# SYNFIX: Dependency-Aware Program Repair via RelationGraph Analysis

Xunzhu Tang*[1], Jiechao Gao*[2], Jin Xu*[3], Tiezhu Sun[1], Yewei Song[1], Saad Ezzini[4], Wendkûuni C. Ouédraogo[1], Jacques Klein[1], and Tegawendé F. Bissyandé[1]

[1]University of Luxembourg
[2]Center for SDGC, Stanford University
[3]Jilin University
[4]King Fahd University of Petroleum and Minerals

## Abstract

Recently, software development automation has been significantly improved by large language model (LLM) advancements, including bug localization, code synthesis, program repair, and test generation. However, most prior work on program repair focuses on isolated elements, such as classes or functions, neglecting their interdependencies, which limits repair accuracy. We present SYNFIX, a RelationGraph-based approach that integrates LLMs with structural search and synchronization techniques for coordinated program repair across codebases. SYNFIX constructs a **RelationGraph** to capture relationships among classes, functions, variables, and their interactions (e.g., imports, inheritance, dependencies). Each RelationGraph node includes detailed code descriptions to help LLMs understand root causes and retrieve relevant contexts. By analyzing one-hop nodes in the RelationGraph, SYNFIX ensures repairs account for dependent updates across components. Patch validation is conducted using regression tests from the SWE-bench benchmark suite. Evaluated on SWE-bench datasets, SYNFIX resolves 52.33% of issues in **SWE-bench-lite** (300 GitHub issues), 55.8% in **SWE-bench-verified** (500 issues), and 29.86% in **SWE-bench-full** (2,294 issues), outperforming baselines such as Swe-Agent, Agentless and AutoCodeRover. The codebase is available at https://github.com/Daniel4SE/SynFixCode.

## 1 Introduction

Large language models (LLMs) have reshaped software engineering by automating diverse tasks such as bug localization, program repair, and test generation (Chen et al., 2021; Austin et al., 2021; Li et al., 2023; Wei et al., 2023). However, applying LLMs to repository-level debugging still presents significant challenges. Although existing methods do well at resolving isolated issues, they often fail to capture the broader context of repository-wide dependencies between files, classes, and functions (Zhou et al., 2012; Austin et al., 2021). Repository-level tasks are inherently difficult as they involve complex interactions across components that require a holistic understanding of the codebase. Traditional approaches struggle to address these challenges, often leading to partial or suboptimal fixes. Consequently, there is a pressing need for solutions that can effectively model and address the interconnected nature of large-scale codebases.

To bridge this gap, benchmarks like SWE-bench and its subset swebenchlite (Jimenez et al., 2024a; swe, 2024) evaluate real-world software development tasks. SWE-bench includes issues from open-source repositories, testing tools on challenges such as multi-file dependencies, parameter synchronization, and regression testing. SWE-bench Lite focuses on bug-fixing in a controlled setting. These benchmarks reveal current limitations and help develop scalable debugging solutions.

Agent-based systems have been proposed to address these repository-level challenges by equipping LLMs with toolsets for iterative decision-making (Zhang et al., 2024a; Yang et al., 2024b). These systems enable LLMs to autonomously decide on actions, perform operations like file edits or test executions, and iteratively refine their solutions based on feedback. While promising in theory, agent-based approaches face key limitations in practice; for example, the complexity of tool usage often introduces abstraction layers that are prone to errors, especially when mapping actions to APIs (Liu et al., 2024a). Moreover, the lack of robust planning mechanisms means that agents frequently make suboptimal decisions, leading to inefficiencies and unnecessary costs (Xia et al., 2024). The iterative, multi-turn nature of these methods exacerbates these issues, as incorrect decisions in early stages can cascade into larger problems, significantly hindering performance (Olausson et al., 2023; Shi

*Equal contribution.



*Findings of the Association for Computational Linguistics: ACL 2025*, pages 4878–4894
July 27 - August 1, 2025 

et al., 2023). Furthermore, their limited ability to self-reflect and filter irrelevant or incorrect feedback amplifies these challenges, making agent-based systems less reliable for large-scale debugging tasks.

To address these limitations, we propose **SYNFIX**, including four-phase modules: RelationGraph, Localization, Synchrony, and Patch Validation. SYNFIX shifts the focus from iterative exploration to structured, deterministic processes that explicitly model the structural relationships within a codebase. Central to SYNFIX is the **RelationGraph**, which captures dependencies among files [1], functions, variables, and classes, enabling a comprehensive understanding of the codebase. This representation ensures that fixes in one part of the codebase are propagated correctly across related components, maintaining repository-wide consistency. By leveraging the RelationGraph, SYNFIX systematically localizes issues, identifies affected contexts, and synchronizes repairs to address interdependencies effectively. Unlike agent-based systems, SYNFIX avoids uncontrolled exploration, ensuring that each repair step is precise and well-informed.

In the RelationGraph phase, SYNFIX models the structural dependencies in the codebase. During the Localization phase, it uses this framework to identify suspicious files, classes, and specific lines of code. Synchrony ensures that fixes propagate correctly across interconnected components, while Paired Patch Validation rigorously tests the patches using regression suites to ensure correctness and compatibility with existing functionality. This deterministic and interpretable design eliminates the inefficiencies of agent-based systems, offering a cost-effective solution for real-world software debugging. SYNFIX has demonstrated exceptional performance on SWE-bench benchmarks (swe, 2024), achieving state-of-the-art results while maintaining scalability and reliability. Its success highlights the potential of structured, agentless approaches to set new standards for efficient, repository-level program repair.

### Contributions

- **RelationGraph-Driven Framework**: SYNFIX introduces a novel RelationGraph-based approach that systematically models dependency relationships within a repository. Our approach enables precise localization and coordinated synchronization of fixes.
- **Enhanced Localization and Synchronization**: SYNFIX employs a hierarchical localization strategy to pinpoint issues with high precision and ensures that the fixes propagate correctly across interconnected components through its *synchrony* phase.
- **Efficient Validation Mechanism**: Through Patch Validation, SYNFIX rigorously tests proposed patches using regression suites, ensuring correctness and compatibility while minimizing the risk of introducing new errors.
- **State-of-the-Art Performance**: SYNFIX achieves superior results on SWE-bench benchmarks, outperforming agent-based baselines in resolution rates, scalability, and cost-efficiency.

## 2 Methodology

Figure 2 presents an overview of SYNFIX, which operates in four main phases: **RelationGraph** (1), **Localization** (2), **Synchrony** (3), and **Patch Validation** (4, 5, 6, 7, and 8).

Given a problem statement and an existing codebase as inputs, SYNFIX proceeds as follows. 1 First, SYNFIX constructs a RelationGraph from the input codebase, capturing its hierarchical structure, variables, classes, functions, and their calling relationships. 2 Next, using both the RelationGraph and the problem statement, SYNFIX prompts a low-cost LLM (e.g., GPT-3.5) to identify and rank the top $N$ most suspicious nodes that may require edits to address the issue. These nodes are presented with their surrounding context—such as the corresponding function, class, or code variable line.

3 To account for synchronous changes, SYNFIX then identifies the one-hop neighbors of the suspicious nodes, extracting these neighbors along with their corresponding lines, functions, and classes. These extracted contexts serve two purposes: (1) as additional input for the LLM, they help evaluate and refine the understanding of issues in the target suspicious nodes, ultimately facilitating more accurate modifications; and (2) after addressing the primary target nodes, the LLM evaluates whether corresponding updates are necessary for these one-hop neighbors, especially in cases where changes to parameters or related structures in the primary nodes propagate to their neighbors. The first purpose is discussed in detail in the main text, while the second is elaborated upon in the Appendix.

4 With this preliminary set of suspicious nodes identified, SYNFIX provides their full code contexts

[1] Why file name or folder name is considered as a kind of node: Some files they could have the function with the same name but in different features.

to the LLM. The model refines and narrows down the potential edit locations (e.g., specific lines) for both dynamic and static process. This step ensures that the final chosen edit points are as accurate as possible. ❺ In the repair phase, SYNFIX presents the problem statement and the identified edit locations to the LLM. The model is then prompted to propose patches aimed at resolving the identified issues.❻ Once patches are generated, SYNFIX attempts to apply them to the original codebase. If a patch cannot be cleanly applied, the process returns to the previous step ❹, repeating for a preconfigured number of iterations. ❼ If the patch application is successful, SYNFIX validates the updated codebase by running regression tests. If these tests pass, the patch is considered successful.

❽ Finally, SYNFIX updates the RelationGraph to reflect the changes in the codebase, ensuring that subsequent iterations start from an accurate and up-to-date state. The following sections provide a more detailed explanation of each phase within SYNFIX.

### 2.1 RelationGraph Building

The RelationGraph is a directed graph where each node represents a code entity, and edges capture hierarchical or dependency relationships between these entities. On average, per 10,000 lines of code, the RelationGraph comprises roughly 1,335 nodes (including 1,020 variable names, 278 function names, and 37.4 class names) and about 2,600 edges representing call relationship. We store and compute our RelationGraph using networkx (Hagberg et al., 2008) and DGL (Deep Graph Library) (Wang et al., 2019).

**Nodes:** Each node in the RelationGraph corresponds to a distinct code entity within the project, such as: folder name (with relative path), file name (with relative path), variable name, class name, function name. By including nodes at multiple abstraction levels, the RelationGraph allows for a holistic understanding of the codebase—from its highest-level directory structure down to the granular level of individual methods.

**Edges:** Edges in the RelationGraph represent structural, import-based, and functional dependencies. For example:

- Folders and their subfolders, as well as folders and the files within them, are connected by edges to reflect hierarchical relationships.
- Files that import other files or modules are linked, indicating a dependency between these code units.

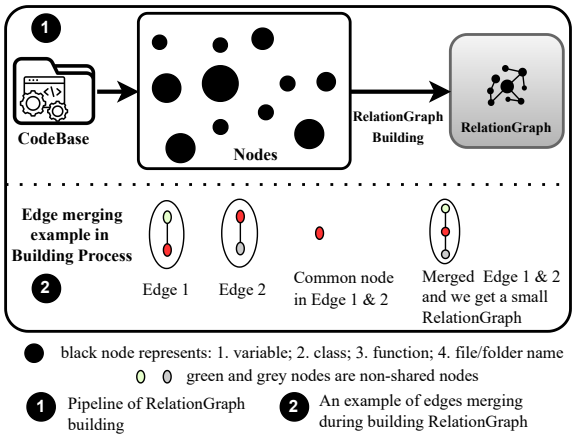


Figure 1: Pipeline of RelationGraph building and an edge merging example during building RelationGraph.

- If a variable, class, or method defined in one file is referenced or invoked by another file or entity, an edge is drawn to represent that relationship (e.g., a method A calling method B).

**Subgraph Construction and Merging:** RelationGraph can be seen as a merging result of a lot of edges. As shown in Figure 1, we firstly take an overall pipeline of building RelationGraph. Then, specifically, we take an example to show how these edges can be merged into a RelationGraph. **Overall pipeline**: Given a codebase, we can extract numerous edges that link two nodes with calling relationships. Then, with these edges, we can merge them them into the final RelationGraph. An example of merging will be introduced in the following. **Example of merging edges**: As Edge 1 and Edge 2 have a common red node, we can merge these two edges into a small graph, just shown in the second part in Figure 1.

### 2.2 Localization

In this section, we focus on code localization for problem statements and obtain necessary contextual information for later patch generation by foundation large language models.

As a codebase contains various codes and complex structural organization, we need to quickly localize the most suspicious code line related to the problem statement. To systematically identify and rank these minimal code regions requiring modification, we integrate a multi-step strategy that includes RelationGraph analysis, embedding-based retrieval, and LLM-based reasoning. The localized code's context serves as input to the LLM for later patch generation. However, considering only the direct context of the localized code line

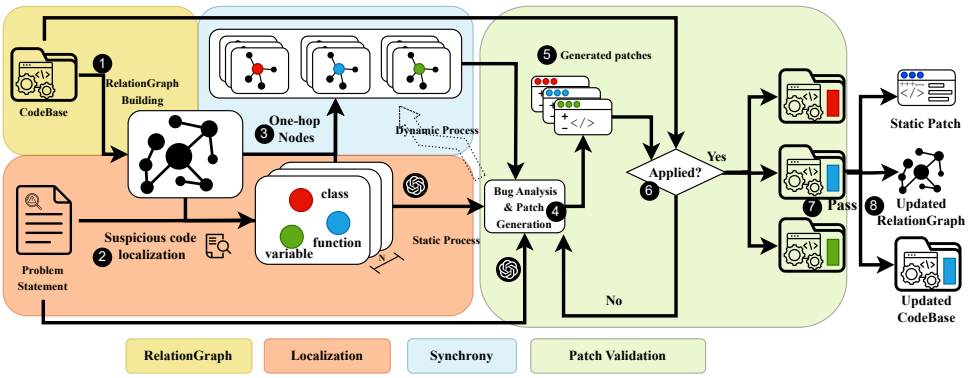


Figure 2: Pipeline of SYNFIX

is insufficient, as critical semantic and structural dependencies may exist elsewhere in the codebase. Thus, step 2 introduces a dynamic process to retrieve additional relevant context, ensuring more informed and accurate patch generation.

Details of the first 2 steps are introduced as follows:

**Step 1: Suspicious Node Identification.** Let the RelationGraph be represented as $G=(V,E)$, where $V$ denotes nodes and $E$ is the set of edges for nodes. Given a problem statement $P$, SYNFIX derives a ranking $R \subseteq V$ of the top-$N$ most suspicious nodes using a hybrid retrieval mechanism:

- **Embedding-based retrieval:** Each node $v \in V$ is converted into a textual representation containing its code snippet, comments, and related files. Using OpenAI's 'text-embedding-3-small' model, embeddings $\phi(v)$ are computed for each node and compared with $\phi(P)$ (the problem statement embedding) using cosine similarity. The top-$N$ candidates with the highest similarity scores are selected.
- **LLM-based reranking:** A structured prompt containing the problem statement and extracted node context is provided to GPT-3.5, which ranks the candidates based on their relevance to the issue.

After **LLM-based reranking**, we can localize the suspicious lines.

**Step 2: Contextualized Error Correction.** Once the most relevant suspicious nodes are identified, SYNFIX refines the error localization through **static process** and **dynamic process**, both of which guide LLM-based correction.

- **Static Process:** The LLM performs error correction using only the local context of the identified suspicious node. This means that the code snippet, along with its immediate surrounding context (e.g., function or class definition), is provided to the model with carefully designed prompts to adjust for potential errors.
- **Dynamic Process:** To improve correction accuracy, SYNFIX extends the context by incorporating one-hop neighbors from the RelationGraph. These additional nodes provide supplementary structural information, such as dependencies and interactions with the suspicious node. The LLM then processes both the direct context and the expanded context from RelationGraph neighbors, improving its ability to generate precise fixes.

Since we have performed localization and provided both static and dynamic process, we supply sufficient contextual information for utilizing LLM in **bug analysis (Step 4) and fixing patch generation**, as illustrated in Figure 2, to address the stated problem. To verify whether the generated patches effectively resolve the issue, we discuss this in **Patch Validation**. Additionally, because the **dynamic process** collects suspicious code one-hop node information from the RelationGraph, these nodes may also exhibit issues since they are structurally related to the suspicious code lines. If such issues arise, further corrective modifications are necessary. This aspect is discussed in **Synchrony**.

### 2.3 Synchrony

In large-scale codebases, modifying a single code entity can introduce inconsistencies in its dependent components. The synchrony phase in SYNFIX ensures that when a suspicious node $v_s$ is modified via a patch $\Delta(v_s)$ (discussed in Appendix E.4), all directly related nodes are checked and updated to maintain consistency. For each suspicious node, SYNFIX identifies its one-hop neighbors, which are components that either reference or are referenced by the modified node. After applying a modification to the suspicious node, SYNFIX

verifies whether any inconsistencies arise in its neighboring components. If inconsistencies are detected, SYNFIX generates additional patches for affected neighbors to maintain consistency.

**Algorithm 1:** Synchronous Repair Process

**Input** : $G=(V,E)$: CallGraph, $v_s$: suspicious node, $\Delta(v_s)$: modification to $v_s$
**Output** : Updated codebase with consistent synchronous modifications

Extract one-hop neighbors:
$N(v_s) \leftarrow \{v \in V \mid (v_s,v) \in E \vee (v,v_s) \in E\}$.
**foreach** $v \in N(v_s)$ **do**
  **if** `ConsistencyCheck`$(v,\Delta(v_s))$ = False **then**
    Apply modification $\Delta(v)$ to $v$.
**return** *Updated codebase.*
**Function** `ConsistencyCheck`$(v,\Delta(v_s))$**:**
  **Input** : $v$: neighbor node, $\Delta(v_s)$: modification to suspicious node
  **Output** : True if consistent, False otherwise
  Evaluate whether modifications to $v_s$ introduce inconsistencies in $v$.
  **if** *Dependencies in v are affected* **then**
    **return** *False*
  **return** *True*

Algorithm 1 details the synchronization process. The algorithm first extracts one-hop neighbors from the RelationGraph, ensuring that all directly dependent components are identified. Each extracted neighbor is subjected to a consistency check, where its local context is analyzed against the modifications made to $v_s$. If dependencies are affected, SYNFIX determines whether a corrective modification is required. When inconsistencies are detected, SYNFIX generates and applies a corrective patch $\Delta(v)$ to update the affected node. This process iterates until all affected components are brought into a consistent state. By systematically iterating through affected dependencies, SYNFIX ensures that modifications do not introduce unintended errors elsewhere in the codebase, thereby preserving functional correctness and structural integrity.

The synchronization process follows a structured procedure: (1) The algorithm first identifies all immediate dependencies of the modified node using the RelationGraph, which provides a representation of interdependencies across the codebase. (2) For each identified neighbor, SYNFIX assesses whether the applied modification affects its functional behavior by examining variable dependencies, function invocations, and data flow relations. If no inconsistency is found, the node remains unchanged. (3) If inconsistencies arise, SYNFIX formulates a corrective patch based on the nature of the dependency violation. This patch may involve updating function calls, adjusting variable assignments, or restructuring the code to preserve consistency. (4) The process repeats iteratively, propagating updates across the affected regions until all dependencies are synchronized.

### 2.4 Patch Validation

Patch validation is the final and critical phase of SYNFIX, ensuring that generated patches not only resolve the identified issues but also preserve the overall functionality of the codebase. In this phase, the updated codebase, denoted as $C_{\text{mod}}$, is rigorously evaluated against a curated test suite $\mathcal{T}$ to verify that all changes are both syntactically correct and semantically consistent with expected behavior.

The process begins as soon as a patch $\Delta$ is applied to the original codebase $C_{\text{orig}}$. SYNFIX executes all tests in $\mathcal{T}$—which include both regression tests for baseline behavior and functional tests targeting specific features—to ensure that the modifications do not introduce new errors. A patch is considered valid only if every test passes.

If any test fails, SYNFIX enters an iterative refinement loop. Diagnostic feedback (such as error logs and stack traces) is collected and used to prompt the LLM to refine $\Delta$. In this refinement process, the LLM may adjust the patch for the originally identified suspicious node or, if new issues emerge, generate a patch for a different node. This iterative cycle continues until a valid patch is produced or a maximum number of iterations $K_{\text{max}}$ is reached. A history of attempted patches is maintained to prevent redundant corrections and cyclical refinements.

Algorithm 2 details the patch validation and refinement process. In summary, SYNFIX applies a patch to $C_{\text{orig}}$, validates the resulting $C_{\text{mod}}$ against $\mathcal{T}$, and, if necessary, iteratively refines the patch based on feedback until the codebase passes all tests.

## 3 Experiment Design

### 3.1 Datasets

**SWE-bench** (Jimenez et al., 2024b): We evaluate SYNFIX using three benchmarks: Lite, Verified, and Full, which include real-world software engineering problems requiring patches. **Lite:** A curated set of 300 high-quality, self-contained problems designed for rapid prototyping and testing. **Verified** and **Full:** Larger benchmarks covering increasingly complex problems and extensive codebases.

**BigCodeBench** (Zhuo et al., 2024): BigCodeBench

**Algorithm 2:** Patch Validation

**Input** : $C_{\text{orig}}$: original codebase, $\Delta$: generated patch, $\mathcal{T}$: test suite, $K_{\max}$: max iterations
**Output** : $C_{\text{mod}}$: validated codebase or failure status

Initialize: $k \leftarrow 0$, $C_{\text{mod}} \leftarrow C_{\text{orig}}$.
**repeat**
  Apply $\Delta$ to $C_{\text{orig}}$, resulting in $C_{\text{mod}}$.
  **if** `Validate`($C_{mod}$, $\mathcal{T}$) = True **then**
    **return** $C_{mod}$.
  **else**
    Increment $k \leftarrow k+1$.
    Request refinement from LLM for $\Delta$.
**until** $k = K_{max}$
**return** Failure (no valid patch found within $K_{\max}$).
**Function** `Validate`($C_{mod}$,$\mathcal{T}$)**:**
  **Input** : $C_{\text{mod}}$: modified codebase, $\mathcal{T}$: test suite
  **Output** : True if all tests pass, False otherwise
  **foreach** $T_i \in \mathcal{T}$ **do**
    **if** Run($T_i$, $C_{\text{mod}}$) $\neq O_i$ **then**
      **return** False.
  **return** True.

is a benchmark designed to evaluate large language models on function-level code generation tasks. It includes 1,140 tasks spanning 139 libraries across seven domains, with an average of 5.6 test cases per task to ensure rigorous evaluation.

### 3.2 Baselines & Metrics

**Baselines** We compare SYNFIX against 34 state-of-the-art baseline approaches on SWE-bench [2], including both open-source and closed-source methods. These baselines encompass a diverse set of techniques, including agent-based, agentless, and retrieval-augmented generation (RAG) models. Closed-source baselines (indicated accordingly) primarily provide final submission patches without revealing their execution trajectories, making it difficult to analyze the intermediate steps taken to generate fixes.

**Metrics** Following prior work (Zhang et al., 2024b; Xia et al., 2024), we evaluate performance using three key metrics: (1) **% Resolved** – the percentage of issues successfully fixed by the tool within the benchmark dataset; (2) **Avg. $ Cost** – the average inference cost per issue, reflecting computational efficiency; and (3) **Avg. # Tokens** – the average number of input and output tokens used per query, indicating the LLM's resource consumption. These metrics provide a comprehensive assessment of effectiveness, efficiency, and scalability across different repair strategies.

[2] https://www.swebench.com

### 3.3 Research Questions

**[RQ1] Performance:** How effective is SYNFIX in resolving problems?
**[RQ2] Ablation Study:** How does removing components like dynamic process or RelationGraph affect performance?
**[RQ3] Effect of Foundation Models:** What is the difference of SYNFIX by using different foundation models?
**[RQ4] Transferability:** How well does SYNFIX generalize to new data, such as BigCodeBench?

## 4 Experimental Results

Table 1: Results on SWE-bench Lite.

| Tool | LLM | % Resolved | Avg. $ Cost | Avg. # Tokens |
|---|---|---|---|---|
| CodeStory Aide (cod, 2024) | GPT-4o+Claude 3.5 S | 129 (43.00%) | - | - |
| Bytedance MarsCode (Liu et al., 2024b) | N/A | 118 (39.33%) | - | - |
| Honeycomb (hon, 2024) | N/A | 115 (38.33%) | - | - |
| MentatBot (men, 2024) | GPT-4o | 114 (38.00%) | - | - |
| Gru (gru, 2024) | N/A | 107 (35.67%) | - | - |
| Isoform (iso, 2024) | N/A | 105 (35.00%) | - | 41,963 |
| SuperCoder2.0 (sup, 2024) | N/A | 102 (34.00%) | - | - |
| Alibaba Lingma Agent (lin, 2024) | GPT-4o+ Claude 3.5 S | 99 (33.00%) | - | - |
| Factory Code Droid (fac, 2024) | N/A | 94 (31.33%) | - | - |
| Amazon Q Developer-v2 (ama, 2024) | N/A | 89 (29.67%) | - | - |
| SpecRover (Ruan et al., 2024) | GPT-4o+ Claude 3.5 S | 93 (31.00%) | $0.65 | - |
| CodeR (Chen et al., 2024) | GPT-4 | 85 (28.33%) | $3.34 | 323,802 |
| MASAI (Arora et al., 2024) | N/A | 84 (28.00%) | - | - |
| SIMA (sim, 2024) | GPT-4o | 83 (27.67%) | $0.82 | - |
| IBM Research Agent-101 (ibm, 2024) | N/A | 80 (26.67%) | - | - |
| OpenCSG StarShip (ope, 2024a) | GPT-4 | 71 (23.67%) | - | - |
| Amazon Q Developer (ama, 2024) | N/A | 61 (20.33%) | - | - |
| RepoUnderstander (Ma et al., 2024) | GPT-4 | 64 (21.33%) | - | - |
| AutoCodeRover-v2 (aut, 2024) | GPT-4o | 92 (30.67%) | - | - |
| RepoGraph (rep, 2024) | GPT-4o | 89 (29.67%) | - | - |
| Moatless (moa, 2024) | Claude 3.5 S | 80 (26.67%) | $0.17 | - |
| | GPT-4o | 74 (24.67%) | $0.14 | - |
| OpenDevin+CodeAct v1.8 (ope, 2024b) | Claude 3.5 S | 80 (26.67%) | $1.14 | - |
| Aider (Gauthier, 2024) | GPT-4o+ Claude 3.5 S | 79 (26.33%) | - | - |
| SWE-agent (Yang et al., 2024b) | Claude 3.5 S | 69 (23.00%) | $1.62 | 521,208 |
| | GPT-4o | 55 (18.33%) | $2.53 | 498,346 |
| | GPT-4 | 54 (18.00%) | $2.51 | 245,008 |
| AppMap Navie (app, 2024) | GPT-4o | 65 (21.67%) | - | - |
| AutoCodeRover (Zhang et al., 2024a) | GPT-4 | 57 (19.00%) | $0.45 | 38,663 |
| RAG (Yang et al., 2024b) | Claude 3 Opus | 13 (4.33%) | $0.25 | - |
| | GPT-4 | 8 (2.67%) | $0.13 | - |
| | Claude-2 | 9 (3.00%) | - | - |
| | GPT-3.5 | 1 (0.33%) | - | - |
| Agentless (Xia et al., 2024) | GPT-4o | 96 (32.00%) | $0.70 | 78,166 |
| **SYNFIX** | GPT-4o | 157 (52.33%) | $0.56 | 39,871 |

(Note: N/A means that the close-source tool does not show its LLM in the official SWE-bench leaderboard.)

### 4.1 [RQ1]: Overall Performance

We evaluated SYNFIX on three benchmark datasets—SWE-bench Lite, SWE-bench Full, and SWE-bench Verified—each reflecting different levels of issue complexity and description quality. In this section, we discuss the performance on SWE-bench Lite. Moreover, we also elaborate on the performance on other two verions SWE-bench in Appendix Sec F.1 and Sec F.2.

On SWE-bench Lite, SYNFIX resolves 157 out of 300 issues (52.33%), a marked improvement over competing agent-based systems (e.g., AutoCodeRover-v2 at 30.67%) and agentless approaches (32.00%). This high resolution rate is

intrinsically tied to SYNFIX 's structured approach: a RelationGraph is built to capture hierarchical and dependency relationships across code entities, which enables a fine-grained, hierarchical localization of faults. By combining static structural analysis with dynamic process that captures runtime behaviors—such as parameter interactions and execution paths—SYNFIX is able to pinpoint error-prone regions with enhanced precision. Furthermore, its paired patch validation mechanism rigorously tests proposed fixes using regression suites, ensuring that corrections maintain overall codebase integrity while minimizing the introduction of new errors.

In addition to its repair accuracy, SYNFIX demonstrates remarkable efficiency. It operates at an average cost of only $0.56 per issue and utilizes 39,871 tokens per query, representing a significant reduction in resource consumption compared to methods like CodeR (which incurs $3.34 per issue and 323,802 tokens). On SWE-bench Verified, where the problem descriptions are of higher quality and demand stricter repair criteria, SYNFIX achieves a 55.8% resolution rate. Its intrinsic localization performance reporting accuracies of 76.4% at the class level, 56.7% at the function level, and 40.7% at the line level—demonstrates the effectiveness of the RelationGraph-driven analysis in handling complex, multi-file dependencies. Overall, integrating dynamic feedback, structured localization, and iterative patch refinement in SYNFIX yields high repair success and ensures scalability and cost-effectiveness in diverse debugging scenarios.

### 4.2 [RQ2]: Ablation Study

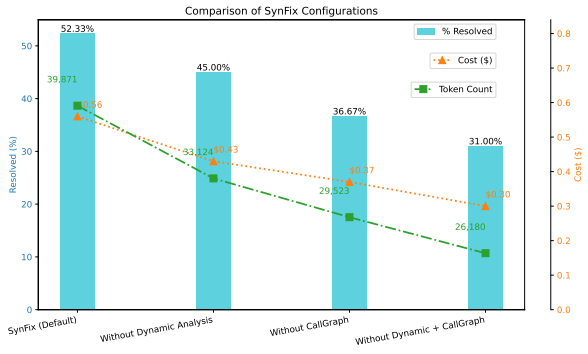


Figure 3: Ablation Study: Impact of removing dynamic process and RelationGraph dependencies on SWE-bench Lite.

We evaluated SYNFIX by selectively disabling dynamic process and RelationGraph dependencies to assess their individual contributions, as shown in Figure 3. In the default configuration—with both components enabled—SYNFIX achieves a resolution rate of 52.33%, an average cost of $0.56 per issue, and consumes 39,871 tokens per query. When dynamic process is removed, the resolution rate decreases to 45.00%, with an average cost of $0.43 and 33,124 tokens used, highlighting the importance of runtime context for parameter synchronization and inter-module interactions. Excluding RelationGraph dependencies further lowers the resolution rate to 36.67%, with an average cost of $0.37 and 29,523 tokens, underscoring the role of structured contextual information in accurate fault localization. In the scenario where both dynamic process and RelationGraph dependencies are disabled, the resolution rate plummets to 31.00%, with the cost reduced to $0.30 and token usage dropping to 26,180. Although these degraded configurations yield marginal savings in cost and token consumption, the significant decline in resolution rate confirms that integrating both dynamic and structural analyses is crucial for optimal repair performance.

### 4.3 [RQ3]: Results on Foundation Models

Table 2 presents an intrinsic performance comparison of foundation models powering SYNFIX on SWE-bench Lite. Our evaluation includes closed-source models (GPT-4o and Claude 3.5 Sonnet) and open-source models (Qwen-2.5 Coder (Yang et al., 2024a) and LLAMA3.1-405B (Dubey et al., 2024)).

The closed-source models exhibit a clear cost-performance trade-off. GPT-4o achieves the highest resolution rate at 52.33% with an average query time of 3.8 seconds, costing $0.56 per query and consuming 39,871 tokens. This suggests that GPT-4o's ability to leverage additional tokens translates into richer contextual understanding and improved accuracy. In contrast, Claude 3.5 Sonnet attains a slightly lower resolution rate of 48.67% but processes queries faster (2.9 seconds) and more cost-effectively ($0.48 per query with 32,124 tokens), making it a strong choice for high-throughput or budget-sensitive applications. Among the open-source models, Qwen-2.5 Coder achieves a 45.67% resolution rate with notably low cost, though its token usage remains comparable to GPT-4o, which may indicate a more efficient, yet less context-rich processing approach. LLAMA3.1-405B, with a resolution rate of 42.33% and a longer average query time (4.3 seconds), appears to be less optimized for the intricacies of debugging tasks.

Table 2: Performance comparison between GPT-4o and Claude 3.5 Sonnet on SWE-bench Lite.

| Model | % Resolved | Avg. Time (s) | Avg. $ Cost | Avg. Tokens |
|---|---|---|---|---|
| GPT-4o | 157 (52.33%) | 3.8 | $0.56 | 39,871 |
| Claude 3.5 Sonnet | 146 (48.67%) | 2.9 | $0.48 | 32,124 |
| Qwen-2.5 Coder | 137 (45.67%) | 3.5 | $0.12 | 39,253 |
| LLAMA3.1-405B | 127 (42.33%) | 4.3 | $0.59 | 37,874 |

### 4.4 [RQ4]: Performance on BigCodeBench

Table 3 summarizes the Pass@1 results obtained from our experiments on the BigCodeBench benchmark using SynFix integrated with GPT-4o, Qwen, and LLAMA3. Our evaluation shows that SynFix-GPT-4o outperforms the baseline models, achieving a Pass@1 of 37.2%, compared to the top o1-2024-12-17 configurations at 35.5% (high reasoning) and 34.5% (low reasoning), DeepSeek-R1 at 35.1%, as well as Gemini-Exp-1206 and DeepSeek-V3-Chat, both at 34.1%. SynFix-Qwen and SynFix-LLAMA3 attained Pass@1 scores of 34.2% and 33.8% respectively, demonstrating that the enhanced reasoning capabilities of GPT-4o within SynFix deliver a significant performance boost on program repair tasks.

Table 3: BigCodeBench Performance Comparison.

| Model | Pass@1 |
|---|---|
| o1-2024-12-17 (temperature=1, reasoning=high) | 35.5 |
| DeepSeek-R1 | 35.1 |
| o1-2024-12-17 (temperature=1, reasoning=low) | 34.5 |
| Gemini-Exp-1206 | 34.1 |
| DeepSeek-V3-Chat | 34.1 |
| **SynFix-Qwen-2.5 Coder** | 34.2 |
| **SynFix-LLAMA3.1-405B** | 33.8 |
| **SynFix-GPT-4o** | **37.2** |

## 5 Related Work

### 5.1 Agent-Based Software Engineering

Agent-based systems leverage large language models (LLMs) to iteratively plan and execute multi-step debugging and repair tasks. Early approaches like Devin (dev, 2024) and OpenDevin (ope, 2024b) introduced agents capable of file editing and environment interaction, while later systems such as SWE-agent (Yang et al., 2024b) and Aider (Gauthier, 2024) enhance these methods with direct repository interactions and targeted fault localization. More recent tools, including Moatless (moa, 2024) and AutoCodeRover (Zhang et al., 2024a), integrate code search and context retrieval to refine patch generation. Despite these advances, the high computational costs and complexity of agent-based approaches motivate our simpler, deterministic pipeline.

### 5.2 Localization and Repair

Fault localization techniques have evolved from dynamic methods (e.g., spectrum-based and mutation-based approaches) that require comprehensive test suites, to static, information-retrieval based methods that compare code with bug reports. While effective for single-file issues, these methods struggle with repository-level defects due to complex interdependencies. Recent LLM-based repair techniques (Xia et al., 2023; Kolak et al., 2022) generate and rank candidate patches to improve scalability and accuracy. Our approach adopts a similar patch-generation strategy while ensuring a structured, interpretable, and cost-effective repair process.

Overall, our method builds on prior work while overcoming key limitations. By integrating RelationGraph analysis, we effectively handle cross-file dependencies that traditional fault localization methods often miss. For further detailed related work, please see the Appendix D.

## 6 Conclusion

SYNFIX addresses the challenges of repository-level program repair with a RelationGraph and Synchronous approach. Its four-phase pipeline: RelationGraph, Localization, Synchrony, and Paired Patch Validation—enables precise issue localization, consistent synchronization of changes, and thorough validation to ensure repository integrity. By leveraging the RelationGraph representation, SYNFIX effectively captures code dependencies, enabling it to handle multi-file and cross-module interactions that are common in real-world codebases. This representation not only ensures that fixes are accurate and localized but also helps maintain consistency across interdependent components of the repository. We evaluate our experiments across all versions of SWE-bench datasets. Experimental results indicate that SYNFIX can outperform the state-of-the-art approaches on all versions of SWE-bench datasets across accuracy, and cost.

To facilitate replication, we have made the source code available at

**https://github.com/Daniel4SE/SynFixCode**

## 7 Acknowledgments

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014). The author Jiechao Gao is partially sponsored by funding from Yonghua Foundation.

## Ethics Statement

This work aims to improve repository-level program repair by developing a structured and deterministic tool. SYNFIX uses pre-trained language models and established benchmarks, such as SWE-bench, without introducing or manipulating sensitive or proprietary data. We ensure that the design and evaluation of SYNFIX adhere to ethical research standards, including transparency, reproducibility, and respect for intellectual property. However, SYNFIX's reliance on language models may inherit biases or inaccuracies present in the underlying models, which should be considered in its application.

## Limitations

While SYNFIX achieves strong performance on SWE-bench benchmarks, several factors may limit its effectiveness in real-world scenarios. The tool's performance may vary when applied to highly domain-specific repositories or programming languages that differ from those used in training the underlying language models, potentially affecting its generalizability. Additionally, SYNFIX relies heavily on regression tests to validate patches, meaning that in environments with incomplete or low-quality test coverage, the reliability of the generated repairs may be compromised. Another limitation is that SYNFIX assumes well-documented code dependencies and accurately structured information within the CallGraph; however, in projects with poorly structured or inadequately documented code, accurate fault localization and repair propagation may be hindered. Furthermore, SYNFIX does not currently account for **version-specific code generation** within repositories (e.g., handling differences between **Torch 2.2 and Torch 1.1**), which presents challenges in evolving software ecosystems where dependencies frequently change. Incorporating version-aware dependency tracking could significantly enhance the method's utility, as highlighted by works such as **VersiCode, LibEvolutionEval, and CodeUpdateArena**. To mitigate these issues, future work could integrate heuristic-based validation and robust static analysis techniques to complement regression tests, reducing dependence on test quality. Additionally, expanding SYNFIX to support a wider range of programming languages, domain-specific characteristics, and version-aware dependency management will be critical for improving its adaptability and broadening its applicability.

## References

2024. Agent-101: A software engineering agent for code assistance developed by ibm research. https://github.com/swe-bench/experiments/blob/main/evaluation/lite/20240612_IBM_Research_Agent101/README.md/.

2024. Aide by codestory. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240702_codestory_aide_mixed.

2024. Alex sima. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240706_sima_gpt4o.

2024. Amazon q developer the most capable generative ai–powered assistant for software development. https://aws.amazon.com/q/developer//.

2024. Appmap speedruns to the top of the swe bench leaderboard. https://appmap.io/blog/2024/06/20/appmap-navie-swe-bench-leader/.

2024. Autocoderover autonomous software engineering. https://autocoderover.dev/.

2024. Devin, ai software engineer. https://www.cognition.ai/introducing-devin.

2024. Factory bringing autonomy to software engineering. https://www.factory.ai/.

2024. Honeycomb. https://honeycomb.sh.

2024. Isoform. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240829_Isoform.

2024. Lingma agent. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240622_Lingma_Agent.

2024. Mentatbot: New sota coding agent, available now. https://mentat.ai/blog/mentatbot-sota-coding-agent.

2024. Moatless tools. https://github.com/aorwall/moatless-tools.

2024a. Opencsg starship. https://opencsg.com/product?class=StarShip/.

2024b. Opendevin: Code less, make more. https://github.com/OpenDevin/OpenDevin/.

2024. Repograph: Enhancing ai software engineering with repository-level code graph. https://github.com/ozyyshr/RepoGraph.

2024. The road to ultimate pull request machine. https://gru.ai/blog/road-to-ultimate-pull-request-machine/

2024. Supercoder. https://superagi.com/supercoder/.

2024. Swe-bench lite. https://www.swebench.com/lite.html.

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*.

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*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Paul Gauthier. 2024. Aider is ai pair programming in your terminal. https://aider.chat/.

Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024a. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024b. Swe-bench leaderboard. https://www.swebench.com/.

James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282.

Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao

Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you!

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024a. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 31–42, New York, NY, USA. ACM.

Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024b. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*.

Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *arXiv preprint arXiv:2406.01422*.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 691–701.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

Mike Papadakis and Yves Le Traon. 2015. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232*.

Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR.

Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*.

Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 1–11.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

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

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*, ICSE '23.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024a. Autocoderover: Autonomous program improvement. *Preprint*, arXiv:2404.05427.

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

Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*, pages 14–24. IEEE.

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. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

## A Efficiency Considerations with GPT-4o

Given the iterative nature of patch refinement, SYNFIX prioritizes computational efficiency by utilizing lightweight LLMs such as GPT-4o during the validation and repair loop. By structuring prompts to focus on localized edits and reusing previously analyzed contexts, the overhead of LLM calls is minimized. Additionally, leveraging a concise representation of the RelationGraph reduces the input size, allowing SYNFIX to maintain cost-effective and rapid iterations.

Paired patches validation ensures that every modification proposed by SYNFIX is rigorously tested for both correctness and consistency. By combining regression testing, iterative refinement, and efficient LLM-based validation, this phase acts as the final safeguard in the pipeline, delivering useable patches.

## B Matching Prompt for Using Chatgpt 3.5

```
You are tasked with identifying suspicious nodes in a software repository based on the given problem description. Each node represents a file, class, or function, along with its structural relationships. The goal is to rank the top N nodes that are most likely relevant to solving the problem. Below is the RelationGraph summary and the problem statement:
RelationGraph: [Insert hierarchical representation here]
Problem Statement: [Insert problem description here]
Task: Provide a ranked list of N suspicious nodes, along with a brief justification for each. The justification should explain why the node is relevant in terms of function, dependencies, or potential issues.
```

## C Analysis Prompt for Using Chatgpt 4o

```
You are tasked with evaluating dependencies in a codebase. The main node has been modified to address an issue, and its neighboring nodes in the dependency graph may require corresponding updates.
Main Node: [Code and proposed modifications for v_s]
Neighboring Nodes: [Skeleton representation of N(v_s)]
Task: For each neighbor, indicate whether it requires updates and, if so, describe the changes needed to maintain consistency.
```

## D Full version of related work

### D.1 Agent-Based Software Engineering

**Rise of Agent-Based Systems in Software Development** Agent-based systems have gained prominence in software engineering, leveraging the iterative and interactive capabilities of large language models (LLMs) to solve complex tasks. Tools like Devin (dev, 2024) and its open-source counterpart OpenDevin (ope, 2024b) pioneered the

integration of agents to handle multi-step processes, such as planning, file editing, and environment interaction. These systems use LLMs to iteratively perform actions, such as utilizing file editors or terminal tools, to achieve task objectives.

Other systems have expanded on this foundation by introducing specialized functionalities. For instance, SWE-agent (Yang et al., 2024b) incorporates a custom agent-computer interface to facilitate direct interactions with repositories, enabling tasks such as reading and editing files or running shell commands. Similarly, Aider (Gauthier, 2024) employs static and dynamic analysis techniques to construct repository maps, allowing LLMs to precisely localize and patch defective code segments.

**Advanced Functionalities in Agent Systems** Expanding on these ideas, Moatless (moa, 2024) integrates code search and retrieval tools with LLM-generated queries to identify relevant code locations, while AutoCodeRover (Zhang et al., 2024a) introduces APIs that allow for targeted searches within specific code contexts, such as class methods. These innovations have enabled iterative refinement of agent workflows, improving localization and debugging accuracy. SpecRover (Ruan et al., 2024), for example, builds on AutoCodeRover by generating functional summaries and feedback messages during debugging steps, ensuring specification-driven repair processes.

Despite their sophistication, agent-based systems often face challenges such as high computational costs, reliance on extensive toolsets, and the risk of cascading errors from incorrect agent decisions. In contrast, AutoFix takes a simpler approach, avoiding these complexities by leveraging a deterministic pipeline for localization, synchronization, and validation. This structured methodology not only reduces computational overhead but also ensures more interpretable and cost-effective debugging workflows.

### D.2 Localization and Repair

**Evolution of Fault Localization Techniques** Fault localization (FL) is a critical step in debugging, aiming to identify the exact code segments responsible for software defects. Traditional FL methods can be broadly categorized into dynamic, static, and learning-based techniques. Dynamic FL approaches, such as spectrum-based FL (SBFL)(Jones and Harrold, 2005; Abreu et al., 2007) and mutation-based FL (MBFL)(Papadakis and Le Traon, 2015; Lou et al., 2020), analyze test executions to identify suspicious code. SBFL identifies lines covered predominantly by failing tests as likely culprits, while MBFL goes further by assessing the impact of code mutations on test outcomes. These methods rely on the availability of a comprehensive test suite, which can limit their applicability in real-world scenarios with sparse testing.

Static techniques address these limitations by using information retrieval (IR) methods (Singhal et al., 2001). These approaches treat fault localization as a search problem, comparing textual similarities between code elements and bug reports (Wang et al., 2015; Saha et al., 2013). While effective for single-file issues, static methods often struggle with multi-file and repository-level defects, where interdependencies between components complicate localization.

**Program Repair Techniques** After localizing faults, program repair techniques focus on generating patches to fix them. Traditional repair methods include template-based (Liu et al., 2019), heuristic-based (Le Goues et al., 2012), and constraint-based (Mechtaev et al., 2016) approaches. While these methods have proven effective in certain scenarios, their scalability and patch diversity are often limited, making them less suitable for complex, multi-component systems.

In contrast, learning-based approaches, particularly LLM-based program repair (Xia et al., 2023; Kolak et al., 2022), have demonstrated superior performance by leveraging modern models' generative capabilities. These tools can sample multiple candidate patches and utilize ranking mechanisms or regression tests to identify the most suitable fix. AutoFix adopts a similar patch-generation strategy, producing concise, diff-format patches (Gauthier, 2024) to minimize the risk of introducing unrelated changes. This design not only improves the reliability of the patches but also makes the repair process more efficient and scalable.

## E Discussion in Details

### E.1 Existing Agentless Approaches

Agentless method (Xia et al., 2024) directly generates patches using static context without iterative agent-based workflows, which reduces computational overhead but limits the ability to handle complex cross-file dependencies and dynamic context shifts. In contrast, our approach integrates structured RelationGraph analysis with iterative refinement, enabling more robust bug

repairs. Table 4 summarizes key differences between the two approaches.

Table 4: Comparison between Agentless and Our Approach.

| Method | Iterative Refinement | Cross-File Analysis |
|---|---|---|
| Agentless | No | Limited |
| Our Approach | Yes | Extensive |

### E.2 RelationGraph Size

The scalability of the RelationGraph is a critical aspect of SYNFIX, as it must efficiently handle large-scale codebases while maintaining accurate structural relationships. To quantify its size, we analyze the SWE-bench datasets and report the following average statistics per 10,000 lines of code:

- 1,020 variable names
- 278 function names
- 37.4 class names

Based on our preliminary analysis, there are approximately 2,600 edges per 10,000 lines of code.

### E.3 Comparison between RelationGraph and Tree-sitter

Table 5 summarizes the key differences between the RelationGraph used in SYNFIX and the Tree-sitter parser.

Table 5: Key Differences between RelationGraph and Tree-sitter

| Aspect | RelationGraph | Tree-sitter |
|---|---|---|
| Cross-File Analysis | Supports cross-file dependencies and inter-file relations, enabling holistic program repair. | Focuses on single-file parsing; lacks built-in cross-file analysis. |
| Scope of Information | Captures only essential elements (e.g., class names, functions, and variables). | Produces a full parse tree with additional grammar rules and nodes, including many syntactic details. |
| LLM Integration | Reduces noise by filtering out irrelevant syntax, focusing on key code elements. | May introduce non-essential structural nodes, potentially adding noise for LLM-based processing. |

**Why RelationGraph over Tree-sitter?**
While Tree-sitter provides detailed per-file parse trees with comprehensive syntactic information, it does not inherently capture cross-file dependencies—an essential aspect for debugging and repair tasks in large-scale codebases. In contrast, RelationGraph tracks file-level dependencies and inter-file relations, offering a minimal yet meaningful representation that is optimized for LLM-based repair. By reducing extraneous details, RelationGraph facilitates more precise error localization and synchronized patch generation, making it particularly well-suited for holistic program repair.

### E.4 Code Modification and Consistency Adjustment

Algorithm 1 includes the line "Apply modification $\Delta(v)$ to $v$ if an inconsistency is detected," which encapsulates the corrective mechanism of SYNFIX. Here, $\Delta(v)$ represents the set of code edits applied to node $v$ when an analysis reveals that its implementation is no longer consistent with a modified node $u$. This mechanism ensures that when a change in one part of the codebase affects dependent components, those dependencies are updated accordingly to maintain overall consistency.

**Example: Python `calculate_discount` Function**

**Original State:**

In Node $u$, a function `calculate_discount` is defined to apply a discount based on two parameters:

```
# Node u: discount_utils.py
def calculate_discount(price
    ↪ : float, discount_rate: float) -> float:
   return price - price * discount_rate
```

Node $v$ calls this function as follows:

```
# Node v: order_processor.py
from discount_utils import calculate_discount

def process_order(price_list: list[float
    ↪ ], discount_rate: float) -> list[float]:
   discounted_prices = []
   for price in price_list:
      discounted_prices.append(
    ↪ calculate_discount(price, discount_rate))
   return discounted_prices
```

**Modification in Node $u$:**

Suppose the design is updated so that `calculate_discount` must enforce a minimum price threshold. A new parameter, `min_price`, is added:

```
# Node u: discount_utils.py
def calculate_discount
    ↪ (price: float, discount_rate
    ↪ : float, min_price: float) -> float:
   discounted = price - price * discount_rate
   return max(discounted, min_price)
```

**Resulting Inconsistency:**

Node $v$ still calls `calculate_discount` with only two arguments, causing a mismatch with the

updated signature.

**Applying $\Delta(v)$:**

A static or dynamic analysis step detects this inconsistency in Node $v$. The corrective patch $\Delta(v)$ is then generated to update the function call, for example by adding a default minimum price value of 5.0:

```
# Node v: order_processor.py
from discount_utils import calculate_discount

def process_order(price_list: list[float
    ↪ ], discount_rate: float) -> list[float]:
   discounted_prices = []
   for price in price_list:
      # Adding
    ↪ the missing min_price argument (e.g., 5.0)
      discounted_prices
    ↪ .append(calculate_discount
    ↪ (price, discount_rate, 5.0))
   return discounted_prices
```

**Outcome:**

After applying $\Delta(v)$, the function call in Node $v$ becomes consistent with the updated definition in Node $u$. This example illustrates how $\Delta(v)$ is used to propagate changes and resolve inconsistencies across interdependent code modules, ensuring that all parts of the codebase remain in harmony following modifications.

# F Results on Full and Verified Versions of SWE-bench

## F.1 Results on SWE-bench Full

**Performance on SWE-bench Full.** The SWE-bench Full dataset, known for its complexity and inclusion of large-scale issues, poses a greater challenge. Table 6 shows that SYNFIX achieves a resolution rate of 29.86%, slightly surpassing leading baselines such as OpenHands + CodeAct (29.38%) and AutoCodeRover-v2.0 (24.89%). Despite the increased difficulty, SYNFIX maintains its competitive edge by leveraging dynamic analysis and fine-grained localization to address issues with multi-level dependencies.

This performance underscores the robustness of SYNFIX across diverse problem types, demonstrating its ability to scale and adapt to more challenging debugging scenarios. By efficiently combining hierarchical localization with paired patches validation, SYNFIX effectively resolves even the most intricate problems in SWE-bench-Full, reinforcing its position as a state-of-the-art tool.

## F.2 Results on SWE-bench Verified

**Performance Comparison.** In terms of resolution rates, GPT-4o achieves the highest performance, resolving 52.33% of issues on SWE-bench Lite, compared to 48.67% for Claude 3.5 Sonnet. This performance gap can be attributed to GPT-4o's advanced contextual understanding, which allows it to handle more complex issues and interdependencies effectively. However, Claude 3.5 Sonnet also delivers competitive results, demonstrating its ability to resolve a significant proportion of issues while being more cost-efficient.

Table 6: Results on SWE-bench Full.

| Tool | LLM | % Resolved |
|---|---|---|
| OpenHands + CodeAct v2.1 | Claude-3.5-Sonnet | 29.38 |
| AutoCodeRover-v2.0 | Claude-3.5-Sonnet | 24.89 |
| Honeycomb | N/A | 22.06 |
| Amazon Q Developer Agent | N/A | 19.75 |
| Factory Code Droid | N/A | 19.27 |
| AutoCodeRover + GPT 4o | GPT 4o | 18.83 |
| SWE-agent | Claude 3.5 Sonnet | 18.13 |
| AppMap Navie | GPT 4o | 14.60 |
| Amazon Q Developer Agent | N/A | 13.82 |
| SWE-agent | GPT 4 | 12.47 |
| SWE-agent | GPT 4o | 11.99 |
| SWE-agent | Claude 3 Opus | 10.51 |
| RAG | Claude 3 Opus | 3.79 |
| RAG | Claude 2 | 1.96 |
| RAG | GPT 4 | 1.31 |
| RAG | SWE-Llama 13B | 0.70 |
| RAG | SWE-Llama 7B | 0.70 |
| RAG | ChatGPT 3.5 | 0.17 |
| **SYNFIX** | Qwen2.5-Coder | **25.72** |
| **SYNFIX** | LLAMA3.1-405B | **24.38** |
| **SYNFIX** | GPT 4o | **29.86** |

# G RelationGraph Driven Error Detection and Synchronous Repair

In this section, we propose a RelationGraph-driven, optimization-based framework for dynamically detecting and resolving bugs within a Python project's call graph. Consider a directed call graph $G = (V, E)$, where each node$v \in V$ represents a code entity (e.g., a file, a class, or a function), and each edge $(u, v) \in E$ encodes an invocation or dependency relationship. We assume that each node $v$ is annotated with two attributes: a binary status status$(v) \in \{0,1\}$ reflecting whether the node has been visited or not, and a nonnegative integer modified_count$(v) \in \mathbb{N}$ counting how many times that node has been modified.

We start by extracting an erroneous node nodeE from the runtime log $L$, which pinpoints a specific line of code and its corresponding node in $G$. We define a traversal $\mathcal{T}$ that, starting at the project's root node $v_{\text{root}}$, proceeds along the directed edges to reach nodeE. All visited nodes along this path

Table 7: Results on SWE-bench Verified.

| Tool | LLM | % Resolved |
|---|---|---|
| Amazon Q Developer Agent (v20241202-dev) | N/A | 55.00 |
| devlo | N/A | 54.20 |
| OpenHands + CodeAct v2.1 | Claude-3.5-Sonnet | 53.00 |
| Engine Labs | N/A | 51.80 |
| Agentless-1.5 + Claude-3.5 Sonnet | Claude-3.5 Sonnet | 50.80 |
| Solver (2024-10-28) | N/A | 50.00 |
| Bytedance MarsCode Agent | N/A | 50.00 |
| nFactorial (2024-11-05) | N/A | 49.20 |
| Tools + Claude 3.5 Sonnet | Claude-3.5 Sonnet | 49.00 |
| Composio SWE-Kit | N/A | 48.60 |
| AppMap Navie v2 | N/A | 47.20 |
| Emergent E1 | N/A | 46.60 |
| AutoCodeRover-v2.0 | Claude-3.5-Sonnet | 46.20 |
| Solver (2024-09-12) | N/A | 45.40 |
| Gru (2024-08-24) | N/A | 45.20 |
| Solver (2024-09-12) | N/A | 43.60 |
| nFactorial (2024-10-30) | N/A | 41.60 |
| Nebius AI Qwen 2.5 72B Generator | LLaMa 3.1 70B Critic | 40.60 |
| Artemis Agent v1 | N/A | 32.00 |
| SWE-agent + Claude 3.5 Sonnet | Claude-3.5 Sonnet | 33.60 |
| nFactorial (2024-10-07) | N/A | 31.60 |
| Lingma Agent + Lingma SWE-GPT 72b | SWE-GPT 72b | 28.80 |
| EPAM AI/Run Developer Agent + GPT4o | GPT4o | 27.00 |
| AppMap Navie + GPT 4o | GPT 4o | 26.20 |
| nFactorial (2024-10-01) | N/A | 25.80 |
| Amazon Q Developer Agent (v20240430-dev) | N/A | 25.60 |
| Lingma Agent + SWE-GPT 7b | SWE-GPT 7b | 25.00 |
| EPAM AI/Run Developer Agent + GPT4o | GPT4o | 24.00 |
| SWE-agent + GPT 4o (2024-05-13) | GPT 4o | 23.20 |
| SWE-agent + GPT 4 | GPT 4 | 22.40 |
| SWE-agent + Claude 3 Opus | Claude 3 Opus | 18.20 |
| Lingma Agent + SWE-GPT 7b | SWE-GPT 7b | 18.20 |
| Lingma Agent + SWE-GPT 7b | SWE-GPT 7b | 10.20 |
| RAG + Claude 3 Opus | Claude 3 Opus | 7.00 |
| RAG + Claude 2 | Claude 2 | 4.40 |
| RAG + GPT 4 (1106) | GPT 4 | 2.80 |
| RAG + SWE-Llama 7B | SWE-Llama 7B | 1.40 |
| RAG + SWE-Llama 13B | SWE-Llama 13B | 1.20 |
| RAG + ChatGPT 3.5 | ChatGPT 3.5 | 0.40 |
| **SYNFIX** | Qwen2.5-Coder | **49.3** |
| **SYNFIX** | LLAMA3.1-405B | **46.4** |
| **SYNFIX** | ChatGPT 4o | **55.8** |

have their status updated to 1, but no modifications are performed during this stage.

Once nodeE is reached, if a bug fix is needed, we perform a local correction to nodeE and increment modified_count(nodeE). This local fix may have implications for context consistency. To model these implications formally, let $\mathcal{X}(v)$ be the local code context of node $v$, and let $\mathcal{N}(v) = N_1(v)$ be its immediate neighbors. We require a multi-context consistency condition:

$$\bigwedge_{v \in V} \text{Consistent}\big(\mathcal{X}(v), \{\mathcal{X}(u) : u \in \mathcal{N}(v)\}\big),$$

i.e., each node's local context must remain consistent with the contexts of its neighbours.

We formulate the modification process as an optimization problem under strict constraints. Suppose each modification to a node $v$ carries a cost $c(v)$ representing the complexity or risk of changing the code. We seek to minimize the total modification cost, subject to constraints on both the recursion depth $D$ of node expansions (for neighbors of nodeE) and the maximum number of total modifications $M_{\max}$ . Specifically:

$$\begin{aligned} \min_{\{x_v\}_{v \in V}} \quad & \sum_{v \in V} c(v) x_v \\ \text{subject to} \quad & x_v \in \{0,1\}, \quad \forall v \in V, \\ & \sum_{v \in V} x_v \leq M_{\max}, \\ & D \leq 3, \\ & \text{Consistent}(\mathcal{X}(v), \{\mathcal{X}(u) : u \in \mathcal{N}(v)\}). \end{aligned} \tag{1}$$

Here, $x_v = 1$ if and only if node $v$ undergoes modification. The constraint $\sum_{v \in V} x_v \leq M_{\max}$ ensures that we do not exceed a predefined global modification budget, which we set to $M_{\max} = 3$. Similarly, $D \leq 3$ ensures that the recursive exploration of neighbors from nodeE does not propagate unbounded.

To implement this procedure, we employ Algorithm 3, which outlines the process of detecting and resolving bugs dynamically. The algorithm first initializes all nodes, then parses the log $L$ to find nodeE, traverses the graph to mark visited nodes, and, if necessary, applies modifications to nodeE and subsequently to its neighbors. A consistency check function $ConsistencyCheck$ is used to ensure that all contexts remain coherent after any series of modifications. If any inconsistency is detected, we reset the relevant nodes' statuses to 0, potentially reintroducing them into the modification pipeline, but still subject to depth and modification count constraints.

This approach yields a highly controlled and theoretically grounded mechanism for error correction. By modelling the resolution as a constrained optimization problem within a bounded recursion framework, we provide guarantees of termination, limited global impact, and multi-context consistency.

**Algorithm 3:** Dynamic and Synchronous Repair

**procedure** DetectResolve($G=(V,E)$, $L$)
- **input :** $G=(V,E)$: call graph of the Python project.
  $L$: runtime log file containing error details.
- **output :** Modified Python project with resolved bugs.
- **foreach** node $v \in V$ **do**
  - Initialize: status$(v) \leftarrow 0$, modified_count$(v) \leftarrow 0$.
- Parse $L$ to locate erroneous node nodeE $\in V$.
- Traverse $G$ from root to nodeE, marking status$(v) \leftarrow 1$.
- **if** Modification required at *nodeE* **then**
  - Modify nodeE;
    modified_count(nodeE) $\leftarrow$ modified_count(nodeE) $+1$.
  - Call ConsistencyCheck(*nodeE*).
- **foreach** neighbor $v \in N_1(nodeE)$ where *status*$(v)=0$ **do**
  - Analyze and modify recursively (respecting $D \leq 3$ and $M_{\max}=3$).
- **return** *Modified Python project.*

**procedure** ConsistencyCheck($v$)
- **input :** $v$: node to verify consistency.
- **output :** Updated dependent nodes if necessary.
- **if** Context consistency is violated **then**
  - Update dependent nodes' status to $0$.