ASYNCODEBENCH: BENCHMARKING COLLABORATION OF ASYNCHRONOUS MULTI-AGENT SYSTEMS IN SOFTWARE ENGINEERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent coding has emerged as an increasingly active direction in software engineering, where complex development tasks are decomposed across multiple specialized agents working on different parts of the problem. Despite the shift from individual problem solving to distributed collaboration, multi-agent systems still lack a direct measure of collaboration and are largely evaluated through task-level outcomes inherited from single-agent coding, conflating individual coding capability with cross-agent coordination. We introduce **AsynCodeBench**, a dependency-centric benchmark for asynchronous multi-agent software engineering that represents each task with an explicit dependency graph and executable Dependency Checkers. Through this dependency-tracking process, we propose two complementary measures: *Asynchronous Dependency Pass Rate* (ADPR), which measures how many cross-agent dependencies are ultimately satisfied, and *Dependency Resolution Step (DRS)*, which measures when each dependency first becomes satisfied during execution. AsynCodeBench comprises 19 tasks from real-world repositories, exposing 52 directed dependencies as explicit units for evaluating cross-agent collaboration. Experiments across model families, scales, and generations reveal a clear gap between coding and collaboration capability: improvements in coding performance do not necessarily translate into stronger collaboration, and task-level metrics can diverge substantially from dependency-level collaboration measures. Dependency-trajectory analysis further reveals that successful coordination often emerges not gradually, but through concentrated bursts in which many dependencies become resolved over a short portion of the execution trajectory, a pattern we term a hopping window. These findings position dependency dynamics as a useful lens for understanding and interpreting collaboration behavior in asynchronous multi-agent software engineering. Resources are available at https://anonymous.4open.science/r/AsynCodeBench-2640/.

1 INTRODUCTION

Large language model agents have rapidly expanded from code generation to repository-level software engineering, where they can modify files, execute tests, and iteratively repair implementations (Jimenez et al., 2024; Yang et al., 2024a). Multi-agent software development has consequently emerged as an active direction, with systems such as ChatDev, MetaGPT, RTADev, CAID, and CodeTeam distributing planning, implementation, testing, and coordination across specialized agents (Qian et al., 2024; Hong et al., 2024; Liu et al., 2025; Wang et al., 2026; Geng & Neubig, 2026). Recent work has also begun to study interaction and coordination more explicitly, including recovery from out-of-sync states and corrective interaction during coding (Guo et al., 2025; Wu et al., 2026). Yet multi-agent software engineering introduces a capability beyond solving the task itself: agents must coordinate their independently produced work. Strong performance in single-agent coding therefore does not necessarily imply effective collaboration in a multi-agent setting.

Evaluation, however, remains largely focused on final task outcomes. Repository-level benchmarks such as SWE-bench assess the final implementation through executable tests (Jimenez et al., 2024), while multi-agent coding systems often report gains using the same end-task metrics over single-agent baselines (Geng & Neubig, 2026; Wang et al., 2026). These comparisons show whether

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

multi-agent execution performs better, but do not directly measure collaboration: final scores conflate individual coding ability with cross-agent coordination and may hide unresolved dependencies across independently implemented components.

We therefore use cross-component software dependencies as a lens for collaboration and introduce **AsynCodeBench**, a dependency-centric benchmark for asynchronous multi-agent software engineering. Unlike existing evaluations that characterize collaboration through final task outcomes or isolated interaction behaviors, AsynCodeBench makes cross-agent dependency resolution observable throughout execution. We introduce Asynchronous Dependency Pass Rate (ADPR) to capture dependency resolution and Dependency Resolution Step (DRS) to capture resolution timing, enabling collaboration to be evaluated both by its outcome and by how it develops over time.

Our study provides a dependency-centric view of collaboration in multi-agent software engineering, separating collaboration capability from conventional coding performance and making its evolution observable throughout execution. Using AsynCodeBench, we find that multi-agent execution is not uniformly stronger than single-agent execution, and that within the evaluated Qwen family, stronger single-agent coding performance does not improve all aspects of collaboration. Dependency trajectories further reveal a recurring *hopping* phenomenon, where substantial dependency resolution is often established through concentrated phases whose timing varies across models and execution protocols. Finally, dependency-aware evaluation can lead to different quality–cost assessments from conventional test-based metrics. These findings highlight collaboration as a distinct capability that cannot be fully characterized by final coding outcomes alone.

2 Related Work

2.1 Repository-Level Coding Agents and Benchmarks

Repository-level coding benchmarks have emerged to bridge the gap between closed-form code generation and real software engineering, where solving a task requires reasoning over an existing codebase rather than completing an isolated function (Ding et al., 2023; Liu et al., 2024; Jimenez et al., 2024). This setting preserves challenges that are largely abstracted away by function-level evaluation, including cross-file reasoning, software dependencies, execution feedback, and consistency across interacting changes (Gautam et al., 2025; Li et al., 2025; Deng et al., 2025; Le et al., 2026; Ding et al., 2026). To retain these challenges, benchmark construction has increasingly relied on real repositories and development artifacts together with executable validation, preserving much of the surrounding software context encountered in practice (Jimenez et al., 2024; Li et al., 2025; Pan et al., 2025; Miserendino et al., 2025; Badertdinov et al., 2025; Vergopoulos et al., 2025). This repository-grounded paradigm has since expanded across software ecosystems and toward increasingly broad and long-horizon development tasks, ranging from feature and library implementation to software evolution and repository-scale transformation (Yang et al., 2024b; Zan et al., 2025; Rashid et al., 2025; Zhao et al., 2024; Yang et al., 2025; Badertdinov et al., 2026; Le et al., 2026; Hong et al., 2026). Yet as the tasks themselves have become more realistic and extended over time, their evaluation still largely reduces execution to final task completion, leaving intermediate progress and failure dynamics comparatively under-characterized.

2.2 Multi-Agent Software Engineering

Multi-agent software engineering has rapidly expanded from role-based workflows toward repository-scale collaboration, with increasingly sophisticated mechanisms for specialization, delegation, integration, and concurrent execution (Qian et al., 2024; Hong et al., 2024; Liu et al., 2025; Wang et al., 2026; Oliveira et al., 2026; Costa, 2026; Kumar et al., 2026). Despite this shift, evaluation still largely inherits a single-agent perspective, judging multi-agent systems by end-task performance or gains over single-agent baselines rather than measuring collaboration itself (Khatua et al., 2026; Donato et al., 2026; Destefanis & Aste, 2026). Recent work has begun to expose this gap through failure analysis, out-of-sync recovery, role-separated coordination, distributed communication, and interactive coding (Cemri et al., 2025; Guo et al., 2025; Kim et al., 2026; Zhang et al., 2026b; Wu et al., 2026). AsynCodeBench goes further by tracing the dependencies that must be resolved across independently implemented components, revealing failed handoffs and coordination dynamics that can remain hidden behind final task performance.

3 BENCHMARK CONSTRUCTION

3.1 PROBLEM DEFINITION

Multi-agent software engineering often decomposes a coding task across multiple agents, each responsible for a different implementation unit. We refer to collaboration as **asynchronous** when these agents proceed independently without necessarily waiting for dependent agents to complete or propagate their updates. When dependencies exist across their subproblems, this execution pattern can create coordination failures: a downstream agent may continue working with incomplete or outdated upstream behavior, causing the independently produced implementations to become inconsistent when they are integrated. We refer to failures arising from this interaction between asynchronous execution and software dependencies as the **asynchronous coordination problem** studied in AsynCodeBench. A key difficulty is that these failures are often hidden by traditional outcome-only evaluation, since dependencies across agents' subproblems are typically implicit and not explicitly tracked. As illustrated in Figure 1, the evaluator may identify failed components without revealing the stale dependency between agents that produced them. This makes it difficult to determine whether the failure reflects limitations in individual coding capability or in multi-agent collaboration.

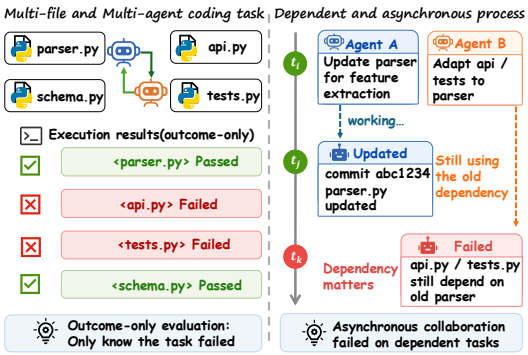


Figure 1: A small illustration of the existing gap.

The key question is therefore whether dependencies between agents' implementations are correctly maintained during collaboration, rather than only whether the final repository succeeds. In AsynCodeBench, a dependency exists when the correctness of code assigned to one agent depends on code implemented by another, such as a required function, interface, data structure, or shared state. We make these dependencies explicit through task-specific dependency graphs, evaluate them with executable Dependency Checkers, and track them across integration checkpoints to determine whether and when they are resolved.

3.2 DEPENDENCY GRAPH AND EXECUTABLE CHECKERS

We operationalize asynchronous coordination through four benchmark components,

$$\mathcal{B} = (\mathcal{R}, \mathcal{G}, \mathcal{C}, \mathcal{E}),$$

where $\mathcal{B}$ denotes AsynCodeBench, $\mathcal{R}$ the repository and target coding task, $\mathcal{G}$ the task-specific dependency graph, $\mathcal{C}$ the executable Dependency Checkers defined over the dependencies in $\mathcal{G}$, and $\mathcal{E}$ the benchmark evaluation.

Each task is represented by a directed dependency graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{A}),$$

where $\mathcal{V}$ is the set of implementation units and $\mathcal{A} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of directed dependency arcs. Each $d \in \mathcal{A}$ connects an upstream producer to a downstream consumer whose correctness depends on behavior provided by the producer. The graph is defined before model evaluation. Appendix A.1 details the dependency construction procedure and supporting repository evidence.

For each dependency $d \in \mathcal{A}$, we pre-register three groups of executable Boolean checkers,

$$\mathcal{C}_d = \left(\mathcal{C}_d^U, \mathcal{C}_d^D, \mathcal{C}_d^I\right), \qquad \mathcal{C} = \{\mathcal{C}_d\}_{d \in \mathcal{A}}.$$

Each constituent checker c returns a binary pass/fail outcome on a repository workspace W, with $c(W) = 1$ indicating that the tested condition is satisfied. The upstream group $\mathcal{C}_d^U$ checks whether the producer provides the required behavior, the downstream group $\mathcal{C}_d^D$ checks whether the consumer

correctly uses or preserves it, and the integrated group $\mathcal{C}_d^I$ checks whether the producer–consumer dependency holds after integration. For $X \in \{U, D, I\}$, the group-level outcome is

$$C_d^X(W) = \mathbf{1}\left[\forall c \in \mathcal{C}_d^X,\, c(W) = 1\right].$$

The upstream and downstream outcomes provide diagnostic evidence about the two sides of a dependency, whereas C_d^I determines whether the dependency itself is satisfied in the integrated workspace.

Let W_t denote the retained integrated workspace at checkpoint t, $t = 1, \ldots, T$. We define

$$I_{d,t} = C_d^I(W_t), \qquad \mathbf{I}_d = (I_{d,1}, \ldots, I_{d,T}),$$

where $I_{d,t} \in \{0, 1\}$ is the satisfaction state of dependency d at checkpoint t. Because the same dependency is evaluated over an ordered sequence of integrated repository states, $\mathbf{I}_d$ records its resolution trajectory over execution. It therefore distinguishes dependencies that never resolve, resolve only after several integration steps, or resolve and later regress—information that is lost when only the final workspace is evaluated. Appendix C details checkpoint construction and indexing.

3.3 Evaluation Metrics

Test Pass Rate. Unlike a binary task success measure, Test Pass Rate quantifies the fraction of evaluation points obtained from the repository-level test suite:

$$\mathrm{TestPass} = \frac{P}{N},$$

where P is the number of passed test points and N is the total number of test points for the current task. It provides proportional credit for partially correct implementations, but does not directly assess whether the benchmark-defined cross-component dependencies are satisfied.

Asynchronous Dependency Pass Rate. Using the pre-registered dependency graph and executable checker states defined above, ADPR measures the fraction of dependencies satisfied in the final integrated workspace:

$$\mathrm{ADPR} = \frac{1}{|\mathcal{A}|} \sum_{d \in \mathcal{A}} I_{d,T}.$$

ADPR therefore provides a dependency-level measure of how successfully the cross-component dependencies identified in AsynCodeBench are resolved after integration.

Dependency Resolution Step. Based on the ordered integrated-checker outcomes defined above, DRS measures the first checkpoint at which a dependency passes its integrated checker:

$$\mathrm{DRS}_d = \min\{t : I_{d,t} = 1\},$$

with $\mathrm{DRS}_d = T + 1$ when dependency d never passes during execution. Since the number of retained checkpoints varies across runs, we normalize the resolution step as

$$\widetilde{\mathrm{DRS}}_d = \frac{\mathrm{DRS}_d}{T + 1},$$

where smaller values indicate earlier resolution and 1 denotes non-resolution. By using the ordered dependency trajectory rather than only its final state, DRS distinguishes dependencies that resolve early from those that become valid only after later integration steps, providing a temporal view of dependency resolution that final ADPR alone cannot capture. DRS is defined over logical integration checkpoints rather than wall-clock time. Since SINGLE exposes only the final workspace, it does not provide a dependency trajectory and is therefore excluded from DRS.

3.4 Task Curation and Human Validation

We curate public repository tasks whose implementations naturally span behaviorally coupled files or modules, preserving existing software boundaries rather than introducing artificial code partitions. AsynCodeBench contains 19 tasks from 16 repositories, covering framework and library engineering, systems and storage, compiler/IR engineering, network protocols, and security. Each

task is packaged with a fixed repository revision, an answer-free task instruction, scoped writable paths, and a task-specific `pytest` evaluator. All execution protocols use the same task package, OpenHands (Wang et al., 2025) harness, and task-specific Docker environment.

To ensure that the dependency structure reflects genuine repository constraints, all dependency graphs undergo independent human validation. Three reviewers verify the validity and direction of each dependency together with its supporting executable evidence, and only validated dependencies are retained. The complete task inventory, packaging details, and dependency-validation protocol are reported in Appendix A.2, with human-review details in Appendix A.3.

4 EVALUATION PROTOCOLS

We organize the five execution protocols as a progression from independent full-task solving to increasingly structured collaboration. SINGLE assigns the entire task to one agent. SERIAL introduces specialist decomposition while preserving producer-to-consumer handoffs, whereas ASYNC-PRIVATE keeps the same decomposition but allows specialists to proceed concurrently in private workspaces. The two manager variants then add an explicit coordination layer, first without write access and then with bounded repair.

The contrast between SERIAL and ASYNC-PRIVATE isolates the effect of asynchronous execution under the same specialist decomposition. In SERIAL, each downstream specialist begins only after the corresponding upstream artifact has been integrated; in ASYNC-PRIVATE, specialists work from the same initial repository state and cannot observe one another's in-flight changes. This creates the setting in which unresolved cross-agent dependencies can emerge.

The manager variants test whether these failures can be mitigated through coordination. ASYNC-MANAGER-RO allows a read-only manager to inspect artifacts, provide feedback, and coordinate integration without modifying production code. ASYNC-MANAGER further permits bounded repair after specialist integration. This distinction separates coordination through information exchange from direct code intervention. Full protocol details are provided in appendix B.

5 EXPERIMENTS

5.1 EXPERIMENT SETTINGS

We evaluate ten open-weight models from the Qwen, DeepSeek, Gemma, Muse, GLM, and Nemotron families, covering dense and mixture-of-experts architectures from 9B dense parameters to 284B total parameters. We use shortened model names. Complete checkpoint identifiers, parameter counts, architectures, and weight precisions are provided in Appendix D.

Using AsynCodeBench, we systematically investigate multi-agent software engineering across four research questions, progressing from collaboration effectiveness and model capability to dependency-resolution dynamics, and resource costs.

- **RQ1** How does multi-agent collaboration compare with single-agent execution, and what aspects of collaboration are overlooked by conventional task-level metrics?
- **RQ2** How do model scale and generation relate to collaborative performance and dependency resolution?
- **RQ3** How do execution protocols shape dependency-resolution outcomes and closure dynamics over successive integration checkpoints?
- **RQ4** How do token costs vary across execution protocols, and how does the choice of quality metric affect the observed quality–cost trade-offs?

5.2 COLLABORATION GAINS AND HIDDEN DEPENDENCY FAILURES

We first ask whether multi-agent execution consistently improves over standalone coding across models. Figure 2(a) compares each model's task-macro Test Pass Rate under Single with its best result among the four multi-agent protocols. This Best Multi score is intentionally optimistic: it

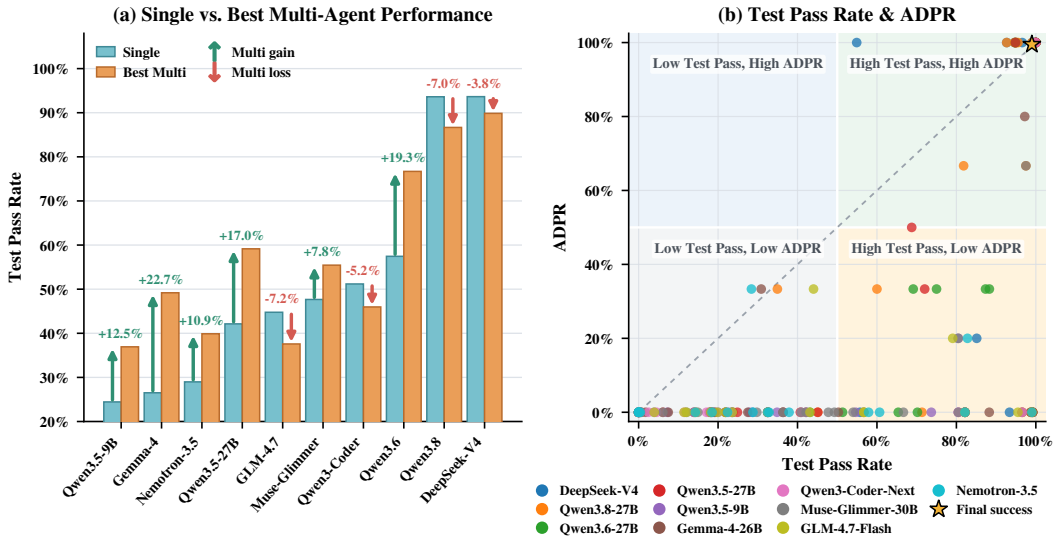


Figure 2: Collaboration gains and hidden dependency failures. (a) For each model, SINGLE is compared with the highest task-macro Test Pass Rate achieved by any of the four multi-agent protocols; arrows show the signed gain. (b) Test Pass Rate versus ADPR for 190 ASYNC-MANAGER-RO model–task outcomes. The dashed line indicates equality between the two metrics, and the 50% boundaries provide a visual partition.

asks whether any collaborative protocol can improve upon the same model acting alone, without yet attributing the gain to a particular protocol.

The answer varies markedly across models. Six of the ten models improve under their best multi-agent protocol, with gains of 7.8–22.7%, while four remain worse than SINGLE even after selecting their best collaborative protocol. The contrast is particularly clear for models with high standalone performance: Qwen3.8-27B and DeepSeek-V4 both reach a 93.6% Test Pass Rate under SINGLE, yet their best multi-agent results decrease to 86.7% and 89.9%, respectively. Thus, multi-agent decomposition can provide substantial gains, but these gains are neither universal nor guaranteed to preserve a model's standalone performance.

Task-level progress, however, does not alone reveal whether collaboration succeeds across component boundaries. Figure 2(b) examines this distinction under ASYNC-MANAGER-RO using all 190 model–task outcomes. Test Pass Rate and ADPR are positively but imperfectly associated (Pearson $r = 0.617$; Spearman $\rho = 0.613$), and their disagreement is strongly asymmetric. Test Pass Rate exceeds ADPR in 136 of 190 outcomes (71.6%), with a mean Test Pass Rate of 48.0% compared with only 18.8% ADPR. More strikingly, 32 outcomes achieve at least 80% Test Pass Rate while resolving fewer than half of their declared dependencies, including 26 with zero ADPR; we observe no corresponding low-test, high-ADPR cases. Results for the other four protocols are in Appendix E.

Overall, multi-agent execution is not uniformly beneficial, and task-level success alone provides an incomplete picture of collaboration. While Test Pass Rate measures broad implementation progress, ADPR directly captures whether dependencies across specialists are successfully resolved, exposing coordination failures that may remain hidden behind high aggregate test performance.

5.3 MODEL CAPABILITY: QWEN SCALE AND GENERATION

The cross-model results above suggest that improvements in standalone coding performance do not necessarily translate into comparable gains under multi-agent execution. To examine this pattern under a more controlled setting, we turn to the Qwen family. We compare Qwen3.5-9B with Qwen3.5-27B to study scaling within the same generation, and Qwen3.5-27B, Qwen3.6-27B, and Qwen3.8-27B to study model evolution at a fixed nominal scale. Figure 3 reports Test Pass Rate, ADPR, and normalized DRS.

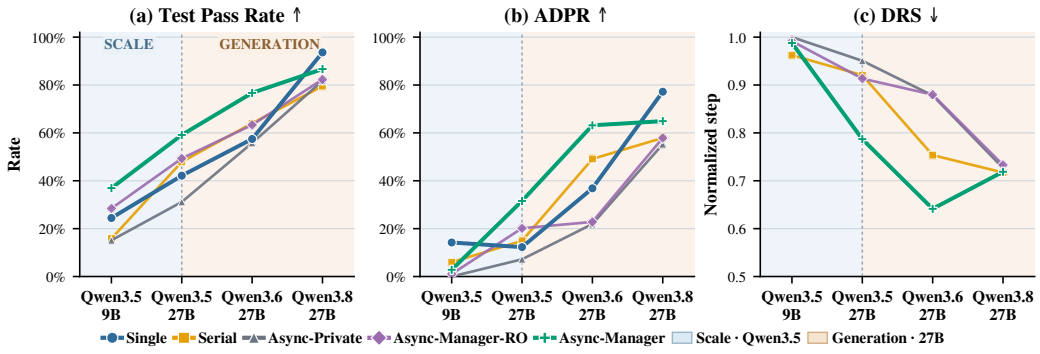


Figure 3: Qwen scale and generation analysis under five matched execution protocols. The left region compares Qwen3.5-9B and Qwen3.5-27B, while the right region compares Qwen3.5, Qwen3.6, and Qwen3.8 at a nominal scale of 27B. Panels report Test Pass Rate, ADPR, and normalized DRS; lower DRS indicates earlier dependency resolution.

Scaling Qwen3.5 from 9B to 27B consistently improves overall test progress, but its effect on dependency resolution is less uniform. Test Pass Rate increases under every protocol, including from 24.4% to 42.1% for SINGLE and from 36.9% to 59.2% for ASYNC-MANAGER. ADPR follows a different pattern: SINGLE changes only from 14.2% to 12.3%, while ASYNC-MANAGER increases from 2.8% to 31.6%. Thus, additional model capacity clearly improves implementation progress, but the extent to which this improvement carries over to cross-component dependency resolution depends strongly on the execution protocol.

The contrast becomes more pronounced across generations. At 27B, SINGLE improves substantially from Qwen3.5 to Qwen3.6 and Qwen3.8, with Test Pass Rate increasing from 42.1% to 57.4% to 93.6% and ADPR from 12.3% to 36.8% to 77.2%. Multi-agent execution also improves, but not at the same rate. Under ASYNC-MANAGER, ADPR rises from 31.6% for Qwen3.5 to 63.2% for Qwen3.6, but changes only marginally to 64.9% for Qwen3.8. In other words, the large improvement observed in standalone dependency resolution from Qwen3.6 to Qwen3.8 is not matched by a comparable gain under collaborative execution.

The trajectory analysis reveals a further distinction that final ADPR alone cannot capture. Under ASYNC-MANAGER, normalized DRS decreases from 0.787 for Qwen3.5 to 0.641 for Qwen3.6, indicating earlier dependency resolution, but rises again to 0.719 for Qwen3.8 despite their similar final ADPR. Two models can therefore reach comparable final dependency coverage while arriving there through different coordination trajectories.

5.4 PROTOCOL EFFECTS AND DEPENDENCY CLOSURE DYNAMICS

The Qwen analysis shows that stronger standalone capability does not uniformly translate into stronger collaboration, and that similar final dependency coverage can arise from different resolution trajectories. We therefore next examine the other side of the interaction: how the execution protocol shapes both the amount and timing of dependency resolution.

Figure 4(a) examines how execution protocols affect dependency resolution by comparing each collaborative setting against SINGLE under the same model. The results reveal substantial model-dependent variation: a protocol that improves dependency resolution for one model may degrade it for another, and even the same model can exhibit opposite outcomes across protocols. For example, Qwen3.6-27B ranges from a 14.9% decrease under ASYNC-PRIVATE to a 26.3% improvement under ASYNC-MANAGER, yielding a 41.2% spread. Moreover, no collaborative protocol consistently improves dependency resolution across all models. These results indicate that collaboration gains depend not only on the underlying model but also on how multi-agent execution is organized.

Figure 4(b) reveals a further pattern that is invisible from final ADPR alone. Using the first-resolution checkpoints captured by DRS, we plot the cumulative fraction of dependencies that have achieved closure over normalized logical progress. The horizontal axis represents each checkpoint's

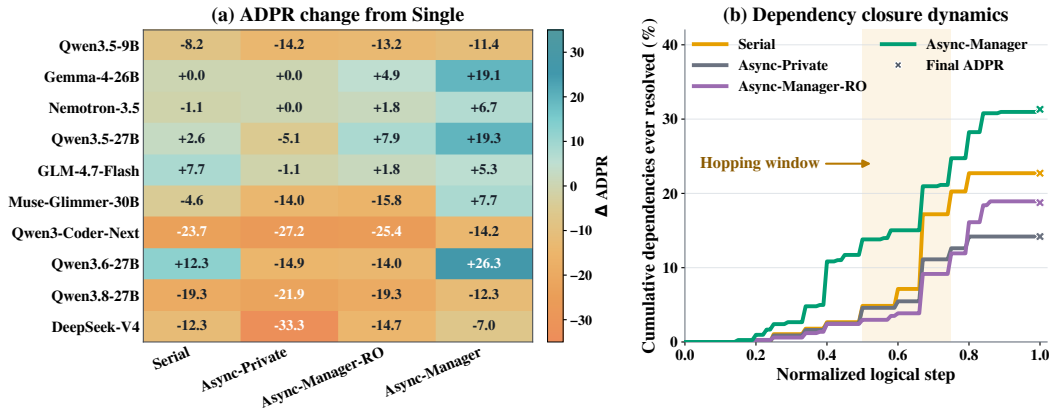


Figure 4: Protocol effects and dependency-closure dynamics. (a) ADPR change relative to Single across models and multi-agent protocols. (b) Cumulative dependency closure over normalized logical steps k/T, showing a recurrent middle-to-late hopping window ($0.50 < k/T \leq 0.75$).

relative position in its execution trajectory, while the vertical axis measures the cumulative fraction of dependencies ever resolved by that point.

Across all four collaborative protocols, the curves exhibit a pronounced increase in the middle-to-late portion of the observed trajectories. Across the aggregated trajectories, a substantial fraction of eventual dependency closures concentrates within $0.50 < x \leq 0.75$. We refer to this region as a representative ***hopping window***, while noting that the onset and duration of the hopping process vary across individual models and protocols. This interval accounts for 34.9%–67.8% of eventual closure mass across protocols. This observation suggests that multi-agent collaboration cannot be fully characterized by its initial decomposition or final outcome. When dependencies become resolved is itself an important dimension of collaboration capability. Appendix F.1 further examines the prevalence of this pattern and the effect of checkpoint normalization.

The aggregate trajectories in Figure 4(b) suggest that the hopping process is closely tied to the formation of substantial dependency resolution. Figure 5 makes this relationship more explicit at the model level. For Gemma-4-26B, protocols whose trajectories remain nearly flat establish little dependency closure, whereas ASYNC-MANAGER develops a pronounced hopping phase and reaches a substantially higher final ADPR. Qwen3-Coder-Next shows that the location of this phase is not fixed: its ASYNC-MANAGER trajectory undergoes a sharp closure burst much earlier in execution, around $x \approx 0.4$, before settling into its final plateau. DeepSeek-V4 exhibits another form of the same phenomenon, with all four protocols undergoing pronounced bursts of dependency closure at later stages of their trajectories.

Across these cases, the common factor is not a particular temporal interval, but the emergence of a concentrated phase in which a substantial share of dependencies becomes resolved. This pattern also holds across the full set of 40 model–protocol configurations: 28 exhibit closure growth within the representative interval $0.50 < x \leq 0.75$, including all 16 configurations reaching at least 20% final ADPR, while configurations with little final dependency resolution often remain nearly flat throughout execution. The onset and duration of the hopping process nevertheless vary substantially across models and protocols, as illustrated by the earlier burst of Qwen3-Coder-Next. These results suggest that substantial dependency resolution is often established through a concentrated hopping process rather than steady accumulation, while the location of that process is configuration-dependent. Complete trajectories for all models are provided in Appendix F.2.

5.5 QUALITY–COST TRADE-OFFS

Beyond collaboration effectiveness, we examine the token costs incurred by different execution protocols. Multi-agent execution introduces additional computational overhead, yet greater token consumption does not necessarily translate into better dependency resolution. To examine this trade-off, we construct within-model Pareto frontiers, identifying protocols that offer quality gains not

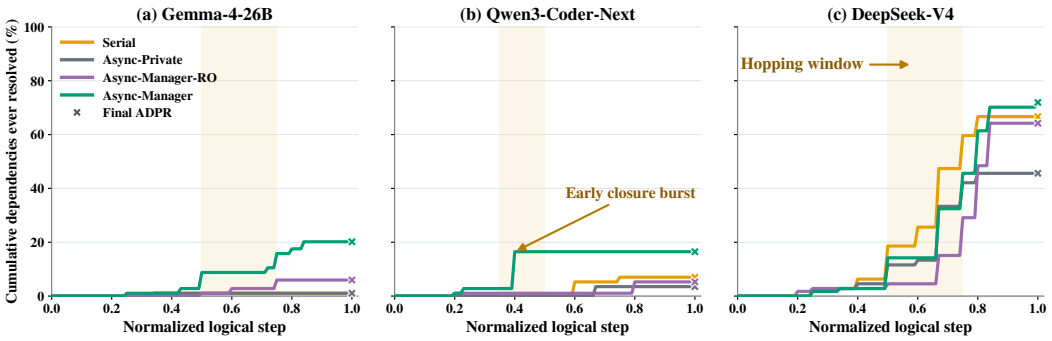


Figure 5: Representative dependency-closure trajectories across execution protocols. Each panel shows the cumulative fraction of dependencies ever resolved over normalized logical steps for one model. The shaded region marks the hopping window.

attainable at a lower token cost. We use mean reported tokens per task as the cost measure and independently evaluate quality using Test Pass Rate and ADPR.

Figure 6 illustrates this metric-dependent trade-off using Nemotron-3.5. Compared with Single, Async-Private consumes more tokens and improves Test Pass Rate from 28.99% to 32.94%, but achieves the same ADPR of 1.05%. It therefore lies on the Test Pass Rate frontier but is dominated by Single under ADPR. Conversely, Async-Manager-RO improves ADPR to 2.81%, placing it on the ADPR frontier. It is dominated by Async-Private under Test Pass Rate, as it incurs higher token costs while achieving lower test performance.

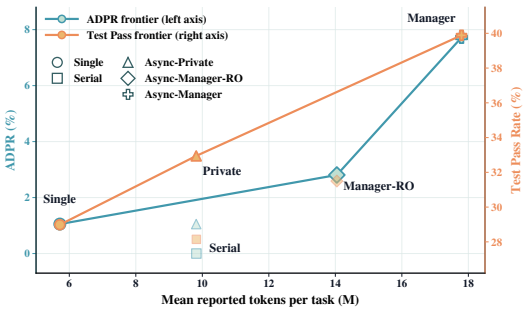


Figure 6: Quality–cost Pareto frontiers for Nemotron-3.5 under ADPR and Test Pass Rate. Both use identical protocols and token costs, yet identify different non-dominated configurations. Marker shapes indicate execution protocols, and lines connect their respective Pareto frontiers.

These differences reveal that the quality–cost assessment of collaboration depends on the metric used to measure quality. Across all ten models, five exhibit different within-model Pareto frontiers under the two metrics, as detailed in Appendix G. Extending our earlier observations of metric mismatch, these results show that task-level performance alone may overlook dependency-resolution gains when evaluating the resource efficiency of multi-agent collaboration. ADPR provides a complementary, dependency-centric perspective on these trade-offs.

6 Conclusion

Asynchronous multi-agent software engineering introduces a challenge beyond task completion: independently produced components must be coordinated correctly. AsynCodeBench studies this challenge through repository-grounded dependencies, making collaboration and its evolution directly measurable during execution. Our results reveal a persistent gap between standalone coding capability and collaborative performance, while also showing that conventional task-level outcomes can hide unresolved coordination failures. By exposing these hidden dependencies and their dynamics over execution, AsynCodeBench provides a complementary view of multi-agent software engineering beyond final task success. We hope this benchmark supports the development of coding agents that are effective both individually and collaboratively.

AI USE STATEMENT

We used OpenAI ChatGPT to assist with language polishing and improving the clarity of the manuscript, literature retrieval and discovery, brainstorming and refinement of research ideas, providing feedback on research methodology and experimental design, and code development and debugging. All methodological choices, benchmark construction, experimental configurations, result verification, and scientific conclusions were independently reviewed, determined, and validated by the authors. The authors take full responsibility for the content of this paper.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide the benchmark tasks, dependency graphs, Dependency Checkers, evaluation scripts, and implementation of all execution protocols in our anonymized repository: https://anonymous.4open.science/r/AsynCodeBench-2640/. The repository also includes the configurations and instructions required to reproduce the experiments and compute the reported task-level and dependency-level metrics. Details of benchmark construction, dependency annotation, evaluation protocols, and metric definitions are provided in the main paper and appendix.

REFERENCES

Anthropic. Building effective agents. Anthropic Engineering, December 2024. URL https://www.anthropic.com/engineering/building-effective-agents. Published December 19, 2024; accessed September 3, 2026.

Anthropic. Claude opus 4.6 system card. Technical report, Anthropic, February 2026a. URL https://www-cdn.anthropic.com/c788cbc0a3da9135112f97cdf6dcd06f2c16cee2.pdf. Section 2.21.3.2 describes a multi-agent evaluation in which the orchestrator has no direct tools and delegates to subagents.

Anthropic. How and when to use subagents in claude code. Claude Blog, April 2026b. URL https://claude.com/blog/subagents-in-claude-code. Published April 7, 2026; accessed September 3, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=nMpJoVmRy1.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench v2: Language-agnostic SWE task collection at scale. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=UCAda9kS57.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025. doi: 10.48550/arXiv.2503.13657. URL https://arxiv.org/abs/2503.13657.

Igor Costa. Agentspawn: Adaptive multi-agent collaboration through dynamic spawning for long-horizon code generation, 2026. URL https://arxiv.org/abs/2602.07072.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Giuseppe Destefanis and Tomaso Aste. When agents coordinate: Measuring coordination in multi-agent ai coding, 2026. URL https://arxiv.org/abs/2608.16801.

Jingzhe Ding, Shengda Long, Changxin PU, Ge Zhang, zhou huan, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, FEI HU, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Chen Qizhi, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Minghao Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Haoran Wang, Zihan Wang, Weihao Xie, Yishuo Yuan, Jiayu Zhang, Yunfei Zhao, He Zhu, LIYA ZHU, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, and Wenhao Huang. NL2repo-bench: Towards long-horizon repository generation evaluation of coding agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=wqQam1muOQ.

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 *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=wgDcbBMSfh.

Benedetta Donato, Noah Hagar-Dent, Aaron Worsnop, Leonardo Mariani, and Valerio Terragni. An evaluation of role-based multiagent code generation on repository-scale problems. *IEEE Software*, 43(5):56–64, 2026. ISSN 1937-4194. doi: 10.1109/ms.2026.3701516. URL http://dx.doi.org/10.1109/MS.2026.3701516.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*, 2024. doi: 10.48550/arXiv.2411.04468. URL https://arxiv.org/abs/2411.04468.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=NiNIthntx7.

Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents, 2026. URL https://arxiv.org/abs/2603.21489.

Xuehang Guo, Xingyao Wang, Yangyi Chen, Sha Li, Chi Han, Manling Li, and Heng Ji. SyncMind: Measuring agent out-of-sync recovery in collaborative software engineering. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 20993–21066. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/guo25l.html.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Swe refactor bench: Can coding agents complete a long-horizon, whole-repository stack migration?, 2026. URL https://arxiv.org/abs/2608.23564.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, zili wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/6507b115562bb0a305f1958ccc87355a-Paper-Conference.pdf.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth*

International Conference on Learning Representations, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann Kaspar Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J Ryan, Jiaxin Pei, and Diyi Yang. Cooperbench: Why coding agents cannot be your teammates yet. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=qo1hehBPVf.

Yubin Kim, Chanwoo Park, Taehan Kim, Eugene Park, Samuel Schmidgall, Salman Rahman, Chunjong Park, Cynthia Breazeal, Xin Liu, Hamid Palangi, Hae Won Park, and Daniel McDuff. Teambench: Evaluating agent coordination under enforced role separation, 2026. URL https://arxiv.org/abs/2605.07073.

Rajesh Kumar, Waqar Ali, Junaid Ahmed, Najma Imtiaz Ali, and Shaban Usman. Agentforge: Execution-grounded multi-agent llm framework for autonomous software engineering, 2026. URL https://arxiv.org/abs/2604.13120.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios, 2026. URL https://arxiv.org/abs/2512.18470.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation, 2025. URL https://arxiv.org/abs/2503.06680.

Jie Liu, Guohua Wang, Ronghui Yang, Jiajie Zeng, Mengchen Zhao, and Yi Cai. RTADev: Intention aligned multi-agent framework for software development. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1548–1581, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.80. URL https://aclanthology.org/2025.findings-acl.80/.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2024. URL https://arxiv.org/abs/2306.03091.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=xZXhFg43EI.

Mariama Celi Serafim De Oliveira, Motunrayo Osatohanmen Ibiyo, Marco Gianrusso, Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. Developing llm-based multi-agent systems in software engineering: A mixed-method experience report, 2026. URL https://arxiv.org/abs/2608.11965.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Cq1BNvHx74.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

Dongsheng Shi, Yue Li, Xin Yi, and Linlin Wang. Emr: Self-evolving medical multi-agent system via experience mining and reuse. *arXiv preprint arXiv:2609.15161*, 2026.

Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks, 2025. URL https://arxiv.org/abs/2503.07701.

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, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Yifei Wang, Ruiyin Li, Peng Liang, Qiong Feng, Zengyang Li, Mojtaba Shahin, and Arif Ali Khan. CodeTeam: An LLM-Powered Multi-Agent Framework for Repository-Level Code Generation. *arXiv preprint arXiv:2606.22082*, 2026.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. Swe-together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026. URL https://arxiv.org/pdf/2606.29957.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=63iVrXc8cC.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=MhBZzkz4h9.

Kaituo Zhang, Zhen Xiong, Mingyu Zhong, Zhimeng Jiang, Zhouyuan Yuan, Zhecheng Li, and Ying Lin. Are tools all we need? unveiling the tool-use tax in llm agents, 2026a. URL https://arxiv.org/abs/2605.00136.

Yuzhe Zhang, Feiran Liu, Yi Shan, Xinyi Huang, Xin Yang, Yueqi Zhu, Xuxin Cheng, Cao Liu, Ke Zeng, Terry Jingchen Zhang, and Wenyuan Jiang. SILO-BENCH: A scalable environment for evaluating distributed coordination in multi-agent LLM systems. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 29379–29398, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1354. URL https://aclanthology.org/2026.acl-long.1354/.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024. URL https://arxiv.org/abs/2412.01769.

A Task Details

A.1 Task-Level Dependency Graphs

For each benchmark task, we first identify the repository-grounded implementation units that must be modified or completed to satisfy the task specification. We then construct a directed graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{A}),$$

where $\mathcal{V}$ contains the implementation units and $\mathcal{A}$ contains directed producer–consumer dependencies among them. An arc $d = (u, v) \in \mathcal{A}$ is included only when the correctness of consumer v depends on behavior, state, an interface, or another artifact provided by producer u. The direction therefore follows the flow of required behavior from producer to consumer.

Dependency arcs are grounded in repository evidence rather than inferred from file proximity or task decomposition alone. For each candidate relation, we inspect the relevant source code, referenced symbols or interfaces, task decomposition, and available public tests to verify that the consumer's correctness genuinely depends on the producer. Relations are excluded when the two implementation units merely co-occur in the same task but can be implemented correctly without satisfying a cross-unit dependency.

The resulting graphs are finalized before model evaluation and remain fixed across all models and execution protocols. Each dependency is manually reviewed for both its direction and supporting evidence, and disagreements during construction are resolved through additional inspection of the repository and associated tests. Freezing the graph before evaluation prevents model behavior or observed outcomes from influencing which dependencies are included.

We present the dependency graph structure for each task. As shown in Figure 7, AsynCodeBench contains nine types of dependency relations. Each dependency specifies well-defined upstream and downstream implementation units, providing the basis for dependency-level evaluation with ADPR.

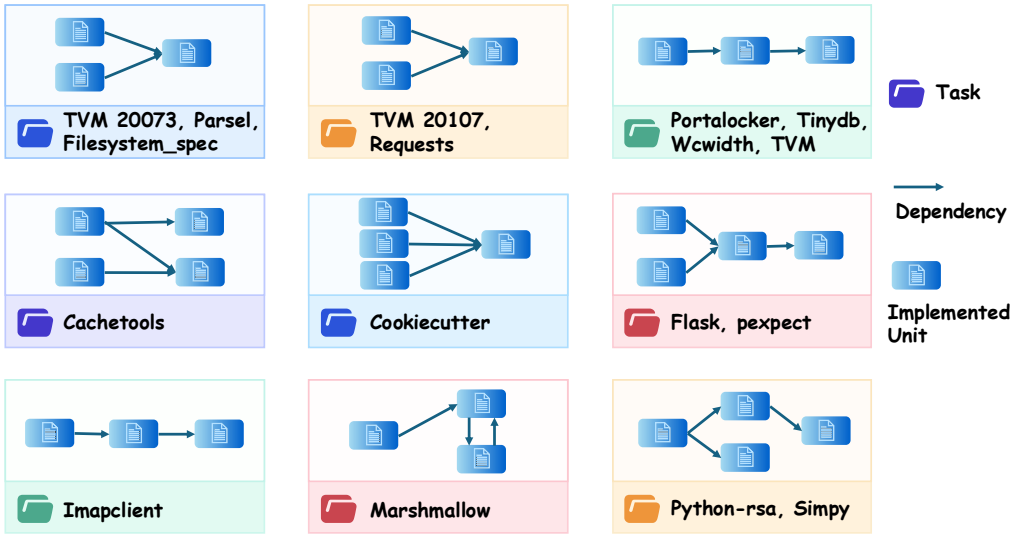


Figure 7: Task-level dependency graphs in AsynCodeBench. Across the benchmark, we identify nine types of dependency relations, each defined by explicit upstream and downstream implementation units. These dependencies form the basis for dependency-level evaluation with ADPR.

A.2 Task Inventory and Provenance

AsynCodeBench contains 19 tasks from 16 public repositories, covering 64 repository-grounded implementation units and 52 directed dependency edges. Fifteen tasks are derived from

Table 1: Complete AsynCodeBench task inventory and provenance. *Agents* denotes the number of specialists used in a multi-agent condition; *Files* denotes the number of distinct implementation files writable by specialists; *Edges* denotes the number of registered dependency edges; *Tests* denotes the number of test files included in the final evaluator; and *Ov.* denotes the number of checksum-pinned public overlays. Domain abbreviations are F/L (framework and libraries), S/S (systems and storage), C/IR (compiler and IR), N/P (network protocols), and SEC (security).

Task	Source record	Domain	Lang.	Agents	Files	Edges	Tests	Ov.
cachetools	Commit0	F/L	Py	2	2	5	12	0
deprecated	Commit0	F/L	Py	2	2	3	8	0
portalocker	Commit0	S/S	Py	2	2	3	5	1
tinydb	Commit0	S/S	Py	2	7	3	8	1
wcwidth	Commit0	S/S	Py	2	2	2	4	0
requests	Commit0	N/P	Py	3	13	3	5	3
simpy	Commit0	S/S	Py	4	9	3	10	0
parsel	Commit0	F/L	Py	3	5	2	6	1
filesystem_spec	Commit0	S/S	Py	3	5	2	4	4
marshmallow	Commit0	F/L	Py	3	9	3	12	4
imapclient	Commit0	N/P	Py	3	9	3	9	5
pexpect	Commit0	S/S	Py	4	9	3	4	1
flask	Commit0	F/L	Py	4	10	3	7	11
python-rsa	Commit0	SEC	Py	4	9	3	8	0
cookiecutter	Commit0	F/L	Py	4	15	3	22	0
apache-tvm-20018	TVM PR #20018	C/IR	Py+C++	3	28	2	14	1
apache-tvm-20073	TVM PR #20073	C/IR	Py+C++	3	7	2	2	1
apache-tvm-20107	TVM PR #20107	C/IR	Py+C++	3	22	2	5	1
apache-tvm-20153	TVM PR #20153	C/IR	Py	3	6	2	3	2

Commit0 (Zhao et al., 2024) and four from merged Apache TVM pull requests; 16 tasks are Python-only, while three involve both Python and C++. Evaluating every task under the five matched execution protocols produces 95 task–protocol scenarios.

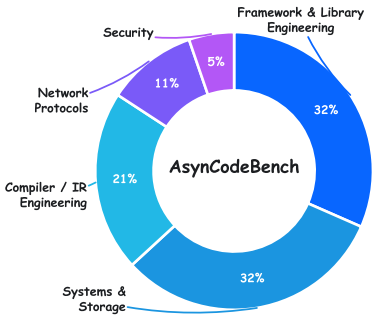


Figure 8: Domain distribution of tasks in AsynCodeBench.

Across all task packages, the evaluators reference 148 task-local public test files and collect 4,884 `pytest` items in completed-state sanity runs. The 52 dependency edges are instrumented with upstream, downstream, and integrated checker groups, yielding 156 checker groups and 508 selector references in total: 125 upstream, 162 downstream, and 221 integrated. After within-task deduplication, these correspond to 274 task-qualified unique test selectors. Task-level evaluators measure overall implementation correctness, whereas Dependency Checkers provide edge-level observations over the execution trajectory; individual selectors may therefore be shared across multiple edges or checker groups.

A.3 Human Review Details

Three reviewers independently examined each benchmark task to verify that it contains meaningful inter-component dependencies and admits a natural decomposition across agents. Each reviewer assessed three fields: `include`, indicating whether the task satisfies the inclusion criteria of AsynCodeBench; `parallelizability_label`, categorized as *parallelizable*, *partially parallelizable*, or *effectively serial*; and `rationale`, providing the justification for the decision. For each dependency edge, reviewers also verified its validity and direction, together with the supporting repository evidence from source code, symbols, tests, or executable Dependency Checkers; disagreements were resolved through discussion. Reviewer judgments were additionally compared

Table 2: Human-review sheet used for task validation. Each task is reviewed independently by three reviewers.

Task Review Sheet	
Task	`<task-name>`
Reviewer	Reviewer 1
Include	☐ Yes ☐ No
Parallelizability	☐ Parallelizable ☐ Partially parallelizable ☐ Effectively serial
Rationale	Reviewer justification for whether the task contains meaningful dependencies and can be naturally decomposed across agents.

Table 3: Execution protocols in AsynCodeBench. "Writable manager" denotes scope-constrained patches produced in a private manager worktree; the manager never directly modifies the integrated workspace.

Protocol	Decomp.	Concurrent	Manager	Writable Manager
SINGLE	–	–	–	–
SERIAL	✓	–	–	–
ASYNC-PRIVATE	✓	✓	–	–
ASYNC-MANAGER-RO	✓	✓	✓	–
ASYNC-MANAGER	✓	✓	✓	✓

against the benchmark dependency graph, and any discrepancy was explicitly recorded. Table 2 provides an example review sheet illustrating the human-review process used for benchmark validation. The results are in our anonymized repository.

B PROTOCOL DETAILS

This section provides the full specification of the five execution protocols summarized in Table 3. Across all protocols, we hold fixed the initial repository state, task instruction, model configuration, final evaluator, and Dependency Checkers. The four multi-agent protocols additionally share the same manifest-defined specialist roles and writable paths, and all specialist artifacts are scope-validated before integration. The protocols differ only in how specialist work is scheduled, what intermediate state is visible, whether a manager coordinates execution, and whether the manager may submit scoped production-code patches.

SINGLE. SINGLE assigns the complete task to one agent with access to the full repository and task instruction. The agent edits the task workspace directly and produces the final implementation without specialist decomposition, handoffs, or manager intervention. This protocol serves as the full-task ownership baseline, avoiding cross-agent coordination while also forgoing specialization and concurrent execution. Because only the final integrated workspace is observed, SINGLE is included in Test Pass Rate and ADPR comparisons but excluded from trajectory-based metrics such as DRS.

SERIAL. SERIAL uses the manifest-defined specialist decomposition but executes specialists sequentially in dependency order. A downstream specialist begins only after the relevant upstream artifact has been scope-validated and integrated, and therefore observes the latest integrated repository state when starting its work. This protocol preserves explicit specialist handoffs while minimizing stale-context effects, at the cost of concurrency.

ASYNC-PRIVATE. ASYNC-PRIVATE uses the same specialist roles and writable scopes as SERIAL, but runs specialists concurrently in isolated private worktrees created from the same initial

repository state. Specialists cannot observe one another's in-flight changes or receive adaptive feedback during their initial execution. Completed artifacts are scope-validated and integrated in a deterministic dependency order. The contrast with SERIAL therefore isolates the effect of asynchronous execution under the same task decomposition: dependent implementations are produced independently and must compose only after integration.

ASYNC-MANAGER-RO. ASYNC-MANAGER-RO retains concurrent private specialist execution and introduces a read-only manager. The manager may inspect the task specification, dependency graph, repository state, specialist artifacts, merge outcomes, and checker diagnostics; it may also plan work, provide feedback, request scoped follow-up attempts, and coordinate integration. It cannot modify production code, so every production patch must originate from a scope-constrained specialist.

This separation follows the orchestrator–worker pattern used in prior multi-agent systems (Fourney et al., 2024; Anthropic, 2024), where a coordinating agent delegates work to specialized agents; related planner-mediated coordination has also been explored in domain-specific multi-agent systems (Shi et al., 2026). Role-specific tool permissions provide a further mechanism for separating coordination from execution (Anthropic, 2026b). A related design appears in Anthropic's reported DeepSearchQA configuration, where the orchestrator acts through delegated subagents rather than direct tools (Anthropic, 2026a). We use read-only authority as an experimental control: it allows explicit coordination without turning the manager into an additional full-context coding agent, while keeping production changes attributable to specialists.

ASYNC-MANAGER. ASYNC-MANAGER preserves the same coordination workflow but additionally permits the manager to submit scoped production-code interventions after specialist integration. The manager operates in its own private worktree, and its writable region is restricted to the union of manifest-defined specialist production paths. Tests, Dependency Checkers, evaluators, task manifests, Git metadata, and out-of-scope files remain protected.

The manager never edits the integrated workspace or specialist worktrees directly. Instead, each proposed patch is recorded, scope-validated, checked against its base revision, committed in the manager worktree, and integrated only if all policy checks pass. Each accepted state-changing intervention creates a new integrated-workspace checkpoint; rejected or no-change interventions do not alter the trajectory. This protocol tests whether coordination benefits from bounded direct repair in addition to planning, feedback, and delegation.

These are controlled system-level contrasts, not an assumed capability ordering. More generally, agentic protocols can themselves introduce execution overhead or alter model behavior beyond the underlying model capability (Zhang et al., 2026a), motivating our explicit comparison of execution protocols rather than treating the surrounding agent framework as neutral.

C INTEGRATED-WORKSPACE CHECKPOINT CONSTRUCTION

Figure 9 illustrates how dependency checkpoints are constructed during execution. The harness records checker outcomes in both private specialist workspaces and the integrated workspace. Private artifact checkpoints are used only for local diagnostics and are not included in the dependency trajectory. Instead, we retain the integrated-workspace observations recorded after specialist integration attempts, accepted state-changing manager interventions when applicable, and final evaluation. These retained observations are ordered by execution order and indexed consecutively as $t = 1, \ldots, T$.

Because the number of integration events and accepted manager interventions varies across runs, T is run-specific. Consequently, the checkpoint index represents logical progress through the observed integrated-workspace trajectory rather than a model turn, tool call, unit of wall-clock time, or fixed amount of computation. SINGLE exposes only the final integrated workspace and is therefore evaluated with ADPR but excluded from temporal DRS analysis.

Table 4: Models evaluated in AsynCodeBench. *Active* denotes the approximate number of parameters activated per token. For dense models, all language-model parameters are active.

Paper name	Evaluated checkpoint	Architecture	Total	Active
Qwen3.5-9B	`Qwen/Qwen3.5-9B`	Dense	9B	9B
Qwen3.5-27B	`Qwen/Qwen3.5-27B`	Dense	27B	27B
Qwen3.6-27B	`Qwen/Qwen3.6-27B`	Dense	27B	27B
Qwen3.8-27B	`Qwen/Qwen3.8-27B`	Dense	27B	27B
Qwen3-Coder-Next	`Qwen/Qwen3-Coder-Next-FP8`	MoE	80B	3B
DeepSeek-V4-Flash	`deepseek/deepseek-v4-flash-0731`	MoE	284B	13B
Gemma-4-26B	`google/gemma-4-26B-A4B-it`	MoE	25.2B	3.8B
Muse-Glimmer-30B	`meta-models/Muse-Glimmer-30B`	Dense	∼29.6B	∼29.6B
GLM-4.7-Flash	`zai-org/GLM-4.7-Flash`	MoE	30B	3B
Nemotron-3.5-Lightning	`nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16`	Hybrid MoE	30B	3B

D Evaluated Model Checkpoints

Table 4 reports the exact model checkpoints used in our experiments and their corresponding names in the main paper. Because the model set includes both dense and mixture-of-experts (MoE) architectures, we report total and per-token activated parameter counts separately. For dense models, the activated parameter count equals the language-model parameter count. For MoE models, the activated count represents the approximate subset of parameters used for each token and can be substantially smaller than the total model size. Weight formats such as FP8 and BF16 describe numerical precision and do not change the parameter count.

E Test Progress and Dependency Resolution Across Protocols

Section 5.2 uses Async-Manager-RO to illustrate the mismatch between task-level test progress and dependency resolution. Figure 10 extends the same analysis to the remaining four protocols. Each panel contains 190 model–task outcomes from the same ten models and 19 tasks, with identical axes and visual encodings to enable direct comparison.

The same asymmetric mismatch appears across protocols. As shown in Table 5, high-test, low-ADPR outcomes account for 22.1–27.4% of observations under every protocol. By contrast, the low-test, high-ADPR region is empty for four protocols and contains only three outcomes under Async-Manager. Thus, substantial test progress frequently occurs without corresponding dependency closure, whereas the reverse pattern is rare.

Table 5: Distribution of model–task outcomes across the four regions in Figure 10 and Figure 2(b). Each protocol contains 190 outcomes. Entries report count (percentage). The 50% threshold is used only as a descriptive visual partition. Blue cells denote the best value according to the indicated direction, while orange cells highlight values discussed in the accompanying analysis.

Protocol	Low Test / Low ADPR ↓	Low Test / High ADPR	High Test / Low ADPR ↓	High Test / High ADPR ↑
SINGLE	97 (51.1%)	0 (0.0%)	**42 (22.1%)**	51 (26.8%)
SERIAL	104 (54.7%)	0 (0.0%)	44 (23.2%)	42 (22.1%)
ASYNC-PRIVATE	116 (61.1%)	0 (0.0%)	49 (25.8%)	25 (13.2%)
ASYNC-MANAGER-RO	105 (55.3%)	0 (0.0%)	**52 (27.4%)**	33 (17.4%)
ASYNC-MANAGER	**83 (43.7%)**	**3 (1.6%)**	50 (26.3%)	**54 (28.4%)**

Table 6: Correlation and mismatch diagnostics by protocol. Percentages are computed over 190 model–task outcomes per protocol. The signed gap is Test Pass Rate minus ADPR; the absolute gap ignores direction. Arrows indicate the preferred direction where applicable. Blue cells denote the best value according to the indicated direction, while orange cells highlight values discussed in the accompanying analysis.

Metric	Single	Serial	Async-P.	Manager-RO	Manager
Pearson r	0.716	0.667	0.599	**0.617**	0.628
Spearman ρ	0.737	0.651	0.559	**0.613**	0.657
Mean Test Pass Rate ↑	51.0%	47.9%	41.9%	**48.0%**	**57.4%**
Mean ADPR ↑	27.4%	22.7%	14.2%	**18.8%**	**31.3%**
Mean signed gap ↓	**23.7%**	25.2%	27.8%	**29.2%**	26.1%
Test Pass Rate > ADPR ↓	**62.6%**	70.5%	77.9%	**71.6%**	67.4%
Test Pass $\geq 80\%$, ADPR $< 50\%$ ↓	**12.6%**	**12.6%**	15.8%	**16.8%**	16.3%
Test Pass $\geq 80\%$, ADPR $= 0$ ↓	**10.0%**	12.1%	12.1%	**13.7%**	14.2%
Final task success ↑	**21.6%**	13.2%	8.4%	13.2%	21.1%

Table 6 provides complementary continuous diagnostics. Test Pass Rate and ADPR remain positively associated under all five protocols, with Pearson correlations of 0.599–0.716 and Spearman correlations of 0.559–0.737. However, Test Pass Rate exceeds ADPR in 62.6–77.9% of outcomes, with mean signed gaps of 23.7–29.2%. Moreover, 12.6–16.8% of outcomes achieve at least 80% Test Pass Rate while resolving fewer than half of their declared dependencies. The discrepancy observed in Section 5.2 is therefore not specific to ASYNC-MANAGER-RO.

F DIAGNOSING DEPENDENCY CLOSURE DYNAMICS

F.1 AGGREGATE CLOSURE CONCENTRATION AND NORMALIZATION

Figure 4(b) shows that first dependency closures are not distributed uniformly over the aggregated execution trajectories. A particularly strong concentration appears within $0.50 < x \leq 0.75$, where x denotes the normalized logical step. We treat this interval as a representative region of the aggregate hopping phenomenon rather than as a fixed temporal definition, since the onset and duration of concentrated closure may vary across individual models and protocols.

Table 7 quantifies this aggregate concentration. The first three columns report the cumulative fraction of dependencies that have achieved closure by the halfway point, the three-quarter point, and at least once during execution, respectively. The final column reports the fraction of eventual closure mass accumulated within the representative interval. Under SERIAL, for example, cumulative closure increases from 4.9% to 20.3% between $x = 0.50$ and $x = 0.75$, accounting for 67.8% of its eventual closure mass. Across the four protocols, 34.9%–67.8% of eventual closure mass is accumulated within this interval, confirming that a substantial portion of dependency resolution is concentrated within a relatively short segment of the execution trajectory.

Table 7: Cumulative dependency closure and hopping-window concentration across execution protocols.

	Cumulative closure (%)			
Protocol	By 50%	By 75%	Ever closed	Closure in $(0.50, 0.75]$ (%)
SERIAL	4.9	20.3	22.7	67.8
ASYNC-PRIVATE	4.6	12.6	14.2	56.5
ASYNC-MANAGER-RO	3.0	11.9	18.9	47.3
ASYNC-MANAGER	13.8	24.7	31.3	34.9

Table 8: Effect of checkpoint normalization on the hopping-window pattern.

Protocol	Peak events from $k = 2, T = 3$	First closure at raw $k = 2$
SERIAL	68.9%	58.6%
ASYNC-PRIVATE	73.9%	60.2%
ASYNC-MANAGER-RO	66.7%	38.4%
ASYNC-MANAGER	50.0%	48.6%

A natural concern is whether this concentration is induced by the normalization used to align trajectories of different lengths. Let k denote the first integrated checkpoint at which a dependency closes and T the total number of integrated checkpoints in the corresponding run, giving the normalized position $x = k/T$. Although this mapping allows trajectories with different numbers of checkpoints to be compared on a common scale, discrete checkpoint structures can accumulate at shared ratios such as $2/3$.

Table 8 shows that normalization indeed sharpens the aggregate peak: 50.0%–73.9% of first closures around this region correspond to $k = 2, T = 3$. However, the underlying concentration remains visible in raw checkpoint space. Across all four protocols, $k = 2$ is still the most common first-closure checkpoint, accounting for 38.4%–60.2% of eventual closures. Normalization therefore accentuates the observed hopping pattern but does not create the underlying burst of dependency closure.

The aggregate concentration, however, does not imply that every model–protocol configuration follows the same temporal pattern. Figure 11 exposes substantial heterogeneity beneath the protocol-level averages: while some configurations place a large share of their eventual first closures within $0.50 < x \leq 0.75$, others contribute little or no closure in this region. The interval therefore captures where closure is most strongly concentrated in aggregate, but not a fixed boundary that governs every execution. This distinction motivates a closer examination of the complete model-level trajectories in Section F.2, where we show that the hopping process can emerge earlier or later depending on the model and execution protocol.

F.2 MODEL-LEVEL VARIATION IN HOPPING DYNAMICS

The variation in Figure 11 raises a natural question: if the aggregate hopping concentration is not shared uniformly across configurations, how does dependency closure actually unfold at the model level? Figure 12 shows the complete trajectories for the remaining seven models, complementing the three representative cases in Figure 5.

The trajectories reveal that the main variation lies not in whether closure must occur within a particular normalized interval, but in when and how strongly a concentrated closure phase emerges. Low-closure configurations, such as many protocols for Qwen3.5-9B, Nemotron-3.5, and GLM-4.7-Flash, remain largely flat throughout execution. By contrast, configurations with substantial final closure exhibit pronounced bursts, but their timing differs considerably. For Qwen3.5-27B and Qwen3.6-27B, ASYNC-MANAGER begins accumulating dependencies well before the representative aggregate interval, whereas Qwen3.8-27B concentrates much of its closure later in the trajectory. Similar differences also appear across protocols within the same model. These trajectories therefore support a broader interpretation of hopping as a configuration-dependent phase of

Table 9: Comparison of within-model Pareto frontiers under ADPR and Test Pass Rate. The two quality metrics are evaluated using identical protocols and token costs. Only metric-exclusive configurations are listed.

Model	Same membership?	ADPR-only protocols	Test-Pass-only protocols
DeepSeek-V4	Yes	–	–
Qwen3.8-27B	Yes	–	–
Qwen3.6-27B	Yes	–	–
Qwen3.5-27B	Yes	–	–
Qwen3-Coder-Next	Yes	–	–
Qwen3.5-9B	No	–	Async-Manager
Gemma-4-26B	No	–	Serial
Muse-Glimmer-30B	No	–	Serial
GLM-4.7-Flash	No	Serial	–
Nemotron-3.5	No	Async-Manager-RO	Async-Private

concentrated dependency closure, whose onset and magnitude vary across models and execution protocols.

G Additional Quality–Cost Analysis

To complement the Nemotron-3.5 case study in the main text, we extend the quality–cost analysis to all ten evaluated models. For each model, we independently construct Pareto frontiers using ADPR and Test Pass Rate as alternative quality measures, while keeping the candidate protocols and their token costs fixed. A protocol is dominated if another protocol achieves equal or higher quality at equal or lower token cost, with at least one strict improvement. Table 9 summarizes the resulting differences in frontier membership. Five models exhibit different frontiers, yielding two ADPR-only and four Test-Pass-only configurations.

Figure 13 presents the other four models with differing frontiers. For Qwen3.5-9B, Single achieves the highest ADPR while consuming the fewest tokens, leaving all collaborative protocols dominated under ADPR. However, Async-Manager substantially improves Test Pass Rate and therefore enters the corresponding frontier. A similar metric-dependent pattern appears in Gemma-4-26B and Muse-Glimmer-30B: Serial improves Test Pass Rate over Single but achieves equal or lower ADPR at a higher token cost, making it non-dominated only under Test Pass Rate. GLM-4.7-Flash exhibits the opposite pattern. Serial improves ADPR over Single and enters the ADPR frontier, whereas Single achieves higher Test Pass Rate at a lower token cost, dominating all collaborative protocols under the conventional metric.

These cases demonstrate that improvements in test performance do not necessarily translate into dependency-resolution gains, and vice versa. Consequently, the same execution protocols can yield different non-dominated configurations depending on the quality metric used. Together with the Nemotron-3.5 case study, these results extend the metric mismatch observed in the main experiments to cost-aware evaluation, illustrating the complementary role of dependency-centric metrics. The complete numerical results underlying these comparisons are reported in Tables 10 and 11.

H Case Study: When High Test Pass Rate Masks Unresolved Dependencies

Figure 14 illustrates why task-level and dependency-level evaluation can lead to different views of the same execution. Under DeepSeek-V4 with Async-Manager on TVM-20073, the final workspace passes 24 of 28 evaluator items, yielding a Test Pass Rate of 85.7%. From the final test outcome alone, the implementation appears largely complete.

The dependency trajectory reveals a different structure behind these results. The task contains two dependencies: source-coordinate mapping and IRBuilder active-span state must both be incorpo-

rated into a shared span-propagation implementation. Source mapping is available from the first retained checkpoint, and both upstream requirements are satisfied by checkpoint 3. However, the downstream propagation logic remains unsatisfied after two subsequent attempts, leaving both dependencies unresolved at final evaluation and yielding an ADPR of $0/2 = 0\%$.

The discrepancy arises because the two metrics count different units. The four failed evaluator items all exercise the same missing span-propagation behavior, whereas ADPR identifies the two unresolved dependencies underlying those failures. Thus, several test failures that appear separately in the final evaluator trace back to a shared coordination bottleneck: the required upstream capabilities are available, but they are never successfully composed in the downstream implementation.

This case highlights the complementary information provided by dependency-aware evaluation. Test Pass Rate summarizes repository-level correctness over evaluator items, while the dependency trajectory identifies whether required cross-agent dependencies are established and when they become available. The latter therefore exposes coordination structure that is not visible from the final test outcome alone.

I Detailed Results of All Models

Tables 10 and 11 report the complete experimental results for all ten models under five execution protocols, covering Test Pass Rate, ADPR, normalized DRS, and mean token consumption per task. The results are aggregated over the same 19 benchmark tasks for each model–protocol configuration, providing the numerical basis for the model capability, protocol, and quality–cost analyses in the main text. Test Pass Rate and ADPR are reported as percentages, with higher values indicating better performance, whereas lower DRS indicates earlier dependency resolution. DRS is not reported for Single, which has no intermediate integration checkpoints. Token consumption is expressed in millions per task; the marked Qwen3.5-9B Async-Manager result represents a lower bound due to one incomplete token record.

We also report the Final Task Success Rate in Table 12. Unlike the Test Pass Rate, the Final Task Success Rate adopts a stricter criterion: a task is considered successful only if it achieves both a complete task pass and full dependency resolution (ADPR = 1). For each model–protocol combination, we report the average success rate across the 19 tasks.

Table 10: Complete experimental results for the Qwen model family. Test Pass Rate and ADPR are reported as percentages. Tokens denote mean reported tokens per task in millions.

Protocol	Test Pass ↑	ADPR ↑	DRS ↓	Tokens (M) ↓
		Qwen3.5-9B		
Single	24.44	14.21	–	5.18
Serial	15.83	5.96	0.962	8.01
Async-Private	15.13	0.00	1.000	8.32
Async-Manager-RO	28.44	1.05	0.992	15.53
Async-Manager	36.93	2.81	0.988	15.48†
		Qwen3.5-27B		
Single	42.13	12.28	–	4.13
Serial	47.91	14.91	0.920	6.18
Async-Private	31.15	7.19	0.951	6.67
Async-Manager-RO	49.33	20.18	0.913	12.41
Async-Manager	59.15	31.58	0.787	14.00
		Qwen3.6-27B		
Single	57.45	36.84	–	4.96
Serial	64.01	49.12	0.754	8.35
Async-Private	55.79	21.93	0.878	8.18
Async-Manager-RO	63.32	22.81	0.880	13.88
Async-Manager	76.70	63.16	0.641	14.49
		Qwen3.8-27B		
Single	93.62	77.19	–	3.45
Serial	79.53	57.89	0.717	8.02
Async-Private	81.76	55.26	0.727	7.09
Async-Manager-RO	82.33	57.89	0.733	12.78
Async-Manager	86.67	64.91	0.719	13.73
		Qwen3-Coder-Next		
Single	51.18	30.70	–	4.69
Serial	39.77	7.02	0.967	9.49
Async-Private	33.55	3.51	0.982	8.66
Async-Manager-RO	28.83	5.26	0.977	17.89
Async-Manager	45.97	16.49	0.886	16.38

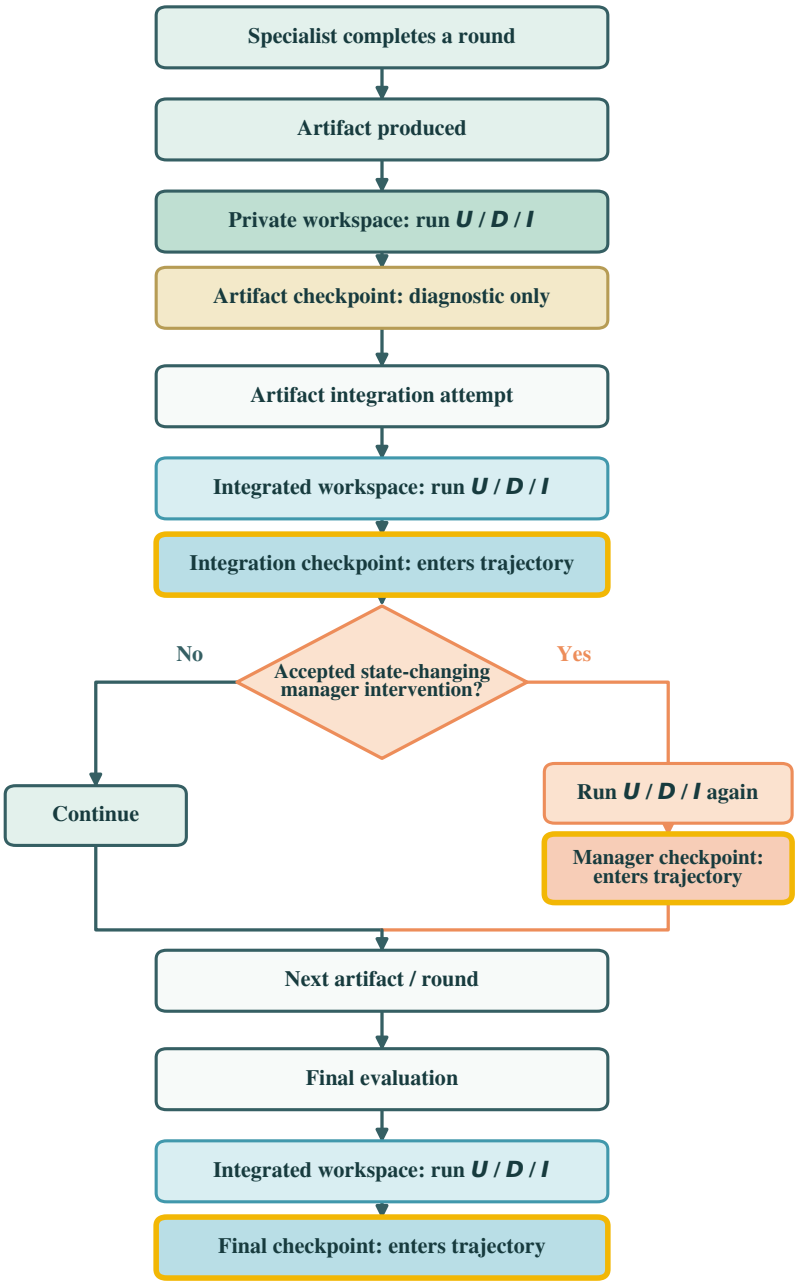


Figure 9: Event-triggered checkpoint construction in AsynCodeBench. The artifact checkpoint evaluates the three checker groups in a private specialist workspace and is used only for local diagnostics. Integrated checker outcomes are recorded at the gold-bordered trajectory checkpoints after integration attempts, accepted state-changing manager interventions when applicable, and final evaluation. Only these ordered integrated-workspace outcomes form the dependency trajectory used by ADPR and DRS.

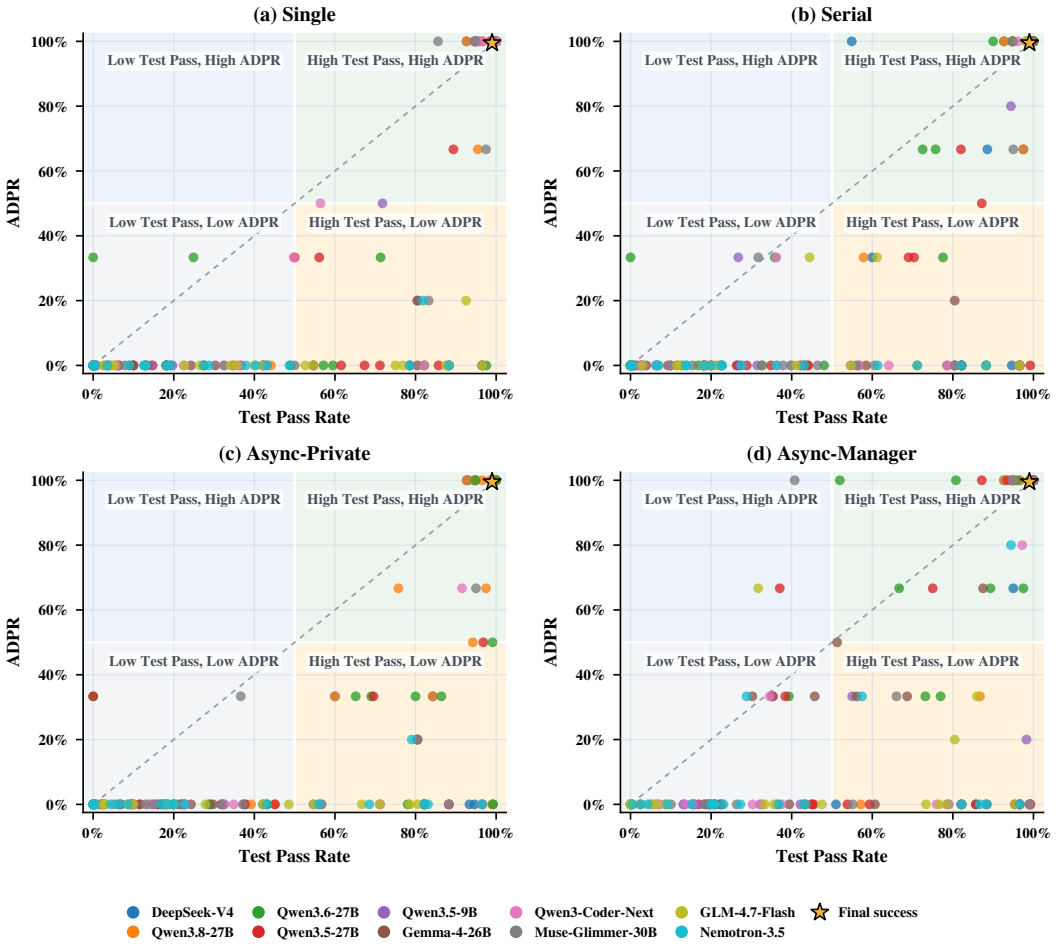


Figure 10: Test Pass Rate versus ADPR under the four protocols not shown in Figure 2(b). Each point represents a model–task outcome and colors denote models. The dashed line indicates equality between Test Pass Rate and ADPR, while the 50% boundaries provide a common visual partition. All panels use identical axes and encodings.

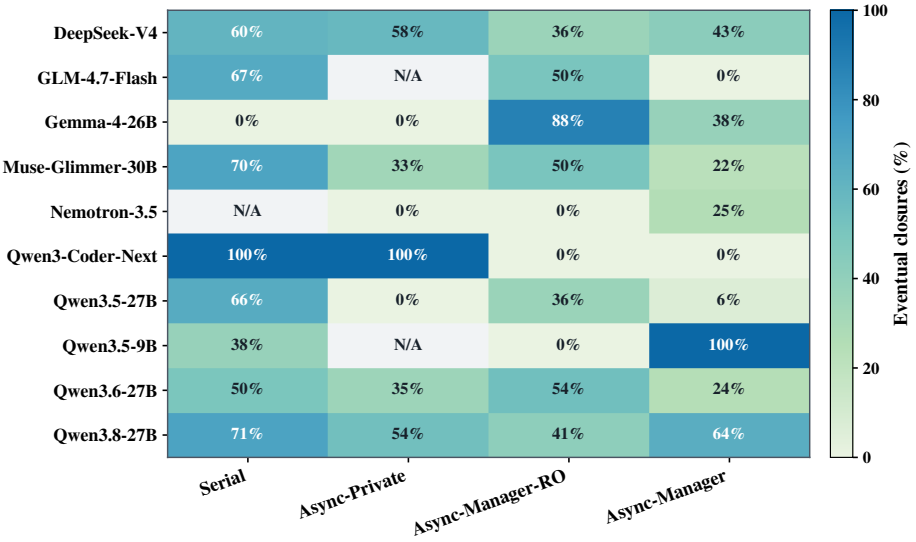


Figure 11: Model–protocol variation in hopping-window concentration. Each cell shows the share of eventual first closures occurring within the hopping window (0.50, 0.75]. N/A denotes no observed dependency closure.

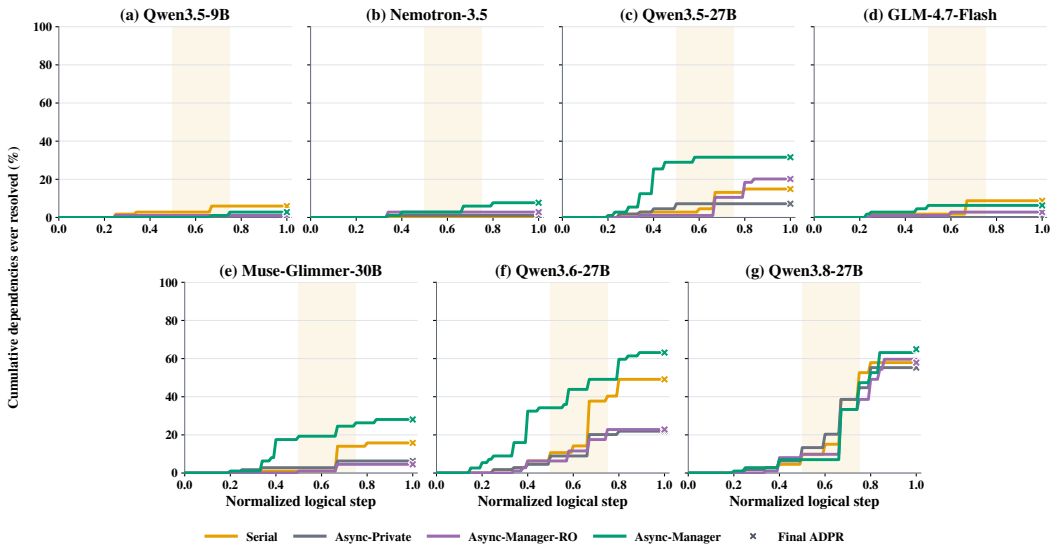


Figure 12: Dependency-closure trajectories for the seven models not shown in the main-text case studies. Curves report cumulative dependencies ever resolved over normalized logical execution, and crosses denote final ADPR. The shaded region marks the representative aggregate concentration $0.50 < x \leq 0.75$. Individual trajectories show that the onset and magnitude of concentrated closure vary substantially across models and protocols.

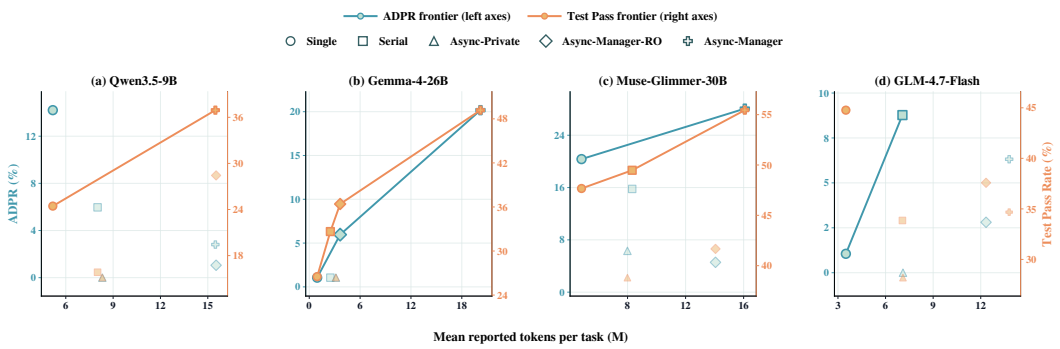


Figure 13: Metric-dependent quality–cost trade-offs for four additional models. Each panel shows the within-model Pareto frontiers under ADPR (blue, left axis) and Test Pass Rate (orange, right axis). Marker shapes distinguish execution protocols, and lines connect non-dominated configurations under each metric. Dominated configurations remain visible. Axis ranges are independently scaled across models.

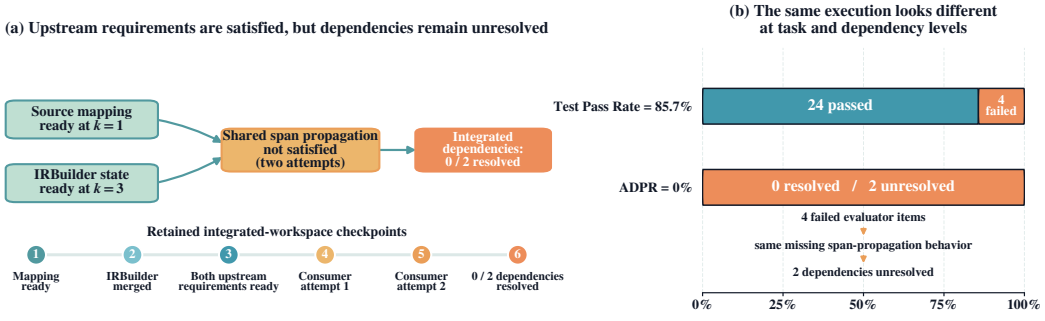


Figure 14: Case study of DeepSeek-V4 under ASYNC-MANAGER on TVM-20073. Although both upstream requirements become available during execution, the shared downstream span-propagation logic remains unsatisfied after two attempts, leaving both dependencies unresolved. The final workspace nevertheless passes 24 of 28 evaluator items (85.7% Test Pass Rate), while ADPR remains 0%, illustrating how task-level performance can obscure unresolved cross-agent dependencies and the coordination failures behind them.

Table 11: Complete experimental results for the remaining model families. Test Pass Rate and ADPR are reported as percentages. Tokens denote mean reported tokens per task in millions.

Protocol	Test Pass ↑	ADPR ↑	DRS ↓	Tokens (M) ↓
DeepSeek-V4				
Single	93.64	78.95	–	2.54
Serial	87.84	66.67	0.665	6.89
Async-Private	75.75	45.61	0.769	6.92
Async-Manager-RO	80.36	64.21	0.748	11.61
Async-Manager	89.85	71.93	0.690	13.65
Gemma-4-26B				
Single	26.51	1.05	–	0.93
Serial	32.65	1.05	0.992	2.49
Async-Private	26.37	1.05	0.992	3.14
Async-Manager-RO	36.40	5.96	0.972	3.64
Async-Manager	49.18	20.18	0.905	20.20
Muse-Glimmer-30B				
Single	47.67	20.35	–	4.83
Serial	49.48	15.79	0.921	8.32
Async-Private	38.81	6.32	0.961	7.99
Async-Manager-RO	41.67	4.56	0.976	14.06
Async-Manager	55.45	28.07	0.832	16.08
GLM-4.7-Flash				
Single	44.77	1.05	–	3.50
Serial	33.85	8.77	0.951	7.06
Async-Private	28.20	0.00	1.000	7.10
Async-Manager-RO	37.58	2.81	0.983	12.34
Async-Manager	34.68	6.32	0.957	13.79
Nemotron-3.5				
Single	28.99	1.05	–	5.72
Serial	28.14	0.00	1.000	9.82
Async-Private	32.94	1.05	0.992	9.82
Async-Manager-RO	31.50	2.81	0.980	14.04
Async-Manager	39.89	7.72	0.959	17.79

Table 12: Number of tasks with successful outcomes across models and execution protocols. Each entry reports the number of successful tasks out of 19, with the corresponding percentage in parentheses.

Single	Serial	Async-Private	Async-Manager-RO	Async-Manager
		Qwen3.5-9B		
2/19 (10.5%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)
		Gemma-4-26B		
0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	1/19 (5.3%)
		Nemotron-3.5		
0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)
		Qwen3.5-27B		
0/19 (0.0%)	1/19 (5.3%)	0/19 (0.0%)	3/19 (15.8%)	2/19 (10.5%)
		GLM-4.7-Flash		
0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)	0/19 (0.0%)
		Muse-Glimmer-30B		
1/19 (5.3%)	2/19 (10.5%)	0/19 (0.0%)	0/19 (0.0%)	3/19 (15.8%)
		Qwen3-Coder-Next		
3/19 (15.8%)	0/19 (0.0%)	0/19 (0.0%)	1/19 (5.3%)	2/19 (10.5%)
		Qwen3.6-27B		
6/19 (31.6%)	5/19 (26.3%)	2/19 (10.5%)	3/19 (15.8%)	7/19 (36.8%)
		Qwen3.8-27B		
14/19 (73.7%)	8/19 (42.1%)	7/19 (36.8%)	7/19 (36.8%)	12/19 (63.2%)
		DeepSeek-V4		
15/19 (78.9%)	9/19 (47.4%)	7/19 (36.8%)	11/19 (57.9%)	13/19 (68.4%)