CascadeFix: Multi-Location Automatic Program Repair via Cascading Planning and Generation

Huan Zhang[1,2], Li Kuang[1*], Yang Yang[1], Yilei Fang[1], Yingjie Xia[3]
[1]Central South University [2]Hunan Normal University
[3]Hangzhou Dianzi University
{z_huan, kuangli, yang978, 8208211311}@csu.edu.cn
xiayingjie@hdu.edu.cn

Abstract

Automated Program Repair (APR) is vital for software maintenance. Despite notable advancements, existing methods still face challenges of insufficient bug dependency modeling and inadequate global repair planning when addressing semantically complex multi-location bugs. We propose CASCADEFIX, a multi-location automatic repair method via cascading planning and generation. Firstly, to improve the modeling of semantic and structural dependencies among bugs, three types of bug relationships-Use, Copy, and Nearby-are defined to characterize semantic connection, patch reusability, and contextual interference. Then, to address inadequate global repair planning, a cascading repair planning algorithm is designed to effectively cluster strongly correlated bugs and intelligently assign reasonable repair priorities and operations to each cluster, ensuring the rationality and consistency of global repair. Finally, taking clusters as the basic repair units, a cascading patch generation mechanism is proposed to dynamically integrate intra-cluster dependency information and cross-cluster repair knowledge, producing patches that maintain syntactic correctness and semantic consistency under global dependency constraints. Experiments on Defects4J show that CASCADEFIX resolves 84 multi-location bugs, achieving a 31% improvement over current state-of-the-art methods.

1 Introduction

Automated Program Repair (APR) plays a crucial role in software development and maintenance. (Li et al., 2025b; Tang et al., 2025; Zhang et al., 2025) Recently, Large Language Models (LLMs) have shown promise in this area by effectively addressing bugs at the line, hunk, and function levels. However, their application in real-world scenarios is limited, mainly because they tend to focus on single-location bugs. As a result, research on *multi-location bugs—which require coordinated corrections across non-contiguous code hunks*—remains relatively scarce (Xin et al., 2024).

In fact, multi-location bugs are surprisingly common. A study shows that at least 40% of real-world bugs involve multiple source files (Zhong and Su, 2015). The prevalence of multi-location bugs is likely higher, as single files often contain multiple bug locations. Furthermore, the widely used bug datasets Defects4J (Just et al., 2014) and Bugs.jar (Saha et al., 2018) contain 62% and 76% multi-location bugs, respectively.

For multi-location bugs, existing methods typically invoke LLMs to generate patches for each location sequentially based on physical line order. Figure 1 (a) exposes the limitations of this paradigm using Defects4J (Just et al., 2014). In Closure-64, existing methods repair the method call (Loc A) before its definition (Loc B) based on their line order, triggering a "Signature Mismatch" error. In Time-26, where seven locations require identical fixes, existing methods redundantly invoke LLMs seven times, incurring high computational costs. Figure 1 (b) reveals a sharp performance decline in state-of-the-art LLM-based APR methods (Bouzenia et al., 2025; Kong et al., 2024; Li et al., 2025a; Yin et al., 2024; Zhang et al., 2023; Xia et al., 2023a; Xia and Zhang, 2024): success rates drop from 52% on single-location bugs to merely 13% on multi-location ones. The root cause is that existing methods overlook the intrinsic inter-location dependencies, resulting in a repair process that lacks a global perspective and dynamic adaptability. To overcome this bottleneck, the following two core limitations must be addressed:

Limitation 1: Insufficient Modeling and Utilization of Bug Dependencies. In multi-location bugs, various dependency patterns often exist between different bug locations, including data dependency, control dependency, and structural coupling. How-

* Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 39374–39385
July 2-7, 2026 ©2026 Association for Computational Linguistics

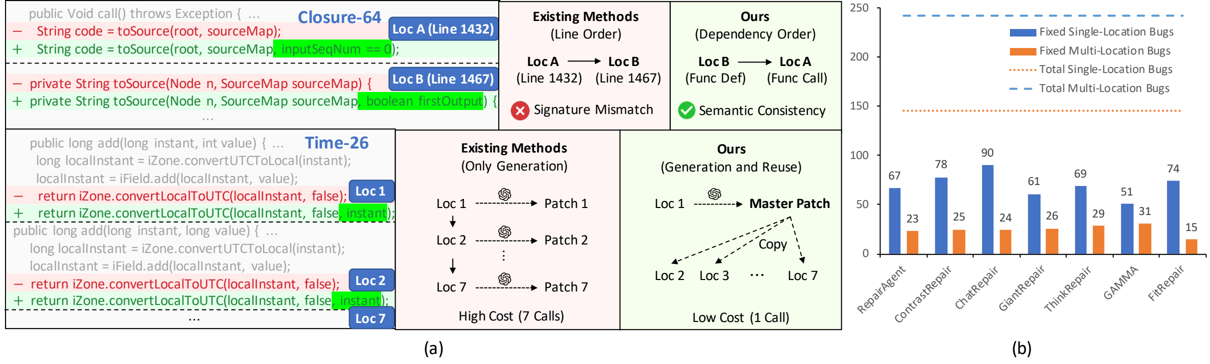

Figure 1: (a) Motivating examples. (b) Comparison of bugs fixed by LLM-based methods on Defects4J.

ever, existing methods typically treat each bug location as an independent repair unit, overlooking the diverse correlations between them. Such a simplified approach can easily lead to semantic conflicts and logical inconsistencies in the generated patches.

Limitation 2: Inadequate Global Repair Planning. Multi-location repair involves a dynamic evolution of code semantics. A fix at one location triggers a cascading effect, reshaping the context for subsequent repairs. Consequently, effective repair cannot simply follow physical line order to invoke LLMs for independent patch generation. The challenge lies in orchestrating a global planning strategy to dynamically determine correlated locations, optimal repair priority, and appropriate operation types, while enabling patch generation that adapts to evolving contexts to ensure global consistency.

To address these challenges, we propose CASCADEFIX, a novel multi-location automatic program repair method via cascading planning and generation. First, to tackle the deficiency in bug dependency modeling, three types of bug relationships—*Use*, *Copy*, and *Nearby*—are defined to characterize semantic connection, patch reusability, and contextual interference between bugs, thereby enhancing the modeling of semantic and structural dependencies among bugs. Secondly, to address inadequate global repair planning, a cascading repair planning algorithm is designed. Through a three-stage progressive workflow, it effectively identifies bugs that require collaborative repair, constructs strongly correlated clusters, and intelligently assigns reasonable repair priorities and operations to each cluster, ensuring the rationality and consistency of global repair. Finally, a cascading patch generation mechanism is proposed, which takes clusters as the basic repair units. Leveraging the code generation capabilities of LLMs, it dynamically integrates intra-cluster dependency information and effectively utilizes historical repair knowledge across clusters, generating patches that ensure syntactic correctness and semantic consistency under global dependency constraints. The main contributions of this paper are:

- We define *Use*, *Copy*, and *Nearby* relationships to model semantic dependencies, patch reusability, and contextual interference among bug locations.
- We propose a graph-driven cascading repair planning algorithm to effectively cluster correlated bugs, determine optimal repair priorities, and assign appropriate repair operations, ensuring global logical consistency.
- We design a context-aware cascading patch generation mechanism that fuses intra-cluster bug dependency information with cross-cluster historical repair knowledge to guide LLMs in generating cross-location collaborative patches.
- Experiments on Defects4J show that CASCADEFIX resolves 84 multi-location bugs, outperforming the state-of-the-art baselines by 31%.

2 Related Work

Traditional APR can be primarily categorized into heuristic-based (Jiang et al., 2018; Yuan and Banzhaf, 2020), constraint-based (Xuan et al., 2016; Le et al., 2017), and pattern-based methods (Liu et al., 2019a; Kim et al., 2013; Liu et al., 2019b). While these methods have proven effective in certain scenarios, their scalability and patch diversity are often limited.

In contrast, learning-based methods (Jiang et al., 2021a; Zhu et al., 2021; Yuan et al., 2022; Ye et al., 2022), particularly LLM-based methods (Li et al., 2025b; Silva et al., 2025; Zhang et al., 2023),

have demonstrated superior performance. They typically either fine-tune LLMs on task-specific datasets (Just et al., 2014; Xia et al., 2023a) or leverage the generative capabilities of pretrained LLMs to analyze buggy code and suggest fixes based on prior knowledge and execution feedback (Xia and Zhang, 2024; Bouzenia et al., 2025; Kong et al., 2024; Yin et al., 2024). Despite their success with single-location repair, they often treat multi-location bugs as independent sub-tasks, lacking the necessary dependency modeling and global planning to effectively resolve complex, coupled bugs.

Although some studies adapt evolutionary search (Yuan and Banzhaf, 2018; Wong et al., 2021), historical code similarity (Saha, 2019), and iterative self-training (Ye and Monperrus, 2024) for multi-location repair, their performance generally falls short of that of LLM-based methods.

3 Method

As shown in Figure 2, we propose CASCADEFIX, a multi-location program repair method via cascading planning and generation. CASCADEFIX comprises three key components: (1) *heterogeneous bug graph construction* to capture dependencies and provide a unified representation for repair; (2) *graph-driven cascading repair planning* to formulate a global repair strategy based on the constructed graph by clustering correlated locations, ranking cluster priorities, and assigning repair operations; and (3) *context-aware cascading patch generation* to fuse intra-cluster dependencies from the bug graph and cross-cluster repair history from prior fixes to generate globally consistent patches.

3.1 Heterogeneous Bug Graph Construction

To effectively model multi-location bugs, we construct a bug-centric heterogeneous bug graph. This graph captures semantic and structural dependencies among bug locations, providing a unified representation for repair.

Formally, we define the graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The node set $\mathcal{V}$ comprises three types of nodes: *File*, *Method*, and *Location*, which respectively represent the buggy files, methods, and code hunks. These nodes are constructed by parsing the buggy code guided by fault localization results. The edge set $\mathcal{E}$ includes seven types of edges: traditional *Import*, *Include*, *Call*, and *Same-Method*, which respectively represent file references, hierarchical structures, method invocation, and same method context relationships. Additionally, it includes *Use*, *Copy*, and *Nearby*, which are designed in this paper to capture semantic connection, patch reusability, and contextual interference among bug locations. The definition and construction of these edges are detailed as follows:

1) Use Edge ($l_i \xrightarrow{Use} l_j$). *Definition:* This edge exists if a code element (e.g., a variable) defined or modified at location l_i is used at l_j via data or control flow. It captures a semantic connection between bug locations, revealing collaborative repair requirements to ensure cross-location logical consistency. *Construction:* We employ static analysis to extract the set of code elements defined at each location. By tracing forward dependencies along data and control flows, we identify downstream bug locations that use these elements.

2) Copy Edge ($l_i \xrightarrow{Copy} l_j$). *Definition:* This edge exists if the code at l_i and l_j is highly similar. It captures patch reusability between bug locations, guiding repair models to apply a patch generated for l_i directly to l_j, thereby reducing redundant LLM invocations. *Construction:* We calculate a similarity score based on the surrounding context lines (s_{line}) and the enclosing method (s_{method}):

$$score(i, j) = \alpha \cdot \text{sim}(s^i_{\text{line}}, s^j_{\text{line}}) + \beta \cdot \text{sim}(s^i_{\text{method}}, s^j_{\text{method}}) \quad (1)$$

where α and β are weights, and sim computes similarity. A *Copy* edge is added if $score(i, j) > \theta$.

3) Nearby Edge ($l_i \xrightarrow{Nearby} l_j$). *Definition:* This edge exists if l_j lies within the local context (e.g., preceding or succeeding lines) of l_i. It captures contextual interference between bug locations, preventing repair models from treating neighboring buggy code as valid context, thereby ensuring accurate bug comprehension. *Construction:* We add a *Nearby* edge if l_j resides within the preceding or succeeding k lines of l_i. Further details are provided in Appendix A.

3.2 Graph-Driven Cascading Repair Planning

Based on the constructed graph, we design a graph-driven cascading repair planning algorithm to formulate a global repair strategy that guides LLMs to achieve logically consistent and computationally efficient repair. As outlined in Algorithm 1, this algorithm employs a three-step progressive mechanism to cluster correlated locations, rank cluster priorities, and assign repair operations.

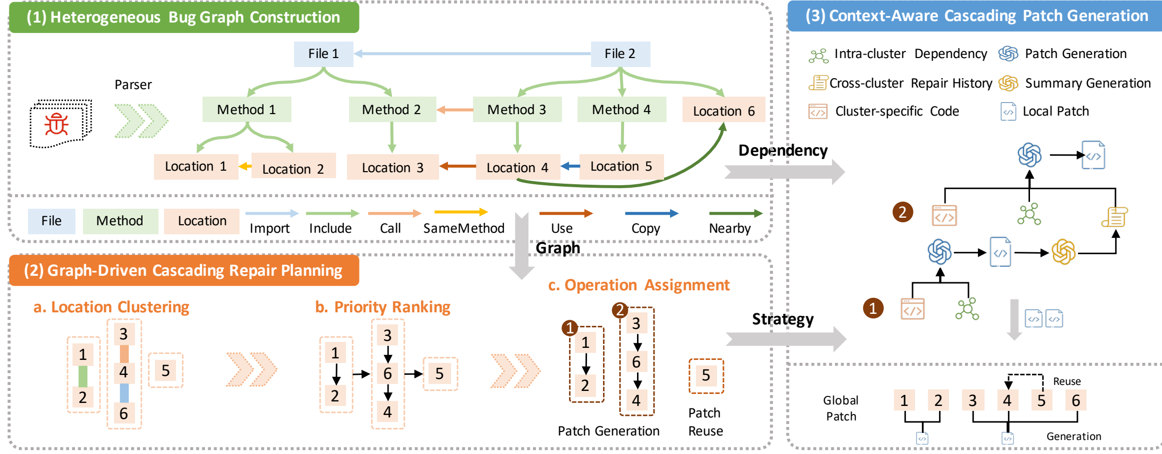

Figure 2: Overview of CASCADEFIX.

1) Correlated Location Clustering This phase aggregates bug locations exhibiting strong semantic and structural dependencies into cohesive clusters to enable collaborative repair. We achieve this by performing connected component analysis on a subgraph composed of *Location* nodes interconnected by *Use*, *Nearby*, and *Same-Method* edges (Lines 2–3).

2) Cluster Priority Ranking This phase determines the global repair order for identified clusters and their constituent bug locations, ensuring alignment with the program's static structure and execution logic. We employ a hierarchical topological sorting strategy, proceeding in a coarse-to-fine manner. First, files are ordered based on *Import* edges (Line 4) to ensure definitions are resolved before usages. Within each file, we prioritize file-level elements (e.g., global fields) (Lines 5–6) and subsequently sort methods via *Call* edges (Lines 7–11), ensuring callees are fixed before callers. At the finest granularity, locations within the same scope are ordered by line number. Finally, the priority of a cluster is derived from the rank of its first appearing location (Line 12).

3) Repair Operation Assignment To improve repair efficiency, this phase assigns repair operations—GENERATION or REUSE—to ranked clusters based on *Copy* edges. The GENERATION operation invokes LLMs to synthesize new patches, whereas the REUSE operation directly applies existing patches from similar locations. Specifically, if the constituent bug locations of multiple clusters exhibit a one-to-one correspondence via *Copy* edges, they are identified as **Equivalent Clusters**. For each equivalence group, the top-ranked cluster is designated as the **Representative Cluster** (Line 13). Consequently, representative and unique clusters are assigned the GENERATION operation, while the remaining clusters are assigned the REUSE operation (Lines 14–17). This mechanism prevents redundant patch generation for bugs with identical patterns and reduces computational cost. Further details are provided in Appendix B.

Algorithm 1. Graph-Driven Cascading Repair Planning

Input: Heterogeneous Bug Graph $\mathcal{G}$
Output: Repair Strategy $\mathcal{S}$

1: **Initialize** $\mathcal{S} \leftarrow \emptyset$, location list $L_{sorted} \leftarrow \emptyset$;
2: **Construct** subgraph $\mathcal{G}'$ with *Location* nodes and edges $\mathcal{E}' \subseteq \{Use, Nearby, Same\text{-}Method\}$;
3: **Identify** connected components in $\mathcal{G}'$ to form cluster set $\mathcal{C}$;
4: **Topologically sort** files based on *Import* edges to obtain F_{sorted};
5: **for each** file $f \in F_{sorted}$ **do**
6: Extract *file-level* locations in f (sorted by line number) and append to L_{sorted};
7: **Topologically sort** methods in f based on *Call* edges to obtain M_{sorted};
8: **for each** method $m \in M_{sorted}$ **do**
9: Extract locations in m (sorted by line number) and append to L_{sorted};
10: **end for**
11: **end for**
12: **Sort** clusters in $\mathcal{C}$ based on the minimum index of their locations in L_{sorted} to obtain $\mathcal{C}_{sorted}$;
13: **Identify** equivalence clusters via *Copy* edges; mark the top-ranked one in each equivalence group as representative;
14: **for each** cluster $c \in \mathcal{C}_{sorted}$ **do**
15: **if** c is representative **or** unique **then** $op \leftarrow$ GENERATION **else** $op \leftarrow$ REUSE;
16: Add (c, op) to $\mathcal{S}$;
17: **end for**
18: **return** $\mathcal{S}$;

3.3 Cascading Patch Generation

The cascading patch generation mechanism adopts clusters as the basic repair units and strictly follows the global strategy formulated by the repair planning algorithm to sequentially invoke the LLM. This mechanism consists of two alternating phases: patch generation and summary generation. Upon completing a cluster fix, the system automatically generates a corresponding summary. This summary is subsequently propagated as structured historical repair knowledge into the generation process of the next cluster, ensuring the orderly transfer and accumulation of the repair context.

The patch generation phase guides the LLM through a prompt that integrates four key components: (1) a task description for patch synthesis; (2) cross-cluster historical repair knowledge and (3) intra-cluster bug dependencies to ensure global semantic consistency; and (4) the relevant buggy code snippets. Subsequently, the summary generation phase distills the core semantic information into structured knowledge. The prompts for this phase consist of two primary elements: (1) a task description for summarization and (3) the fixed code snippets. A concrete example of this cascading process is illustrated in Figure 3.

3.4 Illustrative Example

Figure 3 illustrates the CASCADEFIX workflow on Closure 64, where *Use* edges link Loc 1 and 2 to Loc 3, while a *Nearby* edge links Loc 3 and 4. Based on the identified edges, the planning phase derives repair priorities for three clusters: [Loc 3, Loc 4] $\rightarrow$ [Loc 2] $\rightarrow$ [Loc 1], and assigns the GENERATION operation to each. During the generation phase, the LLM produces patches for each cluster through alternating patch and summary generation, ultimately merging all local fixes into a globally consistent patch.

4 Experiments

4.1 Experimental Setup

Dataset Following prior studies (Xia et al., 2023b; Xia and Zhang, 2022, 2024), we adopt the widely used Defects4J benchmark for evaluation, including both versions v1.2 and v2.0. Defects4J v1.2 comprises 391 bugs from 6 Java projects, while v2.0 introduces 438 new bugs across 9 additional projects. To focus on multi-location repair scenarios, we explicitly filter the datasets to retain only multi-location bugs. As shown in Table 1, this process yields 242 bugs from v1.2 and 278 from v2.0. In total, our experimental dataset consists of 520 real-world multi-location bugs.

Table 1: Statistics of used Defects4J dataset.

Dataset	Project	# Multi-Location Bugs
v1.2	Chart	13
	Closure	77
	Lang	38
	Math	70
	Time	19
	Mockito	25
v2.0	Closure	34
	Cli	26
	Codec	8
	Collections	1
	Compress	31
	Csv	8
	JacksonCore	16
	JacksonDatabind	81
	JacksonXml	5
	Jsoup	50
	JxPath	18

Evaluation Metrics Consistent with previous work (Jiang et al., 2021b; Li et al., 2020), we adopt the *Number of Correct Patches* (# Correct) as our primary metric. It assesses the repair tool's ability to produce accurate patches by counting the number of bugs correctly fixed, which is confirmed through a manual review of plausible patches. Besides, following the study (Yin et al., 2024), we adopt the *Maximum Number of Patches* (# Patch) as a cost metric. It denotes the maximum number of candidate patches that a model is set to loop via running test cases before a correct patch is obtained.

Baselines We compare CASCADEFIX with seven state-of-the-art LLM-based APR methods: FitRepair (Xia et al., 2023a), GAMMA (Zhang et al., 2023), ChatRepair (Xia and Zhang, 2024), RepairAgent (Bouzenia et al., 2025), ContrastRepair (Kong et al., 2024), ThinkRepair (Yin et al., 2024), and GiantRepair (Li et al., 2025a). Additionally, we include four APR methods specifically designed for multi-location bugs: Hercules (Saha, 2019), VarFix (Wong et al., 2021), ITER (Ye and Monperrus, 2024), and DEAR (Li et al., 2022).

4.2 Experimental Implementation

All experiments are conducted on a server equipped with an Intel Core i7-12700KF CPU, 32GB of RAM, and an NVIDIA GeForce RTX 3090 GPU. Regarding hyperparameters, we set the **Copy** edge

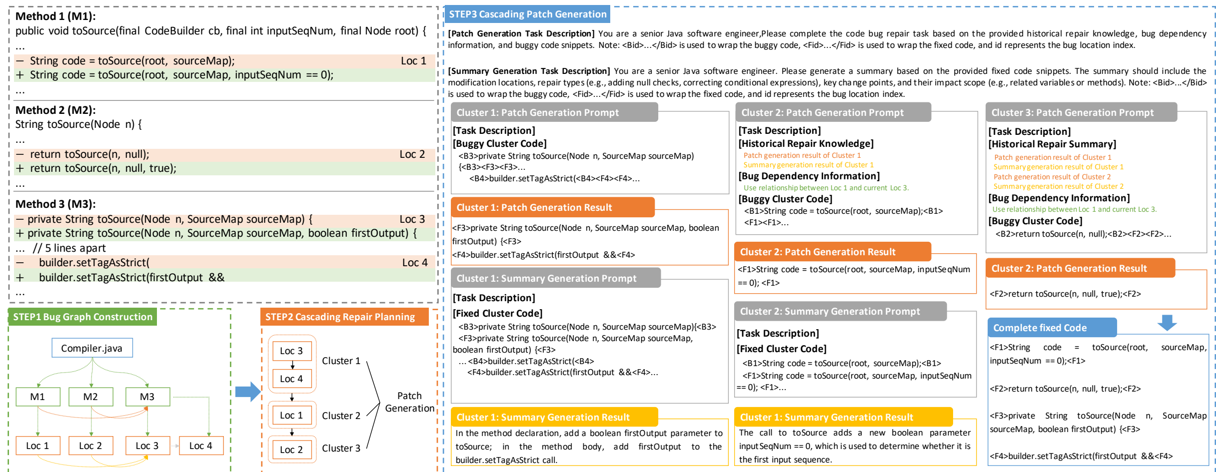

Figure 3: An illustrative example of CASCADEFIX.

weights α and β to 0.5, the similarity threshold to 0.95, and the local context length k to 5. For LLMs, we employ DeepSeek V3 (Liu et al., 2024) and GPT-4o (Hurst et al., 2024) as the backbone models. The temperature is configured to 0.8 with Top-p at 0.95. The timeout for patch validation is fixed at 120 seconds. Additionally, we use `JavaParser` **to** parse the buggy code for bug graph construction.

Since our repair scenarios rely on knowing the bug location, we employ perfect fault localization (where the ground truth location is provided). This is the preferred evaluation setting as it decouples repair performance from localization accuracy, eliminating bias caused by fault localization tools. To ensure a fair comparison, all baseline results reported in this evaluation are also derived under the perfect fault localization setting. For the LLM-based APR baselines, we extract the results from their official replication packages by specifically counting the correct fixes for multi-location bugs. For the direct LLM baselines, we execute the experiments in our environment. For each bug, we **generate** 10 candidate patches.

4.3 Main Results and Analysis

As shown in Table 2, CASCADEFIX demonstrates superior performance and efficiency across both datasets. CASCADEFIX *achieves state-of-the-art results by fixing 84 multi-location bugs, surpassing the best baseline by 31%.* We attribute this improvement to CASCADEFIX's ability to capture semantic and structural dependencies among bug locations. By leveraging these dependencies to formulate a global repair strategy, it generates patches that ensure global semantic consistency. In terms of efficiency, CASCADEFIX *requires a maximum patch budget of only 10, maintaining high effectiveness with minimal computational cost.*

Table 2: Performance of different APR methods for multi-location repair on Defects4J v1.2 and v2.0.

Method	# Correct			# Patch
	v1.2	v2.0	Total	
Multi-Location APR Methods				
Hercules	15	/	15	/
VarFix	16	/	16	/
ITER	15	2	17	1000
DEAR	18	/	18	/
LLM-based APR Methods				
FitRepair	15	0	15	4,000
ChatRepair	24	0	24	200
ContrastRepair	25	0	25	120
RepairAgent	23	11	34	117
GAMMA	31	6	37	250
GiantRepair	26	22	48	/
ThinkRepair	29	35	64	125
CASCADEFIX	**46**	**38**	**84**	**10**

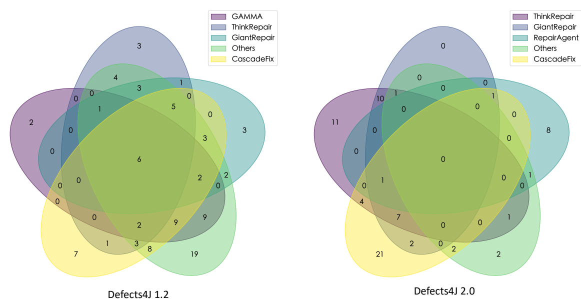

Figure 4: Correct fix Venn diagram on Defects4J.

Figure 4 shows the number of unique fixes (that only one method can fix while others cannot) generated by CASCADEFIX compared with the top-3

performing APR baselines and all other methods (Others). We observe that even compared with all previous APR methods, CASCADEFIX *is able to provide 9 and 21 additional unique fixes that no other APR methods have been able to fix so far on Defects4J v1.2 and 2.0, respectively.*

Table 3: Performance of different APR methods in repairing bugs with varying numbers of locations.

Method	v1.2		v2.0		Total	
	2	**≥3**	**2**	**≥3**	**2**	**≥3**
Multi-Location APR Methods						
Hercules	11	4	-	-	11	4
VarFix	9	7	-	-	9	7
ITER	14	1	1	1	15	2
DEAR	16	2	-	-	16	2
LLM-based APR Methods						
FitRepair	11	4	0	0	11	4
ChatRepair	20	4	0	0	20	4
ContrastRepair	21	4	0	0	21	4
RepairAgent	22	1	8	3	30	4
GAMMA	23	8	5	1	28	9
GiantRepair	19	7	20	2	39	9
ThinkRepair	23	6	**32**	3	55	9
CASCADEFIX	**32**	**14**	29	**9**	**61**	**23**

Table 3 details performance across varying bug complexities. Most fixed bugs involve only two locations. Specifically, 2-location bugs account for 73% of CASCADEFIX's total fixes, compared to 86% for baselines like ThinkRepair. This trend confirms that higher location counts increase repair difficulty by imposing stricter dependency constraints. Notably, CASCADEFIX excels in handling complex bugs (≥3 locations), fixing 14 and 9 in Defects4J v1.2 and v2.0, respectively, while baselines repaired at most 8 and 3.

From a project-level perspective, CASCADEFIX demonstrates superior performance across various projects. As shown in Figure 5, in the Math project of Defects4J v1.2, CASCADEFIX repaired 18 bugs, which is twice the number achieved by the suboptimal method, GAMMA (9 bugs). Regarding the Defects4J v2.0, although CASCADEFIX achieves limited success on the Codec and JacksonXml projects, it significantly outperforms other methods on projects such as Cli, Compress, and JacksonDatabind.

4.4 Ablation Studies

To evaluate the contributions of CASCADEFIX's core components—*graph-driven repair planning* and *context-aware cascading patch generation*—we conduct ablation studies on the Defects4J v1.2. Five variants were designed for comparison:

- **C1**: Removes both cascading repair planning and patch generation, treating each bug location independently using local code context.
- **C2**: Retains cascading repair planning but removes cascading patch generation.
- **C3**: Extends CascadeFix2 by incorporating historical repair knowledge.
- **C4**: Extends CascadeFix2 by incorporating bug dependency information.

Table 4: Ablation study on different components. Cascading patch generation involves Dependency and History.

Method	Description	# Correct	
		DeepSeek	**GPT**
C1	Base LLM	18	12
C2	+ Plannning	38	27
C3	+ Plannning + History	44	38
C4	+ Plannning + Dependency	40	31
Ours	+ Plannning + Dependency + History	**46**	**41**

As shown in Table 4, a comparison between C2 and C1 highlights that cascading repair planning is the most critical component. It leads to a substantial gain of 20 and 15 correct fixes on DeepSeek and GPT, respectively, confirming that explicitly ordering repairs based on dependencies effectively mitigates logical conflicts. Building on this foundation, the cascading patch generation, which fuses historical repair knowledge and bug dependency information, further boosts performance by 8 and 14 fixes in the full CASCADEFIX, thereby validating the effectiveness of our method. Notably, the superior performance of C3 over C4 suggests that *historical repair knowledge plays a more pivotal role in repair success than bug dependency information.*

4.5 Iteration Strategy and Cost

To explore the impact of iterative refinement on repair performance, we design six patch generation strategies, all constrained to a maximum budget of 10 patches per bug but differing in their allocation across rounds. The strategies are defined as follows:

- **Strategy A (10×1):** Generates 10 patches at once without iteration;
- **Strategy B (8–2):** Generates 8 patches in the first round, selects the top 2 for iteration, then generates 2 patches in the second round;

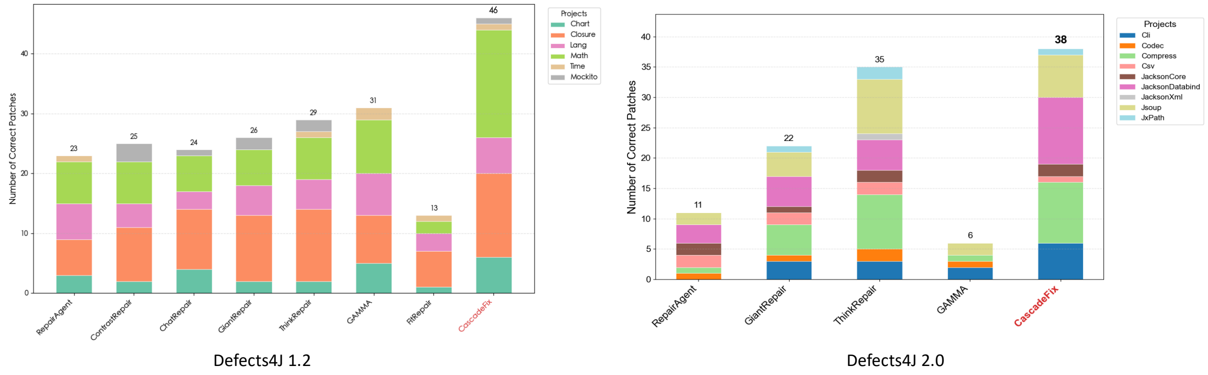

Figure 5: Performance comparison of CascadeFix and baselines across different projects in Defects4J.

- **Strategy C (5×2):** Two rounds of iteration, generating 5 patches per round;
- **Strategy D (6–2–2):** Generates 6 patches in the first round, followed by 2 patches in each of the subsequent two rounds;
- **Strategy E (4–3–3):** Generates 4 patches in the first round, followed by 3 patches in each of the next two rounds;
- **Strategy F (2×5):** Five iterations, generating 2 patches per iteration.

As shown in Table 5, *Strategy B (8–2) achieves optimal performance with 46 correct fixes*, outperforming the non-iterative Strategy A by 4 bugs. This indicates that moderate refinement effectively enhances repair quality for complex multi-location bugs, whereas excessive iterations (Strategies D–F) yield diminishing returns, likely attributable to noise accumulation and error propagation.

Table 5: Number of correct patches of different iteration strategies on Defects4J v1.2 using DeepSeek.

Project	A	B	C	D	E	F
Chart	6	6	6	6	6	5
Closure	14	14	15	10	11	10
Lang	5	6	6	6	4	5
Math	17	18	15	17	16	16
Time	0	1	1	1	1	0
Mockito	0	1	1	1	0	1
Total	42	**46**	44	41	38	37

Further analysis (Figure 6) confirms that while the base model's strong initial capability secures the majority of fixes in the first round, the targeted refinement in Strategy B provides a critical performance boost. Crucially, *Strategy B achieves this peak performance with superior cost-effectiveness (consuming fewer than 6,500 tokens per correct patch), demonstrating the optimal balance between performance improvement and computational cost.*

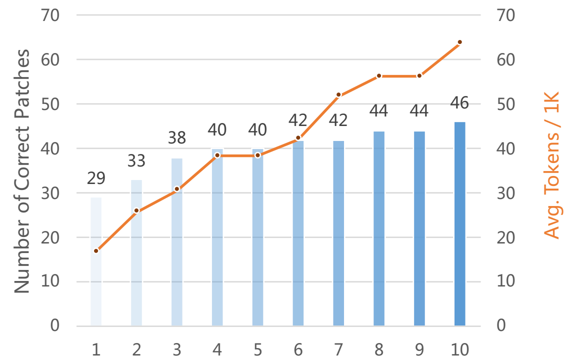

Figure 6: Performance comparison of Strategy B under different patch count thresholds.

5 Conclusion

In this paper, we propose CascadeFix for cascading multi-location program repair. First, we construct a heterogeneous bug graph to capture complex bug dependencies. Guided by this graph, we propose a graph-driven repair planning algorithm to formulate a global repair strategy. Finally, we design a context-aware cascading patch generation mechanism to ensure syntactic and semantic consistency of patches generated by LLMs. Experimental results show that CascadeFix significantly outperforms the state-of-the-art baselines. Code and data used are available at `https://anonymous.4open.science/r/CascadeFix-7CC2/`.

6 Limitations

Our current evaluation is limited to Java projects. Although the extraction of dependencies requires language-specific parsing, the core cascading planning and generation methodology can be adapted to other programming ecosystems. Additionally, con-

structing the heterogeneous bug graph incurs extra computational overhead, which may pose scalability challenges for very large codebases. To address this issue, future work will investigate optimization strategies, such as incremental graph construction and parallel analysis.

7 Ethical Statements

Our study does not pose any ethical concerns. Specifically, our training datasets are publicly accessible and utilized exclusively for research purposes. We have carefully audited our datasets to ensure they are free of unethical content, private information, or offensive topics. Furthermore, the base models employed in this work are also openly available for research use.

Acknowledgments

This work has been supported by the National Natural Science Foundation of China under grant No.62472447, the Science and Technology Innovation Program of Hunan Province under grant No.2023RC1023, and Hunan Provincial Natural Science Foundation of China under grant No.2024JK2006. Besides, this work was carried out in part using computing resources at the High Performance Computing Center of Central South University.

References

Ishac Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. Repairagent: An autonomous, llm-based agent for program repair. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2188–2200. IEEE.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, and 1 others. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021a. Cure: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021b. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE.

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

Dongsun Kim, Jaechang Nam, Jaewoo Song, and 1 others. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 35th international conference on software engineering (ICSE)*, pages 802–811. IEEE.

Jiaolong Kong, Xiaofei Xie, Mingfei Cheng, and 1 others. 2024. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs. *arXiv preprint arXiv:2403.01971*.

Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, and 1 others. 2017. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604.

Fengjie Li, Jiajun Jiang, Jiajun Sun, and 1 others. 2025a. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–28.

Yi Li, Shaohua Wang, and T. N. Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*, pages 511–523.

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*, pages 602–614.

Yingling Li, Muxin slogans Cai, Junjie Chen, and 1 others. 2025b. Context-aware prompting for llm-based program repair. *Automated Software Engineering*, 32(2):42.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Kui Liu, Anil Koyuncu, Dongsun Kim, and 1 others. 2019a. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE.

Kui Liu, Anil Koyuncu, Dongsun Kim, and 1 others. 2019b. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 31–42.

Ripon K. Saha, Yingjun Lyu, Wing Lam, and 1 others. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13.

S. Saha. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE.

Andre Silva, Sen Fang, and Martin Monperrus. 2025. Repairllama: Efficient representations and fine-tuned adapters for program repair. *IEEE Transactions on Software Engineering*, 51(8):2366–2380.

Xunzhu Tang, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. 2025. SynFix: Dependency-aware program repair via RelationGraph analysis. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 4878–4894, Vienna, Austria. Association for Computational Linguistics.

C. P. Wong, P. Santiesteban, C. Kästner, and 1 others. 2021. Varfix: Balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–366.

Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023a. The plastic surgery hypothesis in the era of large language models. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 522–534. IEEE.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023b. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971.

Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831.

Qi Xin, Haojun Wu, Jinran Tang, Xinyu Liu, Steven P Reiss, and Jifeng Xuan. 2024. Detecting, creating, repairing, and understanding indivisible multi-hunk bugs. *Proceedings of the ACM on Software Engineering*, 1(FSE):2747–2770.

Jifeng Xuan, Matias Martinez, Federico Demarco, and 1 others. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55.

He Ye, Matias Martinez, Xiapu Luo, and 1 others. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13.

He Ye and Martin Monperrus. 2024. Iter: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13.

Xin Yin, Chao Ni, Shaohua Wang, and 1 others. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286.

Wei Yuan, Quanjun Zhang, Tieke He, and 1 others. 2022. Circle: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 678–690.

Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067.

Yuan Yuan and Wolfgang Banzhaf. 2020. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067.

Huan Zhang, Qingyang Yan, Weihuan Min, Chenyuan Zhang, Li Kuang, and Yingjie Xia. 2025. Evoapr: Enhancing large language models for automatic program repair with genetic algorithm and dynamic lora. In *2025 IEEE International Conference on Web Services (ICWS)*, pages 946–956. IEEE.

Quanjun Zhang, Chunrong Fang, Tongke Zhang, and 1 others. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 535–547. IEEE.

Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923. IEEE.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, and 1 others. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 341–353.

A Detailed Definition of Multi-location Bug Relationships

Definition 1. Use Relationship. When the data or behavior carried by a code element (e.g., variable, field, method) defined or modified at bug location l_i is directly or indirectly utilized by another bug location l_j through program data flow or control flow, a Use relationship exists between them, denoted as $l_i \rightarrow l_j$. This relationship characterizes data dependencies or behavioral dependencies between bug locations based on program semantics. Modeling Use relationships not only provides context for the direct dependencies of bug locations but also reveals collaborative repair requirements between different locations, ensuring the final patch maintains logical consistency and semantic coherence across locations.

Figure 7 illustrates Use relationships using Lang 4 from Defects4J. The bug location defines the field `lookupMap` in the `LookupTranslator` class, instantiates this field, and accesses the field and calls its `put` method. This forms the data dependency chain, with corresponding usage relationships $l_i \rightarrow l_j$ and $l_j \rightarrow l_k$.

```
public class LookupTranslator extends CharSequenceTranslator {
Loc 1 →   private final HashMap<CharSequence, CharSequence> lookupMap;
          private final HashMap<String, CharSequence> lookupMap;
                      ⋮
          public LookupTranslator(final CharSequence[]... lookup) {
Loc 2 →     lookupMap = new HashMap<CharSequence, CharSequence>();
            lookupMap = new HashMap<String, CharSequence>();
                      ⋮
            if (lookup != null) {
              for (final CharSequence[] seq : lookup) {
Loc 3 →         this.lookupMap.put(seq[0], seq[1]);
                this.lookupMap.put(seq[0].toString(), seq[1]);
                      ⋮
```

Figure 7: Example of Use relationships in Lang 4.

Definition 2. Copy Relationship. When bug locations l_i and l_j exhibit highly similar code contexts, a Copy relationship exists between them, denoted as $l_i \approx l_j$. Its core assumption is that highly similar bug locations typically allow reuse of repair code. Thus, this relationship characterizes patch reusability between bug locations. Modeling Copy relationships helps reduce the repair cost for multi-location bugs. This is because existing methods typically generate patches for both locations when a Copy relationship exists, requiring two LLM invocations. However, in practice, a patch generated for one location can be directly reused for the other, reducing LLM invocations to a single call and avoiding unnecessary overhead from redundant generation.

Figure 8 illustrates the Copy relationship using the Lang 50 bug from the Defects4J dataset as an example. The buggy code at locations l_i and l_j is syntactically identical; in both cases, the behavior involves creating a `Pair` object containing a `key` and `locale` when the `locale` is not null, and subsequently assigning this object to the `key`. Furthermore, the neighboring code and structural context of these two locations are highly similar, leading to the identification of a Copy relationship between them.

```
getDateInstance             ⋮
          key = new Pair(key, timeZone);
        }
Loc 1 → if (locale != null) {
        if (locale == null) {
          key = new Pair(key, locale);
          locale = Locale.getDefault();
        }                   ⋮
getDateTimeInstance         ⋮
          key = new Pair(key, timeZone);
        }
Loc 2 → if (locale != null) {
        if (locale == null) {
          key = new Pair(key, locale);
          locale = Locale.getDefault();
        }                   ⋮
```

Figure 8: Example of Copy relationship in Lang 50.

Definition 3. Nearby Relationship. Let the code lines preceding and following a bug location constitute its preceding context and following context, respectively. When bug locations loc_i and loc_j are situated within the same file, and the buggy code of loc_i resides within the preceding context of loc_j, or the buggy code of loc_j resides within the following context of loc_i, a Nearby relationship is said to exist between them, denoted as $(loc_i, loc_j) \in E_{\text{nearby}}$. The core assumption is that proximal buggy code may trigger contextual interference, leading the model to erroneously reuse buggy code during patch generation. Since some existing methods use a bug location along with its surrounding lines as input, and models tend to reuse context from outside the immediate bug site, the presence of other bugs within that context can easily mislead the model into generating incorrect content. Thus, the Nearby relationship characterizes the contextual interference between bug locations.

Modeling this relationship helps prevent the model from misusing neighboring buggy code, thereby reducing the generation of incorrect patches.

Figure 9 illustrates the Nearby relationship using the Math 67 bug from the Defects4J dataset as an example. When x is set to 5, the following context of bug location loc_1 contains the buggy code from loc_2, specifically `return optimizer.getResult()`.Consequently, a Nearby relationship exists between them.If loc_1 and its surrounding 5 lines of code are provided as input to the LLM, the model might erroneously perceive the usage of the `optimizer` object as correct.This could lead to the incorrect introduction of that object into the patch, ultimately resulting in the generation of an invalid patch.

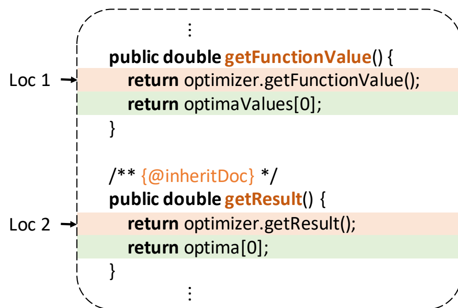

Figure 9: Example of Nearby relationship in Math 67.

B Illustrative Example of Operation Assignment

Figure 10 exemplifies the repair operation assignment process across three clusters. Initially, *Copy* edges (depicted as vertical gray bars) link Location 1 to Location 2 and Location 3 to Location 4, establishing them as equivalent bug locations. Consequently, since the constituent locations of Cluster 1 (Locations 1 and 3) exhibit a strict one-to-one equivalence with those of Cluster 2 (Locations 2 and 4), these two clusters are identified as equivalent clusters.

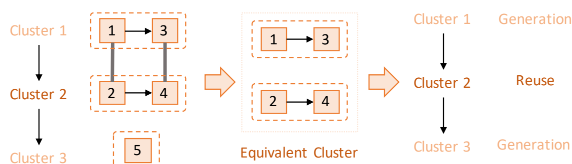

Figure 10: Illustration of equivalent bug locations and equivalent interdependent clusters.

Within this equivalence set, Cluster 1 holds a higher repair priority than Cluster 2 and is thus designated as the representative cluster. Adhering to the assignment strategy, the system assigns the GENERATION operation to Cluster 1, invoking the LLM to synthesize fresh patches. In contrast, the non-representative Cluster 2 is assigned the REUSE operation. This allows it to repair its bugs by directly applying the patches derived from Cluster 1, thereby significantly optimizing computational efficiency.