Effective Strategies for Asynchronous Software Engineering Agents

Anonymous Author(s)

Abstract

AI agents have become increasingly capable at isolated software engineering (SWE) tasks such as resolving issues on Github. Yet long-horizon tasks involving multiple interdependent subtasks still pose challenges both with respect to accuracy, and with respect to timely completion. A natural approach to solving these long-horizon tasks in a timely manner is asynchronous multi-agent collaboration, where multiple agents work on different parts of the task at the same time. But effective application of multi-agent systems has proven surprisingly difficult: concurrent edits by multiple agents interfere with each other, dependencies are difficult to synchronize, and combining partial progress into a coherent whole is challenging. On the other hand, human developers have long relied on mature collaboration infrastructure to manage these challenges in large software projects. Inspired by these collaboration primitives, we introduce Centralized Asynchronous Isolated Delegation (CAID), a structured multi-agent coordination paradigm grounded in three core SWE primitives: centralized task delegation, asynchronous execution, and isolated workspaces. CAID constructs dependency-aware task plans through a central manager, executes subtasks concurrently in isolated workspaces, and consolidates progress via structured integration with executable test-based verification. In empirical evaluation, we find that CAID improves accuracy over single-agent baselines by 26.7% absolute on paper reproduction tasks (PaperBench) and 14.3% on Python library development tasks (Commit0). Through systematic analysis, we find that branch-and-merge is a central coordination mechanism for multi-agent collaboration, and that SWE primitives such as `git worktree`, `git commit`, and `git merge` enable it to be realized in a reliable and executable manner.

CCS Concepts

• **Computing methodologies** → **Multi-agent systems**.

Keywords

Multi-Agent Systems, Centralized task Delegation, SWE-Primitives, Asynchronous Execution, Isolated Workspace, Merge-and-Branch, Long-Horizon SWE Tasks

ACM Reference Format:

Anonymous Author(s). 2018. Effective Strategies for Asynchronous Software Engineering Agents. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As LLM-based software engineering agents improve, we have come to expect more of them. Whereas fixing isolated github issues on real-world repositories was a major challenge a few years ago [8, 27, 31], we are now asking agents to build large apps from scratch [37] or implement entire research papers [24]. One method for performing this implementation is tasking a single agent with a large task, and hoping that it can execute on it from start to finish. While task-completion horizons of agents continue to grow rapidly [10], these systems are still limited in the scope of tasks they can perform reliably, and a single agent performing a large task also takes significant wall-clock time. To this end, in this paper, we study the question: *"how can multiple agents be coordinated to asynchronously collaborate over a shared artifact in an effective way?"*

While much research has focused on coordinating multiple agents, ranging from role-based pipelines that mirror human software engineering teams [6, 21], to hierarchical managers that decompose and delegate subtasks [2], to include verification mechanisms in multi-agent systems [25], and to automated searches over communication topologies [34]—most of these approaches primarily address how tasks are decomposed and allocated across agents. However, the core challenges of *asynchronous* multi-agent collaboration over shared artifacts remain unsolved. Agents operating in this setting face a myriad of challenges such as locally reasonably but globally consistent edits [9], lack of shared state [3], and late discovery of any conflicts [5].

Human software engineering teams face these coordination failures routinely, and they have developed a mature infrastructure to mitigate them. Developers work in isolated copies of the repository (e.g., via `git worktrees`), so parallel edits do not overwrite one another. When changes are ready, version-control integration protocols (e.g., merge-based workflows) consolidate contributions and conflicts explicitly, rather than allowing silent interference. Test suites verify each change automatically, so correctness does not rely solely on any single developer's judgment.

With this in mind, we build CAID (Figure 1), a multi-agent system grounded in SWE primitives, in which a manager agent dynamically decomposes and delegates tasks to multiple engineer agents who execute concurrently in isolated workspaces. In particular, each engineer operates in its own `git worktree`, a fully isolated workspace with a versioned copy of the repository and when an engineer finishes, its changes are integrated back through `git merge`. As in human software teams, each engineer is responsible not only for implementation, but also for executable self-verification and conflict resolution at commit time. Communication between the manager and engineers uses structured `JSON` instructions and git

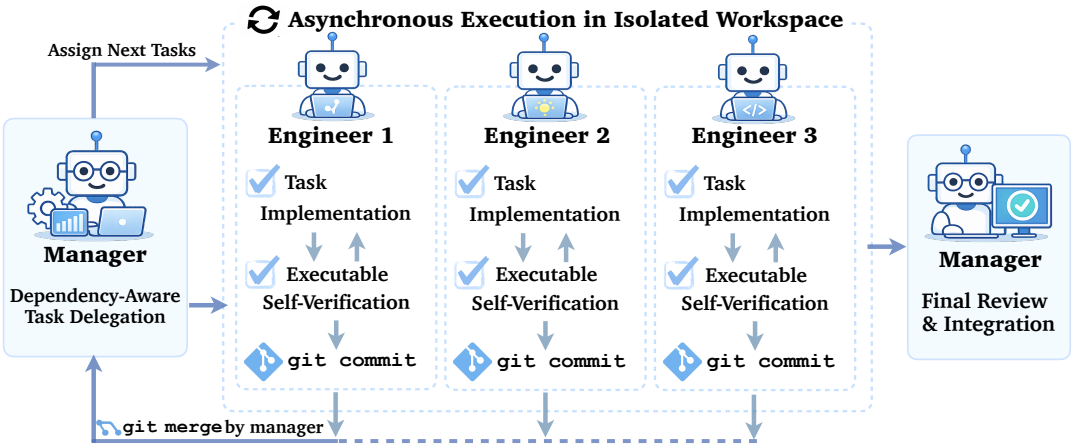

Figure 1: Overview of CAID. The Manager explores the tasks, builds a dependency graph, and creates isolated worktrees for each engineer. In parallel, engineers independently implement, self-verify, and commit. Upon completion, the Manager merges the work, dynamically updates the delegation plan, and reassigns the next task. Finally, the manager does a review of the final product.

commits rather than free-form dialog, avoiding the inter-agent misalignment that has been identified as the primary failure mode in multi-agent systems [3]. We provide further details on the design of CAID in § 2.

We evaluate CAID on two long-horizon, complex software engineering tasks that provide a natural testbed for *shared-artifact* collaboration. Specifically, we use Commit0 [37], which requires agents to implement Python libraries from scratch (e.g., `tinydb`, `minitorch`, `jinja`), and on PaperBench [24], where agents reproduce a conference paper. Together, these benchmarks allow us to evaluate CAID with the lens of branch-and-merge coordination in long-horizon multi-agent software engineering. Based on these experiments show that CAID consistently improves the performance of Commit0 and PaperBench across multiple models.

2 Branch-and-Merge Multi-Agent Coordination with SWE Primitives

CAID's coordination architecture is based on SWE primitives, which support operations such as task decomposition, isolated development, integration, and verification. In Table 1, we associate concrete SWE primitives (e.g., `git worktree`, `git merge`, dependency graphs, and test suites) and their corresponding coordination roles in CAID. CAID consists of task specification and dependency modeling (§ 2.1), dependency-aware task delegation (§ 2.2), workspace isolation and integration (§ 2.3), structured communication with asynchronous execution (§ 2.4), and self-verification with termination control (§ 2.5).

2.1 Task Specification and Dependency Graph

To perform multi-agent delegation, we need to split the overall task into ordered sub-tasks. In our preliminary experience, allowing agents to split tasks arbitrarily causes them to miss important parts as they proceed. Therefore, to proceed with task delegation in a structured way, we instead have the manager create a dependency graph of the repository to organize the work to be done. The repository structure is represented as a directed graph $G = (V, E)$, where each node $v \in V$ corresponds to a unit of work and each directed edge $(v_i, v_j) \in E$ indicates that v_j depends on v_i. Let $C_t \subseteq V$ denote the set of units that have been completed and successfully integrated into the main branch at round t. A unit v_j is eligible for delegation only if all its dependencies have been satisfied: $\texttt{Ready}_t(v_j) \iff \forall (v_i, v_j) \in E,\ v_i \in C_t$. At each round, the manager selects executable units from the ready set $\{v \in V \mid \texttt{Ready}_t(v)\}$ and converts them into task assignments. Depending on the task, the unit of work and dependency analysis method is defined differently. In § 3.1, we describe these definitions for Commit0 and PaperBench respectively. Although granularity differs across benchmarks, in both settings the manager constructs a dependency structure before delegating, and engineers are assigned tasks only after it is established.

2.2 Dependency-Aware Task Delegation

We prompt (see Appendix A.1 and A.2) the manager to convert the dependency structure from Section 2.1 into small executable task units assigned to each engineer. The manager splits implementation work into at most N major task groups, where N is the maximum number of parallel engineers, activating up to N engineers whose dependencies are satisfied (not all N are necessarily activated). Files with strong or circular dependencies are grouped together and assigned to the same engineer to reduce cross-agent coordination.

At each delegation step, the manager selects next tasks with top priority from the major task group, prioritized based on tasks that enable earlier test execution, expose more evaluation signals, or lie closer to the upstream end of the dependency chain. We suggest to the manager that engineers typically start with simpler functions before moving on to more complex ones. The manager dynamically updates dependency state after an engineer merges code and decides whether to assign the next task or keep the engineer idle. We define one round as a complete cycle of delegation, implementation,

SWE Primitive	Coordination Mechanism	Role in CAID
Dependency graph	Scheduling constraints	Dependency order determines safe task delegation
`git worktree`	Workspace isolation	Each agent works in an independent worktree
`git commit` / `git pull request`	Structured signaling	Agents report completion by making commits
`git merge`	Output integration	Completed changes are merged into the main
Merge conflict resolution	Conflict handling	Engineer resolves integration conflicts by themselves
Code review	Verification	Engineer does the self-verification
`asyncio` parallel execution	Concurrent execution	Multiple agents run concurrently
Event loop + `await`	Coordination cycle	Await completion ⟶ integrate ⟶ reassign tasks
`git reset –hard HEAD`	State synchronization	Worktrees sync to latest integrated state

Table 1: Mapping between concrete SWE primitives and multi-agent coordination mechanisms in CAID. Each primitive serves as an operational building block for isolation, delegation, asynchronous execution, and integration.

and dependency update. The process continues until no executable task groups remain or execution limits are reached.

2.3 Workspace Isolation and Integration

We use `git worktree` to ensure that each engineer modifies files only within its workspace, which is always derived from the main branch. Before delegation, we ask the manager to set up the repository in an executable state, including preparing the runtime environment, organizing entry points, or adding minimal function stubs when required by the task. These preparatory changes are committed to the main branch so that all subsequent engineer branches are created from a consistent base state. Certain shared files, such as package initialization files (e.g., `__init__.py`), are marked as restricted, and engineers are explicitly instructed not to commit changes to them. Worktrees are deleted after all assigned tasks are completed or when the engineer reaches the predefined iteration limit.

Integration is performed through standard `git commit` and `git merge` operations. After completing implementation and self-verification, an engineer submits a commit from its branch. The manager attempts to merge this branch into the main branch. If a merge conflict occurs, the engineer who produced the conflicting commit is responsible for resolving it. To solve the conflict, we ask the engineer to pull the latest main branch into its worktree, resolve conflicts locally, and resubmit the updated commit. As the results, the main branch remains the single source of integrated state throughout execution. We observe that this branch-based isolation, combined with explicit merge responsibilities, prevents parallel development from corrupting the shared codebase.

2.4 Communication and Asynchronous Execution

We use a structured JSON protocol as the communication interface between the manager and the engineer agents. When delegated the task, the manager outputs a JSON specification that defines task assignments, file paths, target functions, and dependency information to ensure that the task boundaries, responsibilities, and outputs are explicitly defined and can be programmatically validated. We provide the details in Appendix A.1.

The execution is organized around an asynchronous manager-controlled event loop. Once tasks are delegated, each engineer operates as an independent coroutine. Engineers invoke language model calls, modify code in their worktrees, and execute verification commands such as running tests. These operations are executed concurrently up to a predefined maximum number of active engineers. The manager listens for completion signals and dynamically updates the dependency state when commits are submitted. Engineers who finish early can be assigned new executable task units, while engineers whose dependencies are not yet satisfied remain idle. To manage context growth, the manager maintains a compressed execution history. We use `LLMSummarizingCondenser` to periodically summarize prior interaction rounds while preserving key structured artifacts such as the dependency graph, completed tasks, and unresolved errors. This separation prevents unnecessary context expansion while preserving execution traceability.

2.5 Self-Verification and Termination

To ensure implementation quality, we require each engineer to perform verification before submitting a commit. If executable tests are available, the engineer runs the subset of tests that directly import or reference the modified files. If there is no explicit mapping, the engineer runs the repository's default test command or a minimally runnable entry point. Any failed test or runtime exception must be resolved before submission, and engineers iteratively refine the implementation using concrete error logs and tracebacks. After a verified commit is submitted, the manager integrates it into the main branch and updates the dependency state. The manager does not perform a detailed code review at every step, but monitors the overall progress and remaining implementation units. We terminate execution when all units in the dependency structure have been completed and integrated, or when predefined limits, such as maximum rounds or iteration budgets, are reached.

PaperBench										
		Single-Agent			CAID (2 Engineers)			Single-Agent + CAID		
Model	**SDK**	Score	Runtime	Cost	Score	Runtime	Cost	Score	Runtime	Cost
Claude Sonnet 4.5	v1.11.0	57.2	1803.5	3.3	63.3	2080.4	9.3	**66.8**	3883.9	12.6
MiniMax 2.5	v1.11.0	10.4	2525.3	1.1	36.7	2999.4	2.6	**36.7**	5524.7	3.7
GLM 4.7	v1.11.0	38.0	1177.6	2.8	45.4	1449.4	4.7	**48.5**	2626.9	7.5
Commit0-Lite										
		Single-Agent			CAID (4 Engineers)			Single-Agent + CAID		
Model	**SDK**	Score	Runtime	Cost	Score	Runtime	Cost	Score	Runtime	Cost
Claude Sonnet 4.5	v1.11.0	53.1	692.6	1.9	59.1	1583.2	8.1	**59.5**	2275.8	10.1
MiniMax 2.5	v1.11.0	42.3	752.1	1.6	**57.0**	1908.6	4.5	57.0	2660.7	6.1
GLM 4.7	v1.11.0	42.8	871.0	2.5	**46.5**	1387.8	7.3	46.5	2257.8	9.8

Table 2: Main results on Commit0 and PaperBench. We compare single-agent baselines with CAID under the same model and iteration budget.

3 Main Results

3.1 Evaluation Benchmarks

We evaluate CAID on two long-horizon software engineering benchmarks.

Commit0 [37] tests whether agents can implement a Python library from scratch given a repository skeleton and a suite of unit tests. The task is considered successful only if all tests pass, making it a repository-level integration problem rather than a collection of independent code completions. We use Commit0-Lite as our primary evaluation set. In Commit0, the manager receives an instruction and a repository path containing executable tests (Appendix A.1). It first checks import statements for file-level dependencies, collects test cases, and examines which files those tests exercise. The manager is instructed to first consider file-level delegation, but if a single file contains many unimplemented functions it can further divide work at the function level. After assigning initial tasks to multiple engineers, the manager continues exploring the repository until one engineer completes its tasks, submits a commit for merge, and is ready for the next task.

PaperBench [24] evaluates an agent's ability to reproduce the main contributions of a published conference paper, typically involving multi-step implementation, experimental setup, and result verification. The benchmark emphasizes long-horizon reasoning and structured execution over complex codebases. Due to computational cost constraints, we adopt the `Code-Dev` evaluation protocol instead of running the full evaluation pipeline. Following the benchmark's evaluation paradigm, we use `gpt-5-mini` [18] as the judge model to assess functional correctness and completion quality. As an open-ended task, explicit test-to-file mappings are not always available. The manager reads the paper, considers the main contribution as the central implementation objective, and infers the required implementation order from it. We provide the prompt in Appendix A.2.

3.2 Experimental Setup

We build CAID using the open-source OpenHands agent SDK [27, 28] (v1.11.0), instantiating a centralized manager for dependency-aware task delegation and multiple software-engineer agents in isolated workspaces. We evaluate with three language models: two open-source (`GLM 4.7` [33] and `MiniMax 2.5` [16]) and one closed-source (`Claude-4.5-Sonnet` [1]). Following the Commit0 leaderboard[1] configuration, we use a single-agent setup with `max_iterations` = 100 on both Commit0 and PaperBench. For multi-agent runs, we set `max_iterations` = 50 for the manager and `max_iterations` = 80 for each engineer agent, with 2 implementation rounds. In the main results, we use one manager with 2 engineers on PaperBench and 4 on Commit0. Detailed analysis of configuration choices is in Section 4.[2]

3.3 Baselines

Our primary baseline is a single-agent system built on the same OpenHands agent, isolating the effect of branch-and-merge coordination while holding the underlying framework fixed. This controlled comparison measures the incremental contribution of

[1] https://commit-0.github.io/

[2] All configurations are fixed prior to experimentation to balance correctness and runtime efficiency.

dependency-aware delegation, isolated workspaces, and merge-and-branch integration without introducing variation from framework-level differences such as prompting structure, tool interfaces, memory mechanisms, or execution policies. In Section 4, we further vary coordination and isolation mechanisms to compare different multi-agent architecture design choices.

3.4 Branch-and-Merge Based Coordination Improves Multi-Agent Performance

We compare CAID with the single-agent baseline in Table 2 and observe a consistent advantage for the branch-and-merge-based multi-agent system across both benchmarks and three LLMs. On PaperBench, we observe that multi-agent coordination yields large gains for weaker single-agent runs: MiniMax 2.5 reaches 36.7% under multi-agent execution, while its single-agent score is only 10.4%. The improvement is not limited to weaker models. With Claude 4.5, multi-agent execution achieves 63.3% compared to 57.2% for single-agent. In Commit0-Lite, we find the same pattern. Claude 4.5 improves from 53.1% to 59.1%, and MiniMax 2.5 reaches 57.0% under multi-agent execution. These results indicate that the performance gap is not explained by changing the underlying model, but by changing the execution method. In CAID, engineers work in separate branches and changes enter the main branch only through explicit merge and test validation. This makes parallel work usable by separating implementation from integration: engineers can iterate locally without overwriting each other's intermediate states, while integration failures are surfaced at merge time with concrete test signals tied to specific updates. Our results in Table 2 are consistent with the benefit of making integration explicit and test-gated under long-horizon execution. We provide one-sided t-tests in Appendix C.

Table 2 further reveals an important strategic implication. In long-horizon shared-artifact tasks, multi-agent coordination should not be treated as a fallback after single-agent failure. The Single-Agent + Multi-Agent setting approximates a practical strategy in which a single agent is first attempted, followed by coordinated execution if necessary. However, this sequential strategy incurs nearly additive runtime and cost, while the final performance remains close to the direct multi-agent result. For example, on PaperBench with Claude Sonnet 4.5, the combined strategy reaches 66.8%, only slightly above the multi-agent score of 63.3%, yet runtime increases from 2080.4s to 3883.9s and cost rises from 9.3 to 12.6. On Commit0-Lite with MiniMax 2.5, the multi-agent score is 57.0%, and the combined strategy remains 57.0%, while both runtime and cost increase substantially. These results give a clear strategy insight: treating multi-agent coordination as a fallback is inefficient. A more cost-effective strategy is to adopt coordinated multi-agent execution from the outset rather than switching only after failure.

3.5 Single Agents Fail to Utilize More Iterations

Can a single agent overcome long-horizon shared-artifact challenges simply by running longer? To study this, we run a single agent with $max_iterations = 100$ and $max_iterations = 200$. We control computation through a max iteration budget rather than enforcing a fixed runtime, which better reflects practical agent deployment where iteration-based control is commonly used. As

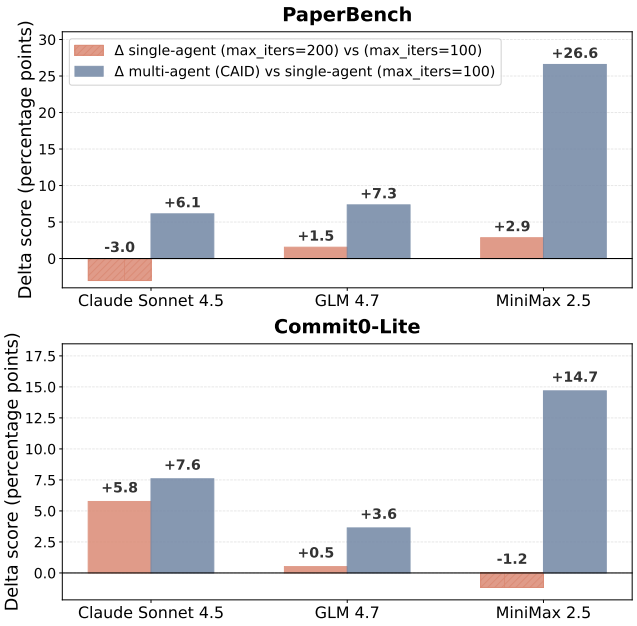

Figure 2: CAID effectively utilizes iteration budgets. We compare the final score and the iteration utilization between single-agent runs with different iteration limits and CAID.

shown in Figure 2, doubling the iteration limit yields only marginal improvements and, in some cases, even degraded results. In PaperBench, Δ from 100 to 200 iterations remains small for GLM 4.7 and MiniMax 2.5, and becomes negative for Claude Sonnet 4.5. In Commit0-Lite, the improvement is similarly limited, and MiniMax 2.5 shows a negative delta. This trend is consistent with the findings in PaperBench, where forcing the agent to run until a time limit does not reliably improve the judge score [24]. In Figure 2, we also show the score gain of CAID relative to the 100-iteration single-agent baseline. Across both benchmarks, these gains are substantially larger than those from increasing the iteration budget. For example, on PaperBench the multi-agent improvement for MiniMax 2.5 exceeds 25 percentage points, while doubling iterations yields only a small change. A similar gap appears in Commit0-Lite. These results show that extending the iteration budget alone does not resolve the fundamental bottleneck of a single agent on long-horizon tasks, whereas multi-agent coordination produces significantly larger gains.

4 Analysis

4.1 `Git worktree` Isolation

In Table 3, we study whether our proposed method of "worktree isolation" is necessary, comparing it with "soft isolation", where all engineers share one workspace, and the central manager attempts to prevent conflicts through instruction-level constraints, such as assigning non-overlapping files and explicitly warning against interference. On Commit0-Lite, soft isolation improves over single-agent from 53.1% to 56.1%, showing that central manager-driven delegation alone already helps when repository structure and file

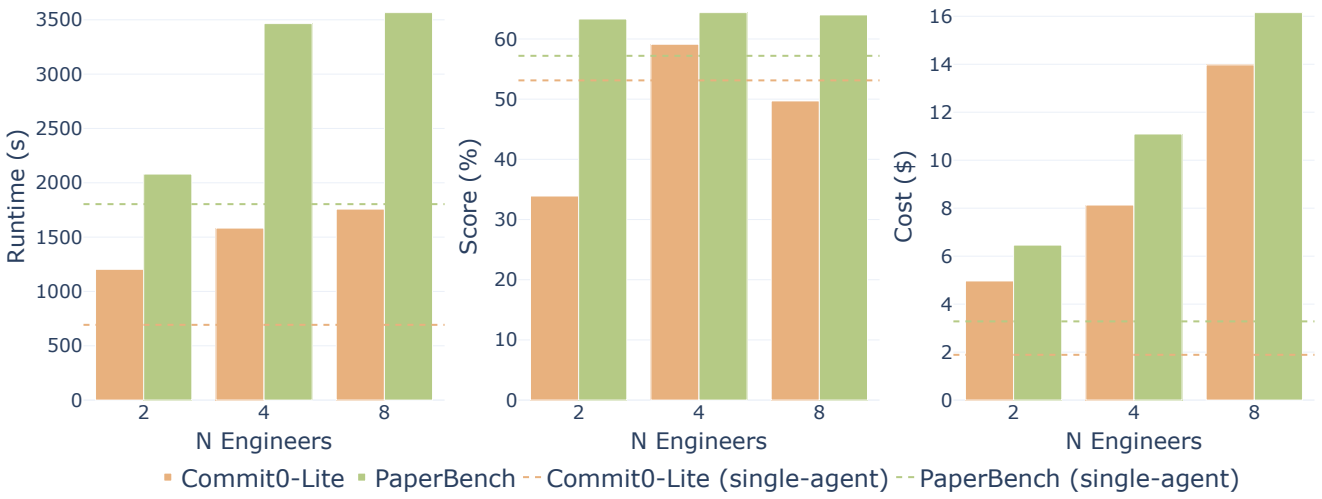

Figure 3: Effect of the number of engineer agents on runtime, pass rate, and cost for Commit0-Lite and PaperBench. We provide the single-agent baselines here for comparison.

PaperBench					
single agent		CAID (worktree isolation)		multi-agent (soft isolation)	
score	iters	score	iters	score	iters
57.2	66.8	63.3	168.3	**55.5**	190.0
Commit0-Lite					
single agent		CAID (worktree isolation)		multi-agent (soft isolation)	
score	iters	score	iters	score	iters
53.1	84.5	59.1	313.3	56.1	335.9

Table 3: Soft context isolation and worktree isolation on PaperBench and Commit0-Lite.

dependencies are explicit. Worktree isolation further increases performance to 59.1%, indicating that instruction-level separation is not sufficient to fully eliminate interference over longer trajectories. In contrast, on PaperBench soft isolation drops to 55.5%, below the single-agent score of 57.2%, while worktree isolation reaches 63.3%. Unlike Commit0, PaperBench does not provide explicit file structure or dependency graphs, and the manager must first infer the global implementation plan from the paper itself. In this case, sharing a workspace causes miscoordination, whereas worktree isolation stabilizes parallel execution.

4.2 Choosing the Degree of Parallel Execution

We analyze how the number of asynchronous engineer agents affects the performance in Figure 3. We find that increasing the number of engineers does not monotonically improve the performance, which align with the results in [32]. The optimal degree of parallelism depends on two factors: the intrinsic parallel structure of the task and the delegation capacity of the central manager. First, tasks differ in how many components can be implemented independently. In Commit0-Lite, performance improves when increasing engineers from 2 to 4 (53.1% single-agent to 59.1%), but decreases when expanding to 8 engineers. Although more agents increase theoretical parallelism, overly fine-grained task delegation introduces integration overhead and conflict resolution cost, especially when multiple engineers modify closely related modules. However, too few engineers can exploit the independent files available in clear-structured repositories, limiting progress within a fixed iteration budget. Second, scalability is constrained by the manager's coordination ability. The central manager must track dependency states, monitor the progress of the engineers, and dynamically assign tasks. When the number of engineers increases, delegation errors or delayed synchronization can propagate and destabilize the overall trajectory. This effect is visible in Commit0-Lite at 8 engineers, where performance declines despite higher computation cost. On PaperBench, where task decomposition is less structurally explicit, increasing engineers beyond 2 yields minimal gain in score while runtime and cost increase steadily. These results show that the number of subagents should be matched to both the inherent modularity of the task and the effective delegation capacity of the manager. Excess parallelism without reliable coordination degrades stability, rather than improving performance. We provide examples of failure in the Appendix D.

4.3 Delegation Shapes Execution Trajectory

In Figure 4, we show two CAID runs and one single-agent run in the `Commit0-Lite` `minitorch` repository to study how task delegation affects execution outcomes. We find that the performance difference between CAID Run 1 (8.7% pass rate) and CAID Run 2 (34.3%) is not simply due to the number of modules implemented, but to which modules are assigned and actively pursued. In Run 2, the

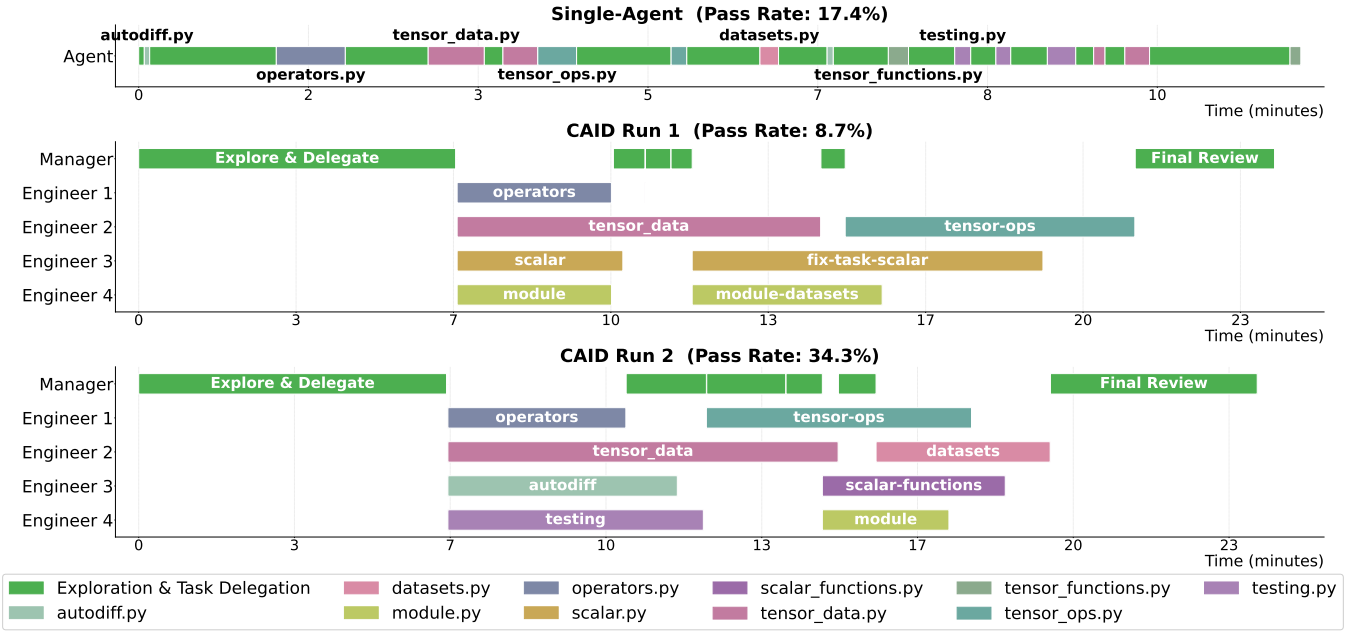

Figure 4: Execution timelines on `minitorch` for one single-agent and two CAID runs. The bars in the Gantt plot indicate file-level implementation intervals and manager phases.

manager assigns an engineer to `autodiff.py`, a file that is critical for passing tests, and sustained effort on this file is followed by broader progress across dependent components. In contrast, Run 1 assigns engineers to several other files, but never assigns work to `autodiff.py`. Although multiple engineers are active, the absence of this key dependency limits the overall pass rate. We observe that the single-agent run touches `autodiff.py` during exploration and implements part of the logic, but the file remains incomplete and the final pass rate reaches only 17.4%. This example shows that the manager's delegation ability, particularly the ability to identify and assign high-impact dependencies, is critical for the success of long-horizon SWE tasks.

4.4 Scaling Asynchronous Parallelism

During our exploration of the multi-agent design, we experimented with different prompt engineering strategies that emphasize distinct objectives, such as prioritizing correctness or efficiency. Figure 5 shows the results on a subset of nine repositories (i.e., *babel, chardet, cookiecutter, imapclient, jinja, minitorch, simply, tinydb*) of Commit0-Lite. In *Round-Manager Review*, the manager explicitly reviews code quality at every implementation round before integration for each engineer, placing stronger emphasis on correctness. In *Engineer Self-Verification*, engineers conduct self-review without repeated managerial inspection, which is closest to the main results we report in Section 3. In *Efficiency-Prioritized*, both manager and engineer agents are explicitly instructed to prioritize runtime efficiency and are reminded that execution time will be evaluated, thereby assigning higher weight to the runtime of implementation in the user instruction. We observe a clear pattern: *Round-Manager Review* achieves the highest pass rate (60.2%) but also incurs the longest

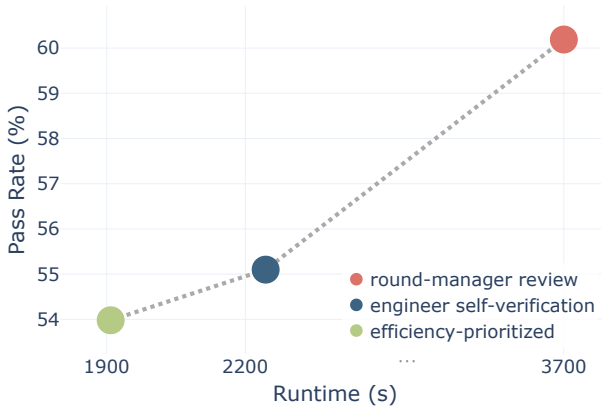

Figure 5: Runtime (s) vs. pass rate (%) of a subset of the Commit0 under three coordination prompts (1) Round-Manager Review: the manager reviews each round before integration; (2) Engineer Self-Verification: engineers verify locally without repeated managerial review; and (3) Efficiency-Prioritized: all agents are instructed to prioritize runtime efficiency.

runtime (3689.1s), *Self-Verification* yields intermediate performance (55.1%) with moderate runtime (2243.9s), and *Efficiency-Prioritized* runs fastest (1908.6s) but achieves the lowest pass rate (54.0%). This development-stage result suggests a trade-off between verification intensity and execution efficiency: emphasizing efficiency

can shorten runtime but may reduce integration robustness, while stricter review improves stability at additional computational cost.

5 Related Work

Multi-Agent Architectures. LLM-based multi-agent systems range from early communication-centric frameworks like CAMEL [12], Generative Agents [20], and ChatDev [21] to more adaptive designs such as EvoMAC [7], AutoAgents [4], and orchestrators like AgentOrchestra [35], MASS [38], and DyLAN [14], which optimize topology and task decomposition but still face communication overhead in long-horizon tasks. Scaling challenges identified by MegaAgent [26] and Qian et al. [22] motivate efficiency improvements via sequential aggregation (Chain-of-Agents [36]), memory abstraction (MemGPT [19]), and context compression (OpenHands [27]). Coordination is typically enforced through standardized procedures and agile-inspired workflows [6, 17], with extensions to implicit coordination [15] and consensus evaluation [39], yet these largely language-mediated approaches struggle in shared-artifact settings like software engineering, where concurrent edits lead to execution conflicts – a limitation highlighted by [9] that motivates our SWE-primitive based coordination design.

Multi-Agent Coordination Challenges. Despite advances in multi-agent architectures, coordination remains limited by communication-driven workflows, particularly in task delegation under uncertainty and shared-workspace conflicts. In dialogue-based systems [29], delegation is often implicit, leading to redundancy and delayed escalation, while more structured approaches – such as hierarchical or orchestrator-based delegation [23, 30] – still struggle at scale, as shown by [13, 22], where larger agent populations increase communication overhead and degrade performance. Unstructured coordination also introduces physical interference, including task overlap and inconsistent execution [11], alongside a superlinear "coordination tax" [13, 22]. Together, these results suggest that language-mediated coordination alone cannot ensure consistent state transitions or integration, motivating our approach of centralized delegation and physically isolated workspaces to reduce conflicts.

Software Engineering for Multi-Agent Coordination. Before the emergence of LLM-based agents, software engineering had already developed mechanisms for coordinating parallel work over shared artifacts, including branching and merging, dependency management, continuous integration, and code review. These mechanisms treat coordination as explicit control over versioned artifacts and their integration. Recent multi-agent work has begun to implicitly adopt parts of the SWE paradigm. Process-driven frameworks such as MetaGPT [6] and AgileCoder [17] mirror role decomposition and lifecycle management. Sandbox-based systems, including the SWE-agent [31], incorporate build–test feedback loops analogous to continuous integration. However, recent empirical studies [9] still report that concurrent modification and merge conflicts remain a primary failure mode when these engineering primitives are not explicitly modeled. These observations suggest that, in shared repositories, the central issue is not only how agents are organized into roles or workflows, but also how concurrent work is isolated, integrated, and verified. In this paper, we focus on branch-and-merge coordination and the SWE primitives that support it in multi-agent software engineering.

6 Conclusion and Future Directions

In this paper, we introduce CAID, a branch-and-merge based multi-agent system for long-horizon software engineering tasks based on existing software engineering primitives. Across Commit0 and PaperBench, CAID not only consistently improves over single-agent baselines, but also show that SWE-primitive supported branch-and-merge is an effective method for multi-agent software engineering. For complex long-horizon, dependency-aware software engineering tasks, CAID provides a strong solution for structuring solutions to enable parallel and coordinated development.

Nonetheless, several challenges remain for future work. CAID introduces notable coordination overhead, leading to higher API costs and limited runtime gains due to sequential, test-gated integration despite parallel execution; this highlights a fundamental trade-off where structured isolation and verification enhance stability but require additional communication and merging steps, suggesting future work on improving scheduling efficiency, reducing redundant verification cycles, and learning when to merge or prune intermediate states to optimize the cost–performance frontier. Additionally, the system's reliance on heuristic-based task delegation limits effectiveness, as suboptimal decomposition can cause locally correct but globally inefficient outputs, motivating exploration of reinforcement learning–based delegation policies, dependency-aware planning modules, and adaptive subtask refinement strategies to improve alignment between global objectives and isolated execution. Finally, since evaluation is confined to software engineering tasks—with clear boundaries and verification mechanisms—extending the approach to other domains like document synthesis, research planning, or multimodal artifact construction will require new methods for adapting isolation mechanisms and designing alternative forms of integration and validation, making the assessment of whether these architectural principles generalize beyond SWE-specific workflows an important direction for future research.

References

[1] Anthropic. 2024. Claude 4.5 Sonnet. https://www.anthropic.com. Large language model developed by Anthropic.

[2] Nikita Benkovich and Vitalii Valkov. 2026. Agyn: A Multi-Agent System for Team-Based Autonomous Software Engineering. *arXiv preprint arXiv:2602.01465* (2026).

[3] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657* (2025).

[4] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288* (2023).

[5] Cognition AI. 2025. Don't Build Multi-Agents. https://cognition.ai/blog/dont-build-multi-agents. Accessed: 2026-02-20.

[6] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*.

[7] Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024. Self-evolving multi-agent collaboration networks for software development. *arXiv preprint arXiv:2410.16946* (2024).

[8] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[9] Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J Ryan, Jiaxin Pei, et al. 2026. CooperBench: Why Coding Agents Cannot be Your Teammates Yet. *arXiv preprint arXiv:2601.13295* (2026).

[10] Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. 2025. Measuring ai ability to complete long tasks. *arXiv preprint arXiv:2503.14499* (2025).

[11] Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. 2024. Agent-oriented planning in multi-agent systems. *arXiv preprint arXiv:2410.02189* (2024).

[12] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems* 36 (2023), 51991–52008.

[13] Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. 2024. More agents is all you need. *arXiv preprint arXiv:2402.05120* (2024).

[14] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2023. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170* (2023).

[15] Alexander Meulemans, Seijin Kobayashi, Johannes von Oswald, Nino Scherrer, Eric Elmoznino, Blake Richards, Guillaume Lajoie, João Sacramento, et al. 2024. Multi-agent cooperation through learning-aware policy gradients. *arXiv preprint arXiv:2410.18636* (2024).

[16] MiniMax. 2024. MiniMax 2.5. https://www.minimaxi.com. Large language model developed by MiniMax.

[17] Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. 2025. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 156–167.

[18] OpenAI. 2025. GPT-5-mini. https://www.openai.com. Large language model developed by OpenAI.

[19] Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. MemGPT: towards LLMs as operating systems. (2023).

[20] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*. 1–22.

[21] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*. 15174–15186.

[22] Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. 2024. Scaling large language model-based multi-agent collaboration. *arXiv preprint arXiv:2406.07155* (2024).

[23] Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, et al. 2025. Coact-1: Computer-using agents with coding as actions. *arXiv preprint arXiv:2508.03923* (2025).

[24] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, et al. 2025. PaperBench: Evaluating AI's Ability to Replicate AI Research. *arXiv preprint arXiv:2504.01848* (2025).

[25] Vishal Venkataramani, Haizhou Shi, Zixuan Ke, Austin Xu, Xiaoxiao He, Yingbo Zhou, Semih Yavuz, Hao Wang, and Shafiq Joty. 2026. MAS-ProVe: Understanding the Process Verification of Multi-Agent Systems. *arXiv preprint arXiv:2602.03053* (2026).

[26] Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. 2025. MegaAgent: A large-scale autonomous LLM-based multi-agent system without predefined SOPs. In *Findings of the Association for Computational Linguistics: ACL 2025*. 4998–5036.

[27] 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).

[28] Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, et al. 2025. The openhands software agent sdk: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690* (2025).

[29] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First conference on language modeling*.

[30] Iris Xu, Guangtao Zeng, Zexue He, Charles Jin, Aldo Pareja, Dan Gutfreund, Chuang Gan, and Zhang-Wei Hong. 2025. BOAD: Discovering Hierarchical Software Engineering Agents via Bandit Optimization. *arXiv preprint arXiv:2512.23631* (2025).

[31] 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. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[32] Yingxuan Yang, Chengrui Qu, Muning Wen, Laixi Shi, Ying Wen, Weinan Zhang, Adam Wierman, and Shangding Gu. 2026. Understanding Agent Scaling in LLM-Based Multi-Agent Systems via Diversity. *arXiv preprint arXiv:2602.03794* (2026).

[33] Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471* (2025).

[34] Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180* (2025).

[35] Wentao Zhang, Liang Zeng, Yuzhen Xiao, Yongcong Li, Yilei Zhao, Ce Cui, Yang Liu, and Bo An. 2025. AgentOrchestra: Orchestrating hierarchical multi-agent intelligence with the Tool-Environment-Agent (TEA) protocol. (2025).

[36] Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Arik. 2024. Chain of agents: Large language models collaborating on long-context tasks. *Advances in Neural Information Processing Systems* 37 (2024), 132208–132237.

[37] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. 2024. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769* (2024).

[38] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. 2025. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533* (2025).

[39] Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934* (2024).

A Prompt Engineering for Multi-Agent Task Delegation

We provide the prompts of user instruction and task delegation for both Commit0 in Section A.1 and PaperBench in A.2.

A.1 Commit0 Prompts

user instruction

```
<uploaded_files>
/workspace/{{ workspace_dir_name }}
</uploaded_files>
I've uploaded a Python code repository in the directory {{
      workspace_dir_name }}.

Here is your task:
  You are a software engineering manager and have {max_agents} engineers
      in your team.
  Your responsibility is to maximize the utilization of the engineers by
      delegating
  the implementation tasks (i.e., the functions with `pass` statements)
      to these
  engineers and guide them to efficiently and effectively complete the
      implementation
  and pass ALL the unit tests. Except for the functions with `pass`
      statements, the
  repository might also contain some missing functions that are not
      defined in any
  files. You need to add them with clear docstrings and `pass` statements
      into the
  files for the engineers to implement. Make sure you submit a local
      commit of
  your changes. Remember you are NOT allowed to generate any code for the
      existing
  functions or classes with `pass` statements. You can only add undefined
  functions as needed.

  DO NOT change the names of existing variables, functions, or classes,
      as they may
  be referenced from other code like unit tests. Do not comment out any
      existing code.

  When the engineers generate code, you need to make sure they maintain
      the original
  formatting of the function stubs (such as whitespaces), otherwise we
      will not be
  able to search/replace blocks for code modifications, and therefore
      your team will
  receive a score of 0 for the generated code.

Here is the command to run the unit tests:
<test_command>
{test_cmd} {test_dir}
</test_command>

Each engineer is expected to proactively submit a local git commit to you
      once their
assigned task is complete. The engineers are responsible for verifying
      their own
implementation quality and running tests before submitting. If no commit
      is submitted,
you should assume the task may be partially complete. In that case,
      manually inspect
the engineer's worktree, determine which parts have already been
      implemented, sync
and merge those completed artifacts into the main directory, and make a
      commit on
their behalf.

When an engineer has completed their task with a successful commit, you
      need to
decide the next task to assign, following these steps:
  1. Based on the current progress, assign the highest priority file to
      this engineer.
  2. Make sure the assigned files do not contain or depend on any missing
      (undefined) functions; if so, add them with clear docstrings and
      `pass` statements into the files and submit a local commit of your
      changes.
  3. Explain the overall progress of the implementation and provide a
      detailed explanation of the purpose of the implementation to
      instruct the engineer
     to complete the task.
  4. If no new implementation is needed for now (e.g., the file is
      already implemented or needs to wait for other engineers to
      complete the dependencies), you can simply say "Thank you for your
      work. I will assign a new task to you later."
  5. You can also assign tasks to idle or inactive engineers if you need
      more capacity to better utilize the engineers.

Engineers are responsible for verifying their own implementation quality
      before
submitting their commits; you do NOT need to review their code quality or
      run tests
yourself. Only focus on delegating tasks (i.e., maximizing the
      utilization of the
engineers to pass more unit tests).

Make sure you DO NOT iteratively overcheck or fix missing functions.
Provide a clear and concise response in the JSON format below.
Please structure your response as JSON:

{
  "assign_task": {
    "reasoning": "Explain your decision",
    "assignments": [
      {
        "engineer_id": "engineer_id",
        "task_id": "task-unique-id or 'fix-<original-task-id>'",
        "file_path": "path/to/file.py",
        "functions_to_implement": ["func1", "func2"],
        "instruction": "Detailed explanation of the purpose of the
      implementation.",
        "complexity": "simple|medium|complex"
      }
    ]
  }
}

If no tasks should be assigned, use an empty assignments array.
```

task delegation

```
Your engineers are waiting for your instructions to start their first
      implementation tasks. Now you need to:

  1. Check for uncommitted changes and commit if needed:
     git status
     git add -A && git commit -m "Add missing stubs from scan phase" ||
       true

  2. Suspend exploration and systematically delegate the implementation
       work
     by outputting a delegation JSON based on your current understanding
     of the repository structure and its dependencies.
     You have up to {max_agents} engineers available.

Suggestions for effective delegation in the first round:

  - First, divide the overall implementation work into up to {max_agents}
    major tasks, balancing complexity and estimated effort as evenly as
       possible. Keep highly interdependent files within the same major
       task. Prefer splitting at the file level. If one file contains a
       disproportionately large number of functions with pass statements,
       you may split by function and assign non-overlapping sets to
       multiple engineers.

  - For each engineer, assign the highest-priority file within their
       major task. If two files are circularly dependent, assign them to
       the same engineer. Engineers are generally more comfortable
       starting from simpler tasks before moving to more complex ones.

  - Engineers are only responsible for implementing functions with pass
    statements. Do not assign them to implement missing functions that
       are not defined in any file. If you previously added undefined
       functions with pass statements, include them in the assignment
       instructions.

  - In each assignment, briefly summarize the relevant repository
       structure and dependencies so engineers do not need to re-explore
       the codebase. Clearly specify which file and which functions to
       implement. Explain the purpose and expected behavior of each
       function. If assigned functions depend on other stub functions not
       assigned to the same engineer, provide a short description of
       those dependencies to avoid confusion.
```

```
Note: Do NOT provide any code snippets or pseudo-code.
Output your delegation plan strictly in the following JSON format:

{
  "delegation_plan": {
    "first_round": {
      "num_agents": <integer (1 to {max_agents})>,
      "reasoning": "Explain why these files are assigned first and why
       this number of engineers is used.",
      "tasks": [
        {
          "engineer_id": "engineer_id",
          "task_id": "task-unique-id",
          "file_path": "path/to/file.py",
          "functions_to_implement": ["func1", "func2"],
          "complexity": "simple|medium|complex",
          "instruction": "Summarize the repository structure and
       dependencies. Then provide detailed instructions for the
       implementation, including the expected behavior of the assigned
       functions and descriptions of any dependent stub functions."
        }
      ]
    },
    "remaining_tasks": [
      {
        "task_id": "task-unique-id",
        "file_path": "path/to/file.py",
        "functions_to_implement": ["func1", "func2"],
        "complexity": "simple|medium|complex",
        "depends_on": ["file_path_1", "file_path_2"]
      }
    ]
  }
}
```

A.2 PaperBench Prompts

user instruction

```
You are tasked with reproducing a research paper.

You have been provided with a research paper located at
       `/workspace/paper`. The directory contains the paper in both PDF
       and markdown format.

Here is your task:
You are a software engineering manager for a research team and have
       {max_agents} engineers in your team. Each engineer has a total of
       {max_rounds} rounds to complete their assigned tasks. Your
       responsibility is to delegate the reproduction tasks to these
       engineers and guide them to efficiently replicate as many of the
       core contributions and results of this paper as possible. Your
       team may not be able to complete everything within the allotted
       time; you should prioritize tasks in order to accomplish as much
       as possible within limited time. You may prepare the environment
       for the reproduction, but you are NOT allowed to generate any code
       for the reproduction. You may only delegate reproduction tasks to
       your engineers. Do NOT create code scaffolds, skeleton files, or
       markdown files -- your engineers will create their own
       implementation files.

Do not worry if your team fails to completely reproduce certain parts of
       the paper. Partial credit will be assigned. If a specific part of
       the reproduction is incorrect, only that part will fail to receive
       credit, while correctly implemented parts will be marked as
       successful. The partial credit will be weighted by importance to
       the paper's main contributions and results. Use your judgment to
       determine which parts of the paper are more important to
       reproduce. We will check that your results match the general
       trends of the original paper within a reasonable margin of error.

Experiments introduced only in the Appendix are not considered core
       contributions and are out of scope, even if referenced in the main
       text. However, if an experiment is described in the main body but
       some implementation details (e.g., prompts or long mathematical
       expressions) are placed in the Appendix, that experiment is still
       in scope.

RESOURCES
---
You may be provided with a rubric file located at
       `/workspace/paper/rubric.json`. The rubric has a hierarchical tree
       structure that breaks down the paper into fine-grained tasks. You
       may receive only a subset of the full hierarchy, or a single
       root-level task. In that case, use your judgment to decompose the
       reproduction into appropriate sub-tasks and assign them to your
       engineers. Do NOT delegate directly at rubric node level; the
       rubric is only a reference for expected outcomes.

An addendum file may also be available at `/workspace/paper/addendum.md`,
       providing additional context or clarifications regarding scope.

Your team may browse and use publicly available resources online.
       However, your team must NOT access the original paper's codebase
       or any resources listed in `blacklist.txt`.

SUBMISSION
---
Your team must produce a git repository located at
       `/workspace/submission/`. Only source code and necessary scripts
       should be committed; the total committed size must not exceed 1GB.
       We will run `git clean -fd` before grading, so any untracked files
       will be removed.

The submission MUST include a reproduction script at:
  /workspace/submission/reproduce.sh

We will copy your submission into a fresh Ubuntu 24.04 LTS Docker
       container and run:
  bash reproduce.sh

The maximum runtime is 7 days. Do not rely on hardcoded absolute paths.
       The container will include an NVIDIA A10 GPU with the NVIDIA
       container toolkit installed.

All graded artifacts must be generated by the reproduction script.

You should regularly update and test your reproduction script throughout
       development.

Finally, include a README.md file after all reproduction tasks are
       completed (or when engineers reach their maximum rounds). The
       README should summarize achieved results, explain the repository
       structure, and describe expected outputs from running reproduce.sh.

ENGINEER SUBMISSIONS
---
Each engineer must submit a local git commit upon completing their
       assigned task. If no commit is submitted, assume partial
       completion.

After reviewing a submission:
  1. Verify whether the expected outputs are produced.
  2. If incomplete, reassign the same task with corrective guidance.
  3. If complete, assign the next highest-priority remaining task.
  4. You may assign tasks to idle engineers to maximize productivity.
  5. Always summarize overall progress and provide detailed instructions
       for the next task.
  6. If onboarding a new engineer, provide a detailed explanation of the
       paper and current reproduction goals.

Output your response strictly in the following JSON format:

{
  "assign_task": {
    "reasoning": "Explain your decision",
    "tasks": [
      {
        "engineer_id": "engineer_id",
        "task_id": "task-unique-id",
        "task_node_id": "rubric task node id if available",
        "requirements": "Specific requirement to implement",
        "task_category": "Code Development|Experiment Running|Results
       Analysis|Other",
        "estimated_complexity": "simple|medium|complex",
        "instruction": "Provide detailed explanation of current progress
       and detailed instructions for this task, including expected
       behavior and outputs, relevant paper details, and required
       dependencies."
      }
    ]
  }
}

If no tasks should be assigned, use an empty tasks array.
```

task delegation

```
The engineers on your team are waiting for instructions to begin their
      first reproduction tasks. You must now delegate the reproduction
      work systematically by outputting a delegation JSON based on your
      current understanding of the paper. You have up to {max_agents}
      engineers available.

Strategies for effective first-round delegation:

- First, divide the overall reproduction effort into up to {max_agents}
      major task groups based on your understanding of the paper.
      Balance complexity and estimated effort as evenly as possible.
      Group related tasks together and carefully consider dependencies
      between tasks (i.e., which components depend on others). Do NOT
      delegate directly at the rubric node level; the rubric (if
      provided) is only a reference for expected outcomes. Remember that
      reproduction includes not only implementation but also experiment
      execution needed to generate expected outputs. When forming task
      groups, consider how experiment orchestration will be structured.

- For each engineer, assign the highest-priority reproduction task within
      their task group (i.e., the task that reproduces the most
      important results).

- Since this is the first assignment round, provide engineers with a
      clear explanation of the overall structure of the paper and a
      detailed summary of the paper based on your exploration. This
      ensures they do not need to re-explore the paper independently.

- Provide detailed instructions for each assigned task. Clearly specify
      which part of the paper is being reproduced and what outputs are
      expected. Include relevant context from the paper and addendum.
      Explicitly mention which dependencies are already available and
      which must be installed. Ensure that each engineer creates and
      modifies only their own files. Do NOT assign multiple engineers to
      modify the same file, as this will cause merge conflicts.

- Do not assign the reproduce.sh script to any engineer. You will create
      it yourself after all engineers have completed their tasks or
      reached their maximum rounds.

- Reproduction involves both implementation and experiment execution.
      Engineers must run experiments and generate concrete outputs
      (e.g., tables, figures, CSV files). Each task group should include
      both implementation and execution steps necessary to produce
      measurable results. The objective is to reproduce as many of the
      paper's core contributions and results as possible within limited
      time.

Note: Do NOT provide any code snippets or pseudo-code. Output your
      delegation plan strictly in the following JSON format:

{
  "delegation_plan": {
    "first_round": {
      "num_agents": <integer between 1 and {max_agents}>,
      "reasoning": "Explain why these tasks are prioritized and why this
       number of engineers is used.",
      "tasks": [
        {
          "engineer_id": "engineer_id",
          "task_id": "task-unique-id",
          "task_node_id": "rubric task node id if available",
          "requirements": "Specific requirement from the rubric to
      implement",
          "task_category": "Code Development|Experiment Running|Results
      Analysis|Other",
          "estimated_complexity": "simple|medium|complex",
          "instruction": "Provide a detailed explanation of the paper and
      detailed instructions for the current reproduction task. Explain
      the expected behavior and outputs. Include relevant details from
      the paper or addendum. Explicitly mention available dependencies
      and required installations. Provide clear, structured guidance to
      ensure correct implementation."
        }
      ]
    },
    "remaining_tasks": [
      {
        "task_id": "task-unique-id",
        "task_node_id": "rubric task node id if available",
        "requirements": "Specific requirement to implement",
        "task_category": "Code Development|Experiment Running|Results
      Analysis|Other",
        "estimated_complexity": "simple|medium|complex",
        "depends_on": ["list of task_ids this depends on, or empty"]
      }
    ]
  }
}
```

B Full Results

We provide the full results of Claude 4.5 Sonnet of each repository on Commit0-Lite and each paper on PaperBench across three LLMs.

C One-sided t-test

Benchmark	Model	Δ	t	p
Commit0	Claude 4.5	+6.0	2.87	0.006
	GLM 4.7	+3.6	1.37	0.095
	MiniMax 2.5	+14.7	2.81	0.007
PaperBench	Claude 4.5	+6.1	1.78	0.046
	GLM 4.7	+7.4	1.93	0.034
	MiniMax 2.5	+0.8	0.23	0.408

Table 4: One-sided paired t-test (H_1: CAID > Single-Agent). Δ: mean score improvement. Bold: $p < 0.05$.

We compute one-sided paired t-tests (H_1: CAID > Single-Agent) across all repositories or papers for each model in Table 4. On Commit0-Lite, the improvement is significant for Claude Sonnet 4.5 ($t = 2.87$, $p = 0.006$) and MiniMax 2.5 ($t = 2.81$, $p = 0.007$), with mean gains of 6.0 and 14.7 percentage points respectively. GLM 4.7 improves by 3.6 points on average but does not reach significance ($p = 0.095$), largely because the per-repository variance is high: CAID brings large gains on some repositories (e.g., +30.7 on `simpy`) but regresses on others (e.g., −10.5 on `tinydb`), which inflates the standard error with only 16 paired samples. On PaperBench, both Claude Sonnet 4.5 ($t = 1.78$, $p = 0.046$) and GLM 4.7 ($t = 1.93$, $p = 0.034$) are significant. The only non-significant case is MiniMax 2.5 on PaperBench ($p = 0.408$), where the mean gain is only 0.8 points. As discussed in Section 4.3, CAID's effectiveness depends on the manager's ability to construct accurate dependency graphs and delegate tasks accordingly. A weaker base model produces less reliable task decomposition on the open-ended PaperBench tasks, limiting the gains that multi-agent coordination can deliver.

D Failure on Scaling the Parallel Execution

We provide an example to show why scaling the parallel execution is not always help. Figure 6 shows the execution timelines on the `simpy` repository under different numbers of engineers ($N = 2, 4, 8$). The performance difference is not solely explained by the number of files touched, but by how the manager structures delegation across engineers. For $N = 4$, delegation remains clean and non-overlapping. Each engineer is assigned distinct files (e.g., `events.py`, `core.py`, `container.py`, `resource.py`), and their implementations proceed largely without interference. The manager avoids assigning closely coupled modules to different engineers

simultaneously, and no two engineers work on the same file at the same time. As a result, integration remains stable and the run reaches a pass rate of 92.1%.

For $N = 8$, although more files are modified and parallel activity increases, the delegation becomes less disciplined. Multiple engineers are assigned different functions within the same file (notably `events.py`), creating overlapping write regions within a shared module. While these edits are logically separable at the function level, they introduce integration risk at the file level. The main branch receives competing updates on the same module, increasing the likelihood of merge conflicts or inconsistent intermediate states. This fragmentation of responsibility prevents clean consolidation and ultimately limits performance to 44.3%. The degradation in $N = 8$ therefore does not arise from excessive parallelism alone, but from a delegation that ignores the ownership boundaries of the file-level. When parallel execution exceeds the manager's ability to enforce coherent task partitioning, local productivity no longer translates into stable global progress. This example illustrates that scaling the number of engineers requires disciplined delegation, not simply increasing concurrency.

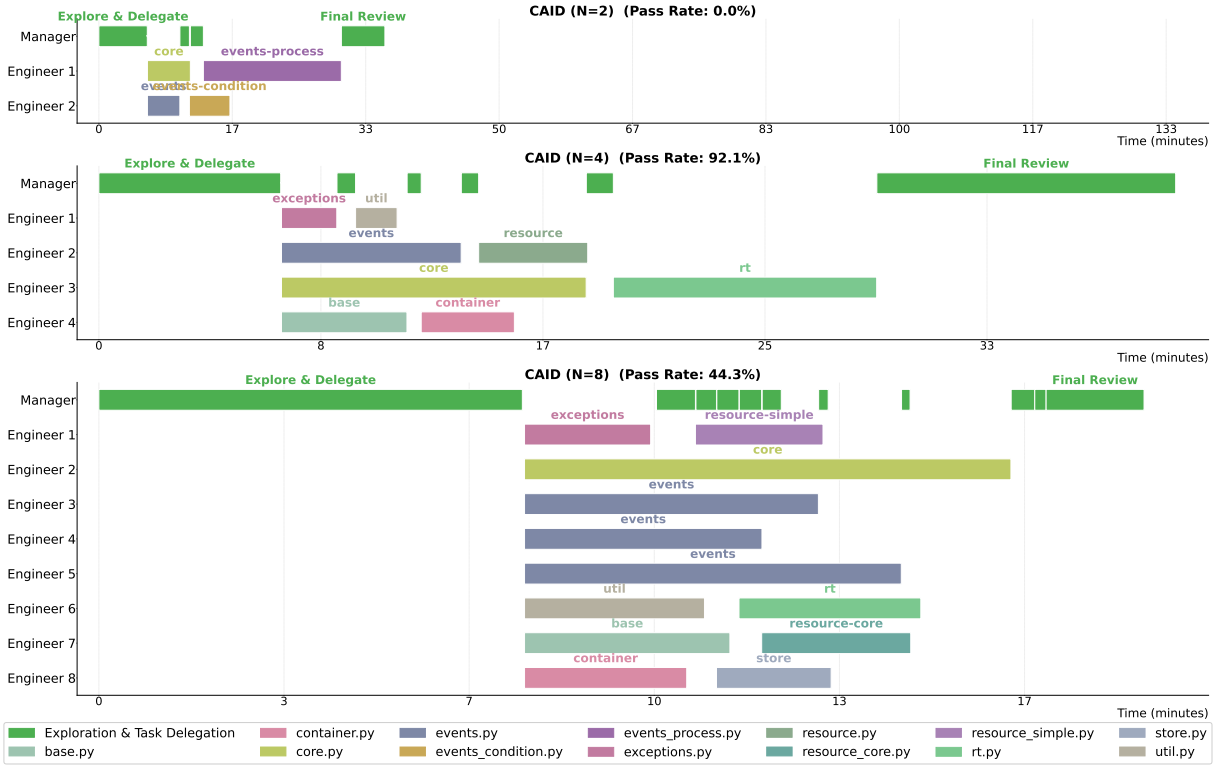

Figure 6: Gantt plot on the `simpy` repository for CAID with different number of engineers, where $N = 2, 4, 8$.

	Single-Agent (100 iters)				CAID (4 engineers)				Single+CAID			
paper_id	**Pass**	**Time**	**Cost**	**Iter**	**Pass**	**Time**	**Cost**	**Iter**	**Pass**	**Time**	**Cost**	**Iter**
babel	0.0	955.3	1.3	100.0	1.5	1749.4	14.0	267.0	1.5	2704.7	15.2	367.0
cachetools	100.0	284.3	1.0	44.0	100.0	863.1	3.3	206.0	100.0	1147.4	4.3	250.0
chardet	6.4	598.0	2.3	31.0	2.4	1112.4	7.5	259.0	6.4	1710.4	9.8	290.0
cookiecutter	35.1	615.8	1.9	100.0	40.2	1246.9	7.0	288.0	40.2	1862.7	8.8	388.0
deprecated	100.0	444.3	1.0	47.0	100.0	1197.2	4.2	165.0	100.0	1641.5	5.2	212.0
imapclient	28.8	596.6	1.9	100.0	42.3	1463.0	9.1	405.0	42.3	2059.6	11.0	505.0
jinja	0.0	647.2	1.7	99.0	5.1	1483.9	10.0	428.0	5.1	2131.1	11.7	527.0
marshmallow	23.1	600.6	2.0	100.0	43.8	1981.0	11.0	444.0	43.8	2581.6	13.0	544.0
minitorch	17.4	689.7	2.0	100.0	34.4	1436.2	8.7	374.0	34.4	2125.9	10.7	474.0
parsel	73.8	782.2	2.2	97.0	72.3	1609.4	7.3	275.0	73.8	2391.6	9.5	372.0
portalocker	79.0	1180.6	1.7	78.0	100.0	2098.5	7.4	275.0	100.0	3279.1	9.1	353.0
pyjwt	61.0	721.7	2.4	99.0	62.2	1513.4	8.0	330.0	62.2	2235.1	10.5	429.0
simpy	77.9	745.2	2.1	100.0	92.1	2424.9	10.7	387.0	92.1	3170.1	12.9	487.0
tinydb	91.0	838.6	2.5	100.0	94.0	1730.0	7.1	285.0	94.0	2568.6	9.7	385.0
voluptuous	56.4	747.5	2.5	100.0	55.7	1801.3	9.2	422.0	56.4	2548.8	11.7	522.0
wcwidth	100.0	634.4	1.7	57.0	100.0	1620.2	5.5	203.0	100.0	2254.6	7.2	260.0
AVERAGE	53.1	692.6	1.9	84.5	59.1	1583.2	8.1	313.3	59.5	2275.8	10.0	397.8

Table 5: Claude 4.5 Sonnet results on Commit0-Lite across different configurations.

	Single-Agent (100 iters)				Multi-Agent (4 engineers)				Single+Multi-Agent (100 iters)			
repo_id	**Pass**	**Time**	**Cost**	**Iter**	**Pass**	**Time**	**Cost**	**Iter**	**Pass**	**Time**	**Cost**	**Iter**
babel	0.4	865.6	3.6	100.0	0.7	1658.4	11.8	395.0	0.7	2524.0	15.4	495.0
cachetools	100.0	314.5	1.5	68.0	100.0	2131.2	4.1	179.0	100.0	2445.7	5.6	247.0
chardet	0.0	438.0	4.1	100.0	0.0	1654.1	8.4	269.0	0.0	2092.1	12.5	369.0
cookiecutter	22.1	3817.7	3.2	100.0	29.0	2047.8	6.4	287.0	29.0	5865.5	9.6	387.0
deprecated	100.0	210.1	0.9	44.0	100.0	2749.6	5.1	190.0	100.0	2959.7	6.0	234.0
imapclient	23.2	550.4	2.6	100.0	24.3	1615.1	13.2	510.0	24.3	2165.5	15.7	610.0
jinja	0.0	419.0	2.4	100.0	0.0	509.2	2.6	150.0	0.0	928.2	5.0	250.0
marshmallow	17.0	392.6	2.3	100.0	38.7	2256.2	18.7	592.0	38.7	2648.8	21.0	692.0
minitorch	17.4	555.5	2.3	100.0	20.0	744.7	9.2	372.0	20.0	1300.2	11.5	472.0
parsel	39.8	486.3	2.6	100.0	47.6	552.8	5.3	240.0	47.6	1039.1	7.9	340.0
portalocker	68.4	2957.4	1.3	56.0	71.1	1287.8	5.6	264.0	71.1	4245.2	6.8	320.0
pyjwt	49.4	1039.5	3.1	100.0	59.5	527.4	1.5	54.0	59.5	1566.9	4.6	154.0
simpy	34.3	672.4	2.6	100.0	65.0	1558.8	7.0	270.0	65.0	2231.2	9.6	370.0
tinydb	82.1	458.9	3.2	100.0	71.6	1124.9	5.9	244.0	82.1	1583.8	9.1	344.0
voluptuous	42.3	419.4	3.2	100.0	32.2	1052.4	6.2	246.0	42.3	1471.8	9.4	346.0
wcwidth	89.5	338.9	0.8	30.0	84.2	734.2	5.5	181.0	89.5	1073.1	6.3	211.0
AVERAGE	42.9	871.0	2.5	87.4	46.5	1387.8	7.3	277.7	46.5	2258.8	9.8	365.1

Table 6: GLM 4.7 results on Commit0-Lite across different configurations.

	Single-Agent (100 iters)				Multi-Agent (4 engineers)				Single+Multi-Agent (100 iters)			
repo_id	**Pass**	**Time**	**Cost**	**Iter**	**Pass**	**Time**	**Cost**	**Iter**	**Pass**	**Time**	**Cost**	**Iter**
babel	0.3	578.6	1.4	100.0	1.2	3972.7	9.4	514.0	1.2	4551.3	10.8	614.0
cachetools	100.0	408.2	0.9	38.0	100.0	469.2	0.7	81.0	100.0	877.4	1.7	119.0
chardet	3.5	612.3	1.7	64.0	31.7	2804.7	4.7	327.0	31.7	3417.0	6.3	391.0
cookiecutter	42.3	901.5	1.6	54.0	47.3	3593.6	6.8	407.0	47.3	4495.1	8.4	461.0
deprecated	100.0	551.5	0.8	33.0	100.0	758.1	1.7	147.0	100.0	1309.6	2.5	180.0
imapclient	18.0	443.9	1.1	100.0	16.9	871.1	1.5	31.0	18.0	1315.0	2.5	131.0
jinja	0.0	419.5	3.6	100.0	0.0	1213.1	1.8	150.0	0.0	1632.6	5.3	250.0
marshmallow	15.5	469.4	1.2	100.0	23.2	1217.8	5.4	242.0	23.2	1687.2	6.6	342.0
minitorch	0.0	461.2	1.2	55.0	40.0	1164.6	2.0	112.0	40.0	1625.8	3.2	167.0
parsel	52.9	857.6	1.8	52.0	100.0	1690.3	5.1	317.0	100.0	2547.9	6.9	369.0
portalocker	76.3	978.6	1.7	70.0	97.4	3394.0	8.3	424.0	97.4	4372.6	10.0	494.0
pyjwt	51.7	793.4	1.8	50.0	51.7	2385.2	7.9	424.0	51.7	3178.6	9.7	474.0
simpy	0.0	1031.0	1.4	61.0	68.6	1578.1	5.6	138.0	68.6	2609.1	7.0	199.0
tinydb	86.1	679.0	1.5	51.0	95.0	2817.1	6.1	171.0	95.0	3496.1	7.6	222.0
voluptuous	37.6	919.5	1.5	69.0	38.3	1172.6	2.7	235.0	38.3	2092.1	4.2	304.0
wcwidth	92.1	1927.9	2.8	39.0	100.0	1436.5	3.0	213.0	100.0	3364.4	5.8	252.0
AVERAGE	42.3	752.1	1.6	64.8	57.0	1908.7	4.5	245.8	57.0	2660.7	6.2	310.6

Table 7: MiniMax 2.5 results on Commit0-Lite across different configurations.

	Single-Agent (100 iters)				CAID (2 engineers)				Single+CAID			
paper_id	**Scores**	**Time**	**Cost**	**Iter**	**Scores**	**Time**	**Cost**	**Iter**	**Scores**	**Time**	**Cost**	**Iter**
adaptive-pruning	33.4	1043.5	3.0	70.0	56.0	2463.0	7.4	191.0	56.0	3506.5	10.5	261.0
all-in-one	68.4	3124.0	3.9	98.0	50.2	1946.9	6.0	146.0	68.4	5070.9	9.9	244.0
bam	57.9	3601.6	3.4	87.0	64.7	2577.7	7.2	223.0	64.7	6179.3	10.6	310.0
bbox	38.6	3397.5	4.0	80.0	68.8	1856.0	9.1	163.0	68.8	5253.5	13.1	243.0
bridging-data-gaps	43.2	1409.7	2.9	78.0	40.5	2078.0	6.6	166.0	43.2	3487.7	9.5	244.0
fre	56.9	1198.6	3.3	92.0	69.6	2193.6	7.6	213.0	69.6	3392.2	10.9	305.0
ftrl	34.6	1499.6	3.2	14.0	61.9	1943.0	7.3	184.0	61.9	3442.6	10.5	198.0
lbcs	79.5	1451.9	3.3	50.0	82.9	2508.9	6.3	170.0	82.9	3960.8	9.6	220.0
lca-on-the-line	59.3	1754.1	3.3	18.0	48.8	2011.9	4.7	205.0	59.3	3766.0	8.0	223.0
mechanistic-understanding	75.0	1771.8	3.0	77.0	63.1	1936.5	6.5	175.0	75.0	3708.3	9.5	252.0
pinn	53.9	2272.6	3.9	44.0	68.4	2222.5	5.3	112.0	68.4	4495.1	9.2	156.0
rice	33.2	2239.4	3.4	72.0	30.0	1870.7	6.4	150.0	33.2	4110.1	9.8	222.0
robust-clip	42.9	1343.8	3.4	83.0	57.2	1899.5	6.4	151.0	57.2	3243.3	9.7	234.0
sample-specific-masks	85.6	1110.3	2.7	22.0	86.3	2081.0	6.2	165.0	86.3	3191.3	8.9	187.0
sapg	28.0	1551.1	3.3	99.0	64.2	1934.8	8.0	139.0	64.2	3485.9	11.4	238.0
sequential-neural-score-estimation	86.5	2011.6	3.2	100.0	86.7	2097.5	4.7	164.0	86.7	4109.1	7.9	264.0
stay-on-topic-with-classifier-free-guidance	66.2	1468.2	3.0	62.0	78.5	1829.0	4.2	140.0	78.5	3297.2	7.2	202.0
stochastic-interpolants	85.8	1260.4	3.5	100.0	74.1	2105.3	6.6	217.0	85.8	3365.7	10.1	317.0
test-time-model-adaptation	62.7	1165.9	2.8	16.0	51.3	1966.1	6.2	165.0	62.7	3132.0	9.0	181.0
what-will-my-model-forget	52.4	1394.8	3.0	74.0	63.2	2086.1	6.5	126.0	63.2	3480.9	9.5	200.0
AVERAGE	57.2	1803.5	3.3	66.8	63.3	2080.4	6.5	168.3	66.8	3883.9	9.7	235.1

Table 8: Claude 4.5 Sonnet results on PaperBench Code-Dev across different configurations.

	Single-Agent (100 iters)				Multi-Agent (2 engineers)				Single+Multi-Agent			
paper_id	**Scores**	**Time**	**Cost**	**Iter**	**Scores**	**Time**	**Cost**	**Iter**	**Scores**	**Time**	**Cost**	**Iter**
adaptive-pruning	44.9	1130.0	2.5	72.0	60.3	1473.6	6.1	187.0	60.3	2603.6	8.6	259.0
all-in-one	19.9	1430.0	3.6	100.0	25.8	1532.0	3.3	140.0	25.8	2962.0	6.9	240.0
bam	63.5	681.6	2.5	53.0	75.3	1315.1	4.9	184.0	75.3	1996.7	7.4	237.0
bbox	15.1	1186.0	2.7	75.0	40.1	1227.9	4.4	163.0	40.1	2413.9	7.1	238.0
bridging-data-gaps	25.6	603.0	2.1	68.0	33.5	1227.1	4.8	190.0	33.5	1830.1	6.9	258.0
fre	42.3	2429.6	2.6	55.0	42.8	1349.6	4.4	177.0	42.8	3779.2	7.0	232.0
ftrl	15.4	1326.4	3.1	95.0	32.7	1850.0	5.2	182.0	32.7	3176.4	8.3	277.0
lbcs	75.0	539.7	3.3	87.0	38.2	1213.2	4.5	145.0	75.0	1752.9	7.8	232.0
lca-on-the-line	34.7	675.7	2.9	58.0	30.2	1974.9	3.2	112.0	34.7	2650.6	6.1	170.0
mechanistic-understanding	0.0	3601.7	3.3	90.0	47.7	1904.2	3.6	164.0	47.7	5505.9	6.9	254.0
pinn	61.0	1158.6	2.4	43.0	43.2	832.6	4.3	155.0	61.0	1991.2	6.7	198.0
rice	28.5	867.8	3.4	96.0	30.0	1870.7	3.6	131.0	30.0	2738.5	7.0	227.0
robust-clip	22.3	728.7	3.7	87.0	29.3	1288.9	7.1	191.0	29.3	2017.6	10.9	278.0
sample-specific-masks	50.4	793.3	2.4	58.0	54.6	1123.5	5.5	217.0	54.6	1916.8	7.9	275.0
sapg	29.4	836.0	4.5	100.0	27.0	952.0	6.0	204.0	29.4	1788.0	10.5	304.0
sequential-neural-score-estimation	58.8	1248.6	2.8	92.0	79.9	1136.1	4.4	176.0	79.9	2384.7	7.2	268.0
stay-on-topic-with-classifier-free-guidance	49.7	807.1	2.6	81.0	59.3	1769.4	4.8	157.0	59.3	2576.5	7.4	238.0
stochastic-interpolants	70.8	1376.5	3.0	67.0	71.0	1586.8	6.7	228.0	71.0	2963.3	9.7	295.0
test-time-model-adaptation	10.3	1106.9	1.0	92.0	32.9	1547.6	3.2	133.0	32.9	2654.5	4.2	225.0
what-will-my-model-forget	42.6	1023.9	1.9	61.0	53.6	1812.9	4.6	76.0	53.6	2836.8	6.4	137.0
AVERAGE	38.0	1177.6	2.8	76.5	45.4	1449.4	4.7	165.6	48.5	2627.0	7.5	242.3

Table 9: GLM 4.7 results on PaperBench Code-Dev across different configurations.

	Single-Agent (100 iters)				Multi-Agent (2 engineers)				Single+Multi-Agent			
paper_id	**Scores**	**Time**	**Cost**	**Iter**	**Scores**	**Time**	**Cost**	**Iter**	**Scores**	**Time**	**Cost**	**Iter**
adaptive-pruning	15.1	3601.5	0.9	50.0	15.2	3558.3	3.1	223.0	15.2	7159.8	4.0	273.0
all-in-one	0.0	2461.6	1.2	35.0	22.3	3635.1	2.7	198.0	22.3	6096.7	3.9	233.0
bam	49.9	2434.8	1.3	11.0	38.5	1852.4	2.6	216.0	49.9	4287.2	3.8	227.0
bbox	0.0	1491.0	0.5	41.0	28.0	1257.8	1.3	136.0	28.0	2748.8	1.9	177.0
bridging-data-gaps	29.4	970.1	0.8	57.0	33.5	2610.4	2.7	181.0	33.5	3580.5	3.5	238.0
fre	0.0	2128.3	2.5	100.0	29.0	2955.0	4.3	284.0	29.0	5083.3	6.8	384.0
ftrl	0.0	3601.2	0.9	55.0	7.1	4130.4	2.4	193.0	7.1	7731.6	3.4	248.0
lbcs	0.0	3601.1	0.6	36.0	0.4	2933.2	3.0	189.0	42.0	6534.3	3.6	225.0
lca-on-the-line	11.8	1045.4	0.6	45.0	0.2	3294.7	3.3	239.0	24.9	4340.1	3.9	284.0
mechanistic-understanding	0.0	3600.7	1.1	64.0	0.3	3129.2	1.3	131.0	34.0	6729.9	2.4	195.0
pinn	0.0	1906.9	1.2	67.0	0.6	2714.2	2.6	192.0	56.0	4621.1	3.8	259.0
rice	0.0	3601.6	0.8	53.0	0.2	2509.7	2.2	173.0	20.6	6111.3	3.0	226.0
robust-clip	0.0	2474.9	1.4	75.0	0.2	3668.2	3.2	250.0	23.9	6143.1	4.6	325.0
sample-specific-masks	0.0	3601.1	0.6	46.0	0.6	4419.1	1.8	78.0	58.7	8020.2	2.4	124.0
sapg	8.4	1780.2	0.6	46.0	0.3	1934.8	0.9	150.0	29.9	3715.0	1.5	196.0
sequential-neural-score-estimation	47.4	2511.2	1.0	75.0	0.7	3759.2	3.0	137.0	71.1	6270.4	4.0	212.0
stay-on-topic-with-classifier-free-guidance	0.5	2882.9	1.2	71.0	0.5	3029.0	3.2	176.0	0.5	5911.9	4.5	247.0
stochastic-interpolants	0.0	2426.3	2.1	100.0	0.7	3608.0	4.8	279.0	0.7	6034.3	6.8	379.0
test-time-model-adaptation	0.0	2990.1	1.5	93.0	0.5	1989.6	1.6	137.0	0.5	4979.7	3.1	230.0
what-will-my-model-forget	0.0	1395.7	0.9	45.0	0.2	3859.7	2.0	49.0	0.2	5255.4	2.9	94.0
AVERAGE	10.5	2525.3	1.1	58.3	36.1	3042.4	2.6	180.6	36.7	5567.7	3.7	238.8

Table 10: MiniMax 2.5 results on PaperBench Code-Dev across different configurations.