To Run or Not to Run: Analyzing the Cost-Effectiveness of Code Execution in LLM-Based Program Repair

ANONYMOUS AUTHOR(S)

LLM-based agents for program repair are increasingly build on a "generate-run-revise" paradigm, iteratively executing tests to evaluate and refine patches. This execution-based approach has become standard practice in state-of-the-art systems. However, executions can be time-consuming and expensive, yet their impact on these agents remain underexplored. In this paper, we conduct a two-stage empirical study over execution behavior in LLM-based program repair. To characterize execution behavior at scale, we first analyze 7,745 agent traces from SWE-bench leaderboard submissions. Second, we evaluate 3,000 end-to-end repair attempts across 200 SWE-bench instances and three agents (Claude Code, Codex, and the open-source OpenCode) under four execution paradigms, which allows for a fine-grained comparison of performance and cost. Our analysis reveals three key observations: ❶ Code execution is used across all agents and models analyzed, with an average of 8.8 test runs per task. Execution behavior varies substantially across agents and models, with frequency ranging from 2 to 19 per task, and late-stage executions (66–100% of conversation) consistently achieve higher success rates than early-stage ones (57.9% average). ❷ Execution restrictions have little effect on repair success: On commercial agents with SOTA models the resolve-rate gap between Prohibited and Unrestricted is only 1.25pp (not statistically significant, $p > 0.05$). The corresponding value for open-source OpenCode with Qwen2.5-Coder-32B is ≈ 0pp, with equivalence holding under both prompt-level and tool-level enforcement of the restriction. Prohibited saves 56–62% tokens and 48–54% wall-clock on Claude Code, and removes the need to maintain per-repository test environments. ❸ Execution benefit is concentrated rather than uniform. For commercial agents, 54–66% of cases complete in a single edit, localization accuracy below Prohibited is over 95%, and 81–100% of failed cases pass agent-executed validation but fail the official evaluation. OpenCode with Qwen2.5-Coder-32B shows another failure mode: it retries more frequently and only 11% of its failed cases pass self-validation. These patterns suggest that current agents apply execution indiscriminately, paying their cost on instances where it provides little benefit. Execution, therefore, should be treated as a resource with an explicit cost-benefit tradeoff, not a default capability.

Additional Key Words and Phrases: automated program repair, LLM agents, SWE-bench, empirical study

ACM Reference Format:
Anonymous Author(s). 2026. To Run or Not to Run: Analyzing the Cost-Effectiveness of Code Execution in LLM-Based Program Repair. 1, 1 (May 2026), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code agents have been widely applied to software engineering tasks, among which automated program repair (APR) stands out as a critical application [38, 41, 44]. Modern code agents typically follow an iterative workflow for program repair, where code execution plays a central role. By executing code, agents can reproduce bugs, localize faults, and validate candidate patches through unit tests. The execution results, such as test outcomes, runtime errors, and logs, serve as feedback that guides subsequent repair steps, enabling agents to progressively improve solution quality.

Despite its utility, code execution is a resource-intensive operation that imposes significant overhead. On the one hand, it introduces considerable token costs, as agents must generate execution

ACM XXXX-XXXX/2026/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

commands, parse execution outputs, and reason over often verbose feedback. On the other hand, code execution incurs non-trivial latency, requiring agents to wait for compilation, runtime, and test results before proceeding. For example, executing a comprehensive test suite can take from several minutes to even hours. A third, less visible cost is per-repository environment setup: running a project's test suite requires a working runtime with the right language version and all dependencies installed, which in practice means maintaining a tested Docker image for every target repository and release. This is a recurring engineering tax that scales with deployment breadth. As code agents are increasingly deployed in real-world and large-scale settings, these costs become significant. Given the substantial resources consumed by code execution, a fundamental question arises: how important is code execution to the effectiveness of code agents in program repair?

However, the research community has only a limited understanding of the role of code execution in code agents for program repair. Prior work on such agents has primarily focused on model architectures [9, 33], prompting strategies [40, 45], search algorithms [46], or benchmark performance [14], often treating execution as a necessary but implicit component of the pipeline. While these studies acknowledge that execution-based feedback is important, there is a lack of systematic investigation into its quantitative impact. A closer work is Agentless [41], which avoids iterative execution by replacing the agent loop with a fixed localization-repair-validation pipeline. However, Agentless removes both the multi-turn loop and execution access at once, so its results cannot isolate execution's contribution. We take a more granular position: the agent loop is valuable, but iterative execution within it may not be.

As a result, the extent to which execution contributes to agent performance in program repair, and whether its benefits justify its cost, remains unclear. To fill this knowledge gap, we conduct the first empirical study that isolates code execution as a single controlled variable within the LLM agent loop for program repair. Unlike prior agentless [41] work, we hold the agent scaffold fixed (Claude Code [1] with Claude-Sonnet-4.5, Codex [29] with GPT-5.2-xhigh, and the open-source OpenCode [31] with Qwen2.5-Coder-32B [12]) and vary only execution access across four paradigms, yielding a controlled measurement of execution's marginal value. Specifically, we target execution behaviors that run code artifacts and produce runtime feedback, including test framework invocations (e.g., pytest and python -m unittest) and Python script execution (e.g., python xxx.py). Through controlled experiments and detailed analysis, we first investigate how execution is conducted by current code agents and then examine how much and why execution influences their effectiveness and efficiency. In the following sections, we introduce the design of our study and present the findings for each research question.

RQ1: How do code agents conduct code execution? Before investigating the effectiveness of execution, we first quantify how agents currently use execution capabilities to provide foundational context for subsequent analyses. Specifically, we analyze 7,745 publicly available agent traces from the SWE-bench leaderboard, covering four prominent execution-based agents (SWE-agent, OpenHands, LiveSWEAgent, and Mini-SWE-agent), twelve different LLMs (GPT-4, GPT-4o, GPT-5, GPT-5.2, Claude-3-Opus, Claude-3.5-Sonnet, Claude-4-Sonnet, Claude-Opus-4.5, Kimi-K2, Qwen3-480B, Gemini-3-Pro, and DeepSeek-V3.2), and two benchmark datasets (SWE-bench Lite and Verified). We examine the frequency, timing distribution, and outcomes of test executions conducted by these agents.

Findings. We observe substantial variation in execution behavior across agents and models. we analyze 7,745 public traces and observe that execution is used across all agent-model combinations (avg. 8.8 runs per task), with frequency ranges from 2 to 19 per task, and recent models tend to use more executions than older ones. For example, OpenHands with Claude-4-Sonnet averages 18.7 executions per task, while Mini-SWE-agent with GPT-5.2 averages only 2.0. Late-stage executions

(66–100% of conversation) consistently achieve higher success rates than early-stage ones across all configurations. For instance, OpenHands with Claude-3.5-Sonnet improves from 42% to 72%. The average success rate is 57.9%, suggesting that agents refine their understanding over time and execute more targeted tests as the repair progresses.

RQ2: What is the impact of code execution on code agents' performance? In this RQ, we perform controlled experiments to measure the impact of code execution. Specifically, we design four experimental settings with progressively increasing access to code execution, ranging from completely restricted to unrestricted access.We analyze the performance of three code agents: Claude Code (Claude Sonnet 4.5), Codex CLI (GPT-5.2-xhigh), and the open-source OpenCode (Qwen2.5-Coder-32B-Instruct). Due to budget constraints, we conduct experiments on two representative subsets of SWE-bench: the first 100 instances of SWE-bench Lite and the first 100 instances of SWE-bench Verified.

Findings. The resolve rate gap between restricted and unrestricted execution is small. For example, Claude Code achieves a 63% resolve rate without execution access, only 1 percentage point lower than the 64% achieved with unrestricted execution, while saving 56% of tokens and 48% of wall-clock time. Across all agent-dataset combinations the difference is not statistically significant ($p > 0.05$, McNemar's test). To rule out data leakage as an explanation, we replicate this on an open-source agent with Qwen2.5-Coder-32B, whose training cutoff predates SWE-bench Verified: OpenCode reaches 10% resolve rate under both Prohibited and Unrestricted while consuming 3× fewer tokens without execution. For the bugs studied, agents pay execution's cost even on instances where it confers no benefit.

RQ3: Under what conditions does code execution benefit code agents? Given the findings from RQ2, we investigate *why* execution does not consistently improve outcomes in order to understand when it is beneficial and when it is not. Specifically, we analyze instances with stable outcomes across modes (Pass→Pass and Fail→Fail) to characterize the conditions under which execution adds value.

Findings. We identify two factors behind the limited benefit on the studied bugs. First, reproduction execution provides little localization benefit: although 55% of Claude Code's successful cases use it, localization accuracy stays above 95% in both modes, and only 48.8% of reproduction executions produce actionable feedback. Second, execution feedback often does not correct errors: 54–66% of commercial-agent cases complete in a single edit regardless of execution access, and 81–100% of failed cases pass agent-conducted validation but fail the official SWE-bench evaluation. This is a gap between agent-chosen tests and ground-truth validation. Execution is not inherently unhelpful; its benefit is concentrated on specific instances. The open question is how an agent should decide *when* to invest in execution.

In summary, this paper makes the following contributions:

- The first empirical study that isolates code execution as a single variable within the agent loop for program repair: scaffold held fixed (Claude Code, Codex, and open-source OpenCode), only execution access varied across four paradigms. Where prior agentless work changes scaffold and execution at once, our design measures execution's marginal value alone. Covers 7,745 public traces plus 3,000 controlled repair attempts on 200 SWE-bench instances.
- Execution is not consistently beneficial: the Prohibited–Unrestricted resolve-rate gap is 1.25pp on commercial agents and ≈ 0pp on the open-source one (none significant, $p > 0.05$). We also expose a boundary regime: an open-source model with a 65K-token context (Qwen2.5-Coder-32B) does best with a single well-chosen execution (Quota-1) rather than Unrestricted. Execution should be treated as a resource with an explicit cost-benefit tradeoff, not a default capability.

- We release a reproducible framework[1] for limiting code execution of program-repair agents.

2 BACKGROUND

In this section, we describe the workflow of code agents and explain how they conduct code execution.

2.1 Code Agents

Code agents are LLM-based systems that autonomously perform software engineering tasks by combining reasoning with tool use [10, 37, 44]. Unlike simple code completion, these agents wrap an LLM in a control loop that persists state across multiple turns, enabling them to tackle complex, multi-step tasks such as bug fixing, feature implementation, and code refactoring.

A typical code agent workflow combines three core capabilities: *file exploration* (traversing directories and searching for relevant code), *code editing* (applying patches via diffs or block replacements), and *code execution* (running shell commands to invoke build systems, linters, and test suites). Representative agents include SWE-agent [44], OpenHands [38], and commercial CLIs such as Claude Code [1] and Codex [29].

For program repair tasks, agents typically follow an iterative loop [42, 44]:

$$\texttt{inspect code} \rightarrow \texttt{propose patch} \rightarrow \texttt{run tests} \rightarrow \texttt{revise}$$

This loop makes test execution the primary feedback channel: agents run tests to validate their patches and use test outputs to guide subsequent revisions. A notable alternative is Agentless [41], which uses a two-phase pipeline (localization followed by repair) without iterative execution, yet achieves competitive results on SWE-bench. This raises a key question: how much value does the execution-heavy agentic paradigm actually provide?

2.2 Code Execution by Code Agents

Code execution is the process by which agents run code artifacts and observe runtime feedback. The typical execution cycle proceeds as follows: the agent first *writes a command* (e.g., `pytest tests/test_foo.py`), the command is then *executed* in a shell environment with access to the project's dependencies, the agent *observes the results* (test outputs, error messages, and stack traces), and based on these results, decides whether to *revise the patch* or submit.

This execution loop incurs substantial costs across multiple dimensions. *Wall-clock time*: agents must wait for execution to complete, with test suites taking seconds to minutes and creating latency in the repair loop. *Token consumption*: execution outputs, including verbose test logs and stack traces, are fed back to the LLM, consuming context window capacity and increasing API costs. *Environment overhead*: each execution requires a properly configured environment with dependencies, databases, and external services. These costs compound when agents engage in trial-and-error behavior, repeatedly executing tests without making substantive progress. Recent work has begun addressing efficiency: Peng et al. [8] show that limiting interaction turns can reduce costs by 24–68% with minimal impact on solve rates. However, turn-level budgets treat all interactions equally, obscuring the distinction between cheap operations (reading files) and expensive ones (running tests). Our study focuses specifically on execution costs, providing a fine-grained analysis of when execution helps and when it merely adds overhead.

[1] https://anonymous.4open.science/r/Run_Less

3 EXPERIMENTAL SETUP

In this section, we introduce the experimental setup of this study, including the settings for code execution, benchmark, agents, evaluation metrics, and implementation details. Our study is guided by three research questions:

- **RQ1**: How do code agents conduct code execution?
- **RQ2**: What is the impact of code execution on code agents' performance?
- **RQ3**: Under what conditions does code execution benefit code agents, and why is its impact not consistently positive?

3.1 Experimental Settings for Code Execution

In this study, we focus on *code execution*, which refers to operations that run code artifacts and produce runtime feedback. This includes test framework invocations (`pytest`, `python -m unittest`, `python manage.py test`, `tox`, `nosetests`) and Python script execution (`python xxx.py`). Other commands such as `ls`, `cat`, `grep`, and `find` are exploratory in nature and are not restricted.

We study four execution paradigms that form a spectrum from no execution to unlimited access. Prohibited mode restricts access to project-specific runtime environments: the agent is instructed via prompt to avoid running test frameworks or project-specific scripts, and project dependencies (e.g., Django, Flask, Sympy) are not installed. Since this is a soft constraint via prompting, agents occasionally still attempt to execute tests or scripts; however, these attempts fail due to missing dependencies and do not provide useful runtime feedback. A basic Python interpreter remains available for simple commands such as `python -c "print(1+1)"`. This design reflects a realistic "read-only analysis" scenario where developers reason about code without setting up the full project environment.

> **Prompt:** `Do not run pytest, unittest, or project test scripts. Project dependencies are not installed.`

Quota-Limited mode allows execution with a point budget, where the agent self-estimates costs before each run and test-framework invocations are treated as expensive operations. We evaluate two budget levels: K=1 (minimal execution) and K=3 (moderate execution), yielding five experimental conditions from four paradigms.

> **Prompt:** `You have a budget of K test run(s). Unused budget is wasted opportunity!`
> **Cost Table:** `pytest`/`unittest`/`python manage.py test` = 1.0 point; `python script.py` = 0.3 point.

Budget-Guided mode permits unrestricted execution but prompts the agent to consider whether each run is worth its cost, testing whether cost awareness alone can reduce unnecessary execution.

> **Prompt:** `You CAN run tests and scripts, but each execution has a cost. Goal: Fix the bug correctly while being mindful of execution costs.`

At the other extreme, Unrestricted mode allows unlimited execution, enabling the typical "generate–run–revise" loop that characterizes most current agents. No execution constraints are specified in the prompt.

This configuration yields 600 unique agent-instance combinations (200 instances × 3 agents), totaling 3,000 end-to-end repair attempts across five execution modes.

3.2 Benchmark

We study execution behavior in the context of automated program repair on SWE-bench [15], a benchmark of real-world GitHub repositories and their issues. Each instance provides a repository snapshot, a problem statement, and a test-based evaluation protocol. Due to budget constraints, we use two commonly used variants: SWE-bench Lite (300 instances) and SWE-bench Verified (500 instances), selecting the first 100 instances from each dataset in their canonical ordering. This yields 200 instances spanning diverse repositories including Django, Flask, Requests, and Sympy. The task requires an agent to understand the bug from the problem statement, locate relevant code in the repository, generate a patch, and optionally validate the fix via test execution. The official SWE-bench harness evaluates predictions by applying the patch in a clean container and running instance-specific tests.

3.3 Models and Agents

Our study adopts a two-stage design. In the first stage (RQ1), we analyze execution behavior across a broad range of publicly available agent traces from the SWE-bench leaderboard, covering four prominent agents (SWE-agent, OpenHands, LiveSWEAgent, and Mini-SWE-agent) and twelve LLMs (GPT-4, GPT-4o, GPT-5, GPT-5.2, Claude-3-Opus, Claude-3.5-Sonnet, Claude-4-Sonnet, Claude-Opus-4.5, Kimi-K2, Qwen3-480B, Gemini-3-Pro, and DeepSeek-V3.2). In the second stage (RQ2–RQ3), we conduct controlled experiments with three agents: Claude Code [1] (Claude Sonnet 4.5), Codex [29] (GPT-5.2-xhigh), and the open-source OpenCode [31] with Qwen2.5-Coder-32B-Instruct [12] served via vLLM [16].[2] All agents operate in the SWE-bench containerized environment. Claude Code and Codex represent state-of-the-art commercial offerings; OpenCode with Qwen2.5-Coder-32B provides an open-source, open-weight model baseline that mitigates data contamination concerns (see Section 5).

3.4 Metrics

We record interaction traces and compute several metrics to characterize agent behavior and outcomes. For execution behavior, we measure *execution frequency* as the average number of test executions per task, *timing distribution* as the percentage of executions occurring in each conversation stage (Early: 0–33%, Middle: 33–66%, Late: 66–100%), and *execution outcome* as the success or failure rate of test executions. For effectiveness and cost, the primary outcome is *resolve rate*, which indicates the proportion of instances where the generated patch passes the official SWE-bench evaluation. We measure *token consumption* (input and output tokens) to reflect API cost, and *wall-clock time* to capture end-to-end runtime. To ensure fair time comparisons, all execution modes for the same instance run concurrently on identical hardware. For understanding execution impact, we measure *localization accuracy* using Hit (at least one edited file matches ground truth) and Recall (proportion of ground truth files edited), *single-edit ratio* as the percentage of instances with only one code edit and no subsequent modifications, *post-execution modification ratio* as the percentage of instances where code is modified after test execution, and *actionable feedback ratio* as the percentage of reproduction executions that provide useful localization information.

[2]Precise CLI and model versions for reproducibility: Claude Code v1.0.16 (Anthropic `claude-sonnet-4-5`); Codex v0.1.2025062301 (OpenAI `gpt-5.2` with reasoning effort `xhigh`); OpenCode v1.4.0 with tensor-parallel vLLM serving `Qwen/Qwen2.5-Coder-32B-Instruct`. For brevity, unless otherwise specified, we refer to these fixed agent–model configurations as *Claude Code*, *Codex*, and *OpenCode*, respectively, in the following.

3.5 Implementation Details

Experimental Scale. Our evaluation comprises **3,000 end-to-end agent runs**: 200 SWE-bench instances (100 from Lite, 100 from Verified) × 3 agents (Claude Code, Codex, OpenCode) × 5 execution modes (Prohibited, Quota-Limited K=1, Quota-Limited K=3, Budget-Guided, Unrestricted). Each run involves a complete repair attempt: checking out the repository, reading the issue, generating and applying patches, and running the official SWE-bench evaluation harness. Critically, all execution modes are evaluated on *exactly the same* 200 instances; this paired design controls for instance-level difficulty variation, enabling reliable statistical analysis through paired comparisons.

We access all agents via their command-line interfaces: Claude Code via `claude -p`, Codex via `codex exec`, and OpenCode via `opencode`. All modes share the same prompt structure, consisting of task description, repository information, problem statement, execution constraint, and output format. Only the execution policy differs across modes, isolating execution access as the primary experimental variable. The point budget serves as a guideline rather than a hard limit; we report outcomes by assigned budget regardless of compliance, allowing us to study how agents respond to cost awareness even when not strictly enforced. The final patch is extracted via `git diff` and evaluated using the official SWE-bench harness, which applies the patch in a clean container and runs the instance-specific evaluation script.

4 EVALUATION

In this section, we present the results of our empirical study, organized around three research questions that examine how agents use execution, the impact on repair effectiveness and cost, and why execution feedback has limited benefits.

4.1 RQ1: How Do Code Agents Conduct Code Execution?

Before examining the effectiveness of execution, we first quantify how frequently execution is conducted in state-of-the-art coding agents. We analyzed 7,745 publicly available agent traces (i.e., the complete logs of agent-environment interactions during repair attempts) from the SWE-bench leaderboard, covering four prominent execution-based agents (SWE-agent, OpenHands, LiveSWEAgent, and Mini-SWE-agent), twelve different LLMs (GPT-4, GPT-4o, GPT-5, GPT-5.2, Claude-3-Opus, Claude-3.5-Sonnet, Claude-4-Sonnet, Claude-Opus-4.5, Kimi-K2, Qwen3-480B, Gemini-3-Pro, and DeepSeek-V3.2), and two benchmark datasets (SWE-bench Lite and Verified).

For each code execution, we record two dimensions. First, we record its *timing*, defined as the normalized position in the agent conversation where 0.0 represents the start and 1.0 represents the end. We categorize timing into three stages: Early (0–33%), Middle (33–66%), and Late (66–100%). Second, we record its *outcome*, classified as either Success or Failure. A successful execution is one where pytest reports "passed" without any "FAILED" messages, unittest outputs "OK", or the command returns exit code 0. A failed execution includes test assertion failures, Python exceptions, or non-zero exit codes. Table 1 presents the test execution analysis across all agent-model combinations. Our observations are as follows:

4.1.1 Statistical Analysis. First, code execution is widely adopted across all agent-model combinations. On average, agents perform 8.8 test executions per task. For models such as Claude-Opus-4.5 in LiveSWEAgent, the execution count can reach 15.9 per instance. Second, execution behavior varies substantially across agents and models. Recent models, with the exception of GPT-5.2, tend to use more executions than older models. For example, OpenHands with Claude-4-Sonnet averages 18.7 executions per task, while Mini-SWE-agent with GPT-5.2 averages only 2.0, representing a 9× difference.

Table 1. Test execution analysis on 7,745 SWE-bench public traces. #Traces is the number of leaderboard submissions for that (agent, model) pair. *Exec/Task* is the mean number of test-framework invocations (`pytest`, `python -m unittest`, `python manage.py test`, `tox`, `nosetests`) per instance, averaged over all traces for that pair. Early/Middle/Late = percentage of executions occurring in each conversation stage (0–33%, 33–66%, 66–100% of turns before the final patch). Success/Failure = percentage of test executions whose exit status indicates all tests passed vs. at least one test failed or errored.

		Stats		Stage			Outcome	
Agent	Model	#Traces	Exec/Task	%Early	%Middle	%Late	%Success	%Failure
SWE-bench Lite								
SWE-agent	GPT-4	254	3.2	42.4%	28.1%	29.6%	46.1%	53.9%
SWE-agent	Claude-3.5-Sonnet	259	6.3	23.5%	33.0%	43.4%	38.7%	61.3%
SWE-agent	GPT-4o	259	7.4	32.3%	33.4%	34.2%	36.7%	63.3%
OpenHands	Claude-3.5-Sonnet	291	6.0	19.8%	33.7%	46.4%	57.8%	42.2%
SWE-bench Verified								
SWE-agent	GPT-4	400	3.3	42.7%	27.9%	29.4%	45.8%	54.2%
SWE-agent	Claude-3-Opus	366	3.7	35.8%	31.2%	32.9%	45.3%	54.7%
SWE-agent	Claude-3.5-Sonnet	434	6.9	24.8%	34.5%	40.8%	40.3%	59.7%
SWE-agent	GPT-4o	425	7.5	31.2%	33.2%	35.6%	30.4%	69.6%
OpenHands	Claude-3.5-Sonnet	488	6.7	22.7%	33.4%	43.9%	58.6%	41.4%
OpenHands	Claude-4-Sonnet	500	18.7	15.4%	30.9%	53.7%	70.4%	29.6%
OpenHands	Kimi-K2	500	15.5	11.7%	34.9%	53.3%	66.4%	33.6%
OpenHands	Qwen3-480B	500	18.7	12.4%	31.7%	55.9%	70.3%	29.7%
OpenHands	GPT-5	479	4.1	11.5%	38.6%	49.9%	68.1%	31.9%
LiveSWEAgent	Gemini-3-Pro	500	12.6	28.7%	38.2%	33.2%	77.8%	22.2%
LiveSWEAgent	Claude-Opus-4.5	500	15.9	23.6%	40.1%	36.3%	79.3%	20.7%
Mini-SWE-agent	Claude-Opus-4.5	495	10.5	12.6%	38.3%	49.1%	78.7%	21.3%
Mini-SWE-agent	Gemini-3-Pro	495	8.6	23.1%	36.6%	40.3%	71.8%	28.2%
Mini-SWE-agent	DeepSeek-V3.2	485	10.5	14.2%	34.9%	50.9%	59.9%	40.1%
Mini-SWE-agent	GPT-5.2	115	2.0	1.8%	21.9%	76.3%	58.0%	42.0%

4.1.2 Timing Analysis. Late-stage execution (66–100% of conversation) is the most common pattern across configurations. For example, OpenHands with Qwen3-480B concentrates 55.9% of executions in the late stage. However, older models like GPT-4 show a different pattern, with more executions in the early stage. SWE-agent with GPT-4 performs 42.4% of executions in the early stage, compared to only 29.6% in the late stage. Notably, some state-of-the-art models show minimal early-stage execution. For example, GPT-5.2 in Mini-SWE-agent has only 1.8% early-stage executions, and Kimi-K2 in OpenHands has 11.7%. This pattern may suggest that these models understand the issue well from the problem description alone, reducing the need for reproduction before editing.

4.1.3 Outcome Analysis. The overall execution success rate is moderately high, with an average of 57.9% across all configurations. Success rates range from 30.4% for SWE-agent with GPT-4o to 79.3% for LiveSWEAgent with Claude-Opus-4.5. Some agent-model combinations achieve particularly high success rates. For example, LiveSWEAgent with Claude-Opus-4.5 and Mini-SWE-agent with Claude-Opus-4.5 both exceed 78%. Among the failures, TestFailure (assertion failures), TestError (collection or setup errors), and Python exceptions (such as ModuleNotFoundError, AttributeError, and TypeError) are the most common types. Due to space constraints, the complete failure categorization is available in our artifacts.

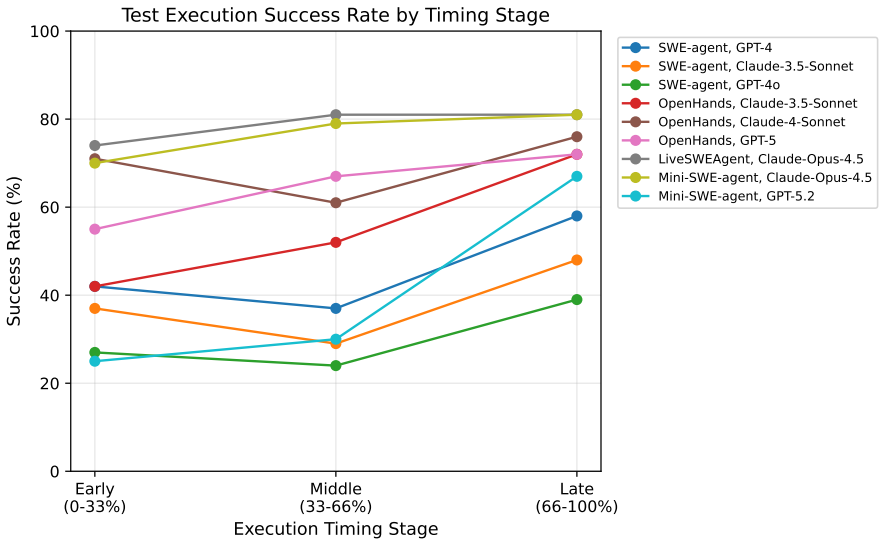

Fig. 1. Test execution success rate by timing stage. Late-stage executions (66–100% of conversation) consistently achieve higher success rates than early-stage ones across all agent-model combinations.

As shown in Figure 1, tests executed in late stages (66–100% of conversation) consistently achieve higher success rates than early-stage tests across all configurations. For example, OpenHands with Claude-3.5-Sonnet improves from 42% (early) to 72% (late), and Mini-SWE-agent with GPT-5.2 improves from 25% to 67%. This pattern suggests that agents refine their understanding over time and execute more targeted, better-formed tests as the repair progresses.

Takeaway for RQ1:

- Code execution is commonly used across all analyzed agent-model combinations, with an average of 8.8 test runs per task.
- Across 7,745 traces spanning 4 agents and 12 models, execution behavior varies substantially: frequency ranges from 2 to 19 per task, and timing patterns differ by agent architecture.
- The average success rate is 57.9%, with late-stage executions consistently achieving higher success rates than early-stage ones (e.g., 42% → 72% for OpenHands with Claude-3.5-Sonnet).

4.2 RQ2: Effectiveness and Cost Analysis

In this RQ, we systematically evaluate the impact of execution access on repair effectiveness and cost. Specifically, we evaluate three agents (Claude Code, Codex, and the open-source OpenCode with Qwen2.5-Coder-32B) on two benchmarks (SWE-bench Lite and Verified) under four execution paradigms (Prohibited, Quota-Limited, Budget-Guided, and Unrestricted), realised as five configurations (Quota-Limited is instantiated at K=1 and K=3). We measure resolve rate, defined as the percentage of instances where the generated patch passes all tests in the official SWE-bench evaluation harness, and record token consumption and wall-clock time following the methodology outlined in Section 3. Additionally, we assess statistical significance between

resolve rates using McNemar's test [27] and equivalence testing (Two One-Sided Tests, TOST [34], with equivalence margin $\delta = 5$ percentage points, or pp). The results are presented in Table 2, where we mark with † cases where Prohibited performance is within 3pp of Unrestricted. The Prohibited–Unrestricted gap stays within ±5pp across the six headline cells and averages 1.25pp on commercial agents. We present the effectiveness result and its data-leakage check first (Tables 2 and 3), then verify that it does not depend on whether agents fully obey the prompt-level constraint (Tables 4 and 5), and end with cost (Tables 6 and 7).

4.2.1 Effectiveness Analysis. Table 2 presents the complete resolve rate comparison across all agents, benchmarks, and execution modes. The central observation is that the gap between Prohibited and Unrestricted is remarkably small: averaged across the four commercial-agent cells the difference is only 1.25pp, and averaged across all six cells (including OpenCode) it is −0.83pp.

Table 2. Resolve rate (%) across all execution modes. Bold indicates best performance per row. †: Prohibited performance is within 3pp of Unrestricted. None of the differences are statistically significant ($p > 0.05$, McNemar's test).

Agent	Benchmark	Prohibited	Quota-1	Quota-3	Budget-Guided	Unrestricted
Claude Code	Lite	63.0†	61.0	62.0	63.0	**64.0**
	Verified	64.0†	64.0	65.0	**67.0**	**67.0**
Codex	Lite	**74.0**†	68.0	69.0	71.0	73.0
	Verified	73.0†	72.0	73.0	71.0	**75.0**
OpenCode	Lite	7.0	**14.0**	7.0	9.0	6.0
	Verified	13.0	**17.0**	11.0	13.0	14.0

Several patterns emerge from these results. First, we do not observe a monotonic relationship between execution access and repair success. For Codex on Lite, Prohibited achieves the highest resolve rate (74.0%) among all modes, outperforming Unrestricted (73%) by 1.0 percentage point. Second, the Quota-Limited modes often perform *worse* than Prohibited. For example, Codex on Lite achieves 74.0% under Prohibited but drops to 68.0% under Quota-1 and 69.0% under Quota-3, a decrease of up to 6 percentage points. This suggests that partial execution access may be counterproductive, possibly because limited feedback is insufficient for effective iteration; we revisit this observation in RQ3. Third, the open-source OpenCode agent (Qwen2.5-Coder-32B) exhibits the same equivalence pattern on a non-commercial stack: despite much lower absolute resolve rates—expected given the 32B parameter count versus frontier models—the Prohibited vs. Unrestricted gap is within ±1pp on both benchmarks (Lite: 7.0% vs. 6.0%; Verified: 13.0% vs. 14.0%), averaging 0pp. Since Qwen2.5-Coder-32B-Instruct's training cutoff predates Verified, this extends the equivalence to a regime with limited exposure to SWE-bench solutions (see Section 5.2 for a dedicated training-data discussion). Overall, these results indicate that the impact of code execution on repair effectiveness is limited.

OpenCode does, however, exhibit one boundary behaviour worth flagging: its highest-resolve mode is Quota-1 (Lite 14.0%, Verified 17.0%; paired-bootstrap 90% CI [+5.9, +15.3]pp vs. Prohibited on Lite), driven by the non-empty-patch rate collapsing from 74/100 (Lite) and 76/100 (Verified) under Quota-1 to 50/100 and 56/100 under Unrestricted as test output crowds the 65K context. We treat this as a small-model boundary effect.

To rigorously assess these differences, Table 3 reports both 95% Wilson confidence intervals (preferred for proportions with small sample sizes) and paired McNemar tests for Prohibited vs. Unrestricted. All six agent-benchmark combinations show substantially overlapping Wilson

intervals and non-significant McNemar results ($p > 0.05$), confirming that observed differences are within statistical noise. The discordant pairs (b = Prohibited succeeds / Unrestricted fails; c = reverse) are few in number and near-symmetric across cells (5:6, 6:9, 4:3, 1:3), consistent with agent stochasticity rather than a systematic execution benefit. Formal TOST equivalence with $\delta = 5$pp holds for Codex/Verified (90% CI: [−0.8pp, +4.8pp]); the remaining five cells keep point estimates within the ±5pp band but do not meet formal equivalence at $n = 100$, a power-vs.-sample-size limitation we revisit in Section 5.2. In summary, the statistical results confirm that, for the bugs studied, execution access does not produce a reliably positive effect on repair outcomes.

Table 3. Resolve rates with 95% Wilson CIs and paired McNemar test (Prohibited vs. Unrestricted). b = Prohib. succeeds / Unrestr. fails; c = reverse; all $p > 0.05$.

Agent	Bench.	Prohib. (CI)	Unrestr. (CI)	b	c	p
Claude Code	Lite	63.0 [53.2,71.8]	64.0 [54.2,72.7]	5	6	1.000
Claude Code	Ver.	64.0 [54.2,72.7]	67.0 [57.3,75.4]	6	9	0.607
Codex	Lite	74.0 [64.5,81.8]	73.0 [63.6,80.7]	4	3	1.000
Codex	Ver.	73.0 [63.6,80.7]	75.0 [65.7,82.5]	1	3	0.625
OpenCode	Lite	7.0 [3.4,13.7]	6.0 [2.8,12.5]	4	3	1.000
OpenCode	Ver.	13.0 [7.8,21.0]	14.0 [8.5,22.1]	4	5	1.000

As a sanity check on whether Prohibited's strong performance reflects *verbatim* memorisation rather than reasoning, we compared the patches produced in the two modes: if memorisation were the dominant mechanism, patches should largely coincide. Using difflib.SequenceMatcher on normalised patches, we find the identical-patch rate is 24% on Claude Code, 1% on Codex, and 14% on OpenCode, with mostly same-file / different-code pairs (Claude Code 15%, Codex 27%, OpenCode 24%); average patch similarity is 42–60%. This is limited but non-trivial overlap, and it rules out *rote recitation* as the dominant mechanism. The OpenCode result is particularly informative: Qwen2.5-Coder-32B-Instruct's training cutoff predates Verified, yet the Prohibited–Unrestricted equivalence still holds on that benchmark, further constraining the role of training-data overlap.

4.2.2 Hard-Constraint Verification. Because Prohibited is enforced at the prompt level, agents may occasionally still attempt to run tests(as shown in Table 4). This raises a concern that whether these unintended executions explain Prohibited's strong performance. We address this in two steps: (i) measure what each configuration actually delivers, and show that the unintended executions in Prohibited do not carry enough usable feedback to explain its resolve rate, and (ii) re-verify on the hard-constraint.

(i) What each configuration actually delivers. Table 4 splits each cell into four per-instance averages: *Attempted* test-framework invocations, *Env-Err* attempts blocked by missing modules or dependency errors before reaching the test stage, *Completed* attempts that reach the test stage and produce non-empty output, and *Actionable* attempts that produce a concrete pass/fail signal the agent could use to guide the next edit. The four columns separate intent (*Attempted*) from usable feedback (*Actionable*), which diverge sharply under Prohibited: missing dependencies turn most attempts into dead ends. Under Prohibited, Codex essentially obeys the prompt outright (0.00–0.01 attempts per instance); OpenCode attempts 0.77 on Lite but drops to zero on Verified; Claude Code is the loudest violator, attempting 0.76–0.79 times per instance. However, environment errors absorb about a quarter of them before any test output is produced, leaving about 0.55–0.59 completed runs per instance; of those, only 0.36–0.39 yield a concrete pass/fail signal the agent could act on. The effective rate of usable unintended feedback is thus well under half the per-instance

quota of Quota-1 and roughly an eighth of the Unrestricted rate (5.30 on Lite, 4.98 on Verified). The same pattern is even more pronounced on OpenCode (0.14 and 0.00 actionable per instance) and essentially zero on Codex.

Table 4. Per-mode compliance: each cell reports *Attempted/Env-Err/Completed/Actionable* as per-instance averages over 100 instances. *Attempted* = test-framework invocations launched; *Env-Err* = blocked by missing modules or dependency errors; *Completed* = reached the test stage; *Actionable* = produced a pass/fail signal (subset of Completed).

Agent	Bench.	Prohib.	Quota-1	Quota-3	B.-Guided	Unrestr.
Claude Code	Lite	0.79/0.22/0.55/0.39	6.39/1.82/4.52/2.69	8.44/2.10/6.25/3.91	8.03/1.81/6.17/3.96	9.52/1.97/7.41/5.30
	Verified	0.76/0.17/0.59/0.36	6.11/1.53/4.52/2.90	7.34/1.57/5.73/3.25	6.78/1.45/5.30/3.65	8.60/1.52/7.00/4.98
Codex	Lite	0.00/0.00/0.00/0.00	1.24/0.55/0.54/0.47	2.65/0.48/1.90/1.72	2.43/0.52/1.58/1.41	3.43/0.54/2.63/2.34
	Verified	0.01/0.00/0.01/0.00	1.19/0.43/0.59/0.50	2.40/0.30/1.93/1.69	2.09/0.38/1.44/1.25	3.19/0.30/2.64/2.29
OpenCode	Lite	0.77/0.06/0.71/0.14	12.93/0.58/12.19/5.49	12.07/1.35/10.54/3.01	11.30/0.73/10.33/3.31	10.66/0.91/9.51/2.22
	Verified	0.00/0.00/0.00/0.00	11.01/0.55/10.35/4.10	9.50/0.91/8.55/1.41	10.42/0.90/9.43/2.86	11.47/0.66/10.74/2.99

(ii) Hard-constraint re-verification. To further demonstrate our hypothesis, we therefore re-run the comparison under two stricter notions of "really Prohibited" to verify the headline directly. The *zero-execution subset* keeps only the instances where the agent made no test-framework attempt at all in Prohibited, which matches the behaviour of an environment-level sandbox on that slice of the data. The Furthermore, for Claude Code on Verified, we execute the benchmark with hard-constraint level (`claude -p --disallowedTools Bash(pytest*)` and other executing commands), while file-editing and read-only shell tools remain available.

Table 5. Resolve rates on compliant subsets. *Zero-execution* = instances where the agent made no test-framework attempts in Prohibited. *Sandboxed* = test-execution sub-commands denylisted at the CLI level. Gap = Prohibited − Unrestricted in percentage points. Env-error-free numbers are summarised in the footnote to the preceding paragraph.

Benchmark	Agent	N	Prohibited	Unrestricted	Gap
Zero-execution subset (attempted = 0 in Prohibited)					
Lite	Claude Code	82	64.6%	65.9%	−1.2
Lite	Codex	100	74.0%	73.0%	+1.0
Lite	OpenCode	93	7.5%	6.5%	+1.1
Verified	Claude Code	84	61.9%	66.7%	−4.8
Verified	Codex	99	73.7%	75.8%	−2.0
Verified	OpenCode	100	13.0%	14.0%	−1.0
Hard-Constraint Prohibited					
Verified	Claude Code	100	63.0%	67.0%	−4.0

As shown in table 5, on the zero-execution subset, every cell stays within the margin (worst: Claude Code Verified at −4.8pp, N=84), which matches the full-sample comparison. The hard-constraint mode resolves 63/100 vs. 67/100 (−4.0pp, within margin), saving 62% tokens and 54% wall-clock (Table 7). Across all three views (full sample, zero-execution subset, hard-constraint), the Prohibited–Unrestricted gap stays within the equivalence margin, demonstrating that the headline is not a result of agents silently breaking the soft constraint.

Table 6. Resource consumption across all execution modes. In/Out = input/output tokens (in thousands). Time = wall-clock seconds. Bold indicates values lower than Unrestricted.

Agent	Benchmark	Metric	Prohibited	Quota-1	Quota-3	Budget-Guided	Unrestricted
Claude Code	Lite	In	**52K**	**79K**	**105K**	**109K**	121K
		Out	**17K**	**26K**	**34K**	**35K**	37K
		Time	**531**	**815**	**876**	**912**	1028
	Verified	In	**48K**	**94K**	**95K**	**101K**	128K
		Out	**15K**	**31K**	**31K**	**33K**	39K
		Time	**573**	**907**	**932**	**990**	1234
Codex	Lite	In	**327K**	**300K**	419K	399K	378K
		Out	**82K**	**75K**	105K	100K	95K
		Time	**570**	**598**	632	618	618
	Verified	In	**431K**	**328K**	463K	**393K**	435K
		Out	**108K**	**82K**	116K	**98K**	109K
		Time	724	**682**	**686**	**682**	723
OpenCode	Lite	In	**187K**	347K	334K	427K	254K
		Out	**9K**	17K	16K	19K	12K
		Time	**585**	**919**	1051	1052	1033
	Verified	In	**105K**	323K	332K	376K	322K
		Out	**4K**	**14K**	19K	18K	15K
		Time	**402**	**1017**	**1105**	**1118**	1232

4.2.3 Cost Analysis. Given that effectiveness differences are minimal, we now examine the cost implications. Table 6 presents token consumption and wall-clock time across all configurations.

Limiting code execution can substantially reduce token consumption. As shown in Table 6, all restricted modes consume fewer tokens than Unrestricted for Claude Code, with savings ranging from 10% (Budget-Guided) to 56% (Prohibited). For Codex, the pattern is more nuanced: Quota-1 achieves the largest savings (21% on Lite, 25% on Verified), while Quota-3 and Budget-Guided sometimes exceed Unrestricted. Moreover, the optimal cost-saving mode differs by agent. For Claude Code, Prohibited delivers the best cost-effectiveness, reducing tokens by 56–62% with only a 1–3pp resolve rate difference. For Codex, Quota-1 is optimal, saving 21–25% of tokens while maintaining comparable resolve rates.

The three agents exhibit notably different cost profiles. Claude Code's token consumption increases by 129–163% from Prohibited to Unrestricted, Codex's increases by only 0.8–15.5%, while OpenCode's increases by 36–208%. This difference stems from their baseline resource usage: Claude Code consumes approximately 65K tokens in Prohibited mode, Codex consumes approximately 470K tokens, and OpenCode consumes approximately 150K tokens. For Claude Code and OpenCode, execution feedback accumulates in the context window, causing rapid token growth; for Codex, the marginal impact of execution feedback is small relative to its already large context. OpenCode shows the most extreme cost explosion on Verified (3.1× tokens, 3.1× time from Prohibited to Unrestricted) for only a +1pp resolve-rate gain, while paying a steeper price still for Budget-Guided (3.6× tokens, 2.8× time) at the same resolve rate. Quota-1 on OpenCode is intermediate in cost (3.1× tokens, 2.5× time over Prohibited) but delivers the highest resolve rate of any mode on both benchmarks. This points to a sweet spot for relatively weak models which are context-constrained: the prompt-level execution cap forces an edit-centric trajectory while preserving enough execution budget to verify candidate patches.

The same pattern holds for wall-clock time, where execution restrictions translate most clearly into runtime savings on Claude Code: Prohibited mode completes tasks in 531–573 seconds, compared to 1,028–1,234 seconds for Unrestricted, a reduction of 48–54%. For Codex, the time savings are more modest: Quota-1 saves 3–6% compared to Unrestricted, while other modes show minimal differences. This pattern mirrors the token consumption results, where Claude Code benefits more from execution restrictions than Codex.

Combining effectiveness and cost, Table 7 characterize the tradeoff across all agent-benchmark combinations. The key finding is that restricting execution achieves a favorable cost-effectiveness tradeoff: agents sacrifice minimal resolve rate while significantly reducing resource consumption. For Claude Code, Prohibited saves 56–62% of tokens and 48–54% of wall-clock time while sacrificing only 1–3 percentage points in resolve rate. For Codex, Quota-1 provides the best tradeoff, saving 21–25% of tokens compared to Unrestricted while achieving similar resolve rates (68% vs 73% on Lite, 72% vs 75% on Verified). Notably, the cost-effectiveness differs substantially between agents: Claude Code benefits significantly from execution restrictions (56–62% token savings), while Codex shows smaller savings (0.8–13%).

Table 7. Cost-effectiveness comparison: Prohibited vs. Unrestricted. Negative Δ values indicate savings.

		Prohibited		Unrestricted				
Agent	Benchmark	In	Out	In	Out	ΔResolve	ΔTokens	ΔTime
Claude Code	Lite	52K	17K	121K	37K	−1.0%	−56.4%	−48.4%
Claude Code	Verified	48K	15K	128K	39K	−3.0%	−62.3%	−53.6%
Codex	Lite	327K	82K	378K	95K	+1.0%	−13.5%	−7.8%
Codex	Verified	431K	108K	435K	109K	−2.0%	−0.9%	+0.1%
OpenCode	Lite	187K	9K	254K	12K	+1.0%	−26.3%	−43.4%
OpenCode	Verified	105K	4K	322K	15K	−1.0%	−67.7%	−67.4%

Takeaway for RQ2:

- The resolve rate difference between Prohibited and Unrestricted is only 1.25 percentage points on average, which is not statistically significant ($p > 0.05$). Execution helps on some instances and hurts on others in roughly equal proportion, indicating that its benefit is not uniform.
- Restricting execution significantly reduces cost: Prohibited saves 56–62% of tokens and 48–54% of time for Claude Code, while Quota-1 saves 21–25% of tokens for Codex.
- Cross-mode patch diversity (76–99% differ) suggests that models adapt solutions to context rather than reciting memorized patches.

4.3 RQ3: When and Why Execution Has Limited Impact

Given the findings from RQ2 that the resolve rate gap between restricted and unrestricted execution is small, we investigate the conditions under which execution adds value. Specifically, we analyze the 600 agent-instance pairs (3 agents × 200 instances) from RQ2, comparing outcomes between Prohibited and Unrestricted modes. We classify instances into four categories based on their outcomes: Pass→Pass indicates that both modes succeed, Fail→Fail indicates that both modes fail, Pass→Fail indicates that Prohibited succeeds but Unrestricted fails, and Fail→Pass indicates the reverse. Table 8 reports outcome-transition counts across the three agents. Overall 547 of 600 cells are stable (269 P→P + 278 F→F), so their outcome is unaffected by whether execution is available;

the remaining 53 split 24 P→F vs. 29 F→P, a near-symmetric pattern consistent with execution helping and hurting in roughly equal proportion (net benefit 5 cells). Our investigation therefore focuses on the two stable groups, Pass→Pass and Fail→Fail, whose trajectories we manually inspect by tracing the decision-making process and code execution patterns; we then stress-test the aggregate conclusion by stratifying over gold-patch complexity.

Table 8. Outcome transitions between PROHIBITED and UNRESTRICTED, summed over Lite+Verified (n=200 per agent). P→F = PROHIBITED succeeds / UNRESTRICTED fails; F→P = reverse.

Agent	P→P	F→F	P→F	F→P	Total
Claude Code	116	58	11	15	200
Codex	141	48	5	6	200
OpenCode	12	172	8	8	200
Total	**269**	**278**	**24**	**29**	**600**

From that inspection we identify two reasons why code execution often fails to alter the final outcome: (1) reproduction execution provides limited localization benefit, and (2) the feedback from code execution is insufficient for agents to correct errors. We close the section with a complexity stratification (§4.3.3) that checks whether this aggregate picture hides a benefit concentrated on harder bugs.

4.3.1 Reason 1: Reproduction Execution Provides Limited Localization Benefit. We first examine whether code execution helps agents locate the correct files to modify. Using the same definition of test execution from RQ1 (pytest, unittest, tox, nosetests, or `python xxx.py` commands), we define *reproduction execution* as test executions occurring *before* the first source code edit (excluding test file edits), used to understand or locate the bug. We compare file localization accuracy between UNRESTRICTED and PROHIBITED modes, where access to code execution is the only difference. Accuracy is measured by two metrics: *Hit* (at least one edited file matches ground truth) and *Recall* (proportion of ground truth files edited).

Table 9 shows that file localization accuracy is nearly identical across execution modes for the commercial agents; execution does not improve their ability to locate buggy files. For Pass→Pass cases, both commercial agents achieve over 95% hit rate and over 93% recall in both modes; OpenCode matches this at the extreme (100% hit, 95.8% recall in both modes on the 12 P→P instances), showing that whenever Qwen2.5-Coder-32B succeeds it does so by correctly identifying the buggy file directly from source reading. For Fail→Fail cases, commercial-agent localization accuracy is lower (85–91% hit rate) but the mode-wise difference remains within 2 percentage points. OpenCode's Fail→Fail subset shows an inversion—PROHIBITED localises at 54.1% while UNRESTRICTED drops to 32.6%. On the P→P subset both modes already saturate at 100% hit, leaving no headroom for a negative effect to show; the inversion therefore surfaces only in F→F, where localization still has room to drop. This is consistent with execution feedback crowding Qwen2.5-Coder-32B's 65K context and degrading source-reading once the model starts iterating, and we characterise it as a small-model boundary effect in Section 5.1. Across all three agents, these results indicate that agents already achieve good localization performance without execution, and that execution access does not substantially improve localization on the subset where it matters most.

We further assess the helpfulness of execution results for file localization. To measure helpfulness, we classify execution feedback as *actionable* or *non-actionable*. Actionable feedback contains useful information for localization, such as file paths, stack traces, or line numbers that point to the buggy code. Non-actionable feedback includes two categories: (1) environment errors that prevent

Table 9. File localization accuracy. Cells are *Hit/Recall* for UNRESTRICTED on the left and PROHIBITED on the right of the slash. Hit = at least one edited file matches ground truth; Recall = proportion of ground truth files edited.

Agent	Pass→Pass (Unrestr. / Prohib.)		Fail→Fail (Unrestr. / Prohib.)	
Claude Code	97.4 / 96.6 %	98.3 / 97.4 %	85.7 / 79.0 %	84.5 / 78.7 %
Codex	98.6 / 96.4 %	95.8 / 93.6 %	90.9 / 83.9 %	89.1 / 82.1 %
OpenCode	100.0 / 95.8 %	100.0 / 95.8 %	32.6 / 31.5 %	54.1 / 51.3 %

code execution (e.g., "ModuleNotFoundError", "OperationalError: table already exists"), and (2) uninformative outputs that provide no localization cues, such as generic success messages (e.g., "All tests passed"), timeout errors without stack traces, or outputs that only show test names without indicating which source files are involved. As shown in Table 10(a), only about half of all reproduction executions are actionable. For example, Claude Code's 164 reproduction executions yield only 80 (48.8%) actionable results, while 84 (51.2%) are non-actionable. This indicates low helpfulness of reproduction execution conducted by current state-of-the-art agents. OpenCode's Pass→Pass subset is too small (12 instances) to analyze reproduction-level actionability, but tellingly *none* of those 12 successes involve a reproduction execution at all: every OpenCode Pass→Pass trace we inspected reaches the edit stage after reading the source code directly, without running any test first. When Qwen2.5-Coder-32B succeeds, it succeeds through pure source-code reasoning; execution feedback contributes nothing to that subset. Moreover, even when reproduction provides actionable feedback, subsequent localization is not significantly improved. As shown in Table 10(b), for the 46 Claude Code instances that received actionable feedback, their localization accuracy in UNRESTRICTED mode (93.5%) is slightly *worse* than in PROHIBITED mode (95.7%), with $\Delta = -2.2$ percentage points. For Codex, the 7 actionable instances show identical localization accuracy across modes ($\Delta = 0$). Given the low ratio of actionable executions and their negligible impact on localization, the overall helpfulness of reproduction execution is minimal.

Table 10. Reproduction-execution analysis (Pass→Pass, UNRESTRICTED). Cols 2–4 count individual executions; the "Loc. Δ" col reports localization hit-rate change between PROHIBITED and UNRESTRICTED on the actionable-feedback subset (Claude Code n=46; Codex n=7; OpenCode n=0).

Agent	Inst. w/ Repro	Actionable Execs	Non-act. Execs	Loc. Δ
Claude Code	64/116 (55.2%)	80 (48.8%)	84 (51.2%)	−2.2pp
Codex	9/141 (6.4%)	11 (64.7%)	6 (35.3%)	±0pp
OpenCode	0/12 (0.0%)	0 (—)	0 (—)	—

4.3.2 Reason 2: Feedback from Code Execution is Insufficient for Agents to Correct Errors. Having established that localization does not require execution, we now examine whether code execution helps *after* the agent begins editing. We define *validation execution* as test runs occurring after the first source code edit, used to validate the patch. Table 12 presents validation execution statistics for Pass→Pass and Fail→Fail cases in UNRESTRICTED mode.

The purpose of validation execution is to reveal potential errors in the generated patch and guide subsequent revisions. However, our analysis reveals that most repairs are completed in a single edit without requiring execution feedback. As shown in Table 11, 54–66% of commercial-agent cases involve only one code edit with no subsequent modifications, regardless of whether execution is available. This suggests that the majority of patches are either correct on the first attempt or contain

issues that execution feedback cannot help resolve. Furthermore, even in UNRESTRICTED mode where execution is available, only 49.2% of commercial-agent instances modify code after execution, with Claude Code showing higher responsiveness to execution feedback (67.5%) than Codex (31.0%). OpenCode with Qwen2.5-Coder-32B shows the opposite editing profile: its single-edit ratio is lower (32–44%) and its post-execution modification rate under UNRESTRICTED is 45.6%. OpenCode with Qwen2.5-Coder-32B iterates on its own patch more often than the commercial agents do. However, the resolve rate does not follow, the extra iterations do not close the gap.

Table 11. Code modification patterns. Single-edit = only one edit with no subsequent modifications; Post-exec = code modified after a test execution. Cells are *Prohib. / Unrestr.* (%).

Agent	Single-edit ratio	Post-exec modification
Claude Code	66.0 / 54.0	4.5 / 67.5
Codex	58.0 / 60.5	0.0 / 31.0
OpenCode	43.7 / 32.3	3.6 / 45.6
Overall	**56.0 / 49.0**	**2.7 / 47.9**

Even when agents iterate based on execution feedback, the feedback often fails to lead to correct fixes. As the right panel of Table 12 shows, 81.2% of Claude Code's Fail→Fail cases and 100% of Codex's Fail→Fail cases achieved at least one validation success (i.e., the agent's executed tests passed) during their repair attempts, yet still failed the final SWE-bench evaluation. OpenCode with Qwen2.5-Coder-32B exposes an additional failure mode *upstream* of this mis-alignment: only 52.3% of its Fail→Fail instances produce any validation execution at all (vs. 94–95% for the commercial models and agents), and only 11.1% of those ever see a passing test. Qwen2.5-Coder-32B therefore typically fails before the "agent tests pass but official tests fail" gap can even arise—it cannot construct self-validation that passes in the first place. For other two settings, their discrepancy arises because agents' self-initiated validation differs from the official evaluation tests, so passing agent-executed validation does not reliably indicate a correct fix. Furthermore, even when execution feedback reveals errors, it often does not translate to a correct fix, as the agent may lack the capability to address the underlying issue.

For Claude Code, UNRESTRICTED mode uses 2.1× more conversation turns, 2.8× more edits, and 8.0× more test executions than PROHIBITED mode, yet both achieve the same success rate. For Codex, the difference is smaller but still present: UNRESTRICTED uses 3.3 test executions per instance compared to zero in PROHIBITED. For bugs solvable through reasoning alone, execution acts as confirmatory validation—confirming an already-correct solution rather than enabling its discovery. This is not to say execution is never beneficial, but that for many instances the cost it imposes exceeds the signal it provides. The preceding analysis focused on stable cases (Pass→Pass and Fail→Fail), which account for the majority of instances. For the remaining cases where outcomes differ between modes, we observe a near-symmetric distribution: 24 Pass→Fail cases (execution hurts) versus 29 Fail→Pass cases (execution helps) across all three agents. This balance suggests that execution is roughly equally likely to help or hurt, with a small net benefit of only 5 cases.

4.3.3 Stratification by Gold-Patch Complexity. The near-symmetric 24 vs. 29 Pass→Fail/Fail→Pass split reported above averages over all bugs; it could still hide a pattern where execution helps specifically on the harder ones while hurting the easier ones. We test that refinement next. A natural hypothesis is that execution feedback becomes more valuable as bug complexity increases: a multi-file, multi-hunk fix might benefit from iterative test-driven refinement in ways that a one-line fix does not. We test this by bucketing each of the 600 (agent, benchmark, instance)

Table 12. Validation-execution outcomes after first edit (UNRESTRICTED). Cols *Success/TestFail/EnvErr* count individual executions and report percentages of the per-(agent, outcome) validation-execution total, which also includes a residual "no test signal" category so the three columns do not sum to 100%; the rightmost two cols (*F→F only*) count instances where the agent's own tests passed at least once while the official evaluation still failed.

		Validation executions			F→F w/ val.	
Agent	Outcome	Success	Test Fail	Env Err	Inst. w/ val.	Any pass
Claude Code	P→P	414 (48.6%)	154 (18.1%)	127 (14.9%)	80/84 (95.2%)	65 (81.2%)
Claude Code	F→F	272 (40.7%)	102 (15.3%)	121 (18.1%)		
Codex	P→P	230 (51.7%)	60 (13.5%)	152 (34.2%)	52/55 (94.5%)	52 (100.0%)
Codex	F→F	85 (55.6%)	17 (11.1%)	49 (32.0%)		
OpenCode	P→P	5 (5.5%)	12 (13.2%)	6 (6.6%)	90/172 (52.3%)	10 (11.1%)
OpenCode	F→F	81 (4.6%)	337 (19.3%)	75 (4.3%)		

cells by the ground-truth patch's complexity—files touched, hunk count, and total added/removed lines—and re-computing PROHIBITED vs. UNRESTRICTED resolve rates within each bucket. Table 13 summarises the hunk-based view on VERIFIED, where within-bucket sample sizes are large enough to be meaningful.

Table 13. Resolve rates on SWE-bench Verified stratified by gold-patch hunk count. Each agent cell shows *Prohibited / Unrestricted / Gap (pp)*. Buckets with $N < 10$ are noisy.

Bucket (N)	#Inst.	Claude Code	Codex	OpenCode
1 hunk	62	72.6/79.0/−6.5	80.6/83.9/−3.2	18.2/18.2/±0
2–3 hunks	25	52.0/60.0/−8.0	60.0/60.0/±0	13.0/13.0/±0
≥ 4 hunks	13	46.2/23.1/+23.1	61.5/61.5/±0	20.0/10.0/+10.0

The sign of the PROHIBITED–UNRESTRICTED gap does not grow monotonically with complexity: on Claude Code, the gap flips from −6.5pp (1 hunk) to +23.1pp (≥ 4 hunks). At the largest bucket, PROHIBITED resolves nearly twice as many instances as UNRESTRICTED, opposite to what a "more complex ⇒ more exec" hypothesis would predict. Codex is roughly flat and OpenCode with Qwen2.5-Coder-32B also flips sign. Stratifications by files touched and total added/removed lines produce the same non-monotonic pattern (file buckets −5.9/±0/+33.3 on Claude Code Verified, though the multi-file bucket has N=6; delta-line buckets −2.2/−11.1/+10.5). A plausible reading is that multi-hunk bugs require holistic reasoning that trial-and-error execution can disrupt by crowding the context with test output; in any case the hypothesis that current execution feedback scales with bug complexity is refuted, reinforcing the main finding that the cost-benefit balance of execution is instance-dependent rather than uniformly positive.

Takeaway for RQ3:

- Reproduction execution provides limited localization benefit: although 55% of Claude Code cases use reproduction, localization accuracy remains >95% in both modes, suggesting execution does not improve localization for bugs in our dataset.
- Validation feedback has low signal quality: 15–34% are environment errors, and 81–100% of Fail→Fail cases passed agent tests but failed official evaluation. Moreover, 54–66% of commercial-agent cases complete in a single edit, suggesting that for many solvable bugs the cost of execution exceeds its informational benefit.
- Execution feedback does not scale with patch complexity. Across single-hunk, 2–3 hunk, and ≥ 4 hunk buckets on Verified, the Prohibited–Unrestricted gap does not grow monotonically; it flips sign across agents and buckets. Complex bugs are not systematically better served by unrestricted execution.

5 DISCUSSION

In this section, we discuss the practical implications of our findings and potential threats to validity.

5.1 Practical Implications

From a practitioner perspective, these results may inspire the design of cost-sensitive agents that strategically minimize execution while maintaining repair effectiveness. First, practitioners may consider restricting the execution behaviors of code agents for program repair when cost matters, as our results show it achieves comparable resolve rates at significantly lower cost; Prohibited additionally eliminates the per-repository testbed setup that industrial deployments incur per repo/release. Second, code execution could be adopted by agents selectively when there is clear evidence that project-level feedback provides value for the specific task or agent. Third, enhancing the quality of the feedback from execution remains a critical area for improving agent-based repair.

Our findings suggest a broader principle beyond program repair: *agents benefit more from deeper reasoning than from more frequent environment interaction*. Our results show that execution feedback is a double-edged sword: helpful for validating correct hypotheses, but harmful when it triggers unproductive search loops or misleads agents away from correct solutions. This motivates future work on *adaptive execution allocation*, where rather than using fixed budgets, agents could learn to request execution only when the expected information gain exceeds the risk of being misled.

5.2 Threats to Validity

5.2.1 Internal Validity. Budget enforcement in Quota-Limited and Prohibited is primarily prompt-level: agents occasionally attempt scripts that fail with environment errors (7–9% of Prohibited attempts), which we count as executions under an intention-to-treat framework. The zero-execution and env-error-free subsets (Table 5) keep the ±5pp Prohibited–Unrestricted band, and a tool-enforced sandboxed re-run on Claude Code Verified reproduces the equivalence (−4pp), confirming the result is not an artefact of soft enforcement. Interaction confounding is mitigated by reporting multiple cost signals (tokens, time, executions) rather than relying on any single metric. To ensure fair timing comparisons, we run all execution modes for the same instance concurrently on identical hardware. To address LLM stochasticity, we employ paired statistical tests (McNemar's test) and equivalence testing (TOST with $\delta = 5$pp); 85% of instances produce identical outcomes across all modes.

5.2.2 External Validity. Our conclusions are scoped to *SWE-bench-style repository-level bug fixing* with three current CLI agents (Claude Code, Codex, and open-source OpenCode+Qwen2.5-Coder-32B). We evaluate on the first 100 instances from each dataset (Lite and Verified), totaling 200 instances spanning diverse repositories including Django, Flask, Requests, and Sympy. Recent work raises memorisation concerns about SWE-bench [21, 30]; our OpenCode configuration uses a model whose training cutoff predates Verified, and the ±5pp equivalence still holds on this low-contamination cell (Lite: 7.0% vs. 6.0%; Verified: 13.0% vs. 14.0%), so the equivalence is not an artefact of training-data overlap. Execution may be helpful for certain tasks (e.g., performance optimization requiring profiling, security vulnerabilities requiring dynamic analysis). Even under Prohibited, agents can use extensive repository inspection and large context windows; "no project runtime" does not mean "no environment access." Furthermore, passing tests does not guarantee patch quality; future work could incorporate human evaluation or static-analysis criteria.

6 RELATED WORK

Program repair. Traditional APR uses search-based [17], learning-guided [23], template-based [22], and neural methods [13, 25, 48]; code-specialized LLMs [5, 9, 19, 24, 28, 33] underpin modern APR agents. Agentic approaches (SWE-agent [44], AutoCodeRover [47], ChatRepair [42], RepairAgent [3], InspectCoder [39], PracAPR [43]) emphasize iterative refinement with execution feedback [6, 26], while Agentless [41] uses a fixed pipeline of localization, synthesis, and test selection; it removes the agent loop and execution access simultaneously.

Resource-aware LLM agents. Prior work targets efficiency via turn budgets [8, 35, 36], token-level compression [7, 11, 20, 32], and inference-level speedups [2, 4, 18]; we study the orthogonal dimension of *execution feedback*, finding near-zero marginal benefit from unlimited execution and suggesting agents are over-resourced along multiple axes.

7 CONCLUSION

This paper presents an empirical study of execution behavior in LLM-based program repair. We analyze 7,745 agent traces from public SWE-bench submissions and conduct controlled experiments on 200 SWE-bench instances across three agents (Claude Code, Codex, and open-source OpenCode+Qwen2.5-Coder-32B) under four execution paradigms, comprising 3,000 end-to-end repair attempts. Our analysis yields several key findings. Test execution frequency varies widely across agents (2 to 19 per task), with execution timing and success rates differing substantially. The resolve rate difference between Prohibited and Unrestricted is 1.25 percentage points on average, which is not statistically significant ($p > 0.05$), and stays within ±5pp on every cell—including when the restriction is tool-enforced at the CLI level on Claude Code Verified (−4pp). Prohibited mode consumes 56–62% fewer tokens and 48–54% less wall-clock time for Claude Code. We find that execution has limited impact because (1) reproduction execution provides limited localization benefit, and (2) validation feedback contains significant noise (15–34% environment errors). These observations provide empirical insights into the role of execution in LLM-based repair. Future work could further investigate the conditions under which execution feedback is most beneficial, develop agents that adapt execution strategies to task characteristics, and extend the analysis to other software engineering tasks.

DATA AVAILABILITY

Our code and results are available at https://anonymous.4open.science/r/Run_Less.

REFERENCES

[1] Anthropic. 2025. Claude Code: An agentic coding tool for the terminal. https://www.anthropic.com/claude-code Accessed: 2026-01-23.

[2] Sangmin Bae. 2025. Accelerating Large Language Model Inference via Early-Exiting Algorithms. *arXiv preprint arXiv:2509.05915* (2025).

[3] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2188–2200. https://doi.org/10.1109/ICSE55347.2025.00157

[4] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. *arXiv preprint arXiv:2302.01318* (2023).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[6] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[7] Viet-Tung Do, Xuan-Quang Nguyen, Van-Khanh Hoang, Duy-Hung Nguyen, Shahab Sabahi, Jeff Yang, Hajime Hotta, Minh-Tien Nguyen, and Hung Le. 2025. Automatic prompt selection for large language models. (2025), 91–102.

[8] Pengfei Gao and Chao Peng. 2025. More with Less: An Empirical Study of Turn-Control Strategies for Efficient Coding Agents. arXiv:2510.16786 [cs.SE] https://arxiv.org/abs/2510.16786

[9] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[10] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[11] Qiushi Huang, Xubo Liu, Tom Ko Zheng, Zhaocheng Liu, Wenwu Zhao, Mark Sherblom, Yvonne Coady, and Wenwu Wang. 2024. Selective Prompting Tuning for Personalized Conversations with LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

[12] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[13] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 1161–1173.

[14] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations*.

[15] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 54107–54157. https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

[17] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1, 54–72.

[18] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[20] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. 2023. Compressing Context to Enhance Inference Efficiency of Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

6342–6353. https://doi.org/10.18653/v1/2023.emnlp-main.391

[21] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. arXiv:2506.12286 [cs.SE] https://arxiv.org/abs/2506.12286

[22] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 31–42.

[23] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *SIGPLAN Not.* 51, 1, 298–312. https://doi.org/10.1145/2914770.2837617

[24] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[25] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 101–114.

[26] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[27] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[28] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction Tuning Code Large Language Models. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[29] OpenAI. 2025. OpenAI Codex: The AI Agent for Software Engineering. https://openai.com/codex Accessed: 2026-01-23.

[30] OpenAI. 2026. Why we no longer evaluate SWE-bench Verified. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/ Accessed: 2026-04-28.

[31] OpenCode Contributors. 2025. OpenCode: An Open-Source AI Coding Agent for the Terminal. https://github.com/opencode-ai/opencode v1.4.0, accessed 2026-04-09.

[32] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Ruhle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 2024. LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression. In *Findings of the Association for Computational Linguistics ACL 2024*. 963–981. https://aclanthology.org/2024.findings-acl.57

[33] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[34] Donald J Schuirmann. 1987. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of pharmacokinetics and biopharmaceutics* 15, 6 (1987), 657–680.

[35] Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. 2026. SWE-Lego: Pushing the Limits of Supervised Fine-tuning for Software Issue Resolving. *arXiv preprint arXiv:2601.01426* (2026).

[36] Nguyen Phu Vinh, Anh Chung Hoang, Chris Ngo, and Truong-Son Hy. 2025. Repeton: Structured Bug Repair with ReAct-Guided Patch-and-Test Cycles. *arXiv preprint arXiv:2506.08173* (2025).

[37] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[38] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[39] Yunkun Wang, Yue Zhang, Guochang Li, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. 2025. InspectCoder: Dynamic Analysis-Enabled Self Repair through Interactive LLM-Debugger Collaboration. *arXiv preprint arXiv:2510.18327* (2025).

[40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[41] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. https://doi.org/10.1145/3715754

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

[43] Qi Xin, Haojun Wu, Steven P. Reiss, and Jifeng Xuan. 2024. Towards Practical and Useful Automated Program Repair for Debugging. *arXiv preprint arXiv:2407.08958* (2024).

[44] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1601, 125 pages.

[45] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[46] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=Lr8cOOtYbfL

[47] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1592–1604.

[48] 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 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3468264.3468544