Multi-agent Collaboration with State Management

Mengyang Liu[1,2] **Taozhi Chen**[3] **Zhenhua Xu**[4] **Xue Jiang**[4] **Yihong Dong**[1,✉]
[1]Shanghai Jiao Tong University [2]Cortices AI [3]Emory University [4]Peking University

Abstract

Recent advances in multi-agent systems have shown great potential for solving complex tasks. However, when multiple agents edit a shared codebase concurrently, their changes can silently conflict and inconsistent views lead to integration failures. Existing multi-agent systems address this through workspace isolation (e.g., one git worktree per agent), but this defers conflict resolution to a post-hoc merge step where recovery is expensive. In this paper, we propose STORM, i.e., STate-ORiented Management for multi-agent collaboration. Specifically, STORM manages agent states by mediating their interactions with the shared workspace, ensuring that each agent operates on a consistent view of the codebase and that conflicting edits are detected and resolved at write time. We evaluate STORM on Commit0 and PaperBench with three LLMs. Averaged over the three models, STORM outperforms the git-worktree-based multi-agent baseline by +26.7 weighted and +17.0 macro pass-rate points on Commit0-Lite and by +5.2 points on PaperBench, while achieving comparable or better cost efficiency. These results suggest that explicit state management is a more effective foundation for multi-agent collaboration than workspace isolation. STORM can also be plugged into existing multi-agent systems. Our code is available at `https://github.com/RISE-X-Lab/STORM`.

1 Introduction

Multiple LLM agents working in parallel can solve tasks that are too large for any single agent to finish within its iteration budget [Dong et al., 2024, Hong et al., 2024, Geng and Neubig, 2026, Qu et al., 2026]. In software engineering, agents can implement different modules concurrently; in scientific research, they parallelize experimental setups. But running agents in parallel on a shared workspace raises a question: when one agent edits a file, how do we know that its assumptions about the rest of the codebase are still valid?

We treat this as a state management problem. An agent interacts with its workspace through file reads and writes. When it modifies a file, its reasoning depends not just on that file but on context from other files it has read (dependencies, interfaces, specifications). The edit is only safe when those context files have not changed since the agent last read them. This is a local consistency requirement: the agent does not need a frozen snapshot of the entire workspace, just assurance that the specific files informing its current edit are up to date.

Existing multi-agent systems mostly avoid this problem by giving each agent its own workspace (e.g., a git worktree) and merging afterward [Geng and Neubig, 2026, Qu et al., 2026, Qin and Xu, 2026]. Isolation prevents interference during editing but pushes all conflict resolution to the merge step, after agents have already committed to potentially incompatible designs. Textual merge conflicts are easy to spot; semantic conflicts, where both sides compile individually but break when combined, are harder, and current tooling cannot resolve them automatically.

In this paper, we propose STORM (STate-ORiented Management), a state management framework for multi-agent collaboration. STORM mediates each agent's file reads and writes. Before accepting

✉Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

a write, it checks whether the agent's view of the target file and its context dependencies is still current. If another agent has modified any of those files in the interim, the write is rejected and the agent receives the updated content so it can retry from a correct baseline. The validation rule adapts classical optimistic concurrency control [Kung and Robinson, 1981] to the read sets of LLM agents.

Our main contributions can be attributed as:

1. A formulation of multi-agent state management as file-level context consistency: an agent's write is valid only if the target file and its read dependencies have not been modified since the agent last observed them.
2. STORM, an architecture-agnostic state management framework for multi-agent collaboration that enforces local state consistency at write time, detecting conflicts immediately and enabling agents to retry from a correct baseline without workspace isolation.
3. Empirical validation on Commit0-Lite [Zhao et al., 2025] and PaperBench [Starace et al., 2025] with Claude Sonnet 4.6, Qwen 3.6 Plus, and DeepSeek V4 Pro. With all three models, STORM achieves the highest weighted pass rate on Commit0-Lite (46.2%, 61.3%, and 32.3%, vs. 24.5%, 16.7%, and 18.5% for GitWorktree) and the highest standalone score on PaperBench (74.1, 55.0, and 66.5, vs. 72.7, 51.6, and 55.8).

2 STORM

Mainstream multi-agent systems give each agent its own git worktree [Geng and Neubig, 2026, Qu et al., 2026, Qin and Xu, 2026] so that agents cannot interfere with each other while working. The cost is paid later: once agents finish their individual task, their branches must be merged back together. If two agents edit the same file, or made incompatible design choices about a shared interface, the merge fails. Worse, because each agent worked in isolation without seeing what others were doing, these conflicts tend to compound. Agent A writes a helper assuming a certain signature; Agent B changes that signature in its own branch; code that depends on both is now broken in a way that neither branch exhibits alone.

We avoid this by putting all agents in the same workspace (Figure 1). The key insight is that an agent does not need a globally consistent view of the entire repository to produce a correct edit. It only needs the files it has actually read to remain unchanged while it reasons. We call this property *local state consistency* and formalize it in Section 2.1. In practice, even with disjoint task assignments, agents sometimes need to edit the same file (e.g., two agents each implementing different functions in a shared module). Most of their edits do not interact, but at the boundaries where they do, agents need a way to exchange information. STORM addresses this with two mechanisms: *write-time conflict control* (Section 2.2), which rejects a write whenever the agent's local view has gone stale and lets it retry with fresh context, and *intent annotations* (Section 2.3), structured comments that agents leave in the code so that when another agent reads the same file, it can see not just the raw code but the intent behind it, enabling coordination at these shared boundaries without explicit messaging.

2.1 Local State Consistency

An LLM agent does not need a frozen snapshot of the entire workspace to produce a correct edit. It only needs the files it has actually read to remain unchanged while it reasons. We formalize this as *local state consistency*.

Workspace and agents. Let the workspace be a set of versioned files $\mathcal{W} = \{(f, v_f) \mid f \in \mathcal{F}\}$, where $v_f \in \mathbb{N}$ is the current version of file f. A manager agent M decomposes a task T into sub-tasks $\{\tau_1, \ldots, \tau_k\}$ and assigns each to an engineer agent a_i together with a *primary file set* $F_i \subseteq \mathcal{F}$:

$$M : T \longrightarrow \{(\tau_i, F_i, a_i)\}_{i=1}^{k}, \quad \text{where } F_i \cap F_j = \emptyset \text{ for } i \neq j. \tag{1}$$

The disjoint assignment reduces but does not eliminate conflicts: agents may still read or edit files outside their primary set (e.g., a shared utility or a common import).

Agent local state. As agent a_i works on τ_i, it accumulates a *read snapshot* S_i recording every file it has observed and the version at observation time:

$$S_i = \{(g, v_g^{\text{obs}}) \mid a_i \text{ has read } g\}. \tag{2}$$

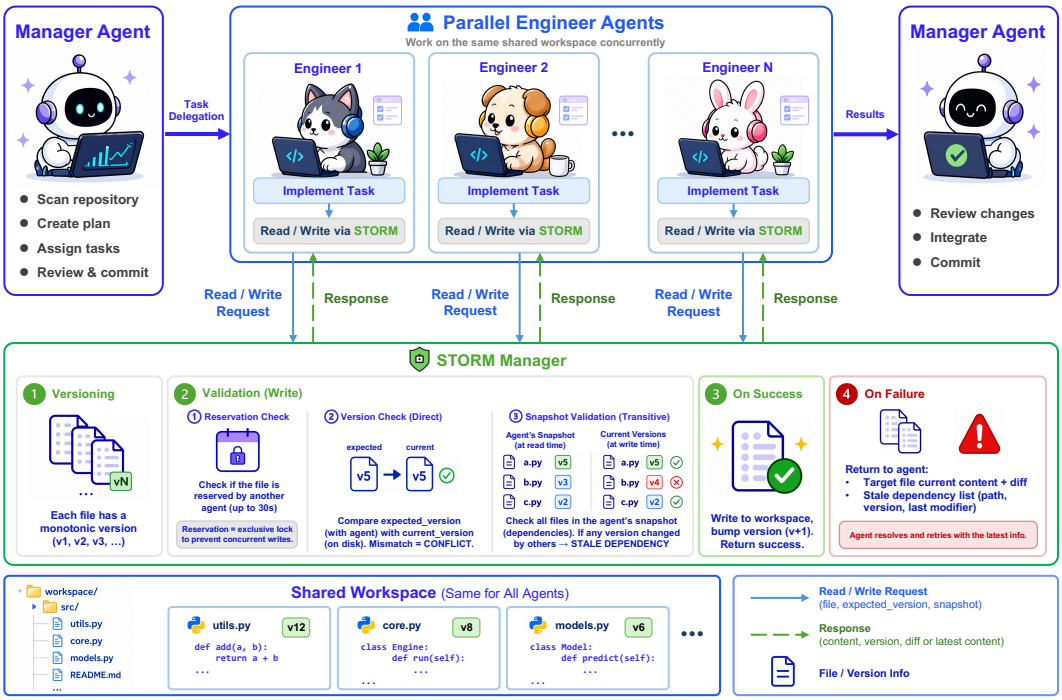


Figure 1: System architecture. The manager analyzes the repository, delegates tasks to parallel engineers, and commits their work. All engineers share one workspace; the STORM manager mediates file operations to detect and resolve conflicts.

When a_i issues a write to file f producing new content c', its generation depends only on S_i, the local context the LLM has seen, not on the full workspace state. This is the key asymmetry we exploit: correctness requires consistency of S_i, not of $\mathcal{W}$.

In practice, each task τ_i requires accessing (reading or modifying) a set of files $A_i \subseteq \mathcal{F}$ that may extend beyond the assigned F_i. When two agents' access sets overlap ($A_i \cap A_j \neq \emptyset$), the shared files form a boundary where conflicts may arise. STORM only needs to coordinate at these boundaries, leaving the non-overlapping majority of work fully parallel.

Write validity. A write (a_i, f, c') is *valid* if and only if the agent's local state is still consistent with the current workspace:

$$\forall\,(g, v_g^{\text{obs}}) \in S_i : \quad v_g^{\text{obs}} = v_g^{\text{cur}}. \tag{3}$$

That is, no file that a_i has read has been modified since its observation. A valid write is applied atomically: $v_f \leftarrow v_f + 1$ and the content of f is updated to c'.

Conflict. A write is *conflicting* when Eq. 3 is violated. Two cases arise:

- **Direct conflict**: the target file itself was updated ($v_f^{\text{obs}} < v_f^{\text{cur}}$), meaning another agent wrote to f concurrently.
- **Stale dependency**: a dependency file $g \neq f$ was updated ($v_g^{\text{obs}} < v_g^{\text{cur}}$), meaning a_i's reasoning may rest on outdated context.

In both cases, STORM rejects the write and returns the current state, enabling a_i to refresh S_i and retry from a correct baseline. Section 2.2 details the mechanism.

Figure 1 shows the architecture. A single manager agent reads the repository, partitions work into sub-tasks scoped to disjoint file sets, assigns each to an engineer, reviews diffs after engineers finish, runs tests, and commits accepted changes. Only the manager commits. Each engineer receives a scoped task (e.g., "implement all functions in `tensor_ops.py`") and accesses the workspace only through the STORM-mediated `file_editor`. Engineers do not communicate directly; coordination happens through the shared codebase and intent annotations (Section 2.3).

2.2 Write-time Conflict Control

STORM enforces the validity condition in Eq. 3 via a mechanism inspired by optimistic concurrency control [Kung and Robinson, 1981]. The key observation is that in a well-decomposed task, most concurrent edits touch different files; the system lets all operations proceed without blocking and only intervenes when a conflict actually occurs.

Implementation. Each file maintains a monotonically increasing version counter (v_f, starting at 1). Every `file_editor` read returns the content together with v_f; every write must declare the expected version. The STORM layer validates the write against Eq. 3 by comparing the agent's full read snapshot S_i to the current workspace state. If validation passes, the write is applied atomically ($v_f \leftarrow v_f + 1$). If it fails, the write is rejected.

Rejection payload. On rejection, STORM returns: (1) the current content of the target file, (2) a unified diff showing what changed since the agent's last read (for direct conflicts), and (3) a list of stale dependencies with their version deltas. This gives the LLM enough context to re-plan from the current baseline without needing to re-read every file. The payload is the response to the rejected `file_editor` call itself, so the agent refreshes its context in the same turn and reissues the write; there is no separate retry planner.

Reservation. After a rejection, a short reservation is granted to the rejected agent on the target file. The reservation is a liveness safeguard, not part of the consistency guarantee: stale writes are rejected with or without it. Without it, under high contention a rejected agent could refresh its view only to be invalidated again before it retries, and repeated alternating invalidations can cause livelock or starvation.

2.3 Intent annotations

Version tracking catches file-level conflicts but not semantic ones. Two agents might implement the same helper with different signatures, or make incompatible assumptions about a shared data structure. To reduce this, engineers annotate their code with structured intent comments:

```
# engineer_1: validate numeric inputs before summing
def add(a, b):
    if not isinstance(a, (int, float)):
        raise TypeError("a must be numeric")
    return a + b
```

Each comment identifies the author and describes what the block accomplishes. Engineers preserve annotations left by other agents unless their task requires changing the annotated block. When another agent reads the file, these comments provide a lightweight channel for semantic coordination: agents can see what others have done and avoid duplicate or conflicting work. The convention is injected into each engineer's system prompt automatically.

3 Experiments

We evaluate on two agent benchmarks: Commit0-Lite and PaperBench Code-Dev. We compare three standalone methods across three LLMs (Claude Sonnet 4.6, Qwen 3.6 Plus, DeepSeek V4 Pro): (1) Single-agent with 100 iterations; (2) GitWorktree, engineers in isolated worktrees integrated by the manager after completion; and (3) STORM, a manager with engineers sharing one workspace. GitWorktree and STORM share the same manager–engineer workflow, models, and budgets, and differ mainly in post-hoc worktree integration versus write-time validation. Commit0-Lite uses 4 engineers; PaperBench uses 2 engineers with 2 rounds of delegation each.

Evaluation. For Commit0-Lite, we run `pytest` on the final workspace and report **Score$_\mathbf{w}$** (total tests passed / total tests) and **Score** (mean per-repository pass rate). For PaperBench, we use the Code-Dev subset following Geng and Neubig [2026] due to cost constraints: the LLM judge (Sonnet 4.6) grades only "Code Development" nodes in the rubric tree, evaluating whether the submitted source code correctly implements each criterion without requiring experiment execution. To check for

Table 1: Results on Commit0-Lite and PaperBench Code-Dev. Among the standalone methods, the best per model is in **bold** and the second best underlined.

	Commit0-Lite				PaperBench		
	$\text{Score}_{\text{w}} \uparrow$	Score ↑	$\text{Cost}_{\text{eff}} \downarrow$	$\text{Time}_{\text{eff}} \downarrow$	Score ↑	$\text{Cost}_{\text{eff}} \downarrow$	$\text{Time}_{\text{eff}} \downarrow$
Claude Sonnet 4.6							
Single-Agent	20.7	66.4	**3.2**	18.8	68.7	12.5	**11.6**
GitWorktree	24.5	63.8	8.6	20.9	72.7	17.2	22.5
STORM	**46.2**	**82.5**	6.3	**16.8**	**74.1**	**11.6**	24.4
GitWorktree-Combined	31.1	77.8	8.2	26.0	77.6	27.2	31.3
STORM-Combined	49.0	87.3	6.3	21.2	78.2	22.0	33.3
Qwen 3.6 Plus							
Single-Agent	34.0	**75.3**	**1.3**	13.0	47.7	**1.0**	**16.2**
GitWorktree	16.7	57.4	6.9	55.8	51.6	2.7	35.9
STORM	**61.3**	70.5	2.5	**12.5**	**55.0**	1.6	21.0
GitWorktree-Combined	35.1	79.9	3.9	32.0	57.9	3.2	45.4
STORM-Combined	74.1	85.6	2.1	13.6	58.6	2.3	32.9
DeepSeek V4 Pro							
Single-Agent	26.8	**65.3**	**1.7**	**31.3**	62.9	**0.8**	**16.5**
GitWorktree	18.5	44.0	3.8	45.0	55.8	1.8	35.7
STORM	**32.3**	63.2	3.0	39.0	**66.5**	1.9	35.4
GitWorktree-Combined	29.3	72.9	4.0	56.7	66.2	2.3	45.8
STORM-Combined	39.6	75.3	3.5	51.0	71.2	2.5	47.7

same-family judge bias, we also re-judge the Sonnet runs with GPT-5.5; these results are reported in Appendix C.1. We report **Score** (mean per-paper judge score $\times 100$). For both benchmarks, we report $\textbf{Cost}_{\textbf{eff}}$ (total cost divided by Score_{w} on Commit0-Lite and by Score on PaperBench, lower is better) and $\textbf{Time}_{\textbf{eff}}$ (total wall-clock minutes across all tasks divided by the same score, lower is better).

Implementation. All agents run on OpenHands [Wang et al., 2025]. Multi-agent runs follow a two-round protocol: in the first round the manager decomposes the task and dispatches engineers in parallel (each with 80 iterations); in the second round it reviews outputs, runs tests, and may reassign failed sub-tasks for one retry. The manager gets 50 iterations total. Model pricing and token accounting are in Appendix A.1, and the remaining setup is in Appendix A.

3.1 Main results

Table 1 reports one run per configuration on both benchmarks for all three models, and Appendix C.1 repeats the Sonnet 4.6 configurations three times (Table 10). Our primary comparison is among the standalone methods. On Commit0-Lite, STORM achieves the **highest weighted score for every model** (**46.2**, **61.3**, **32.3** for Sonnet, Qwen, DeepSeek), with gains concentrated on large repositories with cross-file dependencies where shared-workspace coordination matters most. Over three Sonnet runs, STORM leads GitWorktree by **21.3** weighted and **16.0** macro points. The macro score is less uniform: with Qwen and DeepSeek, single-agent keeps a higher macro score (75.3 vs. 70.5 and 65.3 vs. 63.2) because it solves small, self-contained repositories that STORM sometimes over-decomposes. GitWorktree has the lowest macro score for every model. On PaperBench, STORM has the highest standalone score for all three models: **74.1** vs. 68.7 single-agent (Sonnet), **55.0** vs. 47.7 (Qwen), and **66.5** vs. 62.9 (DeepSeek). Because the Sonnet 4.6 judge shares a model family with the Sonnet agents, we re-judge the original Sonnet trajectories and two additional runs with GPT-5.5. STORM still ranks first (**72.3** vs. 67.7 GitWorktree and 65.7 single-agent), and it also ranks first for Qwen and DeepSeek agents judged by Sonnet.

The Combined rows use a verifier-guided cascade: single-agent runs first, and on tasks it does not fully solve (under the test suite on Commit0-Lite or the rubric on PaperBench), we run the multi-agent method and keep the higher-scoring of the two codebases. STORM-Combined scores highest for every model on both benchmarks, showing that the two approaches are complementary: single-agent excels on small, self-contained tasks, while STORM covers larger, coupled ones. Per-task results are in Appendix B.

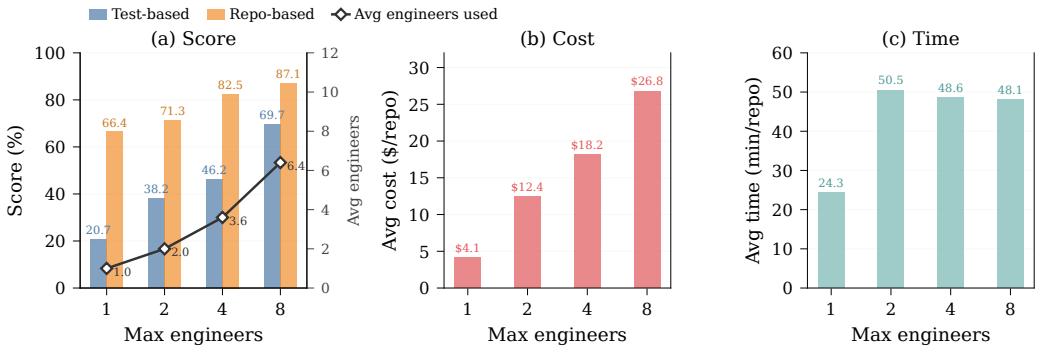


Figure 2: Scaling engineers with Sonnet 4.6 on Commit0-Lite. (a) Both test-based and repo-based scores improve monotonically from 2 to 8 max engineers, and the line shows the average number actually deployed. (b) Cost scales linearly with engineer count. (c) Wall-clock time remains roughly constant due to parallel execution.

The single agent wins on cost-efficiency across most models due to zero coordination overhead. On Commit0-Lite, STORM is more cost- and time-efficient than GitWorktree for every model, because conflicts are resolved incrementally rather than in a costly merge step.

3.2 Scaling to More Engineers

A common finding in prior multi-agent systems is that performance *degrades* as more agents are added [Geng and Neubig, 2026, Lin, 2026]. The reason is straightforward: for a fixed-size task, splitting work among more agents means each agent's sub-task becomes smaller, but the file coupling between sub-tasks increases. More agents need to touch shared interfaces, read overlapping files, and make mutually consistent design choices. Under worktree isolation, this coupling manifests as merge conflicts that grow combinatorially with the number of branches. STORM inherently avoids it: since conflicts are detected and resolved at write time, higher coupling leads to more frequent but individually cheap rejections rather than a catastrophic merge failure at the end. Each conflict is resolved in isolation while the agent's reasoning context is still fresh, so scaling up agents does not increase the difficulty of final integration.

Figure 2 shows the effect of increasing the maximum number of engineers from 2 to 8 on Sonnet 4.6 with STORM. Both the test-based score (overall pass rate) and the repo-based score (macro-average per-repository pass rate) improve: from 38.2% to 69.7% (+31.5) and 71.3% to 87.1% (+15.8), respectively. The gains are not uniform across the two transitions. Moving from 2 to 4 engineers yields +11.2 macro points but only +8.0 overall points, because the improvement concentrates on medium-sized repositories (`cookiecutter` 40.9→98.6, `imapclient` 16.5→89.1, `jinja` 0.0→47.1). Moving from 4 to 8 engineers yields a smaller macro gain (+4.6) but a much larger overall gain (+23.5), driven almost entirely by `babel` (20.2→57.5) and `jinja` (47.1→66.5), repositories with thousands of tests and deep cross-file dependencies that benefit from aggressive parallelization.

Notably, the manager does not always use all available engineers. The average number of engineers actually deployed is 2.0, 3.6, and 6.4 for max settings of 2, 4, and 8 respectively. Simple repositories (`cachetools`, `deprecated`, `wcwidth`) consistently receive only 2 engineers regardless of the maximum, avoiding unnecessary coordination overhead. The number of repositories solved at $\geq$99% pass rate grows from 7 (max=2) to 8 (max=4) to 10 (max=8) out of 16.

Cost scales approximately linearly with the number of engineers ($199→$292→$429), while wall-clock time remains roughly constant ($\sim$13 hours total across all 16 repositories) because engineer tasks execute in parallel. The cost-efficiency ratio (dollars per percentage point of overall score) is stable at $5–6 per point across all configurations, indicating that additional engineers provide proportional value rather than diminishing returns at this scale.

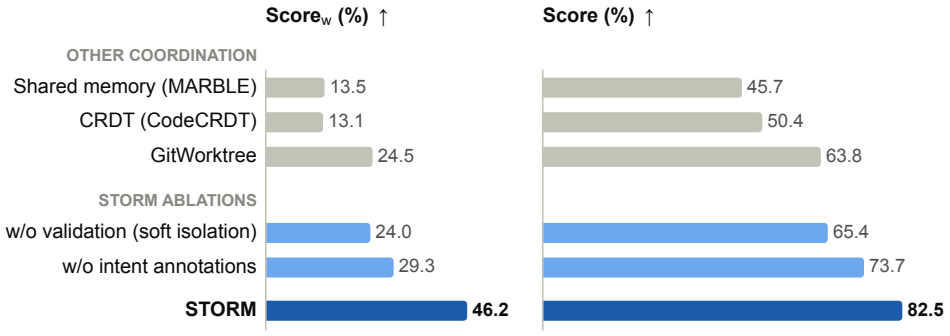


Figure 3: Coordination mechanisms and ablations on Commit0-Lite with Claude Sonnet 4.6 and 4 engineers.

3.3 Coordination Baselines and Ablations

Figure 3 compares STORM with other coordination mechanisms under the same manager–engineer workflow, models, and budgets (Sonnet 4.6, 4 engineers) and removes STORM's components (details in Appendix C.2). Soft isolation keeps the shared workspace and manager loop but replaces write-time validation with prompt-level file scopes; it performs about the same as GitWorktree (24.0 vs. 24.5 Score$_w$) and far below STORM (46.2), so the manager review loop alone does not explain STORM's gain. Shared-memory (MARBLE-based [Zhu et al., 2025]) and CRDT (CodeCRDT-based [Pugachev, 2025]) coordination, ported into the same harness, reach only 13.5 and 13.1 Score$_w$. Removing intent annotations while keeping validation lowers STORM's Score$_w$ by 16.9 points and Score by 8.8 points.

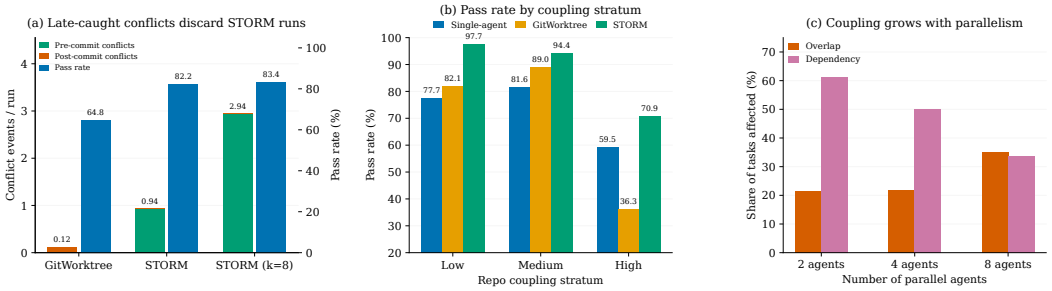


Figure 4: Analysis summary on Sonnet 4.6. (a) STORM surfaces conflicts pre-commit, while GitWorktree leaves most conflicts for late (post-commit/merge) resolution; late-caught conflicts are associated with lower final pass rates (blue overlay). (b) STORM's pass-rate advantage over the single-agent and GitWorktree baselines grows with cross-file coupling. (c) Narrow task scopes are not always independent scopes: first-round overlap and dependency signals remain common, and rise with k. Quantities in (a) and (c) are proxy measurements derived from manager-review events and declared task scopes.

3.4 Further Analysis

STORM helps by exposing invalid parallel work early. STORM's benefit is converting hidden integration errors into explicit write-time feedback. As Figure 4(a) shows, on Sonnet 4.6 GitWorktree leaves most conflict events until post-commit merge while STORM surfaces them pre-commit at both k=4 and k=8, and the configurations that catch conflicts early also achieve higher final pass rates. The effect is not that STORM eliminates conflicts, but that it relocates them from a fragile post-hoc merge to a cheap write-time check. GitWorktree produces 3.81 reviewed write attempts per run at 91.8% acceptance, versus 6.00 at 81.2% for STORM (k=4) and 10.25 at 67.1% for STORM (k=8); the lower acceptance reflects a stricter consistency filter rather than weaker engineering, and STORM still outperforms GitWorktree in final repository quality.

STORM remains robust on high-coupling repositories where alternatives collapse. STORM's advantage is sharpest where parallel coordination matters most. Stratifying repositories by a proxy coupling score (from task overlap, dependency signals, multi-file scope, and rejection evidence), STORM's lead over GitWorktree widens from $+15.6$ points on low-coupling and $+5.4$ on medium-coupling subsets to $+34.6$ points on the high-coupling stratum (Figure 4(b)), because GitWorktree collapses outright there (36.3% vs. STORM 70.9%). STORM's margin over the single-agent baseline remains positive across all strata ($+20.0$, $+12.8$, and $+11.4$ points from low to high); it narrows in the high-coupling regime only because the absolute ceiling falls for every method, not because STORM loses its edge. Concretely, as coupling scales from low to high, STORM degrades gracefully ($97.7\% \rightarrow 94.4\% \rightarrow 70.9\%$), whereas single-agent falls from 77.7% to 59.5% and GitWorktree plummets from 82.1% and 89.0% to 36.3%: STORM is the *only* method that does not break down under coupling. The repository-level pattern agrees: `marshmallow` rises $0.0\% \rightarrow 82.3\%$ and `imapclient` $9.7\% \rightarrow 89.1\%$ (Table 4), precisely where branch-level isolation is most brittle. As a result, STORM more than doubles the Sonnet weighted score of the single agent (46.2 vs. 20.7) and nearly doubles that of GitWorktree (46.2 vs. 24.5). Since the coupling score is a heuristic, we check a rejection-free variant against static properties of the unmodified repositories (Appendix C.3): it correlates with the number of source files, tests, and internal import edges (Spearman $\rho = 0.56$–0.67), and STORM's gain over GitWorktree correlates with the same properties ($\rho = 0.58$–0.64), all significant after FDR correction.

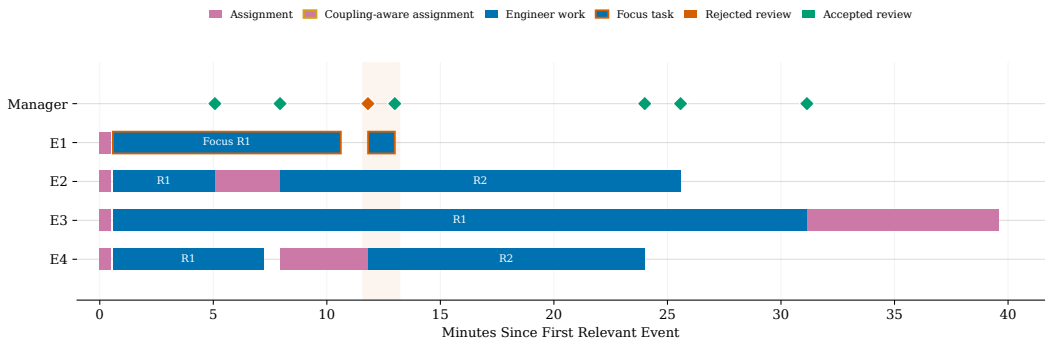
Scaling is limited more by decomposition quality than by STORM itself. The diminishing returns of additional engineers come from decomposition, not STORM. As Figure 4(c) illustrates, STORM already produces narrow scopes: 91.4% of k=4 tasks are single-file, yet 50.0% still carry dependency signals and 21.7% of first-round tasks overlap another's file scope. At k=8, single-file share stays high (85.1%) while first-round overlap rises to 35.1%. First-round overlap correlates with rejected-review rate ($r = 0.28$ overall, $r = 0.78$ within k=8): once the manager can no longer partition the repository into disjoint units, additional engineers create contention faster than useful parallel work.

Case study: pre-hoc coordination versus post-hoc recovery. A paired run on `jinja` makes the aggregate pattern in Figure 4(a) concrete at the level of a single repository. Under GitWorktree (Figure 5a), four engineers work on isolated branches and the coupling around `utils.py` stays invisible until merge: the manager rejects the focus diff with an explicit *merge conflict* reason (red diamond) and the same engineer reworks the task in a second round inside the shaded recovery window before acceptance. Under STORM (Figure 5b, restricted to the coupling task set), the manager reasons about the same coupling at decomposition and instruction time: it packs `utils.py` and `async_utils.py` into a single assignment and explicitly sequences downstream consumers against it (gold-outlined bars carry this interdependence reasoning); the focus task then passes on the first review and no rejection appears in the coupled task set. The two timelines describe the same mechanism from opposite sides, GitWorktree lets coupling surface *post hoc* at the merge boundary, while STORM converts the identical signal into a *pre hoc* coordination decision baked into how tasks are defined and ordered.

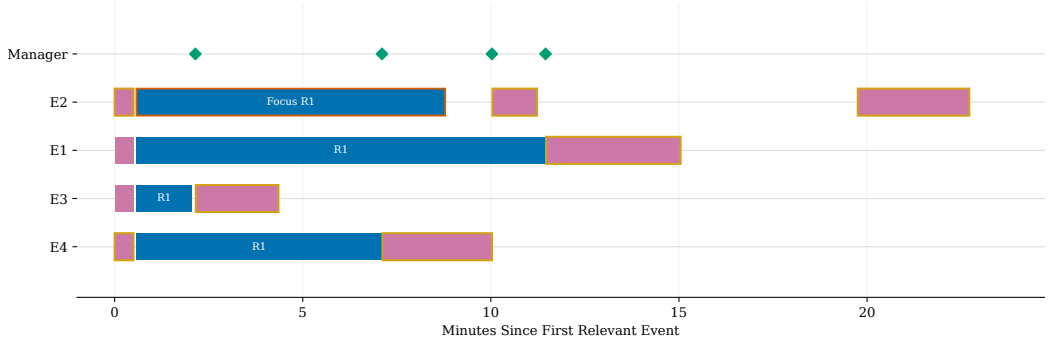
In summary, the analysis suggests a clear interpretation of STORM. Its main contribution is not cheaper computation or perfect recovery from every rejected edit. Its contribution is to replace delayed, fragile integration with immediate consistency checks that make parallel collaboration substantially more reliable on the repositories where parallelism is most useful. The residual failure modes of STORM are further analyzed in Appendix D.

4 Related Work

Multi-Agent Collaboration Multi-agent collaboration decomposes complex tasks into specialized roles [Dong et al., 2024, Hong et al., 2024, Wu et al., 2023, Dong et al., 2026, Hsu et al., 2026]. Compared with a single-agent workflow, a multi-agent system can separate planning, implementation, testing, debugging, and review [Tao et al., 2024, Li et al., 2025, Kumar et al., 2026]. Existing work mainly studies role-based software development, repository-level issue resolution, and workspace-isolated parallel development. Self-Collaboration [Dong et al., 2024] first introduced the multi-agent framework for code generation [Dong et al., 2025a]. Subsequently, MetaGPT model software development as a structured workflow among specialized agents [Hong et al., 2024]. MAGIS and



(a) GitWorktree: post-hoc detection.



(b) STORM: pre-hoc coordination.

Figure 5: Paired run timelines on `jinja`; the legend in (a) applies to both panels. (a) GitWorktree exposes the `utils.py` coupling only at merge review: the red diamond marks the rejected review, the shaded band the rework window, and *Focus R2* the retry that is eventually accepted. (b) STORM detects the same coupling at decomposition time and co-assigns or sequences the dependent tasks (gold-outlined manager instructions carry explicit interdependence reasoning); the focus task passes on its first review and no rejection remains within the coupled task set.

SWE-Debate study multi-agent issue resolution on SWE-style benchmarks [Tao et al., 2024, Li et al., 2025]. AgentForge adds execution feedback through Docker-based validation [Kumar et al., 2026]. Git worktree is a common mechanism for workspace-isolated parallel agents. Systems such as Git Worktree, CORAL, and StatsClaw use git worktree to separate agent workspaces [Geng and Neubig, 2026, Qu et al., 2026, Qin and Xu, 2026]. Each agent works in a separate branch or worktree, while shared memory or a central coordinator supports information exchange. Workspace isolation enables parallel exploration and reduces direct interference between agents' code changes. However, git worktree only provides low-level workspace isolation. It does not solve task decomposition, dependency tracking, semantic conflicts, or merge selection. Different agents may edit related files under incompatible assumptions. Some errors may only appear after integration. Industry systems also report that optimistic concurrency control improves over lock-based shared-state coordination [Lin, 2026], but remains insufficient without hierarchical task decomposition. Therefore, worktree-based systems still require higher-level coordination, shared memory, review, and execution-based verification.

State Management in Agentic Systems Agentic systems require persistent state across multi-step interaction. The state may include conversation history, plans, tool outputs, execution feedback, repository changes, and intermediate artifacts. Existing work manages agent state through reflection, memory, skill libraries, and execution traces [Shinn et al., 2023, Park et al., 2023, Packer et al., 2023, Wang et al., 2024, Hu et al., 2025, He et al., 2026, Liu et al., 2026]. Reflexion and Generative Agents use reflection and episodic memory to improve later decisions [Shinn et al., 2023, Park et al., 2023]. MemGPT manages long-term context through memory tiers [Packer et al., 2023]. Voyager stores reusable skills as executable code, while HiAgent manages working memory with subgoals [Wang et al., 2024, Hu et al., 2025]. However, state management remains difficult in multi-agent software

development. Most existing methods focus on a single agent or a single memory store. Multi-agent systems must also manage local states, shared states, workspace states, and execution states. Git Worktree uses isolated workspaces and branch-and-merge coordination for asynchronous software engineering agents [Geng and Neubig, 2026]. CORAL and StatsClaw use shared memory and isolated worktrees to support parallel execution [Qu et al., 2026, Qin and Xu, 2026]. CodeCRDT instead coordinates agents through observable shared state: it accepts and merges concurrent edits with deterministic character-level convergence and leaves semantic conflicts to later diagnosis [Pugachev, 2025]. MARBLE, the framework of MultiAgentBench, evaluates collaborating and competing LLM agents [Zhu et al., 2025]. However, workspace isolation and shared memory do not guarantee state consistency, dependency tracking, or safe integration. STORM works at a different layer: it validates a write against the versions of the files the agent observed and rejects it when those dependencies are stale. We view CodeCRDT, shared-memory, and worktree mechanisms as related and potentially complementary to STORM, and compare against shared-memory and CRDT baselines in Section 3.3.

5 Conclusion

We presented STORM, a state management framework that replaces workspace isolation with local state consistency for multi-agent collaboration. All agents share one workspace; writes are accepted only if the agent's observed files remain unchanged, and intent annotations coordinate semantics at shared boundaries. STORM outperforms the GitWorktree baseline by +26.7 weighted and +17.0 macro pass-rate points on Commit0-Lite and by +5.2 points on PaperBench, and it also outperforms shared-memory and CRDT-based coordination in the same harness. On PaperBench Code-Dev, STORM has the highest standalone score for all three models and under an independent GPT-5.5 judge. With Qwen and DeepSeek, STORM again achieves the highest weighted pass rate and outperforms GitWorktree on both metrics. Scaling to 8 engineers improves score with roughly constant wall-clock time. Within the baselines we evaluate, these results suggest that explicit state management is a more effective foundation for multi-agent collaboration than workspace isolation.

References

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7):189:1–189:38, 2024.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for A multi-agent collaborative framework. In *ICLR*. OpenReview.net, 2024.

Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents. *CoRR*, abs/2603.21489, 2026.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. CORAL: towards autonomous multi-agent evolution for open-ended discovery. *CoRR*, abs/2604.01658, 2026.

Tianzhu Qin and Yiqing Xu. Statsclaw: An ai-collaborative workflow for statistical software development. *CoRR*, abs/2604.04871, 2026.

H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *ICLR*. OpenReview.net, 2025.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate AI research. In *ICML*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *ICLR*. OpenReview.net, 2025.

Wilson Lin. Scaling long-running autonomous coding. Cursor Blog, January 2026. URL `https://cursor.com/blog/scaling-agents`.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You. MultiAgentBench: Evaluating the collaboration and competition of LLM agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. URL `https://aclanthology.org/2025.acl-long.421/`.

Sergey Pugachev. Codecrdt: Observation-driven coordination for multi-agent LLM code generation. *CoRR*, abs/2510.18893, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155, 2023.

Yihong Dong, Jiaru Qian, Haoran Zhang, Peixu Wang, Binhua Li, Zhi Jin, Yongbin Li, Ge Li, Xiaokang Yang, and Xue Jiang. Agentic Reverse Engineering: From I/O to code with discovery agent. *arXiv preprint arXiv:2605.15334*, 2026. URL `https://arxiv.org/abs/2605.15334`.

Chun-Wah Hsu, Kai Gong, Yu Wu, Xianhe Chen, Mengyang Liu, Jie Li, Hanyu Li, Zhixuan Liu, Naisheng Tang, Jiaying Chi, Ziheng Fan, Xuning He, Xiaokang Yang, Xue Jiang, and Yihong Dong. OpenCollab: A multi-agent coding framework with programmable collaboration and controllable runtime. *arXiv preprint arXiv:2609.38345*, 2026. URL `https://arxiv.org/abs/2609.38345`.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: llm-based multi-agent framework for github issue resolution. In *NeurIPS*, 2024.

Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. Swe-debate: Competitive multi-agent debate for software issue resolution. *CoRR*, abs/2507.23348, 2025.

Rajesh Kumar, Waqar Ali, Junaid Ahmed, Najma Imtiaz Ali, and Shaban Usman. Agentforge: Execution-grounded multi-agent LLM framework for autonomous software engineering. *CoRR*, abs/2604.13120, 2026.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents. *CoRR*, abs/2508.00083, 2025a.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *UIST*, pages 2:1–2:22. ACM, 2023.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *CoRR*, abs/2310.08560, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024, 2024.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *ACL (1)*, pages 32779–32798. Association for Computational Linguistics, 2025.

Xuning He, Kai Fan, Runyu Zhang, Binghuan Wang, Yuxiang Zhang, Xue Jiang, and Yihong Dong. In-model context compression with agentic soft tokens for long-horizon LLM agents. *arXiv preprint arXiv:2610.41291*, 2026. URL https://arxiv.org/abs/2610.41291.

Mengyang Liu, Weiming Wang, Haixin Nan, Yu Wu, Peize Li, Shihong Huang, Xuning He, Xue Jiang, and Yihong Dong. SPROUT: Scaling agentic reinforcement learning for long-horizon software engineering. *arXiv preprint arXiv:2610.14405*, 2026. URL https://arxiv.org/abs/2610.14405.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33 (7):182:1–182:30, 2024.

Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. ROCODE: integrating backtracking mechanism and program analysis in large language models for code generation. In *ICSE*, pages 334–346. IEEE, 2025.

Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Yanhao Li, Yue Liu, Rongyu Cao, Jue Chen, Fei Huang, and Binhua Li. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute. In *ASE*, pages 3730–3741. IEEE, 2025.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *EMNLP*, pages 2471–2484. Association for Computational Linguistics, 2023.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. Repofusion: Training code models to understand your repository. *CoRR*, abs/2306.10998, 2023.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *NeurIPS*, 2023.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *ACL (Findings)*, Findings of ACL, pages 3603–3614. Association for Computational Linguistics, 2024.

Xue Jiang, Tianyu Zhang, Ge Li, Mengyang Liu, Taozhi Chen, Zhenhua Xu, Binhua Li, Wenpin Jiao, Zhi Jin, Yongbin Li, and Yihong Dong. Think anywhere in code generation. *CoRR*, abs/2603.29957, 2026.

Mofei Li, Taozhi Chen, Guowei Yang, and Jia Li. Memcoder: Multi-dimensional evolving memory for private-library-oriented code generation. *CoRR*, abs/2604.24222, 2026.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ACM Trans. Softw. Eng. Methodol.*, 34(3):77:1–77:22, 2025b.

A Detailed Experiment Setup

We evaluate on **Commit0** [Zhao et al., 2025], a repository-level code implementation benchmark. Each instance is a real Python repository with its test suite intact but source implementations removed (replaced with stubs or `pass` statements). The task is to implement the missing code so that the tests pass. We use the Commit0 Lite subset: 16 repositories spanning a range of domains and sizes. Several repositories require cross-file reasoning, making them difficult for a single agent and a natural fit for studying multi-agent collaboration. We also evaluate on **PaperBench** [Starace et al., 2025], a benchmark for evaluating AI agents on end-to-end research replication. Each instance is based on an ICML 2024 Spotlight or Oral paper. The task is to understand the paper, implement the required codebase from scratch, run experiments, and reproduce the key results. PaperBench uses hierarchical rubrics to evaluate fine-grained replication progress rather than only final execution success. These tasks require long-horizon planning, cross-file implementation, experiment execution, and result verification, making them a natural fit for studying multi-agent collaboration.

Configurations. We compare five configurations across three LLMs (Claude Sonnet 4.6, Qwen 3.6 Plus, DeepSeek V4 Pro): (1) **Single-agent** with 100 LLM iterations; (2) **GitWorktree** [Geng and Neubig, 2026] with 4 engineers (2 on PaperBench) in isolated worktrees, merged after completion; (3) **STORM** with a manager and 4 engineers (2 on PaperBench) sharing one workspace; (4) **GitWorktree-Combined**; and (5) **STORM-Combined**. The Combined variants are a verifier-guided cascade. Single-agent runs first and its complete codebase is checked by the verifier (the repository test suite on Commit0-Lite, the rubric score on PaperBench). If it passes all tests, it is kept. Otherwise, the multi-agent method runs on the same task, producing a second, separate codebase, and the codebase with the higher verifier score is kept. All files and test outcomes for a task come from the selected codebase; tests, patches, and files are never mixed across submissions. The benchmark score is then aggregated over the selected codebases, and the multi-agent cost is counted only on tasks where it was invoked. The cascade requires a representative test suite at deployment time and pays for a second run on unsolved tasks, so we treat it as a complementary result rather than a standalone method.

Implementation. All agents run on the OpenHands SDK [Wang et al., 2025] inside Docker containers. The manager gets up to 50 LLM iterations; each engineer gets up to 80 per task and can be reassigned once. During analysis, the manager recommends how many engineers to use based on repository complexity; for simple repositories it may use as few as 2. Evaluation uses `pytest` with `--json-report`; if the JSON report is unavailable we fall back to parsing terminal output. The total number of tests per repository is taken from the ground-truth counts in the Commit0 paper [Zhao et al., 2025].

Metrics. We report $\textbf{Score}_{\textbf{w}}$ (weighted pass rate: total tests passed / total tests, dominated by large repositories) and **Score** (macro: mean of per-repository pass rates). For efficiency we report $\textbf{Cost}_{\textbf{eff}}$ (total dollars / Score_{w}) and $\textbf{Time}_{\textbf{eff}}$ (total minutes / Score_{w}), both lower-is-better.

PaperBench. We use the PaperBench Code-Dev subset, which evaluates code development only and skips the reproduction execution step. The judge grades only "Code Development" nodes in the rubric. Table 1 and the per-paper tables use Sonnet 4.6 as the LLM judge. Because Sonnet 4.6 is also an agent model, we re-judge the original Sonnet trajectories and two additional Sonnet runs with GPT-5.5 for Table 10; in these runs Sonnet 4.6 is the agent model and GPT-5.5 is used only as the judge.

Baselines. GitWorktree and STORM use the same manager model, manager prompt, decomposition protocol, and iteration budgets, and share most engineer prompts and the same two-round scheduling, review, and reassignment structure. Their realized assignments can differ because the manager observes different workspace states, but STORM receives no stronger decomposition prompt or scheduler. GitWorktree does not rely on a plain `git merge`: its manager inspects and semantically integrates the engineers' changes, tests the combined repository, and can repair or reassign failed work. Manager-level reassignment after a failed review is available to all multi-agent methods, including after a GitWorktree integration failure. Soft isolation runs the same workflow in a shared workspace with prompt-level file scopes but without STORM's state validation. For the shared-memory and CRDT baselines (Figure 3), we port the coordination mechanisms of MARBLE [Zhu et al., 2025] and

CodeCRDT [Pugachev, 2025] into the same harness, with the same base model (Claude Sonnet 4.6) and settings.

A.1 Cost accounting

All models are accessed through OpenRouter's OpenAI-compatible endpoint (`https://openrouter.ai/api/v1`). Table 2 lists the prices we use, in USD per million tokens as listed by OpenRouter in 2026. For GPT-5.5 prompts above 272K tokens, OpenRouter lists tiered prices of $10.00/M input, $1.00/M cached input, and $45.00/M output. Sonnet 4.6 also lists a $6.00/M one-hour cache-write tier; the table reports its standard $3.75/M cache-write price. The OpenHands SDK records token usage and billed cost for every call. Table 3 reports one complete STORM run on Commit0-Lite with Claude Sonnet 4.6 for both the manager and engineer agents; costs vary across runs (compare Table 4).

Table 2: Model pricing in USD per million tokens, as listed by OpenRouter in 2026.

Model	Role (OpenRouter ID)	Input	Cached input	Cache write	Output
Claude Sonnet 4.6	Agent (`anthropic/claude-sonnet-4.6`)	3.00	0.30	3.75	15.00
Qwen 3.6 Plus	Agent (`qwen/qwen3.6-plus`)	0.50	0.05	0.625	3.00
DeepSeek V4 Pro	Agent (`deepseek/deepseek-v4-pro`)	0.435	0.003625	–	0.87
GPT-5.5	PaperBench judge (`openai/gpt-5.5`)	5.00	0.50	–	30.00

Table 3: Token usage and billed cost for one complete STORM run on Commit0-Lite with Claude Sonnet 4.6 (4 engineers).

Repository	Prompt tokens (M)	Completion tokens (M)	Total tokens (M)	Cost ($)
babel	38.66	0.36	39.02	23.47
cachetools	3.82	0.07	3.88	3.26
chardet	31.61	0.28	31.89	19.40
cookiecutter	34.77	0.43	35.20	22.49
deprecated	2.37	0.05	2.42	2.18
imapclient	22.56	0.26	22.82	14.46
jinja	112.92	0.70	113.62	58.72
marshmallow	74.35	0.54	74.89	42.93
minitorch	59.37	0.52	59.88	34.59
parsel	17.65	0.22	17.88	12.17
portalocker	9.10	0.19	9.30	8.33
pyjwt	22.61	0.28	22.90	15.53
simpy	39.98	0.55	40.52	28.30
tinydb	13.87	0.17	14.04	9.21
voluptuous	44.55	0.40	44.95	27.10
wcwidth	1.92	0.05	1.97	1.80
Total	530.11	5.07	535.17	323.94

B Full experimental results

Tables 4–7 report per-instance results for all models. We highlight several patterns.

STORM excels on cross-file repositories. The largest gains over both baselines come from repositories with heavy inter-file dependencies. On Sonnet, `marshmallow` jumps from 0.0% (single) and 60.9% (GitWorktree) to **82.3%** under STORM; `imapclient` from 9.7% / 37.8% to **89.1%**; and `jinja` from 0.0% / 5.8% to **47.1%**. On Qwen, `babel` shows the most dramatic improvement: 3.6% (single) and 0.2% (GitWorktree) to **74.2%** under STORM, demonstrating that shared-workspace coordination is critical for large, tightly coupled codebases.

Single-agent strengths. The single agent remains competitive on small, self-contained repositories (`chardet`, `deprecated`, `parsel`) where decomposition overhead outweighs parallelism benefits.

Table 4: Per-repository pass rate, cost, and time (seconds) for Claude Sonnet 4.6. Best pass rate per row in bold.

	Single-Agent			**GitWorktree (4 Eng)**			**STORM (4 Eng)**		
Repository	Rate	Cost	Time	Rate	Cost	Time	Rate	Cost	Time
babel	0.6	4.3	934	1.3	30.8	3301	**20.2**	7.9	1678
cachetools	**100.0**	0.7	249	**100.0**	2.6	657	**100.0**	1.5	400
chardet	**99.7**	7.0	768	0.3	3.2	817	22.1	18.7	2932
cookiecutter	74.9	7.2	1958	94.3	24.2	2548	**98.6**	24.8	5234
deprecated	**100.0**	1.1	698	94.7	1.4	528	**100.0**	3.4	898
imapclient	9.7	2.6	3141	37.8	6.3	1698	**89.1**	24.9	3736
jinja	0.0	4.7	665	5.8	29.2	3332	**47.1**	39.3	3733
marshmallow	0.0	5.9	1150	60.9	31.7	3162	**82.3**	32.8	3896
minitorch	**70.4**	6.8	2093	53.0	16.1	2337	**70.4**	35.9	4439
parsel	**100.0**	6.3	1568	93.2	4.3	1202	**100.0**	11.0	1974
portalocker	**100.0**	7.2	2273	**100.0**	8.8	1741	**100.0**	11.1	1727
pyjwt	**100.0**	3.5	716	5.8	2.2	639	**100.0**	18.6	1783
simpy	58.6	0.5	2109	78.6	6.2	1627	**100.0**	21.9	2655
tinydb	**99.5**	2.5	940	**99.5**	10.3	1353	**99.5**	11.1	1474
voluptuous	49.7	4.2	3621	**96.0**	32.3	4928	90.6	25.9	9181
wcwidth	**100.0**	1.5	442	**100.0**	2.4	956	**100.0**	2.9	869

Table 5: Per-repository pass rate, cost, and time (seconds) for Qwen 3.6 Plus. Best pass rate per row in bold.

	Single-Agent			**GitWorktree (4 Eng)**			**STORM (4 Eng)**		
Repository	Rate	Cost	Time	Rate	Cost	Time	Rate	Cost	Time
babel	3.6	3.7	1709	0.2	2.0	557	**74.2**	30.1	5502
cachetools	**100.0**	0.6	487	**100.0**	3.2	966	**100.0**	2.2	927
chardet	**99.7**	3.0	1159	**99.7**	3.2	966	38.8	12.1	2350
cookiecutter	38.7	3.0	1485	36.5	8.6	4234	**56.4**	9.2	3733
deprecated	**100.0**	1.0	1135	**100.0**	3.8	2369	**100.0**	2.1	1124
imapclient	**100.0**	2.3	1193	0.0	12.1	4801	44.6	19.2	4441
jinja	**98.8**	0.6	366	0.0	2.6	744	0.0	2.6	744
marshmallow	**28.5**	3.9	2714	0.0	15.1	3106	23.7	7.7	2386
minitorch	33.9	4.3	1664	**52.2**	8.0	2463	30.4	5.4	1551
parsel	**100.0**	1.4	591	6.3	1.1	546	82.0	10.7	3239
portalocker	83.3	0.7	3623	**100.0**	4.0	8099	**100.0**	5.9	4074
pyjwt	97.3	4.3	1866	89.2	14.8	6453	**100.0**	13.9	3814
simpy	76.4	3.8	3132	51.4	12.8	8926	**81.4**	8.7	3528
tinydb	88.1	3.0	1365	**99.5**	9.4	2940	96.0	7.0	3550
voluptuous	55.7	4.5	1346	83.2	10.9	2491	**100.0**	12.2	2685
wcwidth	**100.0**	3.9	2599	**100.0**	4.5	6393	**100.0**	4.0	2544

On Sonnet, `chardet` scores 99.7% single vs. 22.1% STORM, showing that STORM's decomposition can occasionally hurt when the repository is better solved monolithically.

PaperBench patterns. On PaperBench Code-Dev (Table 7), STORM leads on 11 of 20 papers, GitWorktree on 5, and single-agent on 4. STORM's largest wins come on papers requiring substantial code organization (`what-will-my-model-forget`: 99.8 vs. 82.9 single, `lbcs`: 95.6 vs. 84.1 single). GitWorktree wins on papers where independent sub-tasks map cleanly to separate files (`sample-specific-masks`: 98.2 vs. 72.8 STORM), suggesting that when task boundaries align perfectly with file boundaries, isolation incurs no penalty.

Table 6: Per-repository pass rate, cost, and time (seconds) for DeepSeek V4 Pro. Best pass rate per row in bold.

	Single-Agent			**GitWorktree (4 Eng)**			**STORM (4 Eng)**		
Repository	Rate	Cost	Time	Rate	Cost	Time	Rate	Cost	Time
babel	1.2	3.9	3078	0.7	7.3	6153	**19.3**	4.9	6344
cachetools	71.2	0.1	1275	**100.0**	1.1	2016	**100.0**	1.3	1561
chardet	**99.7**	2.6	2044	0.3	0.9	987	1.3	3.9	3824
cookiecutter	**69.5**	4.3	3627	34.3	4.1	2927	35.1	3.0	2569
deprecated	**100.0**	0.5	3308	9.9	0.3	399	**100.0**	1.6	2148
imapclient	16.9	3.5	3622	9.7	3.2	4337	**49.1**	6.4	6320
jinja	**29.3**	5.0	3347	3.3	13.6	6531	5.8	11.7	6778
marshmallow	38.6	5.2	3282	**49.0**	12.8	4745	47.9	9.3	5402
minitorch	36.1	3.4	3636	**51.7**	3.9	2613	44.3	4.4	3018
parsel	97.6	3.3	2861	**100.0**	4.7	3434	**100.0**	12.5	8645
portalocker	36.1	0.1	3621	**97.2**	2.5	5894	**97.2**	3.1	3708
pyjwt	88.8	4.4	3625	**91.9**	4.2	2214	81.1	6.3	6668
simpy	**73.6**	3.2	3625	52.9	6.6	4507	66.4	6.9	4990
tinydb	99.5	2.4	2188	**100.0**	4.5	3086	96.5	4.2	4873
voluptuous	**85.9**	4.3	3621	0.0	-	-	67.8	15.2	6271
wcwidth	**100.0**	0.7	3513	2.6	0.3	277	**100.0**	2.1	2487

Table 7: Per-paper scores, cost, and wall-clock time for Claude Sonnet 4.6 on PaperBench Code-Dev. Best score per row in bold.

	Single-Agent (100 iter)			**GitWorktree (2 Eng)**			**STORM (2 Eng)**		
Paper	Score	Cost	Time	Score	Cost	Time	Score	Cost	Time
sample-specific-masks	93.9	28.40	2721	**98.2**	38.81	5555	72.8	22.64	6156
bam	97.4	5.34	1832	97.7	110.97	7067	**97.9**	14.27	3806
what-will-my-model-forget	82.9	207.31	3601	94.7	215.28	3149	**99.8**	315.79	3937
pinn	81.0	42.22	3600	**91.1**	20.61	6459	69.6	10.58	5328
sequential-neural-score-estimation	**92.6**	28.33	2060	88.7	35.29	4412	91.0	39.91	6699
mechanistic-understanding	**94.4**	15.69	2558	81.9	17.66	4404	61.9	17.64	6246
test-time-model-adaptation	58.5	5.75	2084	**81.4**	36.87	3694	80.8	40.42	4666
robust-clip	51.1	21.03	1843	78.4	34.59	3677	**82.6**	45.45	4029
fre	72.5	68.11	1438	78.0	79.55	2971	**78.4**	20.49	5448
stochastic-interpolants	78.4	13.66	1902	76.4	18.47	2423	**79.3**	22.63	4888
adaptive-pruning	43.6	34.99	2863	72.7	45.88	6724	**72.8**	26.60	5688
all-in-one	**87.3**	27.88	2991	72.0	46.90	5894	72.6	20.66	6389
stay-on-topic-with-classifier-free-guidance	67.5	22.92	2651	**70.4**	29.61	5604	69.2	38.31	4933
bbox	43.8	34.22	2061	**68.0**	46.35	3438	50.5	12.44	5443
lca-on-the-line	72.0	62.13	1791	66.6	147.78	5730	**75.7**	25.93	4899
ftrl	53.5	37.82	1965	63.4	51.09	5506	**66.1**	26.62	5302
rice	41.0	44.08	1934	59.7	76.31	7122	**61.9**	70.25	6730
bridging-data-gaps	42.1	5.37	1719	52.9	35.81	5958	**67.9**	43.24	4865
lbcs	84.1	137.75	3369	46.7	145.55	6729	**95.6**	19.75	6491
sapg	**35.7**	16.09	2710	14.6	15.48	1567	34.8	28.80	6434

Table 8: Per-paper scores, cost, and wall-clock time for Qwen3.6-Plus on PaperBench Code-Dev. Best score per row in bold.

	Single-Agent (100 iter)			**GitWorktree (2 Eng)**			**STORM (2 Eng)**		
Paper	Score	Cost	Time	Score	Cost	Time	Score	Cost	Time
pinn	72.5	1.64	1860	77.4	5.99	4671	**81.2**	4.92	6043
lbcs	43.3	2.66	2138	77.2	17.72	6564	**88.5**	2.90	4631
sample-specific-masks	**78.7**	2.35	2499	75.9	4.89	4847	68.7	3.20	4604
stochastic-interpolants	74.7	1.63	1553	71.7	4.79	3313	**76.8**	2.47	2525
what-will-my-model-forget	27.4	1.84	2785	**69.2**	4.17	5456	55.2	3.39	2079
mechanistic-understanding	70.6	2.36	2536	65.0	6.39	6377	**84.4**	3.86	6889
bridging-data-gaps	30.0	1.88	2366	**62.9**	7.84	6070	50.2	5.96	3078
lca-on-the-line	52.1	2.90	2712	**59.2**	17.85	8140	42.9	5.08	3180
sequential-neural-score-estimation	67.8	3.26	3345	56.6	2.82	8542	**71.6**	3.42	2344
stay-on-topic-with-classifier-free-guidance	**58.8**	2.05	3007	50.6	6.43	4746	52.9	3.13	3420
bam	76.7	2.72	1565	50.5	5.40	4819	**76.9**	9.87	6708
fre	31.2	1.59	1326	50.4	5.27	3624	**53.0**	6.73	3556
ftrl	25.2	2.31	2491	**48.7**	11.43	6813	43.7	7.18	3204
adaptive-pruning	23.4	3.21	1912	**43.1**	5.85	5563	31.6	4.28	3885
test-time-model-adaptation	54.4	2.66	3600	40.6	4.15	4191	**64.7**	7.11	3368
rice	33.4	2.29	1708	**36.7**	5.17	4094	22.4	3.96	2147
robust-clip	13.5	2.03	1266	**30.5**	5.04	4202	29.7	4.15	2035
all-in-one	**59.5**	2.84	2614	28.1	5.33	6073	47.1	2.60	1480
bbox	21.4	1.96	1715	20.1	5.65	6254	**42.4**	2.41	1248
sapg	**39.3**	2.60	3220	18.3	5.63	7000	15.2	3.42	2967

Table 9: Per-paper scores, cost, and wall-clock time for DeepSeek-V4-Pro on PaperBench Code-Dev. Best score per row in bold.

	Single-Agent (100 iter)			**GitWorktree (2 Eng)**			**STORM (2 Eng)**		
Paper	Score	Cost	Time	Score	Cost	Time	Score	Cost	Time
sample-specific-masks	82.8	1.79	3600	**90.8**	7.22	8200	82.8	7.71	7715
pinn	87.4	2.07	3536	88.4	6.79	4730	**88.9**	4.66	3745
what-will-my-model-forget	73.1	2.69	3600	**82.3**	2.65	5533	79.7	6.37	8477
lbcs	72.5	2.01	3601	78.0	3.73	4596	**95.8**	5.64	6477
stochastic-interpolants	**81.3**	1.22	2513	69.0	3.57	4813	60.2	4.51	5717
sequential-neural-score-estimation	82.5	2.48	1873	68.7	4.30	5973	**84.2**	5.52	8880
bbox	39.2	2.42	3600	**65.6**	4.33	4136	53.3	4.71	6289
bam	**91.9**	4.37	3600	65.3	5.57	7399	79.1	6.48	7407
fre	**65.0**	2.58	2112	64.4	8.38	8058	55.8	8.77	8355
all-in-one	87.8	1.87	3570	63.8	9.28	6804	**97.6**	8.63	8035
test-time-model-adaptation	60.2	2.34	1875	**62.9**	2.86	4533	57.2	3.61	5615
adaptive-pruning	37.6	3.28	3600	47.1	6.79	8182	**53.5**	4.98	5644
bridging-data-gaps	45.7	3.54	3600	45.7	6.15	6034	**58.1**	5.80	6667
ftrl	41.5	2.80	3600	**45.1**	4.81	5377	43.6	8.77	6679
mechanistic-understanding	76.1	2.42	3226	42.8	2.53	5204	**88.3**	8.05	6821
lca-on-the-line	47.1	2.80	3408	36.5	4.44	6418	**75.8**	8.53	6330
robust-clip	37.0	2.13	2633	33.6	5.40	6135	**65.1**	11.68	8752
stay-on-topic-with-classifier-free-guidance	62.7	3.31	2579	29.7	3.25	7393	**72.2**	4.62	6227
rice	**51.8**	3.30	2954	28.1	2.82	5696	20.3	6.45	7823
sapg	**34.1**	2.18	3225	8.6	3.24	4474	19.2	3.51	9851

C Additional Analyses

C.1 Repeated runs and judge sensitivity

Table 10 repeats the Claude Sonnet 4.6 configurations three times on both benchmarks and reports the mean and sample standard deviation. On Commit0-Lite, STORM leads GitWorktree by 21.3 Score_w and 16.0 Score points, and its standard deviation is at most 0.3 points, the smallest of the three standalone methods. For PaperBench, we re-judge the original Sonnet trajectories and two additional runs with GPT-5.5, which is used only as the judge; STORM keeps the highest mean (72.3 vs. 67.7 for GitWorktree and 65.7 for single-agent).

Table 10: Repeated runs with Claude Sonnet 4.6: mean $\pm$ sample standard deviation over three complete evaluations. Commit0-Lite uses 4 engineers. PaperBench Code-Dev uses 2 engineers and is judged by GPT-5.5, a different model family from the agents.

	Commit0-Lite		**PaperBench (GPT-5.5 judge)**
Method	Score_w ↑	Score ↑	Score ↑
Single-Agent	21.70 ± 1.41	$\underline{69.90 \pm 4.95}$	65.71 ± 2.74
GitWorktree	$\underline{25.11 \pm 0.71}$	66.30 ± 3.53	$\underline{67.74 \pm 0.95}$
STORM	$\mathbf{46.40 \pm 0.28}$	$\mathbf{82.29 \pm 0.30}$	$\mathbf{72.27 \pm 2.28}$

C.2 Coordination baselines and component ablations

Isolation strategy. In Table 11, we compare isolation strategies on Commit0-Lite with Claude Sonnet 4.6 using 4 engineers. Soft isolation keeps STORM's shared workspace, manager loop, and budgets, and gives each engineer prompt-level file scopes, but removes write-time version and snapshot validation; it is therefore also the ablation of STORM without local state validation. Prompt-based soft isolation and git worktree isolation achieve similar results (Score_w 24.0 vs. 24.5, Score 65.4 vs. 63.8), both improving over the single-agent baseline in weighted score (20.7) while remaining comparable in unweighted score. This suggests that delegation alone provides gains on larger repositories where multiple agents can cover more test cases in parallel, but the choice between instruction-level constraints and physical branch separation has limited impact on final pass rate. STORM outperforms both isolation strategies (Score_w 46.2, Score 82.5) despite using the same number of engineers. First, STORM detects conflicts at write time rather than deferring them: when an edit violates local state consistency (Eq. 3), it is rejected immediately, letting the agent re-plan while its reasoning context is still fresh. Soft isolation instead relies on instruction-level constraints that are frequently violated, causing silent overwrites, while worktree isolation defers conflicts to a merge step where multi-file resolution often fails. Second, because all engineers share a single workspace, any read returns the latest committed state, including other engineers' recent changes. Agents working on adjacent modules naturally observe each other's implementations and can adapt interfaces accordingly, whereas worktree isolation keeps engineers blind to concurrent progress until merge time, by which point incompatible decisions may have already propagated.

Table 11: Effect of isolation strategy on Commit0-Lite with Claude Sonnet 4.6 and 4 engineers (one run per configuration, as in Table 1).

Method	Score_w ↑	Score ↑	Cost_eff ↓	Time_eff ↓
Single-Agent	20.7	$\underline{66.4}$	**3.2**	$\underline{18.8}$
Soft Isolation	24.0	65.4	7.7	29.1
GitWorktree	$\underline{24.5}$	63.8	8.6	20.9
STORM	**46.2**	**82.5**	$\underline{6.3}$	**16.8**

Local state validation. Soft isolation is STORM without write-time version and snapshot validation, with the manager-led decomposition, delegation, review, testing, and integration loop held fixed. Adding validation raises Score_w by 22.2 points and Score by 17.1 points, so the manager review loop alone does not explain the gain.

Intent annotations. Removing intent annotations while keeping validation, the manager, four engineers, and the same budgets lowers Score_w by 16.9 points and Score by 8.8 points. We read this as outcome-level evidence that annotations improve coordination, not as evidence that agents use every annotation correctly.

Other components. The reservation (Section 2.2) is a liveness safeguard and does not change which writes are accepted. The rejection payload is returned within the same `file_editor` call, so STORM has no separate retry planner. Manager decomposition and manager-level reassignment are shared with GitWorktree and soft isolation (Appendix A), so neither is a STORM-specific advantage.

C.3 Validation of the coupling proxy

The coupling score in Section 3.4 is a heuristic. To validate it independently, we analyze the unmodified repositories before any agent runs and measure three static properties: the number of source files, the number of tests, and the number of internal import edges. We correlate each with a rejection-free variant of the coupling index and with STORM's improvement over GitWorktree. Neither the static measures nor this coupling index uses write-rejection or test-result information. We use the gain over GitWorktree because that comparison directly contrasts shared-state coordination with workspace isolation under the same model, prompts, manager, scheduling, agent count, and budgets.

Table 12: Spearman correlations between static repository properties and (i) the rejection-free coupling index and (ii) STORM's pass-rate gain over GitWorktree, on the 16 Commit0-Lite repositories with Claude Sonnet 4.6. FDR-corrected q-values in parentheses.

Static repository property	Coupling index: ρ (q)	STORM gain over GitWorktree: ρ (q)
Number of source files	$+0.555$ (0.039)	$+0.622$ (0.030)
Number of tests	$+0.666$ (0.019)	$+0.584$ (0.032)
Internal import edges	$+0.659$ (0.019)	$+0.637$ (0.030)

All three properties agree (Table 12): repositories with more files, tests, and internal dependencies receive higher coupling scores, and STORM's improvement over GitWorktree is larger on those repositories. All six correlations remain significant after FDR correction and stay positive when any single repository, including `babel`, is removed. This indicates that the coupling index captures real repository scale and dependency structure rather than restating observed conflicts. With only 16 repositories, we still treat it as a proxy, not a ground-truth or causal measure.

D Failure analysis details

The failure analysis clarifies the boundary of what file-level state management can and cannot guarantee, restricted to STORM runs at $k{=}4$ and $k{=}8$. Failed tests are categorized from pytest traceback symptoms into coarse buckets: assertion or semantic failures, missing API or symbol failures, type or contract errors, not-implemented errors, and other runtime failures (Figure 6, left). These categories are directly derived from tracebacks, but they remain symptom labels rather than manually adjudicated root causes.

Run-level cause tags are likewise heuristic proxies (Figure 6, right). *Incomplete API* is inferred from missing-symbol or not-implemented failures. *Scope drift* is inferred when accepted writes modify files outside the task's declared scope. *Budget/runtime* is inferred from explicit agent errors such as max-iteration or timeout failures. *Accepted same-file overlap* is inferred when multiple task ids have accepted writes to the same file in a failed run. Failed tests themselves are dominated by assertion mismatches, missing APIs, and type or contract errors, indicating that unsuccessful runs still produce substantial but behaviorally incorrect implementations. Among failed STORM runs, cross-module scope drift and accepted same-file overlap appear in nearly all failed cases, and budget or runtime failures remain common at both $k{=}4$ and $k{=}8$. These proxies explain why STORM is not sufficient by itself: many remaining failures arise after writes have already been accepted as file-version consistent. STORM can ensure that accepted writes are based on a current workspace state, but it cannot guarantee that the chosen task boundaries are semantically correct, that independently accepted edits compose cleanly, or that agents complete within budget.

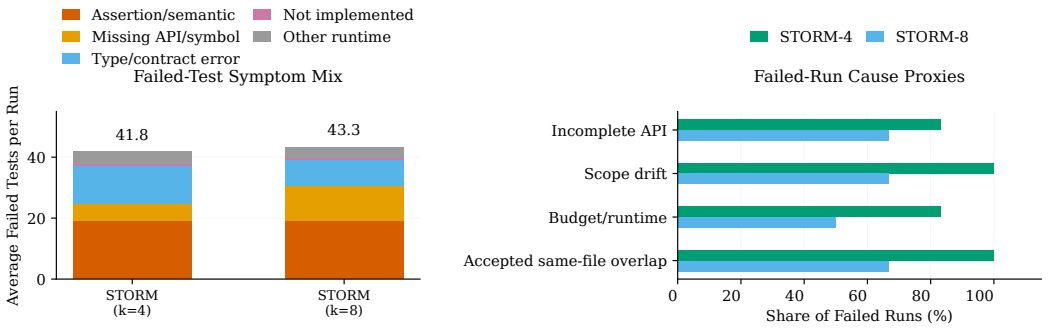


Figure 6: Failure analysis for STORM runs at k=4 and k=8. Left: average failed-test symptom mix per run, decomposed into assertion/semantic, missing API/symbol, type/contract, not-implemented, and other runtime failures. Right: share of failed runs in which each run-level cause proxy fires, covering incomplete API, scope drift, budget/runtime, and accepted same-file overlap.

E Extended Related Work

Code Generation for Software Development LLM-based code generation has moved from function-level synthesis to broader software development tasks. Recent code generation agents can decompose tasks, write code, use tools, run tests, and debug failures [Jiang et al., 2024, 2025, Ma et al., 2025]. Repository-level methods further retrieve and use cross-file context from existing projects, including APIs, dependencies, tests, and coding conventions [Zhang et al., 2023, Shrivastava et al., 2023]. Benchmarks such as CrossCodeEval and DevEval show that realistic code generation requires cross-file reasoning and repository-level dependency understanding [Ding et al., 2023, Li et al., 2024].

However, code generation alone is not enough for reliable software development. Real tasks often require long-horizon interaction, repeated testing, debugging, and revision. Existing methods mainly improve planning, retrieval, or generation quality [Jiang et al., 2026, Li et al., 2026]. They provide limited support for multi-step collaboration, state control, and conflict management across agents. Execution-grounded agents add test feedback to the generation loop [Kumar et al., 2026, Dong et al., 2025b], but reliable coordination across long software development workflows remains an open problem.

F Limitations

Terminal bypass. STORM mediates the `file_editor` tool but not direct filesystem writes through bash (`sed`, `echo >`, Python scripts). A post-hoc diff mechanism detects these but cannot reject them preventively.

No command coordination. Concurrent shell commands (e.g., two agents running formatters on overlapping files) are not serialized. Extending concurrency control to arbitrary terminal side effects remains open.

File-level granularity. Two agents editing different functions in the same file trigger a false-positive rejection. Heavily shared files (e.g., `__init__.py`) become serialization bottlenecks. Line-level or hunk-level tracking would reduce this at the cost of managing shifting offsets after each edit.

Scope of the consistency guarantee. STORM's guarantee covers only the files an agent has read; it cannot detect a semantic dependency on a file outside the observed read set. This is not specific to coordination, since a dependency an agent never inspects is equally invisible to single-agent and worktree-based agents. In STORM, testing, manager review, and replanning progressively expose such dependencies: once an agent reads a file during diagnosis, it enters the read set and later writes are validated against it. Because test coverage can be incomplete, STORM does not infer every latent semantic dependency or guarantee repository-wide semantic correctness.

Intent annotations. Annotations provide no formal semantic guarantee and can communicate incomplete or incorrect rationale between agents. The ablation in Appendix C.2 shows an outcome-level benefit, not that every annotation is read or used correctly.

High contention. Within the scale and dependency density covered by Commit0-Lite, we see no evidence that write rejection overwhelms the coordination benefit: STORM's advantage over GitWorktree grows with source-file count, test count, and internal import edges (Appendix C.3). Our benchmarks do not establish behavior beyond this range. At sufficiently high contention, repeated invalidations could make retry overhead dominate; the reservation prevents livelock but does not bound this cost.

Evaluation scope. Our baselines are implementations of worktree isolation, soft isolation, shared memory, and CRDT coordination in a common harness; they do not cover every isolation-based, shared-memory, or communication-heavy system. Repeated runs are reported for Sonnet 4.6 only, and the Qwen and DeepSeek results are single runs. With those two models, STORM improves the weighted score but not the macro score on Commit0-Lite, so its benefit is concentrated on larger repositories. PaperBench is evaluated only on the Code-Dev subset and relies on LLM judges (Sonnet 4.6, and GPT-5.5 for the repeated Sonnet runs).

G Prompt templates

We summarize the key prompts used in the manager-engineer protocol. The full prompt text is available in the released code.

Manager scan instruction (excerpt):

Start by exploring the repository structure to understand what your team needs to work on and identify which functions have top priority to be implemented (i.e., those with `pass` statements). To explore the repository structure and dependencies:

1. Check the imports and the actual functions used in the files with `pass` statements and review the relevant tests to understand the EXPECTED BEHAVIOR of the functions and the dependencies between the files.

2. Run `pytest -collect-only -continue-on-collection-errors` to have a better understanding of the scope and the dependencies of this codebase.

Collect all the undefined functions from the test collection errors and add them with clear docstrings and `pass` statements into the files and make a local commit. DO NOT make any changes to the functions with `pass` statements since those need to be implemented by the engineers.

Manager delegation.

Manager delegation instruction (excerpt):

Split the overall implementation work into up to k major tasks, balancing complexity and estimated effort as evenly as possible. Make sure the high dependent files are in the same major task. Try to split the major tasks at file level first; if a single file contains a disproportionately large amount of functions with `pass` statements, you can delegate at the function level and assign non-overlapping sets of functions to multiple engineers.

For each engineer, assign the first task that has the highest priority within their major task. When you provide instruction to each engineer, briefly summarize the relevant repository structure and the dependencies so they don't need to re-explore the repository. Then clearly specify which file/functions to implement and explain the purpose of this task. If the assigned functions depend on other stub functions, include a brief description of what each dependency does.

Engineer task prompt. Each engineer receives the repository structure summary, its assigned functions, dependency descriptions, the test command, and shared-workspace rules. The key constraint is shown below:

Engineer task prompt (excerpt):

You are a software engineer working on implementing a python code repository in a group. You are responsible for implementing the functions instructed by your manager (i.e., the functions with `pass` statements) and passing the unit tests.

Shared Workspace: Your workspace is `{worktree_path}`. ALL engineers on your team share this SAME directory. DO NOT modify files that belong to other engineers. Only edit the specific files and functions assigned to you. Check before modifying any file to make sure another engineer hasn't already changed it.

Scope Discipline: You are ONLY responsible for implementing the functions listed below. If you see test failures caused by functions in OTHER files or functions NOT assigned to you, DO NOT attempt to fix them. Report the failure to your manager and move on.

Multi-agent coordination: Every edit you make MUST include a one-line comment in the form `# {engineer_id}: <short intent>` IMMEDIATELY above the block you added or changed. If you see `# <other-engineer-id>: ...` comments, preserve both the comment and the block below it unless your task explicitly requires changing them.

You are assigned to implement the following functions in the file: `{file_path}`: `{functions}`. After you finished the implementation, make sure it will not cause any hanging issues. Do NOT commit; the manager will review your changes and commit on your behalf.

Manager review and reassignment. After each round, the manager checks commit status. For successful commits, it assigns the next task. For failed commits, it reassigns the same task. In the final review, the manager merges all engineers' work, resolves integration issues (import mismatches, naming inconsistencies), and checks for hanging code (infinite loops, `input()` calls).

Manager delegation output format (excerpt):

```
{
"engineer_id": "engineer_1",
"file_path": "path/to/file.py",
"functions_to_implement": ["func1", "func2"],
"instruction": "Implement the tensor storage
layout. The TensorData class uses ..."
}
```