

Beyond Accuracy: Behavioral Dynamics of Agentic Multi-Hunk Repair

NOOR NASHID, The University of British Columbia, Canada

DANIEL DING, The University of British Columbia, Canada

KEHELIYA GALLABA, Queen's University, Canada

AHMED E. HASSAN, Queen's University, Canada

ALI MESBAH, The University of British Columbia, Canada

Automated program repair has traditionally focused on single-hunk defects, overlooking multi-hunk bugs that are prevalent in real-world systems. Repairing these bugs requires coordinated edits across multiple, disjoint code regions, posing substantially greater challenges. We present the first systematic study of LLM-driven coding agents (CLAUDE CODE, CODEX, GEMINI-CLI, and QWEN CODE) on this task. We evaluate these four state-of-the-art agents on 372 multi-hunk bugs from the HUNK4J dataset, yielding 1,488 repair trajectories for large-scale behavioral analysis. We employ fine-grained metrics to assess localization, repair accuracy, regression behavior, and operational dynamics across agents. We find that localization capability varies substantially, with CODEX achieving the highest success rate (75.0%) and QWEN CODE the lowest (38.2%). Repair accuracy also differs widely, ranging from 25.81% (QWEN CODE) to 93.28% (CLAUDE CODE), and consistently declines with increasing bug dispersion and complexity (hunk divergence and spatial proximity). High-performing agents (CLAUDE CODE and CODEX) demonstrate superior semantic consistency, achieving positive average regression reduction (+2.47 and +2.25, respectively), whereas lower-performing agents often introduce new test failures. Notably, agents do not *fail fast*; failed repairs consume substantially more resources (39%–343% more tokens) and require longer execution time (43%–427%). Additionally, we developed MAPLE to provide agents with repository-level context. Empirical results show that MAPLE improves repair accuracy of GEMINI-CLI by 30% through enhanced localization. By analyzing fine-grained metrics and trajectory-level analysis, this study moves beyond accuracy to explain how coding agents localize, reason, and act during multi-hunk repair. Our findings underscore the impact of bug divergence and spatial proximity on multi-hunk repair success for coding agents.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Multi-Hunk, Program Repair, Large Language Models, Coding Agents

1 Introduction

Automated program repair (APR) has progressed substantially in recent years, yet most existing techniques remain limited to *single-hunk* bugs—defects that can be corrected by modifying a single, localized code region [7, 12, 14, 19, 23, 32, 37, 38, 41, 45, 46]. In contrast, *multi-hunk* bugs, which require coordinated edits across multiple, disjoint code regions, are prevalent in real-world software systems [21, 30], yet have received limited attention [15, 17, 35, 42, 43]. Repairing such bugs poses significantly greater challenges, as it requires not only identifying all defective locations but also reasoning about their interdependencies and synthesizing coherent edits across the codebase.

Multi-hunk bugs differ greatly in structure and complexity, spanning edits within a single function to changes across multiple files or even packages. Our recent work [21] characterizes this variability along two dimensions: *hunk divergence*, measuring lexical and structural differences among edits, and *spatial proximity*, capturing their dispersion across the codebase. Our empirical

Authors' Contact Information: Noor Nashid, The University of British Columbia, Vancouver, Canada, nashid@ece.ubc.ca; Daniel Ding, The University of British Columbia, Vancouver, Canada, dyxd2003@ece.ubc.ca; Keheliya Gallaba, Queen's University, Kingston, Canada, gallabak@sigsoft.org; Ahmed E. Hassan, Queen's University, Kingston, Canada, ahmed@cs.queensu.ca; Ali Mesbah, The University of British Columbia, Vancouver, Canada, amesbah@ece.ubc.ca.

study across six large language models (LLMs) reveals that multi-hunk repair accuracy declines consistently with increasing hunk divergence and spatial dispersion [21].

Recent advances in AI have enabled the emergence of *coding agents*, LLM-driven agents that can iteratively interact with software projects through actions such as searching, editing, building, and testing [1, 2, 8, 25]. Unlike LLM prompting [24], these agents operate autonomously over multiple steps, making decisions based on intermediate feedback from compilers, test results, and source code changes. However, understanding how these agents behave during complex software engineering tasks remains difficult. Their decision-making process is often *non-deterministic*, meaning that identical inputs may lead to different sequences of actions due to the underlying model. They also adapt their strategies *dynamically* as they observe new feedback, making their reasoning paths hard to interpret. This dynamic and unpredictable nature becomes even more pronounced in the context of *multi-hunk* bugs, where successful repair requires identifying and consistently editing multiple related code regions. Simple outcome-based metrics, such as overall success or failure, fail to capture these nuanced behaviors. To advance automated repair, it is therefore important to analyze how coding agents reason, plan, and coordinate their actions when faced with complex, dispersed bug scenarios.

We present the first systematic study of coding agents in the context of multi-hunk program repair. Our investigation addresses two complementary aspects. First, we evaluate how well current coding agents perform when repairing multi-hunk bugs of varying complexity. Second, we examine their underlying capabilities and behaviors during the repair process, which have not been systematically examined before. We analyze trajectory-level traces of ordered action sequences, such as search, edit, build, and test, together with the corresponding feedback from the environment, including compiler diagnostics and test results. Since each coding agent generates trajectories that differ in structure and interaction semantics, we normalize them into a unified representation, enabling a consistent analysis of behavioral patterns across agents. To capture distinct aspects, we introduce a set of fine-grained metrics that separately assess localization accuracy, repair accuracy, regression, and tool usage patterns. In addition, we develop MAPLE (MODEL CONTEXT PROTOCOL FOR AUTOMATED LIGHTWEIGHT REPOSITORY CONTEXT EXTRACTION), a lightweight code analysis system that provides coding agents with repository-level context. Our analysis reveals clear behavioral differences that cannot be observed through overall accuracy alone. Coding agents perform well on bugs with low dispersion but struggle as divergence increases, often failing to coordinate edits across multiple locations. To the best of our knowledge, this is the first work to systematically characterize both the accuracy and the behavioral mechanisms of coding agents on multi-hunk program repair tasks.

We make the following contributions:

- (1) We conduct a systematic evaluation of four coding agents, CLAUDE CODE (Anthropic), CODEX (OpenAI), GEMINI-CLI (Google), and QWEN CODE (Alibaba) on 372 multi-hunk bugs from HUNK4J [21], analyzing 1,488 repair trajectories.
- (2) We introduce a set of behavioral metrics that isolate distinct aspects of the repair capability of coding agents. *Localization Success* measures the ability to identify buggy files; *Compilation Success* quantifies whether agent-proposed changes are syntactically coherent and can be compiled without any errors; *Repair Accuracy* captures the correctness of the generated patch; and *Regression Reduction* quantifies whether an agent introduced fewer bugs than it fixed during the repair process.

Results: We observe significant variation in effectiveness across agents. We find that localization success varies dramatically from 38% to 75%, and final repair accuracy ranges from a low of 26% to a high of 93%. This highlights the substantial differences in the reasoning and

code navigation capabilities of current agents. A deeper analysis of bugs not repaired by all agents reveals that high divergence can render even spatially localized bugs intractable.

- (3) We analyze tool-use sequences of agents to identify concrete behavioral patterns that distinguish successful repairs from failures.

Results: The two primary failure modes exemplified in the coding agents are *over-modification without validation* and *over-exploration without action*.

- (4) We develop MAPLE, a lightweight code analysis Model Context Protocol (MCP) tool that provides coding agents with repository-level context.

Results: Empirical results show that scope-driven, context retrieval assistance provided via MCP integration enhances the bug localization capabilities and thus improves the repair accuracy of weaker coding agents.

2 Characterization of Coding Agents

The behavior of coding agents in multi-hunk code repair cannot be meaningfully characterized by a single success or failure outcome. Real-world software defects frequently span multiple, interdependent code regions [30], creating complex and intertwined challenges [21]. First, the agent must accurately localize several spatially distributed fault sites, each of which may require different contextual reasoning. Second, edits performed at one location can influence the correctness of others, either improving or degrading the overall repair outcome. This complexity is further compounded by the iterative nature of coding agents, which reason over intermediate feedback such as compiler diagnostics and test results and continuously adapt their strategies as they proceed. A comprehensive understanding of agent behavior, therefore, requires fine-grained, process-level analysis that extends beyond aggregate measures of repair accuracy.

To formalize this analysis, we employ a set of fine-grained metrics to characterize the behavioral and functional capabilities of coding agents during multi-hunk repair. These metrics collectively capture how agents identify, modify, and validate buggy code, offering a structured basis for evaluating both their effectiveness and operational behavior.

2.1 Agent Execution Model

We consider a set of multi-hunk bug instances \mathcal{B} , where each bug $b \in \mathcal{B}$ is associated with a set of ground-truth buggy hunks $\mathcal{H}_{\text{gt}}(b) = \{h_1, h_2, \dots, h_m\}$ and $m \geq 2$. Let \mathcal{A} denote the set of coding agents under study, and let $\mathcal{T} = \{\text{search}, \text{edit}, \text{build}, \text{test}, \dots\}$ denote the action space comprising all available tool invocations. When an agent $A \in \mathcal{A}$ is executed on a bug instance b , it interacts with the development environment by performing a sequence of actions drawn from \mathcal{T} , which collectively form its *trajectory* [39, 40, 44].

More formally, the *trajectory* of a coding agent A for a given bug b , denoted $S_A(b)$, is the ordered sequence of actions executed by the agent during the repair process:

$$S_A(b) = \langle a_1, a_2, \dots, a_n \rangle, \quad (1)$$

where each $a_i \in \mathcal{T}$ represents an atomic tool invocation from the available action space. Each action a_i produces an observation o_i (e.g., compiler output, test results, or runtime logs), which informs the agent's subsequent decision a_{i+1} . The trajectory, therefore, encodes the complete interaction loop between the agent and its environment, capturing how the agent plans, acts, and adapts based on feedback throughout the multi-hunk repair process.

At the end of execution, the agent produces a candidate patch, denoted by $P_A(b)$, which contains the final code modifications proposed for bug b . If the agent fails to complete execution, encounters a runtime error, or exceeds its time limit, the resulting patch is recorded as empty. The generated

patch serves as the observable artifact of the agent's repair process, and is later used to evaluate localization accuracy and fixing effectiveness.

From the patch, we derive the set of code regions modified by the agent:

$$H(S_A(b)) = \{ f \mid f \text{ is modified in } P_A(b) \}. \quad (2)$$

If $P_A(b)$ is empty, due to execution failure, runtime error, or timeout, then $H(S_A(b))$ is also empty. This set represents the files examined by the coding agent and is used to analyze which files were actually modified relative to the ground-truth buggy regions.

We further record $\text{Tests}_A^{\text{before}}(b)$ and $\text{Tests}_A^{\text{after}}(b)$ as the numbers of passing tests before and after the agent's execution on b , respectively. These quantities measure the overall impact of the agent's repair attempt on program correctness. Overall, this execution model represents multi-hunk repair as a structured sequence of reasoning and tool-mediated interactions, forming the foundation for analyzing how coding agents plan, act, and adapt during complex repair processes.

2.2 Effectiveness Metrics

This section defines the *effectiveness metrics* used to characterize the outcome of each repair attempt. These metrics capture the observable end states of the repair process, including whether the bug can be localized, whether generated patches compile successfully, and whether they restore program correctness without introducing new regressions.

2.2.1 Localization. A coding agent cannot repair a bug without first identifying where the buggy code resides. In the multi-hunk setting, this challenge is exacerbated, as successful repair requires editing *all* buggy regions within the codebase and performing coordinated modifications to maintain patch coherence. Localization capability, therefore, reflects how effectively the agent explores and covers the relevant buggy files required for such interdependent code changes across the project. Insufficient localization may lead to wasted edits or incomplete fixes, even when some modifications are correct.

We define the *file localization success (LS)* metric as the extent to which the agent has interacted with all ground-truth buggy files while producing a plausible patch. If the generated patch $P_A(b)$ is empty or invalid as the agent failed to complete execution, encountered a runtime error, or produced uncompileable code, then the localization metric is set to 0.

When a patch is generated, the LS metric is defined as:

$$\text{LS}(S_A(b)) = \begin{cases} 1, & \mathcal{H}_{\text{gt}}(b) \subseteq H(S_A(b)), \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

An LS value of 1 indicates that the agent has covered all ground-truth buggy files in its generated patch, even if additional files were also modified. This definition captures *localization completeness*, emphasizing whether the agent successfully identified every relevant buggy file required for a coherent fix. The LS metric thus isolates the localization ability of the agent independently of the correctness of the final patch.

2.2.2 Compilation. Multi-hunk repairs require coordinated modifications across spatially separated code regions, where changes in one location may introduce syntactic dependencies on edits elsewhere. For instance, renaming a variable in one hunk necessitates consistent updates at all usage sites, while modifying a method signature requires corresponding changes at all call sites. Failure to maintain such consistency across hunks results in compilation errors, even when individual edits are locally correct.

We define the compilation success metric as:

$$\text{Compilation}_A(b) = \begin{cases} 1, & P_A(b) \neq \emptyset \wedge \text{the patch compiles without errors,} \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

A compilation value of 1 indicates the agent synthesized a syntactically coherent patch across all modified regions. This metric serves as a necessary precondition for functional correctness: patches that fail to compile cannot be tested and thus cannot achieve repair success.

2.2.3 Repair. Localization alone does not ensure a correct repair. In the multi-hunk repair setting, fixing requires coordinated edits across buggy files so that the resulting patch compiles, passes all tests, and does not introduce new failures. Repair ability measures how effectively a coding agent produces coherent and correct patches that restore program correctness.

Following prior work [21, 31], repair accuracy is defined as whether the agent generated a plausible patch that passes all available tests:

$$\text{Accuracy}_A(b) = \begin{cases} 1, & P_A(b) \neq \emptyset \text{ and } \text{Tests}_A^{\text{fail, after}}(b) = 0, \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

An accuracy value of 1 means that the agent produced a complete multi-hunk fix with no remaining test failures.

2.2.4 Regression Reduction. While knowing whether a generated patch compiles and passes all originally failing tests is essential, such binary success indicators do not capture the broader behavioral impact of a repair. Two coding agents may achieve identical repair success rates yet differ substantially in how their patches affect the overall test outcomes. One agent may fix the intended failures without side effects, whereas another may resolve some tests but introduce new regressions. Binary metrics are unable to distinguish these scenarios.

To address this limitation, we introduce the metric *regression reduction*, which quantifies the net change in the number of failing tests after applying a patch. For an agent A attempting to repair a bug b , regression reduction is defined as:

$$\text{RR}_A(b) = \text{failed_tests}_{\text{before}} - \text{failed_tests}_{\text{after}} \quad (6)$$

Here, $\text{failed_tests}_{\text{before}}$ denotes the total number of failing test cases before applying the patch, and $\text{failed_tests}_{\text{after}}$ denotes the total number afterwards. Positive values ($\text{RR}_A(b) > 0$) indicate that the agent's patch reduced the number of failing tests, either by fixing originally failing cases or by introducing fewer new failures than it resolved. A value of zero ($\text{RR}_A(b) = 0$) signifies no net change in the number of failing tests, implying that the patch compiled successfully but did not improve the test outcome. Negative values ($\text{RR}_A(b) < 0$) indicate that the patch increased the number of failing tests, suggesting a regression in correctness. When compilation fails, regression reduction is undefined and excluded from the analysis.

Regression reduction thus provides a finer-grained measure of repair impact than binary success metrics. It distinguishes agents that merely achieve correctness from those that progressively improve or degrade the test landscape. Agents with consistently positive regression reduction contribute net improvements, whereas those with negative averages introduce additional faults, requiring developers to correct both original and newly induced errors.

2.3 Hunk Characteristics: Divergence and Proximity

Multi-hunk patches differ not only in the number of edits they contain but also in how those edits vary internally and how they are distributed across the codebase. To characterize these dimensions

of repair complexity, we focus on two complementary properties introduced in our recent work [21]: *hunk divergence*, which captures internal heterogeneity among edits, and *spatial proximity*, which reflects their distributional layout within the program structure. Together, these metrics provide a comprehensive view of how coordination effort scales with both the diversity and the dispersion of edits in a patch.

2.3.1 Hunk Divergence. While the number of hunks in a patch provides a coarse measure of its fragmentation, it does not fully capture the internal diversity among the edits. Patches with the same number of hunks can differ substantially in how those hunks vary lexically, structurally, or spatially. To quantify this variation, we adopt the notion of *hunk divergence* [21], which characterizes the heterogeneity and coordination complexity within a multi-hunk patch.

The *hunk divergence* of a multi-hunk patch P measures the degree of variation among its constituent hunks in terms of lexical, structural, and spatial dissimilarity. Given a patch $P = \{h_1, h_2, \dots, h_n\}$, its divergence is defined as:

$$\text{Div}(P) = \ln(n) \cdot \left(\frac{2}{n(n-1)} \sum_{1 \leq i < j \leq n} \text{Div}(h_i, h_j) \right) \quad (7)$$

where $\text{Div}(h_i, h_j)$ denotes the pairwise dissimilarity between hunks h_i and h_j . It quantifies how different two hunks within a patch are from each other. It combines three complementary dimensions: lexical, structural, and file-level separation. The lexical component measures the dissimilarity of code changes in terms of their textual tokens; the structural component captures differences in their abstract syntax tree (AST) representations; and the file-level term accounts for whether the edits occur within the same file or across different files. A context-sensitive weighting factor amplifies the file-level contribution when hunks belong to separate files, reflecting the higher coordination effort required in distributed repairs. The resulting score ranges from 0 to 1, where higher values indicate greater divergence between edits. This metric jointly captures the internal heterogeneity of edits and the coordination complexity induced by the number of hunks in a patch. The overall hunk divergence score satisfies $\text{Div}(P) \in [0, \ln(n)]$.

2.3.2 Spatial Proximity. In addition to internal diversity among edits, multi-hunk patches vary in how their modifications are distributed throughout the codebase. *Spatial proximity* [21] characterizes this distribution by describing the degree to which individual hunks are clustered within, or dispersed across, program components such as methods, classes, files, and packages. Whereas hunk divergence measures how edits differ, spatial proximity reflects where they occur within the program's structural hierarchy.

Spatial proximity is represented categorically rather than as a continuous distance metric. Each multi-hunk patch is assigned to one of five dispersion categories, *Nucleus*, *Cluster*, *Orbit*, *Sprawl*, or *Fragment*, that collectively represent increasing levels of edit dispersion. *Nucleus* patches contain hunks co-located within the same method or class; *Cluster* patches extend across nearby functions or classes within a single file; *Orbit* patches span multiple files within a module; *Sprawl* patches affect distant files or packages; and *Fragment* patches are highly scattered across the repository. This hierarchical representation offers an interpretable view of edit dispersion that aligns with developers' reasoning about code locality and coordination effort.

Hunk divergence and spatial proximity together serve as key indicators of repair difficulty. Divergence captures how heterogeneous the edits are, while proximity captures how far apart they are distributed. Patches with high divergence and low proximity tend to involve varied yet localized changes, whereas those with both high divergence and high dispersion require extensive coordination across multiple interdependent code regions. These complementary perspectives

allow us to characterize the structural and spatial complexity that coding agents must overcome during multi-hunk repair.

2.4 Operational Dynamics of Coding Agents

Beyond outcome-oriented measures such as repair accuracy and regression reduction, we examine a complementary set of behavioral metrics that characterize operational dynamics of coding agents during the repair process. These metrics capture the efficiency, runtime dynamics, and interaction patterns underlying each repair attempt.

2.4.1 Token Consumption. We quantify *token consumption* to measure the number of language model tokens exchanged between the agent and the API throughout the repair trajectory. For an agent A repairing bug b , we distinguish two complementary components: *input tokens* transmitted to the model and *output tokens* generated by the model across all invocations. Let K_b denote the number of model calls made during the repair process. The total counts are defined as:

$$\text{InTok}_A(b) = \sum_{k=1}^{K_b} \text{tokens_sent}_A^{(k)}, \quad \text{OutTok}_A(b) = \sum_{k=1}^{K_b} \text{tokens_generated}_A^{(k)}. \quad (8)$$

The overall token consumption is then given by:

$$\text{TC}_A(b) = \text{InTok}_A(b) + \text{OutTok}_A(b) \quad (9)$$

Input tokens capture all prompt content, tool calls, and contextual information provided to the model, while output tokens correspond to the model's generated responses. These quantities together characterize the operational footprint of a repair attempt, serving as a proxy for the computational efficiency and resource utilization of coding agents.

2.4.2 Runtime. Runtime measures the wall-clock time required for an agent to complete a repair task, from task initialization to termination. This metric captures the end-to-end execution duration encompassing all agent activities, including model inference calls, tool invocations, code modifications, compilation checks, and test executions throughout the repair process.

For a bug b and agent A , runtime is defined as:

$$\text{Runtime}_A(b) = t_{\text{end}} - t_{\text{start}} \quad (10)$$

where t_{start} is the timestamp when the agent receives the repair task, and t_{end} is the timestamp when the agent terminates, either through successful repair, timeout, or explicit failure.

Runtime serves as a practical efficiency metric complementary to token consumption. While token consumption measures computational cost in terms of model inference, runtime reflects the actual elapsed time required for repair completion, which is critical for assessing agent responsiveness in practical scenarios.

2.4.3 Agentic Tool Utilization. We characterize coding agent behavior through trajectory analysis, which models the complete sequence of actions performed during repair, including tool invocations such as `search`, `edit`, `build`, and `test`. These trajectories capture how agents localize buggy regions, decide when and where to apply edits, and adapt to feedback. By examining the frequency and distribution of tool usage within these trajectories, we identify recurring behavioral patterns that reveal how agents allocate effort and coordinate progress across multiple bug locations.

For a trajectory $S_A(b)$ (Equation 1), we record the number of calls to each action type $a \in \mathcal{A}$:

$$\text{calls}(a) = |\{i \mid a_i = a\}| \quad (11)$$

The relative frequency of each tool is then defined as:

$$U_a = \frac{\text{calls}(a)}{\sum_{a' \in \mathcal{A}} \text{calls}(a')} \quad (12)$$

We leverage these metrics to understand how much each tool contributes to the agent’s repair process. Moreover, we investigate whether the agent distributes effort across multiple tools or relies on a limited subset of actions.

2.4.4 Tool Sequencing Pattern. Developers typically follow structured tool-use sequences while debugging. For instance, a common workflow involves diagnosing the problem, applying edits, running tests to observe behavior, and refining the changes based on feedback. We examine whether coding agents exhibit similar sequencing patterns in their workflow.

Given that trajectories capture the temporal order of the agent’s reasoning and tool usage during repair, we analyze each trajectory $S_A(b)$ using a fixed window size w to extract local subsequences of actions:

$$p_i = \langle a_i, a_{i+1}, \dots, a_{i+w} \rangle, \quad 1 \leq i \leq n - w \quad (13)$$

For each agent, we aggregate all subsequences across its trajectories and identify the top- n most frequent patterns. These top- n sequences capture the agent’s characteristic repair workflows during multi-hunk fixing.

By examining these top- n sequences, we aim to assess whether coding agents exhibit coherent diagnostic, such as test driven development patterns (edit–test–refine) loops similar to human developers, or whether they follow distinct workflows that reflect different reasoning strategies. This analysis seeks to understand how agents coordinate actions, whether they leverage test feedback to guide refinement, and to what extent their operational structure aligns with systematic debugging practices. Consistent sequencing patterns would indicate that the agent organizes its repair process into structured operational cycles, whereas fragmented or highly variable patterns may reveal reactive or uncoordinated reasoning during multi-hunk repair.

3 MAPLE

Effective debugging and modification depend on structured navigation and context awareness. To repair multi-hunk bugs, coding agents must locate and reason over multiple, distributed code regions. However, large repositories exceed the model’s context window, preventing agents from loading entire projects at once. This limitation makes efficient repository exploration essential for scaling automated repair.

We present a systematic mechanism that equips agents with developer-like navigation capabilities through repository-level context extraction. Given a codebase, our method parses and indexes its structure, extracting type definitions (classes, interfaces, enums) and method declarations. It provides multi-granularity access—global, file-level, and class-level—enabling agents to explore codebases incrementally rather than monolithically. Agents can query repository structure, retrieve class skeletons, or fetch specific methods and code snippets on demand.

Instead of implementing ad-hoc tools for each agent, we adopt the Model Context Protocol (MCP) [3], which defines a unified API layer between coding agents and external context servers. This abstraction allows integration across agents while ensuring reproducibility and interoperability. We follow established approaches for repository-level context extraction [20, 22, 44].

Our MCP server, MAPLE, implements nine commands organized by scope and granularity, each prefixed with `maple_` to prevent naming conflicts with other MCP endpoints. Table 1 summarizes these capabilities.

Table 1. MAPLE MCP tools

Scope	Tool	Input	Returns	Purpose
Class	maple_find_method_in_class	method, class	Method source code	Locate method within given class
File	maple_find_class_in_file	class, file	Class declaration	Locate class within specific file
	maple_find_method_in_file	method, file	Method source code	Locate method within specific file
	maple_find_code_in_file	snippet, file	Code with context	Search code in specific file
	maple_extract_class_skeleton	file name	Package, imports, signatures	Extract class structure
Global	maple_find_class	class name	Class declaration	Locate class in codebase
	maple_find_method	method name	Method source code	Locate method in codebase
	maple_find_code	code snippet	Code with context	Search code in codebase
	maple_repo_structure	project path	Directory tree	View repository structure

To support these operations efficiently, we construct structural indices based on Abstract Syntax Trees (ASTs), which precisely capture program elements such as class declarations, method definitions, and type hierarchies. Unlike text-based search utilities (e.g., `grep`, `awk`), AST parsing distinguishes code structure from literals and comments, enabling accurate retrieval of semantic entities. For each file, we extract types and methods with their fully qualified names and locations, building hierarchical indices that map methods to their enclosing classes and classes to their source files. Unparseable files are logged but excluded from the index. These indices are generated at session initialization and persist throughout the repair process, allowing fast, context-aware lookups without repeated parsing.

4 Evaluation

Our study addresses the following research questions:

- **RQ1:** How effectively can coding agents localize multi-hunk bugs?
- **RQ2:** What is the repair accuracy of coding agents when addressing multi-hunk bugs?
- **RQ3:** How do bug characteristics, such as hunk divergence and spatial proximity, affect the repair success rate of coding agents?
- **RQ4:** To what extent do coding agents introduce regressions during multi-hunk bug repair?
- **RQ5:** How do the operational dynamics of coding agents, including token consumption, tool utilization, sequencing behavior, and runtime reflect their internal mechanisms during multi-hunk bug repair?
- **RQ6:** Does MAPLE, our scope-driven context-assisting MCP, enhance the ability of coding agents to repair multi-hunk bugs?

4.1 Dataset

To enable a systematic evaluation of multi-hunk program repair, we use HUNK4J [21], a curated dataset of reproducible multi-hunk bugs with test-suite validation collected from open-source Java projects.

HUNK4J contains 372 developer-written multi-hunk bugs with patches containing multiple, distinct modification hunks. Single-file multi-hunk bugs constitute the majority of the cases, with 244 instances where all changes are contained within a single file. Among these, 140 involve two hunks, 55 contain three hunks, and 49 include four or more. Such cases represent intra-file repair scenarios that primarily demand localized reasoning. In contrast, 128 bugs extend across multiple files, reflecting more fragmented and distributed repair patterns. Of these, 68 include four or more hunks, indicating substantial dispersion of changes throughout the codebase.

By integrating test-suite-validated patches with fine-grained hunk annotations, HUNK4J enables systematic analysis of bug heterogeneity across multiple dimensions, such as lexical variation, structural complexity, file-level dispersion, and spatial distribution of edits.

4.2 Agents

We evaluate four state-of-the-art coding agents in our multi-hunk bug repair study: CODEX¹ (OpenAI), GEMINI-CLI² (Google), CLAUDE CODE³ (Anthropic), and QWEN CODE⁴ (Alibaba). These agents were selected because they represent four leading industrial efforts in autonomous code generation and repair. Together, they capture diverse design philosophies and interaction paradigms across the current landscape of tool-augmented LLM systems.

CODEX, developed by OpenAI, serves as one of the most widely adopted code generation models, forming the foundation for tools such as GitHub Copilot. GEMINI-CLI, part of Google's Gemini Code Assist ecosystem, integrates closely with real-world developer environments through command-line and IDE interfaces. CLAUDE CODE, from Anthropic, focuses on interpretable code manipulation and task execution, enabling multi-step problem solving through structured reasoning loops. Finally, QWEN CODE, released by Alibaba Cloud, is an open-source multilingual code model designed for reproducible research and extensibility across programming tasks. These four agents collectively span a broad spectrum of model architectures, tool integrations, and accessibility levels, from proprietary commercial platforms to open research frameworks.

4.3 Procedure

We developed a fully automated framework to evaluate each coding agent on all bugs in HUNK4J. While each agent has distinct command-line interfaces, underlying models, and output formats, we execute them in a standardized, automated fashion without manual intervention.

For each bug, we execute a five-stage pipeline. First, we checkout the buggy version into an isolated workspace directory. Second, we inject bug-specific metadata (title and description from the original bug report) into a standardized prompt template (Figure 1). Third, we provide additional scripts (`run_bug_exposing_tests.sh`, `run_all_tests_trace.sh`) that generate complete test execution trace logs with full stack traces, enabling agents to diagnose failures more effectively. Fourth, we launch the agent in headless mode. As an example, for QWEN CODE, we invoke `qwen-code -sandbox -yolo -p "Read and execute AGENT.md" -telemetry`, where `-sandbox` restricts file system operations to the workspace, `-yolo` enables autonomous action execution without approval prompts, and `-telemetry` captures complete trajectory traces including tool invocations and model responses. Fifth, after agent termination, we run `compile` and `test` commands to determine patch correctness.

All agents receive identical task descriptions through a standardized prompt template. Figure 1 shows the condensed structure. We add bug-specific metadata, including the title and description from the original bug report. The template specifies the repair workflow and documents the available build and test commands. For RQ6, the prompt includes additional descriptions of MAPLE MCP tools for structured codebase search. The complete prompt is provided in our replication package.

Each agent executes within its own isolated workspace directory containing the checked-out buggy project, eliminating cross-contamination between bug instances. Agents have unrestricted access to their workspace file system, the build and test framework, and their respective API endpoints. We access CLAUDE CODE through the Claude Max plan (\$200/month), CODEX through

¹<https://openai.com/codex/>

²<https://cloud.google.com/gemini/docs/codeassist/gemini-cli>

³<https://www.claude.com/product/claude-code>

⁴<https://github.com/QwenLM/qwen-code>

Task Description

Task: Fix a bug in a Java project. The project contains failing test cases. Investigate and resolve the underlying defect to restore full test passage.

Bug Context:

- **Title:** {{bug_title}}
- **Description:** {{bug_description}}

Workflow: Identify failing tests, diagnose root cause, apply fixes, verify correctness through comprehensive testing, and ensure no regressions.

Build & Test Commands:

- **Compile:** `defects4j compile`
- **Test:** `defects4j test, ./run_bug_exposing_tests.sh, ./run_all_tests_trace.sh`

Maple MCP – Structured Codebase Search:

- **Class:** `maple_find_method_in_class`
- **File:** `maple_find_class_in_file, maple_extract_class_skeleton`
- **Global:** `maple_find_class, maple_repo_structure`

Success Criteria: Zero compilation errors, zero test failures.

Fig. 1. Standardized prompt template provided to all coding agents (condensed).

Table 2. Multi-hunk bug repair with coding agents

Agent	Total Bugs	Localization Success (%)	Compilation Success (%)	Regression Reduction (Avg)	Repair Accuracy (%)
QWEN CODE	372	142 (38.17%)	364 (97.85%)	-1.34	96 (25.81%)
GEMINI-CLI	372	173 (46.51%)	347 (93.28%)	-1.92	155 (41.67%)
CODEX	372	279 (75.00%)	367 (98.66%)	2.25	324 (87.10%)
CLAUDE CODE	372	238 (63.98%)	372 (100.00%)	2.47	347 (93.28%)

ChatGPT Pro (\$200/month), GEMINI-CLI using GEMINI-2.5-FLASH from Google, and QWEN CODE using QWEN3-CODER-FLASH through OpenRouter⁵. All agents operate in headless mode. We conducted the evaluation between September and November 2025. At the beginning of the study, the following agent CLI versions were used: CLAUDE CODE 2.0.13, CODEX 0.21.0, GEMINI-CLI 0.10.0, and QWEN CODE 0.0.11.

After agent termination, we first compile the patched code. Compilation failures result in immediate rejection. For compilable patches, we run the full test suite. Test outcomes are recorded in structured CSV files. We further collect trajectory logs and patch diffs from each agent.

4.4 Localization Ability (RQ1)

As defined in Section 2.2.1, an agent successfully localizes bug b if the set of files it modifies is a superset of the files modified in the developer patch. This criterion ensures that for multi-hunk bugs spanning multiple files, the agent must identify all fault locations. Localization success is necessary but not sufficient for repair success: an agent that fails to localize all buggy files cannot produce a correct fix. However, multiple semantically equivalent patches may exist for the same bug, potentially modifying different sets of files. Developer patches provide a consistent, reproducible ground truth for comparative evaluation across agents. This file-level localization criterion does not

⁵<https://openrouter.ai/>

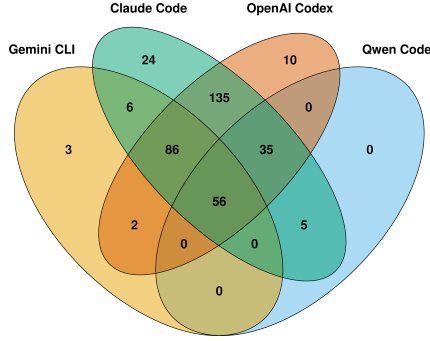


Fig. 2. Overlap of fixes across coding agents

account for patch equivalence, but ensures all agents are evaluated against the same localization target.

Table 2 presents localization success rates across all four agents. CODEX yields the highest success rate at 75.0% (279 bugs), followed by CLAUDE CODE at 64.0% (238 bugs), GEMINI-CLI at 46.5% (173 bugs), and QWEN CODE at 38.2% (142 bugs). These results suggest that current coding agents differ substantially in their ability to navigate complex repository structures.

4.5 Repair Ability (RQ2)

Table 2 presents compilation success and repair accuracy across all four agents on 372 HUNK4J bugs. We first examine compilation success, then analyze repair accuracy.

Compilation Success. Compilation success measures whether the generated patch produces syntactically valid code that compiles without errors. CLAUDE CODE achieves perfect compilation success at 100% (372 bugs), indicating all generated patches compile successfully. CODEX and QWEN CODE also achieve high compilation rates at 98.66% (367 bugs) and 97.85% (364 bugs) respectively. GEMINI-CLI exhibits the lowest compilation success at 93.28% (347 bugs), with 25 bugs producing patches that fail to compile. These near-perfect compilation rates across all coding agents indicate that syntactic correctness is not a primary bottleneck for multi-hunk bug repair. Even the lowest-performing agent compiles successfully 93.28% of the time.

Repair Accuracy. CLAUDE CODE yields the highest repair accuracy at 93.28% (347 bugs), followed by CODEX at 87.10% (324 bugs). GEMINI-CLI achieves 41.67% (155 bugs) and QWEN CODE achieves 25.81% (96 bugs). Thus, repair accuracy varies substantially across agents, ranging from 25.81% (QWEN CODE) to 93.28% (CLAUDE CODE). While compilation success remains high across agents (93.28%-100%), repair accuracy spans a much wider range. This gap indicates that syntactic correctness is necessary but insufficient for successful repair.

CLAUDE CODE and CODEX demonstrate superior ability to coordinate edits across multiple interdependent locations while preserving semantic consistency. These agents, built on SONNET-4.5 and GPT-5 respectively, exhibit a stronger capacity to reason over distributed context and reconcile dependencies across files and hunks. In contrast, GEMINI-CLI and QWEN CODE frequently produce patches that compile but fail to fix bugs. GEMINI-CLI's 41.67% accuracy indicates less than half of compilable patches achieve functional correctness, while QWEN CODE's 25.81% indicates only one in four patches successfully repair bugs.

Overlap of Agent Capability. Figure 2 shows the intersection of correctly repaired bugs across the four coding agents. The central overlap contains 135 bugs that all agents, CLAUDE CODE,

Table 3. Hunk divergence for fixed and unfixed bugs, and spatial proximity distribution (% fixed) across coding agents

Agent	Hunk Divergence (Fixed)			Hunk Divergence (Unfixed)			Spatial Proximity (% Fixed)				
	Median	Mean	Max	Median	Mean	Max	Nucleus	Cluster	Orbit	Sprawl	Fragment
QWEN CODE	0.33	0.37	1.15	0.46	0.50	1.60	22.03	31.35	19.40	24.00	0.00
GEMINI-CLI	0.38	0.40	1.12	0.47	0.51	1.60	44.07	48.11	26.87	36.00	18.18
CODEX	0.41	0.46	1.56	0.51	0.54	1.60	91.53	84.32	91.04	92.00	63.64
CLAUDE CODE	0.41	0.45	1.56	0.58	0.68	1.60	100.00	93.51	89.55	92.00	81.82

CODEX, GEMINI-CLI, and QWEN CODE —successfully repair, representing a shared subset of defects consistently solvable across different model architectures. Beyond this common subset, CLAUDE CODE contributes 24 unique repairs, followed by CODEX with 10 and GEMINI-CLI with 3, while QWEN CODE provides no unique fixes.

The three-way intersection of CLAUDE CODE, CODEX, and GEMINI-CLI covers an additional 86 bugs, whereas smaller overlaps appear for CLAUDE CODE, CODEX, and QWEN CODE (35 bugs) and for other mixed triads (56 bugs in total). Two-way intersections are relatively minor, including 6 shared repairs between CLAUDE CODE and GEMINI-CLI, 5 between CODEX and QWEN CODE, and 2 between CODEX and GEMINI-CLI. These results indicate measurable complementarity among the agents: high-accuracy systems such as CLAUDE CODE and CODEX not only provide broad repair coverage but also contribute distinct, non-overlapping fixes.

This diversity suggests that ensemble or cooperative repair strategies leveraging multiple agentic models could improve overall coverage and robustness in multi-hunk repair.

4.6 Multi-hunk Repair Complexity (RQ3)

We next examine how repair accuracy varies with the structural complexity of multi-hunk bugs, measured through hunk divergence and spatial proximity (see Section 2.3). Table 3 summarizes divergence statistics for fixed and unfixed bugs, along with the proportion of successfully repaired cases across the five spatial proximity classes.

Across all coding agents, fixed bugs exhibit consistently lower hunk divergence than unfixed ones. For instance, the median divergence for fixed bugs ranges from 0.33 in QWEN CODE to 0.41 in CLAUDE CODE and CODEX, whereas unfixed bugs show substantially higher values, reaching up to 0.68 in CLAUDE CODE. This pattern confirms that increasing lexical, structural, and file-level dispersion among hunks correlates with decreased repair accuracy. Agents struggle to coordinate edits when patches involve dissimilar or widely distributed code fragments, reinforcing the role of hunk divergence as a predictor of repair difficulty.

The spatial proximity results further support this observation. Repair success declines steadily from localized to dispersed bug classes. CLAUDE CODE achieves 100% repair accuracy in Nucleus bugs (single-method edits) and maintains high success rates across all categories: 93.51% for Cluster (single-file), 89.55% for Orbit (neighboring files), 92% for Sprawl (distributed), and 81.82% for Fragment (highly dispersed). CODEX exhibits similar strength with 91.53% (Nucleus), 84.32% (Cluster), 91.04% (Orbit), 92% (Sprawl), and 63.64% (Fragment). In contrast, GEMINI-CLI shows moderate performance ranging from 44.07% (Nucleus) to 18.18% (Fragment), while QWEN CODE achieves only 22.03% (Nucleus), 31.35% (Cluster), 19.40% (Orbit), 24% (Sprawl), and 0% (Fragment). These results demonstrate that most repairs occur within localized categories, while highly dispersed Fragment bugs remain exceptionally challenging, with only the strongest agents (CLAUDE CODE, CODEX) achieving any success. Notably, CLAUDE CODE demonstrates superior cross-file reasoning compared to other agents, maintaining consistently high accuracy even in more dispersed settings.

In contrast, QWEN CODE exhibits no successful repairs in the Fragment category, highlighting the limitations of lightweight models in handling highly scattered code dependencies.

Overall, the results indicate that both divergence and spatial dispersion strongly influence agentic repair accuracy. As multi-hunk complexity increases, agents require deeper contextual understanding and more precise coordination across edit locations. Enhancing models with divergence-aware retrieval and proximity-sensitive context integration remains an important direction for improving multi-hunk repair effectiveness.

Intersections by Spatial Proximity. Figure 3 visualizes the intersection patterns of multi-hunk bugs repaired by the four coding agents, with each intersection decomposed by spatial proximity class. Unlike the Venn diagram, this Upset plot provides a more granular view of overlap structure, revealing how spatial dispersion interacts with agent collaboration. Each vertical bar represents a unique combination of agents, while the stacked color segments indicate the spatial distribution of fixed bugs—Nucleus (dark blue), Cluster (light blue), Orbit (orange), Sprawl (tan), and Fragment (green). The numbers above the bars denote the total number of bugs in each intersection.

Smaller intersections, ranging from 2 to 10 bugs, correspond to isolated or agent-specific fixes—for instance, two bugs repaired jointly by GEMINI-CLI and CODEX, and ten repaired exclusively by CODEX. Medium-sized overlaps capture more substantial collaboration: CLAUDE CODE alone repairs 24 bugs, the combination of QWEN CODE, CLAUDE CODE, and CODEX accounts for 35, and the trio of GEMINI-CLI, CLAUDE CODE, and CODEX handles 56. The largest intersections reveal the most significant collaborative regions: 86 bugs fixed by GEMINI-CLI, CLAUDE CODE, and CODEX (excluding QWEN CODE), and 135 bugs repaired by all four agents.

The spatial distribution within these intersections exposes a clear trend in repair difficulty. Cluster-class bugs dominate across most groups, comprising 39% of the 135 bugs fixed by all agents, followed by Orbit (23%), Nucleus (19%), Sprawl (17%), and Fragment (2%). Fragment-class bugs remain exceptionally rare, appearing mainly in the CLAUDE CODE-only and all-agent combinations. Bugs with higher proximity—those in the Cluster and Nucleus categories—are consistently solvable across models, while spatially dispersed bugs (Orbit, Sprawl, and Fragment) are handled by fewer agents and require stronger reasoning capabilities.

Cluster-class bugs dominate across intersections, showing that spatially localized repairs are consistently solvable by most agents. In contrast, Fragment-class bugs remain exceptionally rare, appearing only in a few cases, such as the CLAUDE CODE-only and all-agent overlaps. The largest consensus group, consisting of 135 bugs repaired by all four agents, represents the most stable subset of defects and reflects tasks that are broadly tractable for current models. Three-agent overlaps—particularly those excluding QWEN CODE, which account for 86 and 56 bugs—demonstrate that CLAUDE CODE, CODEX, and GEMINI-CLI collectively handle a broader range of complex, multi-hunk scenarios. Nucleus-class bugs appear consistently across intersections, suggesting that moderate spatial dispersion remains within the reasoning capacity of most agents.

Hunk Divergence as a Predictor of Agentic Repair Success. Figure 4 presents a faceted violin plot showing the distribution of hunk divergence values for bugs that were successfully repaired (Pass) versus those that were not (Fail) across the four coding agents. Each facet corresponds to one agent, QWEN CODE, GEMINI-CLI, CODEX, and CLAUDE CODE, with the y-axis representing hunk divergence, ranging from approximately -0.5 to 2.0. Within each facet, embedded boxplots show medians and quartile ranges.

Across all agents, a consistent trend emerges: successful repairs cluster at lower divergence values, typically between 0.3 and 0.5, while failed repairs extend beyond 0.6 and reach as high as 2.0. For QWEN CODE, passing bugs have a median divergence near 0.33 (concentrated between 0.2 and 0.5), whereas failed repairs rise to about 0.46, spreading up to 1.6. GEMINI-CLI follows a similar

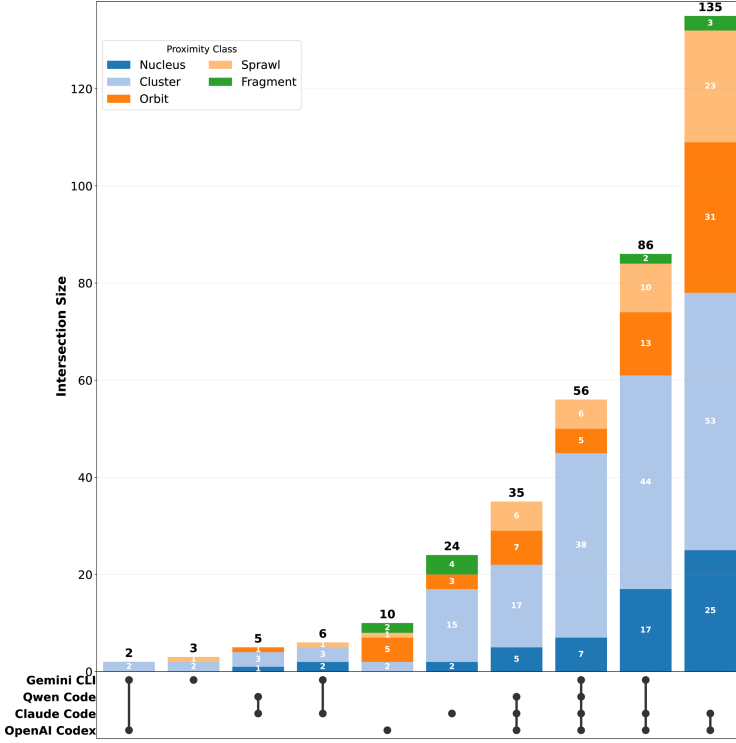


Fig. 3. Upset-style plot showing bugs correctly repaired by the selected coding agents, grouped by spatial proximity class. Bars indicate intersections among coding agents; color segments denote proximity categories.

pattern, with pass cases centered around 0.38 and fail cases near 0.47, both extending to 1.6. CODEX shows a slightly higher pass median of 0.41 and a fail median of 0.51, though its fail distribution is narrower, aligning with its higher overall repair accuracy. CLAUDE CODE demonstrates the sharpest separation: pass cases have a median of 0.41 (ranging from 0.2 to 0.8), while the few failures cluster tightly around 0.58 and extend to 2.0. This narrow, tall fail distribution indicates that CLAUDE CODE succeeds on nearly all low-divergence bugs but struggles with highly fragmented patches.

Overall, the visualization shows a clear and consistent relationship between repair success and hunk divergence. Bugs with compact, semantically aligned edits, reflected in lower divergence values, are consistently solvable across agents, whereas those involving scattered or heterogeneous changes remain challenging for all models. These results indicate that hunk divergence is a strong predictor of repair success in agentic systems, underscoring that spatial and structural coherence across edits remains critical for effective multi-hunk repair.

Failure Case Analysis: Bugs Unfixed by Any Agent. While the majority of bugs were successfully repaired by at least one agent, 10 bugs (2.7% of the dataset) remained unsolved by all four agents. We conducted a comprehensive analysis of these failure cases to understand the fundamental limitations of current agentic repair approaches. Our analysis reveals several findings that challenge conventional assumptions in automated program repair with coding agents. The analysis for these bugs is shown in table 4.

We first analyze localization accuracy by examining whether agents correctly identified the files requiring modification. Surprisingly, 50% of unsolved bugs (5 out of 10) were successfully localized

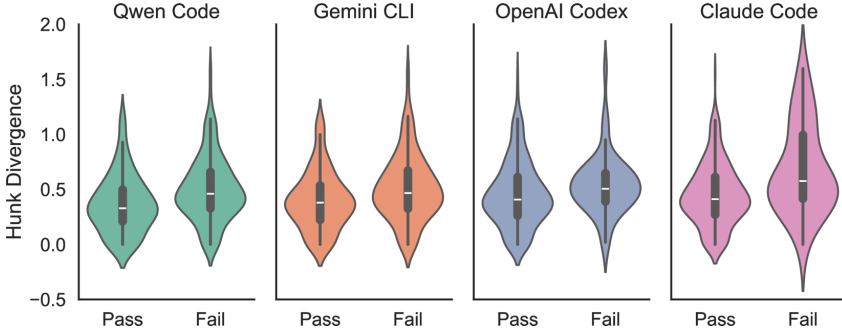


Fig. 4. Faceted violin plots of average hunk divergence, grouped by coding agents and split by outcome.

Table 4. Characteristics of 10 bugs not fixed by all four agents.

Bug ID	Localization Success				Multi-Hunk Complexity		
	QWEN CODE	GEMINI-CLI	CODEx	CLAUDE CODE	Hunks	Divergence	Proximity
JacksonDatabind_108	✗	✗	✓	✗	2	0.268	Cluster
JacksonDatabind_81	✗	✗	✓	✗	7	1.158	Cluster
JacksonDatabind_105	✗	✗	✗	✗	2	0.588	Cluster
JacksonDatabind_110	✓	✓	✓	✗	3	0.584	Cluster
JacksonDatabind_104	✓	✗	✓	✗	3	0.463	Cluster
JacksonDatabind_80	✗	✗	✓	✗	3	0.414	Cluster
Closure_171	✗	✗	✗	✗	2	0.364	Orbit
JacksonDatabind_109	✗	✗	✗	✗	5	0.702	Orbit
Closure_157	✗	✗	✗	✗	7	1.078	Sprawl
JacksonDatabind_103	✗	✗	✗	✗	26	1.599	Sprawl
Rate				Mean			
10 Non-fixed Bugs	0.20	0.10	0.50	0.00	6.0	0.722	—
All 372 Bugs	0.38	0.47	0.75	0.64	3.86	0.467	—
Diff (Δ)	-47%	-79%	-33%	-100%	+55%	+55%	—

by at least one agent. Specifically, CODEX successfully localized 5 bugs, QWEN CODE localized 2 bugs, and GEMINI-CLI localized 1 bug. Notably, bug JacksonDatabind_110 was correctly localized by three different agents (GEMINI-CLI, QWEN CODE, and CODEX), yet none could generate a correct repair. This reveals a distinct *localization-repair gap* that agents can identify where the bug is located, but fail to synthesize an appropriate fix.

We also observe unsolved bugs exhibit significantly higher complexity as measured by hunk divergence. The mean hunk count for unsolved bugs is 6.0 compared to 3.86 for the overall dataset, a 55% increase. Also, a surprising finding concerns the spatial distribution of changes. We classified bugs by proximity class based on how close the required changes are to each other. Intuitively, bugs requiring spatially close changes (classified as “Cluster”) should be easier to fix because related modifications are localized. However, 60% of unsolved bugs are Cluster-type, with a mean divergence of 0.579, higher than the overall mean of 0.467. This analysis reveals that physical co-location of changes does not guarantee a successful repair by coding agents. Despite changes occurring in the same file or nearby lines, high divergence indicates that agents struggle to generate a coherent fix.

The distribution of unsolved bugs across projects is highly skewed. Eight of the ten unsolved bugs (80%) originate from the JacksonDatabind project, despite JacksonDatabind bugs comprising only 11% of the overall dataset. JacksonDatabind bugs are particularly challenging because they

involve complex type systems with deep inheritance hierarchies, annotation-driven behavior where semantics are specified through metadata rather than code, and polymorphic serialization logic requiring runtime type resolution.

Overall, despite localization success for half of these bugs, none were successfully repaired, demonstrating a distinct localization-repair gap. The high mean hunk count and divergence values reveal a complexity threshold beyond which all agents fail.

4.7 Regression (RQ4)

Table 2 presents regression reduction statistics across all four coding agents. Our results reveal a stark effectiveness divide. CLAUDE CODE and CODEX achieve positive regression reduction (+2.47 and +2.25, respectively). In contrast, QWEN CODE and GEMINI-CLI exhibit negative regression reduction (-1.34 and -1.92, respectively), introducing an average of 1–2 new test failures per repair attempt.

We observe a strong positive correlation between repair success rate and regression reduction (Pearson $r = 0.96$, $p = 0.045$, $n = 4$). CLAUDE CODE achieves both the highest repair rate (93.28%) and the highest regression reduction (+2.47), while QWEN CODE exhibits both the lowest repair rate (25.81%) and the lowest regression reduction (-1.34). Although the sample size is small, our observation suggests that higher repair rates do not sacrifice code quality for quantity. This suggests that effective bug repair requires not aggressive code generation, but rather *context-aware* and *dependency-conscious* program synthesis. Tools exhibiting positive regression reduction demonstrate a deeper understanding of code structure and test relationships.

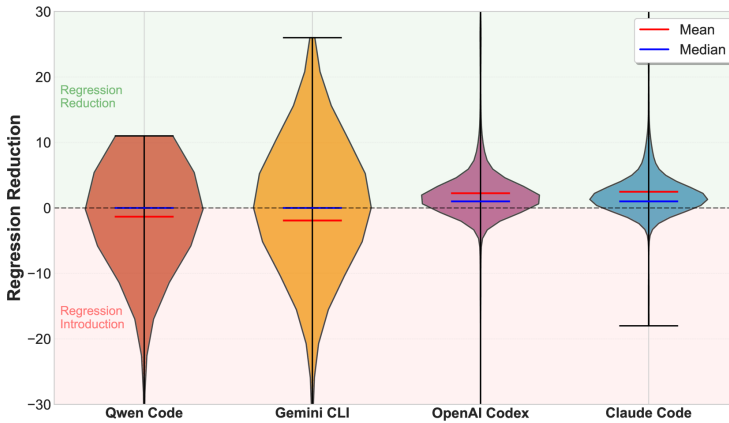


Fig. 5. Regression reduction across coding agents

In real-world deployment, the introduction of regression is particularly problematic. A “repair” that fixes one bug while introducing new failures creates net negative value, imposing a *regression tax* on development teams who must debug both the original issue and the newly introduced failures. Our results suggest that QWEN CODE and GEMINI-CLI, despite achieving non-trivial repair rates (25.81% and 41.67%), would require extensive human review before production deployment. The repair success rate alone provides an incomplete and potentially misleading evaluation of tool quality. For example, Tool A achieves a 60% repair success rate with an average regression change of -2.0, while Tool B achieves a 50% repair success rate with an average regression improvement of +1.0. This example shows that evaluations should be more nuanced and consider the overall practical value of using agentic coding tools.

Variability in Regression Reduction. Figure 5 presents violin plots that reveal the complete probability density of regression reduction outcomes. The width at any vertical position represents probability density; wider sections indicate common outcomes, narrower sections indicate rare outcomes. Red and blue lines mark the mean and median, respectively.

The violin shapes reveal a stark categorical divide across coding agents. CLAUDE CODE achieves 93.0% positive outcomes (346 of 372 repairs), 5.6% zero-change (21 repairs), and only 1.3% negative (5 repairs). CODEX shows similar behavior with 89.9% positive, 7.9% zero, and 2.2% negative. Both exhibit narrow distributions with low variance ($\sigma = 5.91$ and 6.64), indicating predictable behavior. Their worst-case scenarios are bounded (min -18 and -47). In contrast, QWEN CODE achieves only 30.2% positive outcomes (110 of 364 repairs), with 59.9% zero-change (218 repairs compile but produce no improvement) and 9.9% negative (36 repairs introduce regressions). GEMINI-CLI shows 46.1% positive, 43.5% zero, and 10.4% negative. Both display wide, dispersed distributions with high variance ($\sigma = 28.57$ and 31.25), nearly $5\times$ larger than Claude/Codex—and catastrophic worst-cases (min -543 and -488), where a single repair breaks hundreds of tests.

The variance ratio ($\sigma_{\text{Gemini}}^2 / \sigma_{\text{Claude}}^2 \approx 28$) quantifies operational risk: high-variance tools require substantially more quality assurance resources per repair. Distribution shape predicts automation feasibility—narrow violins enable autonomous deployment in CI/CD pipelines, while wide violins necessitate supervised deployment with mandatory human review. GEMINI-CLI’s range $[-488, +26]$ includes catastrophic outcomes that could halt development, whereas CLAUDE CODE’s $[-18, +73]$ contains only manageable scenarios. This tail risk asymmetry explains why practitioners need to be careful while leveraging high-variance coding assistance tools despite seemingly acceptable averages. When selecting tools to build production-ready applications, practitioners must consider different aspects: central tendency (mean, median) indicates typical outcomes, dispersion (standard deviation) reveals consistency, and worst-case tail risk (minimum/maximum values) exposes catastrophic failures.

Overall, *regression reduction* reveals the hidden cost of automated repairs. Tools with negative regression reduction impose a *regression tax* on development teams, potentially creating more problems than they solve. This aspect needs to be carefully examined for the evaluation of future coding agents in software engineering tasks.

4.8 Operational Dynamics (RQ5)

Table 5. Token consumption per bug on multi-hunk bugs with coding agents

Agent	Pass		Fail	
	Input	Output	Input	Output
QWEN CODE	1.70M	7.1K	2.37M	9.5K
GEMINI-CLI	1.33M	4.2K	5.90M	12.1K
CODEX	57.1K	5.7K	97.3K	15.4K
CLAUDE CODE	969	18.1K	308	32.0K

4.8.1 Token Consumption. While repair success measures functional correctness, it overlooks a critical operational dimension: token consumption. We analyze token consumption patterns across agents to understand the computational cost of repair attempts and whether successful repairs differ from failed ones in token consumption.

We measure API token consumption in two dimensions. Input tokens represent the total tokens sent to the model across all repair attempts. Output tokens represent the total tokens generated by the model across all repair attempts. For QWEN CODE, GEMINI-CLI, and CLAUDE CODE, token counts are extracted directly from agent trajectory logs and aggregated. For CODEX, which does not log

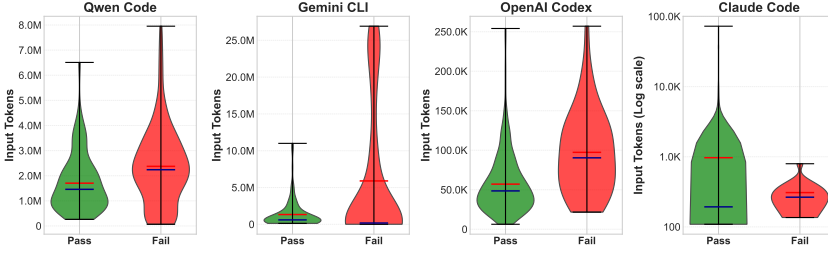


Fig. 6. Violin plot distributions of input token consumption across coding agents. Each agent has independent y-axis scaling.

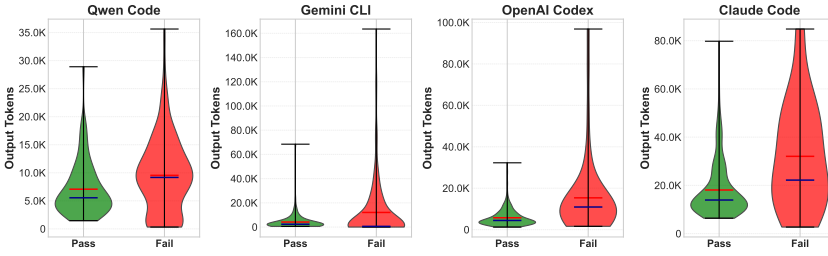


Fig. 7. Violin plot distributions of API output token generation across coding agents. Each agent has independent y-axis scaling.

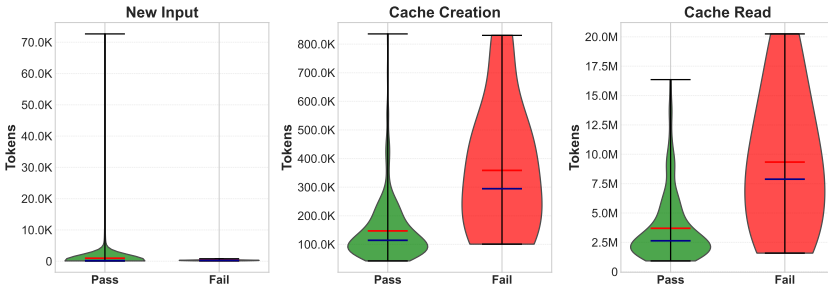


Fig. 8. CLAUDE CODE input token distribution by type and repair outcome.

token counts, we parse all input prompts and output responses using tiktoken⁶ to compute token count, then aggregate across all repair attempts.

Token Consumption Across Agents. Table 5 presents token consumption statistics across all four agents and Figures 6–7 show violin plots of token distributions by outcome. CLAUDE CODE demonstrates the lowest per-bug input token consumption, representing $57\times$ less than CODEX (57.1K), $1,373\times$ less than GEMINI-CLI (1.33M), and $1,758\times$ less than QWEN CODE (1.70M). Prompt caching enables this reduction by reusing previously cached project context. For output tokens per bug, GEMINI-CLI generates the fewest at 4.2K tokens on average, followed by CODEX at 5.7K, QWEN CODE at 7.1K, and CLAUDE CODE at 18.1K.

⁶<https://github.com/openai/tiktoken>

Figure 8 shows new input tokens only for CLAUDE CODE (not total input including cache). CLAUDE CODE’s prompt caching divides input tokens into three distinct categories. New input tokens represent uncached prompt content unique to each request, such as bug-specific information, test failures, and iterative repair instructions (successful repairs average 969 per bug). Cache creation tokens represent prompt content written to the cache for reuse across multiple API calls, such as project source code, dependency files, and static context (successful repairs average 147K per bug). Cache read tokens represent previously cached content retrieved in subsequent API invocation (successful repairs average 3.70M per bug). Figure 8 reveals that cache read tokens dominate at 96% of total input volume.

Token Consumption by Repair Outcome. Failed repairs exhibit distinct token consumption patterns across agents, as shown by wider failure distributions (red violins) in Figure 6 and Figure 7. QWEN CODE’s failed repairs use 39.3% more input tokens and 35.0% more output tokens compared to successful repairs. GEMINI-CLI exhibits the most dramatic increase, with failed repairs consuming 343.2% more input tokens and 190.5% more output tokens. CODEX shows 70.4% more input tokens and 170.2% more output tokens for failures. These patterns indicate that failures do not terminate early; agents persist through multiple unsuccessful attempts before giving up.

CLAUDE CODE shows a distinct pattern due to its prompt caching mechanism. Failed repairs use 68.2% fewer new input tokens (308 vs. 969) but generate 77.1% more output tokens (32.0K vs. 18.1K). The lower number of new input tokens results from cache reuse across repeated repair attempts, where most of the prompt context remains unchanged. Figure 8 shows that failed repairs consume 143.7% more cache creation tokens and 152.4% more cache read tokens than successful ones. This indicates that caching enables the reuse of stored context and avoids the cost of resending large inputs. Overall, prompt caching reduces expensive input token usage while preserving access to the required contextual information.

Table 6. Mean runtime (in seconds) for successful and failed repairs across coding agents.

Agent	Successful Repairs (s)	Failed Repairs (s)	Pass–Fail Gap (s)
QWEN CODE	341.71	490.20	148.49
GEMINI-CLI	148.17	781.30	633.13
CODEX	313.84	739.36	425.52
CLAUDE CODE	345.89	671.20	325.31
Average	287.40	670.52	383.11

4.8.2 Runtime. Table 6 summarizes the mean runtime of successful and failed repair attempts across coding agents. Overall, failed repairs consistently require substantially more time than successful ones, with an average runtime of 670 s compared to 287 s. GEMINI-CLI exhibits the shortest average runtime for successful repairs but the longest for failed repairs, while QWEN CODE demonstrates the smallest pass–fail gap, indicating more stable runtime behavior across outcomes.

The consistent pattern across all agents that failed repairs require substantially longer runtimes than successful ones suggests that agents invest more time exploring alternative solutions when initial approaches fail, with different agents exhibiting varying degrees of persistence and exploration strategies.

4.8.3 Understanding agentic actions. The actions $a \in \mathcal{A}$ (Equation 11) can be implemented in two distinct ways. First, coding agents may invoke *native tools* provided directly by the agent framework. For example, CLAUDE CODE provides `read_file` to retrieve file content, `edit` to modify code, and `search_file_content` to search within files. Similarly, QWEN CODE offers `write_file` to edit

files, `list_directory` to enumerate directory contents, and `search_file_content` to locate code patterns. Second, agents may execute *shell-invoked commands* through bash interfaces, including standard Unix utilities (`cat`, `sed`, `grep`), build systems (`mvn test`, and domain-specific frameworks (`defects4j compile`, `defects4j test`).

Table 7. Unique command diversity across coding agents

Agent	Native Tools	Shell-Invoked	Total Unique
QWEN CODE	10	81	91
GEMINI-CLI	11	27	38
CODEX	0	96	96
CLAUDE CODE	7	60	67

Tool Diversity. Table 7 presents the number of unique tools invoked by each agent across all repair attempts in HUNK4J. The total unique count represents distinct commands used at least once during evaluation. Notably, CODEX relies exclusively on shell-invoked commands (96 unique commands), executing every action through shell interfaces without native tool support. This terminal-centric design reflects a general-purpose interaction model where file operations, code editing, build execution, and version control are all performed via command-line invocations. QWEN CODE and CLAUDE CODE exhibit similar patterns, using primarily shell-invoked commands (81 and 60 unique commands, respectively) with minimal native tool integration (10 and 7 native tools). In contrast, GEMINI-CLI demonstrates more balanced usage, employing 11 native tools alongside 27 shell-invoked commands.

These differences in tool diversity and implementation paradigms have implications for how agents allocate effort across repair activities. Agents relying on shell-invoked commands must construct command-line invocations and interpret unstructured output, whereas agents leveraging native tools benefit from structured interfaces and reduced parsing overhead.

Tool Usage Patterns in Successful vs Failed Repairs. To analyze how agents allocate effort differently in successful versus unsuccessful repairs, we abstract all tool invocations within the action space \mathcal{A} (Equation 11) into functional categories based on operational purpose. Commands are grouped by their function rather than implementation: `read_file` and `cat` both constitute READ operations, while `edit` and `write_file` both represent WRITE operations.

Table 8. Tool category usage: successful and failed repairs across coding agents

Category	QWEN CODE			GEMINI-CLI			CODEX			CLAUDE CODE		
	Pass (%)	Fail (%)	Total (%)	Pass (%)	Fail (%)	Total (%)	Pass (%)	Fail (%)	Total (%)	Pass (%)	Fail (%)	Total (%)
WRITE	14.4	15.1	14.9	25.6	41.7	36.9	44.6	41.4	44.0	22.1	17.3	21.5
READ	25.2	23.1	23.6	26.6	22.4	23.6	5.6	5.0	5.5	22.8	24.1	23.0
TEST	16.8	15.6	15.9	22.8	17.0	18.7	14.9	14.6	14.8	14.9	15.3	14.9
BUILD	9.2	10.6	10.3	13.0	12.0	12.3	6.1	10.2	7.0	12.3	12.8	12.3
SEARCH_CONTENT	8.8	8.3	8.4	2.0	1.7	1.8	9.4	10.3	9.6	8.9	13.9	9.5
SEARCH_FILES	6.7	6.3	6.4	4.7	2.1	2.8	5.9	4.1	5.5	11.0	7.5	10.6
NAVIGATE	14.9	16.2	15.9	0.0	0.0	0.0	0.0	0.0	0.0	0.8	1.4	0.9

Table 8 presents the distribution of the top seven most frequently invoked categories across all agents, separated by repair outcome. The categories include WRITE (code modification), READ (file content retrieval), TEST (test execution), BUILD (compilation), SEARCH_CONTENT (pattern search), SEARCH_FILES (file system queries), and NAVIGATE (directory changes). Each value

represents the percentage of commands in that category relative to the agent’s total command executions for that outcome class.

The data reveals two primary failure modes: *over-modification without validation* and *over-exploration without action*. GEMINI-CLI demonstrates the first mode: WRITE increases from 25.6% to 41.7% while TEST decreases from 22.8% to 17.0%, yielding a modification-to-comprehension ratio shift from 0.96:1 (Pass) to 1.86:1 (Fail). CLAUDE CODE demonstrates the second mode: WRITE *decreases* from 22.1% to 17.3% while SEARCH_CONTENT *increases* from 8.9% to 13.9% in failed repairs, producing a lower modification-to-comprehension ratio (0.72:1) when failing. CODEX maintains consistently minimal comprehension effort (5% READ) regardless of outcome, yielding an 8:1 modification-to-comprehension ratio in both successful and failed repairs. Failed repairs increase BUILD operations from 6.1% to 10.2%, indicating repeated recompilation attempts. QWEN CODE allocates 15.9% to NAVIGATE operations, unique among all agents, suggesting excessive directory traversal overhead.

A common pattern emerges: failed repairs allocate more effort to BUILD operations across three agents (QWEN CODE: 9.2%→10.6%, CODEX: 6.1%→10.2%, CLAUDE CODE: 12.3%→12.8%), suggesting iterative debugging without convergence. The absence of a universal Pass-Fail pattern indicates that effective repair strategies are agent-specific: GEMINI-CLI requires reducing excessive modification and increasing validation, while CLAUDE CODE requires reducing excessive search and increasing code modification.

Tool Sequence Patterns in Successful vs Unsuccessful Repairs. To understand how repair strategies differ between successful and unsuccessful attempts, we analyze tool sequence patterns using a sliding window approach with $w = 3$. A relatively short window ($w = 3$) is chosen to capture local action dependencies without diluting interpretability across longer and more variable trajectories. Additional results for window sizes $w = 4$ and $w = 5$ are included in our replication package.

We extract consecutive 3-step tool invocations from each trajectory and count pattern frequencies for the two lowest-performing agents, QWEN CODE and GEMINI-CLI. Figures 9 and 10 show the top five patterns separated by repair outcome (abbreviated as WR=WRITE, RD=READ, TST=TEST, BLD=BUILD, NAV=NAVIGATE, SC=SEARCH_CONTENT). The distribution of tool-use patterns follows a long-tail characteristic, with a large number of distinct sequences occurring infrequently. We report both frequency counts and normalized percentages for the most frequently observed patterns to aid interpretability. The relatively low percentages stem from the extensive diversity of distinct action sequences.

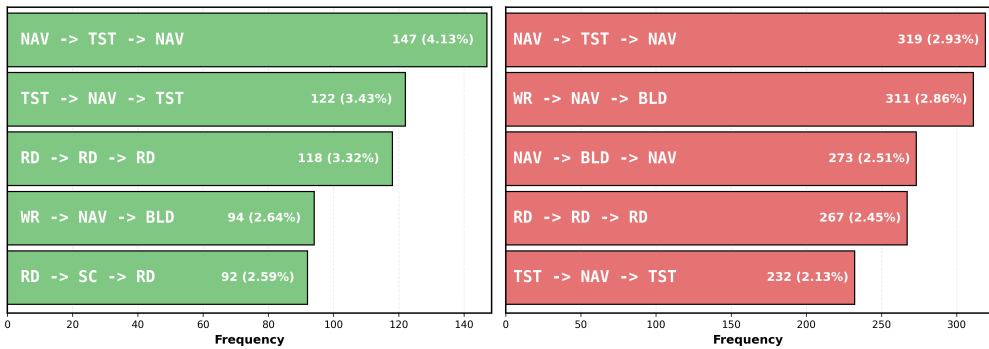


Fig. 9. Tool sequence patterns for QWEN CODE.

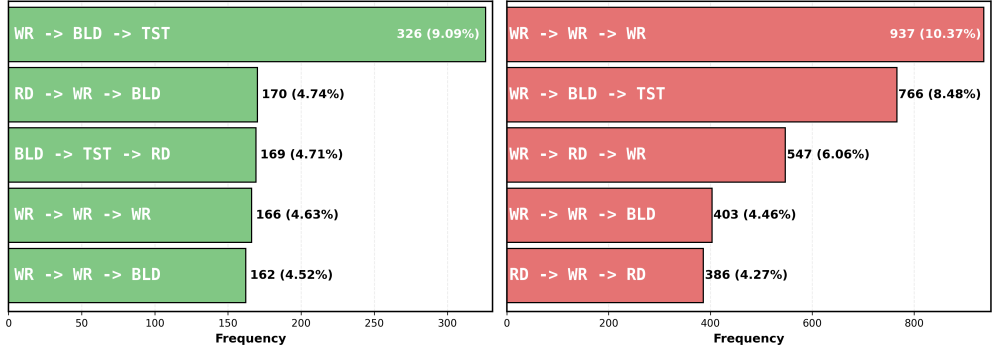


Fig. 10. Tool sequence patterns for GEMINI-CLI.

QWEN CODE exhibits navigation overhead in both successful and unsuccessful repairs. The most frequent pattern is NAVIGATE → TEST → NAVIGATE (4.13% successful, 2.93% unsuccessful). NAVIGATE appears in 3/5 successful patterns and 4/5 unsuccessful patterns. WRITE → NAVIGATE → BUILD occurs at 2.64% in successful repairs and 2.86% in unsuccessful repairs. Navigation overhead persists regardless of outcome.

GEMINI-CLI exhibits different patterns in successful versus unsuccessful repairs. Successful repairs use WRITE → BUILD → TEST most frequently (9.09%). Unsuccessful repairs use WRITE → WRITE → WRITE most frequently (10.37%), a 2.24-fold increase over successful repairs (4.63%). WRITE → READ → WRITE occurs at 6.06% in unsuccessful repairs. Unsuccessful repairs accumulate consecutive modifications without intermediate compilation or testing.

QWEN CODE and GEMINI-CLI exhibit distinct failure modes. QWEN CODE shows navigation overhead in both successful and unsuccessful repairs. GEMINI-CLI shows a behavioral shift: successful repairs use WRITE → BUILD → TEST (9.09%), while unsuccessful repairs use WRITE → WRITE → WRITE (10.37%). QWEN CODE requires architectural changes to reduce navigation overhead. GEMINI-CLI requires strategies to prevent consecutive modifications without validation.

4.9 Role of context-assistance with MAPLE (RQ6)

As shown in Table 2, QWEN CODE and GEMINI-CLI achieve substantially lower repair accuracy than CLAUDE CODE and CODEX. We examine whether providing repository-level context through MAPLE can improve the accuracy of these less effective agents.

We selected 10 random bugs from each of the five spatial proximity classes in HUNK4J, yielding 50 bugs in total. We ran both QWEN CODE and GEMINI-CLI on this subset under two conditions: baseline (without MAPLE) and with MAPLE. MAPLE provides structured search capabilities through MCP tools (Table 1) that equip coding agents with intra-repository tools to query class definitions, method implementations, and code snippets across the repository. Table 9 presents the impact of MAPLE on these agents.

Impact on Localization. For QWEN CODE, localization success remains unchanged at 13 bugs (26.00%) with and without MAPLE, indicating that the agent’s localization challenges may stem from fundamental reasoning limitations of the underlying model QWEN3-CODER-FLASH. In contrast, GEMINI-CLI shows substantial improvement, increasing from 16 bugs (32.00%) to 21 bugs (42.00%), a relative gain of 31.25%. This improvement indicates that GEMINI-CLI can effectively leverage structured repository search tools when such tools are available, successfully identifying 5 additional fault locations that were previously missed. This finding suggests that context-assistance tools yield

Table 9. Impact of MAPLE on coding agents for multi-hunk bug repair

Agent	Localization Success (%)		Compilation Success (%)		Regression Reduction (Avg)		Accuracy (%)	
	Baseline	+MAPLE	Baseline	+MAPLE	Baseline	+MAPLE	Baseline	+MAPLE
QWEN CODE	13 (26.00%)	13 (26.00%)	49 (98.00%)	48 (96.00%)	0.31	-1.00	11 (22.00%)	12 (24.00%)
GEMINI-CLI	16 (32.00%)	21 (42.00%)	49 (98.00%)	45 (90.00%)	1.00	0.36	20 (40.00%)	26 (52.00%)

benefits primarily for agents with sufficient baseline reasoning capacity to exploit the additional information.

Impact on Repair Accuracy. The repair accuracy results reveal a promising pattern. GEMINI-CLI yields a substantial accuracy gain, rising from 20 bugs (40.00%) to 26 bugs (52.00%) with 6 additional correct repairs representing a 30% relative improvement. With the inclusion of MAPLE, GEMINI-CLI transitions from fixing fewer than half of the bugs to successfully repairing the majority. QWEN CODE shows more modest improvement, increasing from 11 bugs (22.00%) to 12 bugs (24.00%).

These gains come with trade-offs in compilation success and regression metrics. Compilation success decreases for both agents: QWEN CODE drops from 98% to 96%, while GEMINI-CLI declines from 98% to 90%. Regression reduction also changes: QWEN CODE moves from +0.31 to -1.00, and GEMINI-CLI decreases from +1.00 to +0.36. These changes suggest that MAPLE encourages more exploratory repair attempts, occasionally leading to syntactic errors or unintended side effects. However, the overall repair success for GEMINI-CLI, 6 additional correct fixes, demonstrates that the accuracy gains outweigh the quality penalties in the aggregate. The trade-offs reflect a fundamental challenge in agentic repair: a richer context enables more ambitious fixes but also increases exploration risk.

5 Discussion

Our empirical study of four LLM-driven coding agents on multi-hunk repair tasks reveals critical insights into their capabilities and limitations. The results extend beyond benchmarking coding agents, providing a nuanced view of agent behavior that has implications for practitioners, researchers, and tool builders.

Beyond Accuracy to Trustworthiness. Our findings demonstrate that repair accuracy is an incomplete metric when considered in isolation. Agents like QWEN CODE and GEMINI-CLI, despite achieving moderate success rates, introduce a substantial “regression tax” by frequently causing new test failures. This behavior makes them a liability in automated CI/CD pipelines, where stability is crucial. In contrast, high-performing agents like CLAUDE CODE and CODEX show positive regression reduction, indicating they produce more reliable fixes. Practitioners should pay attention to selecting an agent that prioritizes not only its success rate but also its regression behavior and the high computational cost of its failed repairs. Agents with high-variance outcomes require mandatory human oversight, while those with predictable, high-quality output are closer to enabling autonomous deployment.

Localization Criterion and Patch Equivalence. Our localization metric operates at the file level and measures whether agents modify the same files as those changed in developer patches. File-level localization provides a clear and reproducible basis for comparing agents, using developer patches as the ground truth. However, it overlooks several nuances that affect interpretation.

Semantically equivalent patches may modify different files. For example, a null pointer exception can be fixed either by adding a null check at the call site or by updating the called method to handle

null inputs. Both eliminate the defect but involve different files, which may cause the metric to underestimate localization capability when agents identify valid alternative repair locations.

Codec_13 provides a concrete example. The bug presents itself as a potential null pointer exception in `DoubleMetaphone.java` when comparing encoding results. The developer patch addresses this through a multi-file refactoring: it removes `CharSequenceUtils.java`, adds a null-safe `equals()` method to `StringUtils.java`, and modifies `DoubleMetaphone.java` to use `StringUtils.equals()` instead of direct comparison⁷. This approach spans three files. In contrast, CLAUDE CODE implements explicit null checking directly within `DoubleMetaphone.java`, using intermediate variables and conditional logic to safely handle null values⁸. Both patches eliminate the defect and pass all tests.

These factors help explain cases where localization success appears lower than repair accuracy for high-performing agents such as CODEX and CLAUDE CODE. These agents may generate correct patches by modifying semantically equivalent file sets or by applying broad edits within a correctly localized file that happen to resolve the defect.

Localization-Repair Gap. A key finding is the *localization-repair gap*, where agents can accurately identify all files requiring modification yet still fail to synthesize a correct patch. This gap highlights a limitation of current agentic architectures; effective localization does not necessarily translate into coherent reasoning or coordinated editing across interdependent code regions. However, for complex bugs such as the ones unfixed by all four agents (See Table 4), localization remains a challenge. Metrics such as hunk divergence and spatial proximity are predictors of repair difficulty and can serve as benchmarks for evaluating future progress in this area.

Agent Efficiency. Failed repairs impose substantial computational costs, as coding agents often continue consuming resources rather than terminating early. For practitioners evaluating production-ready systems, this highlights the importance of assessing not only repair accuracy but also the efficiency of failure handling. High-performing agents such as CLAUDE CODE and CODEX justify their resource usage through consistently higher success rates, whereas low-performing agents incur compounded costs—low success combined with high resource expenditure per failure, making them less suitable for large-scale deployment.

These findings point to concrete directions for tool improvement. For agentic tool developers, our results emphasize that consistency and semantic understanding are as critical as raw coding ability. Agents should be engineered to detect unproductive repair paths and “fail fast,” conserving computational resources for promising attempts. The effectiveness of CLAUDE CODE’s caching mechanism further illustrates the value of efficient context management in reducing redundant computation and improving overall repair efficiency.

Implications for Context-Driven Repair. Our findings with MAPLE suggest that context-assistance tools can be helpful to coding agents. The 30% improvement in accuracy for GEMINI-CLI with MAPLE is particularly encouraging, as it shows that structured repository tools can help coding agents. The limited impact on QWEN CODE suggests a threshold below which additional context provides diminishing returns, indicating that context-assistance is most effective when paired with agents possessing baseline reasoning capabilities. The compilation and regression trade-offs highlight opportunities for refinement. Future work may explore adaptive context retrieval strategies that balance exploration with conservatism or integrate feedback mechanisms that guide agents toward more targeted repairs. Future agentic systems should integrate more sophisticated, structure-aware context retrieval mechanisms, as demonstrated by the performance gains from MAPLE.

⁷https://github.com/agentic-se/agentic-multihunk-repair/blob/main/Codec_13_patches/Codec_13_ground_truth.diff

⁸https://github.com/agentic-se/agentic-multihunk-repair/blob/main/Codec_13_patches/Codec_13_claude.diff

Non-Deterministic Nature of Runs. Agentic multi-hunk repair is inherently non-deterministic. Even when provided with identical inputs, repeated runs of the same agent may follow distinct trajectories and yield different outcomes. This variability stems from the probabilistic nature of the underlying LLM and the continuous feedback-driven interaction between the agent and the development environment.

During repair, even a minor variation in the internal reasoning of the model, such as proposing an alternative fault hypothesis, selecting a different file to edit, or generating a slightly different patch, can alter the trajectory of the agent. The agent continuously updates its plan based on compiler diagnostics, test outcomes, and observed code changes, and each reasoning step directly affects the subsequent actions. This feedback loop introduces path dependency, where small perturbations in reasoning responses can propagate into substantially different sequences of actions.

These non-deterministic effects are particularly amplified in multi-hunk repair. Coordinated modifications across multiple interdependent code regions require consistent reasoning over distributed contexts. Small deviations, such as addressing one hunk prematurely or misinterpreting a diagnostic message, can disrupt this coordination and lead to incoherent or incomplete patches.

Auto-Update Behavior of the Agents. A controlled evaluation of coding agents requires stable and reproducible environments. During our experiments, we observed unexpected instances of automatic software updates initiated by both the QWEN CODE and CLAUDE CODE agents. In the case of QWEN CODE, the agent initiated a self-update without explicit user permission while executing a repair task, displaying the message `Update successful! The new version will be used on your next run.`

For QWEN CODE, the experiment began with version 0.0.11, which automatically upgraded to version 0.0.14 during execution, as verified using the command `qwen -version`. Similar update behavior, though without explicit notification, was observed for CLAUDE CODE.

Operational Instability of Models. The execution of agentic systems depends on the continuous availability and stability of their hosted model infrastructure. During our evaluation, we observed several external factors that affected the operational consistency of CLAUDE CODE and other hosted agents.⁹ The first factor involves service outages, where the Claude status dashboard reported elevated error rates for the Sonnet 4.5 model and temporary unavailability of the Claude API and CLAUDE CODE interfaces.¹⁰ A second factor concerns degraded service quality, caused by an upstream provider error that exposed a fault in Anthropic infrastructure, leading to intermittent access issues and increased latency across the agent and API.¹¹ A third factor relates to dynamic service recovery, where fluctuating system states during mitigation intervals resulted in inconsistent agent responses and incomplete repair trajectories.

These operational variations highlight the dependency of coding agents on third-party hosted environments. Even short-lived degradations or transient failures can interrupt long-running repair sessions, influence success rates, and reduce experimental reproducibility. Establishing controlled execution environments and reliable service baselines remains essential for rigorous empirical evaluation of agentic systems.

Human-in-the-loop. In this work, we evaluate agents in a fully autonomous mode. In reality, these are used as assistants to human developers. A failed patch might still provide a valuable starting point for a human to fix the remaining issues. Future work could explore how agents can serve as assistants, generating partial or candidate fixes for complex bugs, with a human developer

⁹<https://status.claude.com/>

¹⁰<https://status.claude.com/incidents/ggjp05h790b3>

¹¹<https://status.claude.com/incidents/gr1vrcvz9jd4>

guiding, validating, and completing the repair. This would also necessitate developing new metrics, such as the ‘closeness to the ideal fix’, to quantify the utility of imperfect patches.

6 Threats to Validity

Construct Validity. This concerns the relationship between the theoretical concepts being studied and the specific metrics used to measure them. Our defined metrics, namely Localization Success, Compilation Success, Regression Reduction, and Repair Accuracy, are designed to capture key aspects of the repair process. However, they are simplifications of a complex reality. For example, our Repair Accuracy metric is binary (a patch either passes all tests or it does not), which does not account for partially correct patches that might still be useful to a developer. Similarly, Localization Success operates at the file level, meaning an agent could modify the correct files but still fail to address the correct buggy hunks within them. Nevertheless, we believe these metrics provide a clear, objective, and reproducible benchmark for comparing agent performance on the core task of fully automated repair. To provide a more nuanced view, we complemented these outcome-based metrics with a qualitative analysis of Tool Sequence Patterns, which reveals the underlying behaviors and strategies of the agents, offering insights beyond simple success or failure.

Internal Validity. This refers to the confidence that the observed outcomes are caused by the experimental variables and not by confounding factors. As discussed in Section 5, LLM-based Agentic systems are inherently non-deterministic. The same agent, given the same initial prompt, can produce different action trajectories and final patches across multiple runs due to the probabilistic nature of the underlying LLM. This means a single successful or failed run may not be representative of the agent’s true capability. While eliminating this non-determinism is currently impossible, we mitigated its impact by performing the evaluation across a wide variety of problems (i.e., on 372 distinct multi-hunk bugs from the Hunk4J dataset). This allows us to observe overarching trends and patterns in agent behavior that are less susceptible to the randomness of a single execution.

As detailed in Sections 5 and 5, the performance of the agents can be affected by external factors such as API outages, service degradation, and automatic background updates. These events are outside of our control and could have impacted the outcomes of specific repair attempts. We addressed this by using a standardized evaluation harness that provided identical access to tools and environments for all agents. We logged all interactions and, where possible, noted the agent versions before and after any auto-updates. While these factors represent a real-world challenge of using agentic systems, our reporting on these instabilities contributes to a more transparent understanding of the current state of these tools.

External Validity. This concerns the generalizability of our findings to other contexts, such as different programming languages, projects, bug types, or coding agents. We evaluated four prominent agentic coding tools: CODEX, GEMINI-CLI, QWEN CODE, and CLAUDE CODE. However, the field of AI is rapidly evolving, and new, more capable agents may have emerged since our evaluation was conducted. The reported performance numbers may not accurately reflect the capabilities of the latest models. The agents selected were state-of-the-art and widely used at the time of our study, to the best of our knowledge. More importantly, our primary contribution is not just the performance benchmark itself, but the conceptual framework, the behavioral metrics, and the qualitative insights into the challenges of multi-hunk repair. These findings, such as the difficulty in coordinating edits across dispersed locations, are fundamental to the problem and are likely to remain relevant for future generations of agents. Our evaluation methodology can be readily applied to benchmark new tools as they become available.

Our study relies exclusively on the HUNK4J dataset, which contains multi-hunk bugs from Java projects. Therefore, our findings may not directly generalize to other programming languages (e.g.,

Python, C++, JavaScript) or to types of bugs not represented in this dataset (e.g., concurrency or performance bugs). However, Hunk4J is, to our knowledge, the largest and most systematically curated dataset specifically designed for multi-hunk bug repair research, making it the most appropriate choice for this foundational study. By focusing on a single language, we control for a significant variable, allowing for a clearer analysis of agent behavior. Future work should undoubtedly seek to replicate these findings in other languages and on different project domains to establish the broader generalizability of our conclusions.

7 Related Work

Autonomous agents iteratively reason, invoke tools, and adapt based on environmental feedback [28]. Bug repair agents include RepairAgent [4], AutoCodeRover [44], OpenHands [34], SWE-Agent [39], Agentless [36], Magis [33], AgentCoder [11], MarsCode Agent [16], FixAgent [13], and Passerine [26]. Extensions such as SpecRover [29] employ dedicated reproduction and testing phases. These agents employ diverse architectures: iterative reasoning and testing, structured search-locate-fix workflows, and specialized agent-computer interfaces. There are also specialized agents that handle project setup [10], test execution [6], notebook debugging [9], package installation [18], and log analysis [27].

While these agents are employed on diverse software engineering tasks, understanding their internal decision-making processes, operational dynamics, and failure modes remains largely unexplored. Recent work [5] analyzes trajectories of three agents (RepairAgent, AutoCodeRover, OpenHands) to identify debugging anti-patterns through qualitative analysis. However, their study differs from ours in several fundamental ways. First, they evaluate research prototypes while we evaluate production coding agents used by actual developers (CLAUDE CODE, CODEX, GEMINI-CLI, QWEN CODE) which leverage latest SOTA models (SONNET-4.5, GPT-5, GEMINI-2.5-FLASH, QWEN3-CODER-FLASH). Second, they sample 120 trajectories, whereas we conduct a large-scale study on 372 multi-hunk bugs (1,488 repair trajectories). Third, they lack systematic multi-hunk bug characterization—our work introduces hunk divergence and spatial proximity metrics to quantify multi-hunk complexity, showing that bug characteristics strongly predict repair success. Fourth, their analysis focuses on qualitative action patterns, while we provide a multi-dimensional quantitative evaluation. Finally, we propose and evaluate Maple, an MCP-based context-assistance mechanism that improves mid-tier agent accuracy by 30%. The BIRCH benchmark [21] specifically targets multi-hunk bugs and introduces the hunk divergence and spatial proximity metrics we employ. Our work builds on BIRCH by evaluating autonomous agents rather than direct LLM prompting, revealing how agent architectures, tool usage patterns, and operational dynamics affect multi-hunk repair success.

8 Conclusion

This work presented the first systematic study of LLM-driven coding agents on the complex task of multi-hunk program repair. We evaluated four agents (CLAUDE CODE, CODEX, GEMINI-CLI, and QWEN CODE) on 372 multi-hunk bugs from the HUNK4J dataset, introducing fine-grained metrics for localization, repair accuracy, regression behavior, and operational dynamics. We also developed MAPLE, an MCP-based tool, which successfully improved the repair accuracy of less effective agents. Our analysis revealed substantial performance variation, with repair accuracy ranging from 25.81% (QWEN CODE) to 93.28% (CLAUDE CODE). Success was strongly influenced by bug complexity, as repair accuracy consistently declined sharply with increasing bug dispersion (hunk divergence). We identified a distinct localization-repair gap; 50% of the 10 bugs unfixed by any agent were successfully localized by at least one agent, yet a correct fix could not be synthesized. The operational assessment revealed a stark effectiveness divide based on regression reduction. High-performing

agents, CLAUDE CODE (+2.47) and CODEX (+2.25), achieved positive average regression reduction, demonstrating superior semantic consistency. Conversely, QWEN CODE (-1.34) and GEMINI-CLI (-1.92) exhibited negative regression reduction, implying they introduced new test failures, thus imposing a “regression tax”. Furthermore, agents do not “fail fast”; failed repairs are expensive, often consuming significantly more resources (tokens) than successful ones (e.g., GEMINI-CLI failed repairs consumed 343.2% more input tokens). These results characterize the accuracy and behavioral mechanisms of coding agents, underscoring the necessity of moving beyond simple binary accuracy metrics to assess operational dynamics, resource efficiency, and regression reduction for developing production-ready agentic systems.

9 Data Availability

The paper is currently under review, and the code artifacts will be made available soon.

References

- [1] Alibaba Cloud. 2024. Qwen Code: Large Language Models for Code. <https://qwenlm.github.io/qwen-code-docs/en/>. Accessed: 2024-12-15.
- [2] Anthropic. 2024. Claude Code. <https://www.claude.com/product/claude-code>. Accessed: 2024-12-15.
- [3] Anthropic. 2024. Model Context Protocol Specification. <https://modelcontextprotocol.io>. Accessed: 2024-01-15.
- [4] I. Bouzenia, P. Devanbu, and M. Pradel. 2025. RepairAgent: An Autonomous, LLM-based Agent for Program Repair. In *Proceedings of ICSE 2025*.
- [5] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*.
- [6] I. Bouzenia and M. Pradel. 2025. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. In *Proceedings of ISSTA 2025*.
- [7] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations*.
- [8] Google Cloud. 2024. Gemini CLI: Code Assist Command-Line Interface. <https://cloud.google.com/gemini/docs/codeassist/gemini-cli>. Accessed: 2024-12-15.
- [9] K. Grotov, A. Borzilov, M. Krivobok, T. Bryksin, and Y. Zharov. 2024. Debug Smarter, Not Harder: AI Agents for Error Resolution in Computational Notebooks. In *arXiv preprint arXiv:2410.14393*.
- [10] R. Hu, C. Peng, X. Wang, and C. Gao. 2025. An LLM-based Agent for Reliable Docker Environment Configuration. *arXiv preprint arXiv:2502.13681* (2025).
- [11] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui. 2024. AgentCoder: Multi-agent-based Code Generation with Iterative Testing and Optimisation. In *arXiv preprint*.
- [12] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 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*. 298–309.
- [13] C. Lee, C. S. Xia, L. Yang, et al. 2024. A Unified Debugging Approach via LLM-based Multi-agent Synergy. In *arXiv preprint arXiv:2404.17153*.
- [14] 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*. 602–614.
- [15] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*. 511–523.
- [16] Y. Liu, P. Gao, X. Wang, C. Peng, and Z. Zhang. 2024. MarsCode Agent: AI-native Automated Bug Fixing. In *arXiv preprint arXiv:2409.00899*.
- [17] 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*. 691–701.
- [18] L. Milliken, S. Kang, and S. Yoo. 2024. Beyond Pip Install: Evaluating LLM Agents for the Automated Installation of Python Projects. In *arXiv preprint arXiv:2412.06294*.
- [19] Marjane Namavar, Noor Nashid, and Ali Mesbah. 2022. A Controlled Experiment of Different Code Representations for Learning-Based Bug Repair. *Empirical Software Engineering Journal* (2022).
- [20] Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. 2026. Issue2Test: Generating Reproducing Test Cases from Issue Reports. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. IEEE.

- [21] Noor Nashid, Daniel Ding, Keheliya Gallaba, Ahmed E Hassan, and Ali Mesbah. 2025. Characterizing Multi-Hunk Patches: Divergence, Proximity, and LLM Repair Challenges. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*.
- [22] Noor Nashid, Taha Shabani, Parsa Alian, and Ali Mesbah. 2024. Contextual api completion for unseen repositories using llms. *arXiv preprint arXiv:2405.04600* (2024).
- [23] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Embedding context as code dependencies for neural program repair. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 95–106.
- [24] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.
- [25] OpenAI. 2024. OpenAI Codex. <https://openai.com/codex/>. Accessed: 2024-12-15.
- [26] P. Rondon, R. Wei, J. Cambroner, et al. 2025. Evaluating Agent-based Program Repair at Google. *arXiv preprint arXiv:2501.07531* (2025).
- [27] D. Roy, X. Zhang, R. Bhave, et al. 2024. Exploring LLM-based Agents for Root Cause Analysis. In *Proceedings of FSE 2024 Companion*. 208–219.
- [28] A. Roychoudhury, C. Pasareanu, M. Pradel, and B. Ray. 2025. Agentic AI Software Engineer: Programming with Trust. *arXiv preprint arXiv:2502.13767* (2025).
- [29] H. Ruan, Y. Zhang, and A. Roychoudhury. 2024. SpecRover: Code Intent Extraction via LLMs. In *arXiv preprint arXiv:2408.02232*.
- [30] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE Press, 13–24.
- [31] André Silva and Martin Monperrus. 2024. *RepairBench: Leaderboard of Frontier Models for Program Repair*. Technical Report 2409.18952. arXiv. <https://arxiv.org/abs/2409.18952>
- [32] Mifta Sintaha, Nashid Noor, and Ali Mesbah. 2023. Dual Slicing-Based Context for Learning Bug Fixes. *Transactions on Software Engineering and Methodology (TOSEM)* (2023), 27 pages.
- [33] W. Tao, Y. Zhou, W. Zhang, and Y. Cheng. 2024. Magis: LLM-based Multi-Agent Framework for GitHub Issue Resolution. In *arXiv preprint arXiv:2403.17927*.
- [34] X. Wang, B. Li, Y. Song, et al. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *Proceedings of ICLR 2024*.
- [35] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 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*. 354–366.
- [36] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. In *arXiv preprint arXiv:2407.01489*.
- [37] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.
- [38] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. Transplantfix: Graph differencing-based code transplantation for automated program repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
- [39] J. Yang, C. E. Jimenez, A. Wettig, et al. 2024. SWE-Agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 2024*.
- [40] S. Yao, J. Zhao, D. Yu, et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of ICLR 2023*.
- [41] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *44th International Conference on Software Engineering*. ACM, 1506–1518.
- [42] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM.
- [43] Yuan Yuan and Wolfgang Banzhaf. 2019. A hybrid evolutionary system for automatic software repair. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1417–1425.
- [44] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of ISSA 2024*. 1592–1604.
- [45] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 341–353.
- [46] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-aware neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1443–1455.