SCALING INFERENCE-TIME COMPUTE WITH TEAMS OF CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent systems promise a new axis for scaling inference: rather than having one agent think longer or sample more attempts, a task is split across many agents that work in parallel and coordinate. However, in practice, more agents can easily produce duplicate work, coordination failures, and higher costs. We therefore formulate team design as a compute allocation problem: starting from a solo agent, where should additional inference compute go to reach the reward-cost Pareto frontier? We build a minimal multi-agent harness from the ground up and trace this frontier by varying, one at a time, the workflow stage at which a lead delegates to its workers (planning, implementation, or verification), the models that form the team, and the number of workers. On long-horizon challenging software-engineering tasks (e.g., ProgramBench), teams outperform solo agents but cost three to seven times as much. With the right compute allocation, however, a heterogeneous 16-agent team comes within 1.4 points of a strong homogeneous 4-agent team at 70% lower cost. This efficiency stems from three findings. First, planning is the most valuable use of workers: in a homogeneous team whose lead decides how many workers to spawn, a lead that delegates only planning to its workers, and then implements and verifies the code itself, matches the reward of a lead that delegates to workers throughout planning, implementation, and verification, at half the cost. Second, cheap models pay off as planners but not as implementers: a strong lead with cheap planners recovers 47.5% of the reward gain of an all-strong team for only 13.9% of its additional cost, whereas cheap implementers lower reward. Third, planners keep improving reward as the team grows, whereas implementers and verifiers plateau beyond four workers: additional implementers fragment the code into more files for the lead to integrate, and additional verifiers add little new information about the program. Together, our results offer a guide to spending an agent budget where it counts, given the task, cost, and latency constraints a user has.

1 INTRODUCTION

Coding agents can now solve long-horizon software-engineering tasks that require navigating large repositories, editing code, invoking tools, and iteratively debugging their own solutions (Yang et al., 2024; DeepSeek-AI et al., 2026; Z.ai, 2026; Team et al., 2026; OpenAI, 2026; Anthropic, 2026). The dominant way to make such an agent stronger at inference time is to scale its compute sequentially, by letting it reason or revise for longer (Jaech et al., 2024; Guo et al., 2025; Muennighoff et al., 2025), or in parallel, by sampling independent attempts and aggregating them (Wang et al., 2022; Brown et al., 2024; Snell et al., 2024). Both are bounded by a single agent's trajectory. Longer traces increase latency and eventually exhaust the context window, and independent attempts repeat the work rather than dividing it. A team of agents offers a third axis: it divides a single attempt among agents that work on different parts of the task, whose work must then be consolidated into one solution. Team members can explore competing hypotheses, inspect different parts of a repository, implement separable changes, and independently verify a candidate solution. This provides a new test-time scaling strategy for when a single agent's context window or time budget becomes limiting.

Adding agents, however, does not automatically add useful work. More agents bring more compute, which can raise solution correctness, but in practice these gains can be lost to duplicated work, conflicting edits, and the cost of consolidating what each agent produces (Geng & Neubig, 2026;

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.

Yang et al., 2026b). Whether a team helps therefore depends less on how many agents it has than on how it spends its compute.

We make this question precise by treating team design as a compute allocation problem (Section 2). Starting from a solo agent, the next dollar of inference can go to a stronger lead, to stronger or cheaper workers, to more workers, or to a particular stage of the software-engineering workflow (Royce, 1987): planning (**P**), implementation (**I**), or verification (**V**). Each choice moves the team to a different point in reward and cost. We ask which choices reach the reward-cost Pareto frontier, and which buy the most reward per added dollar.

To answer this, we build a minimal, configurable harness with a hierarchical lead-worker topology (Figure 1): a lead agent decomposes the task, delegates subtasks to workers, and consolidates their results. Because every part of the harness is exposed, we can vary one design choice at a time: the stage at which the lead delegates (P, I, V, or all three, PIV), the lead and worker models, the number of workers, whether the lead waits for its workers or keeps working while they run, and whether workers can message each other. We focus on ProgramBench (Yang et al., 2026a), a long-horizon repository-reconstruction benchmark in which agents receive only a black-box executable and its documentation, and must discover its behavior, plan, implement, and verify a complete program that reproduces it. We complement it with Commit0 (Zhao et al., 2024), which also asks agents to build a repository from scratch but provides its skeleton, including the file layout and function stubs, together with its unit tests. We evaluate frontier models from the GPT and Claude Opus families.

Our results show that **teams outperform solo agents, but at a steep price**: a team composed of all strong agents raises reward over a single agent working alone, yet costs 3 to 7 times as much in inference dollars and, in most configurations, takes longer to finish (Section 4). We find, however, that **with the right allocation of compute, a heterogeneous 16-agent team can come within 1.4 points of a strong homogeneous 4-agent team at 70% lower cost** (Section 7). This configuration rests on where the compute goes. On ProgramBench, **planning emerges as the most valuable use of workers**: when the lead delegates planning to a swarm of workers and then writes and verifies the code itself, the team matches the reward of a lead that delegates all three stages (planning, implementation, and verification) to its workers, at **half its cost**, and each dollar spent on planning buys up to twice the reward of a dollar spent on any other stage (Section 5). Adding more workers to implementation or verification, in contrast, quickly plateaus. Planning wins here because it is the bottleneck of this task; on Commit0, where the repository already dictates the plan, implementation benefits most instead (Appendix C.13). Because planning workers only gather information for the lead to act on, **they can be cheap**: a strong lead with cheap planners recovers 47.5% of the reward gain of an all-strong team for only 13.9% of its additional cost(Section 6).

2 Team Design as Budget-Constrained Optimization

A single agent spends its inference budget along one trajectory. A team, by contrast, distributes it across several design dimensions: the capability of the lead, the capability and number of workers, and the workflow stage at which workers are deployed. These dimensions are not independent, since the value of cheaper workers depends on the lead that directs them, and the value of additional workers depends on the stage to which they are assigned. We therefore formulate team design as a budget allocation problem.

Formulation. We consider hierarchical teams in which a lead agent decomposes the task, delegates subtasks to workers, and consolidates their results (Section 3). A team configuration $c = (s, w, n, \ell)$ is specified by the stage $s \in \{\mathrm{P}, \mathrm{I}, \mathrm{V}, \mathrm{PIV}\}$ toward which the lead is instructed to direct its workers, the worker model w, the number of workers n, and the lead model ℓ. On a task distribution $\mathcal{T}$, each configuration attains a mean reward $R(c; \mathcal{T})$ at a total inference cost $C(c; \mathcal{T})$.[1] Ideally, designing a team under a budget B would amount to solving

$$R^*(B; \mathcal{T}) \;=\; \max_{c} \; R(c; \mathcal{T}) \quad \text{subject to} \quad C(c; \mathcal{T}) \le B, \tag{1}$$

[1] We treat wall-clock latency as a secondary objective rather than a constraint in Eq. 1. It is governed mainly by the scheduling policy (blocking or asynchronous), and we report it alongside reward and cost in Section 4.

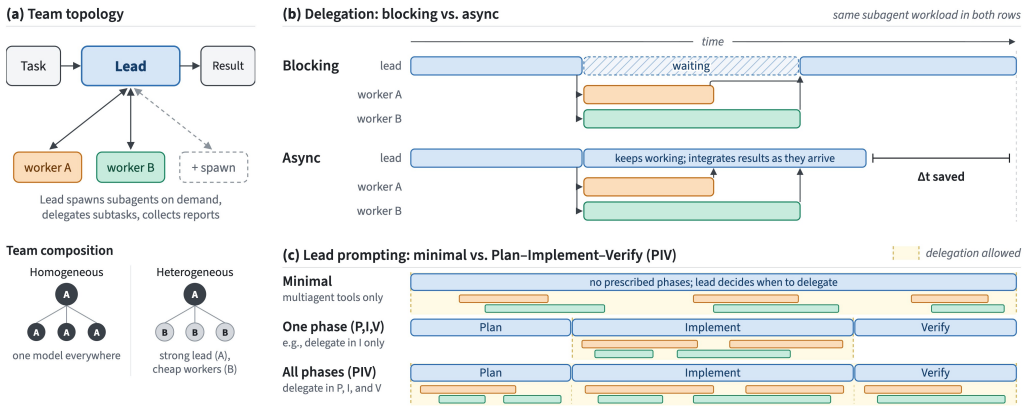

Figure 1: Overview of our multi-agent harness. A lead agent delegates subtasks to workers operating in a shared sandbox. We vary team composition (a), whether the lead blocks or keeps working while workers run (b), and which stages of the task the lead is prompted to delegate (c).

whose solution, traced across budgets, defines the Pareto frontier of performance and cost. The optimal configuration is task-dependent: extra workers pay off most at the stage that bottlenecks the task (Section 5, Appendix C.13).

Efficiency. Solving Eq. 1 exactly is infeasible, as each configuration requires a full benchmark evaluation, and the stage instruction s steers where the lead delegates rather than fixing how much it spends. We therefore do not match budgets across configurations. Instead, we ask a marginal question: *starting from a solo agent, where does the next dollar of inference buy the most reward?* We define the *efficiency* of a configuration c as the reward it gains per dollar of added inference cost relative to the solo agent:

$$\eta_c = \frac{R_c - R_{\text{solo}}}{C_c - C_{\text{solo}}}, \tag{2}$$

where R_c and C_c denote the mean reward and total inference cost of configuration c, and R_{solo} and C_{solo} those of the solo agent under the same guidance. We report η_c in reward points per \$100; a higher η_c means the configuration spends its added compute more productively, and a negative η_c means added compute lowers reward. A configuration *dominates* another if it achieves at least the same reward at no greater cost, with one of the two strictly better. We compute η_c only when $C_c > C_{\text{solo}}$; a configuration that costs no more than the solo agent while achieving higher reward dominates it, and we mark it as such.

Tracing the frontier. We approximate the frontier by varying one design choice at a time and fixing each subsequent study at the most efficient choice found so far. We vary, in turn, the workflow stage s at which the lead delegates to workers, whether planning, implementation, verification, or all three (Section 5); the lead and worker models ℓ and w (Section 6); and the number of workers n (Section 7). Since this coordinate-wise search does not solve Eq. 1, the resulting configurations are empirical operating points rather than guaranteed optima.

3 Experimental Setup

Harness Architecture To study multi-agent orchestration under controlled conditions, we design a configurable harness that isolates key orchestration primitives such as scheduling, message delivery, agent prompts, and model choice. Following the minimalist design of `mini-SWE-agent` (Yang et al., 2024), agents use a standard `bash` tool to interact with the environment and follow a ReAct-style loop (Yao et al., 2022). We adopt a top-down hierarchy where a lead agent receives the task instruction and coordinates workers through a `delegate_agent(task, agent_id=None)` tool. Omitting `agent_id` instantiates a fresh worker with its own context window and role prompt. Providing an existing ID resumes that worker, preserving its conversation history (e.g., to revise tests or continue debugging an authored module). In some configurations,

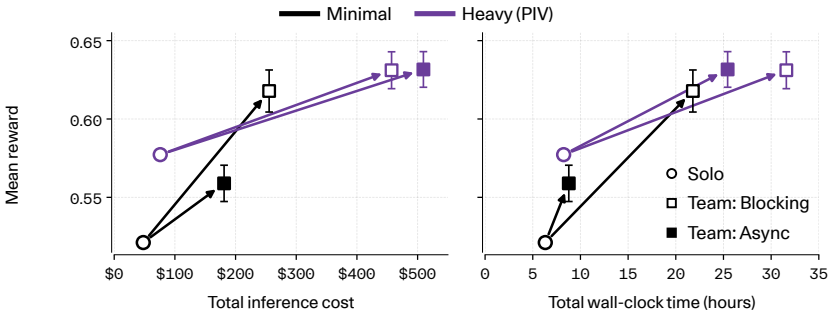

Figure 2: **Teams outperform solo agents, but incur higher inference and wall-clock costs.** Mean ProgramBench reward versus total inference cost (left) and total recorded wall-clock time (right) for Sol solo agents and homogeneous blocking and asynchronous teams under minimal and heavy (PIV) guidance. Points are means over three runs; error bars are 95% intervals over tasks of each team's paired difference from the solo agent with the same guidance.

`list_agents` and `wait_for_updates` tools are also provided to the lead, allowing it to check the status of all agents or block until the next worker update is surfaced.

Agent Team Configurations. Given a task, the lead dynamically determines the team size and delegates well-scoped subtasks to worker agents. Each worker observes only its assigned subtask: it does not receive the full task description or observe other workers' conversations. We evaluate two scheduling strategies. Under *blocking* scheduling, the lead waits for a worker to finish before continuing; under *asynchronous* scheduling, the lead continues working while workers execute concurrently and receives their outputs upon completion. Unless otherwise noted, Sections 4–7 use only communication at delegation boundaries: the lead provides a subtask when it spawns or resumes a worker, and the worker's terminal output is delivered to the lead upon completion. Agents cannot exchange direct messages while a worker is running, either between the lead and workers or among workers. Section 8 adds and evaluates this capability. We also compare two prompting regimes. Under *minimal* prompting, the lead is only told to assign workers non-conflicting tasks and how its delegation tools behave; when and how to delegate is left to the lead. For guided prompting, we adopt a three-stage abstraction inspired by waterfall-style software development(Royce, 1987): planning and exploration (**P**), implementation (**I**), and verification (**V**). Each stage-specific guidance instructs the lead to complete the full workflow while using workers only in the designated phase. This design isolates the value of delegation at each phase. We then compose all three interventions into **PIV**, our *heavy-guidance* variant, to evaluate their joint effect when workers are used throughout the workflow (full prompts in Appendix D).

Datasets. We primarily use ProgramBench (Yang et al., 2026a), a from-scratch repository reconstruction benchmark comprised of 200 unique tasks ranging from simpler command-line utilities like `CMatrix` and `Ripgrep` to complex programs like `FFmpeg` and `The PHP Interpreter`. In each task, an agent is provided access to a fresh workspace, a black-box reference executable, and relevant behavioral documentation from the original repository. Rather than repairing existing codebases or satisfying pre-written unit tests (Jimenez et al., 2024), the agent must actively probe the reference executable to discover its interface, edge cases, and runtime behavior, and then reconstruct a compatible implementation that is evaluated against held-out behavioral test suites. Additionally, we also explore Commit0-Lite (Zhao et al., 2024)[2] a benchmark comprised of 16 Python libraries that an agent is asked to reimplement given a repository skeleton and a set of unit tests.

Models. We conduct our main experiments with three tiers of the GPT-5.6 model family (OpenAI, 2026): Sol, Terra, and Luna, with all models set to use medium reasoning effort. Homogeneous teams use the same model for the lead and all workers. Heterogeneous teams pair a Sol or Terra lead with Luna workers. We also run additional experiments with Claude and Gemini model families (Appendix C).

[2]Commit0-Lite is only reported in the Appendix C.13: its 16 tasks limit statistical power, and it is near saturation, likely because frontier models are trained on its public libraries.

4 Does a team of agents outperform a solo agent?

Results. Teams outperform solo agents under both minimal and heavy PIV guidance, and heavy guidance pushes them further ahead, yielding the best reward (Figure 2). Teams spawn between 4.2 and 5 workers depending on whether minimal or heavy guidance is provided. Compared to a solo agent that issues between 19 and 22 tool calls, (Appendix C.6), teams issue between 44 and 140 tool calls per task based on the guidance respectively. Across configurations, gains are largest on easy tasks and are less pronounced on harder ones (Appendix C.7). A team is thus an effective way to scale inference-time compute, but it incurs substantially higher inference cost. On Sol, blocking teams take 3.5× longer under minimal guidance and 3.8× longer under heavy guidance. Whereas asynchronous teams take 1.4× and 3.1× longer respectively, a setting where the lead can also progress simultaneously with the workers as opposed to blocking where the lead idles while the workers run. This raises the central question of our study: *how should a team allocate its compute to reach the Pareto frontier of performance and cost?*

5 Where should workers be utilized?

We first vary the stage s to which workers are directed. We ask whether directing workers to specific stages of the software-engineering workflow (planning a design, implementing it, and verifying its behavior) changes this tradeoff. We consider 3 variants besides heavy guidance prompting (**PIV**) that differ in where the lead delegates work. Under *heavy guidance (PIV)*, the team follows a plan-implement-verify workflow, and the lead decides which stages to delegate to workers (Appendix D). To isolate the value of each stage, in the other three variants we restrict all delegation to a single stage and let the lead handle the rest: under *planning* (P), workers explore and plan while the lead implements and verifies; under *implementation* (I), workers write code while the lead plans, integrates, and verifies; and under *verification* (V), workers test the lead's work after it plans and implements. We compare each variant against PIV to identify which allocations lie on the Pareto frontier of performance and cost.[3]

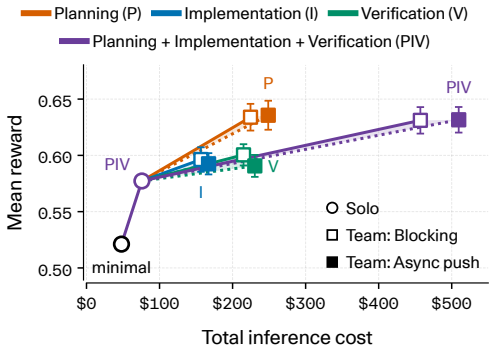

Figure 3: Planning provides the strongest observed reward–cost tradeoff under both schedules on ProgramBench (Sol teams; means over three runs). Solid lines lead to blocking teams and dotted lines to asynchronous teams. Error bars are paired 95% intervals relative to Solo PIV.

Results. Planning is by far the most efficient place to spend compute (Figure 3, Figure 4). With $\eta_{\mathrm{P}} = 3.8$ versus at most 2.4 for any other stage (Eq. 2), each dollar spent on planning buys roughly 1.6× more reward than the same dollar spent on implementation, verification, or the full workflow. Planning alone matches the reward of heavy PIV guidance at half the cost, and thus dominates it on the frontier. Because ProgramBench provides the target program only as a black-box executable, its behavior must be uncovered through probing, and planning workers parallelize this discovery across its interface, edge cases, and outputs. Additional compute therefore pays off most before any code is written, and delegating beyond planning adds cost without improving reward.

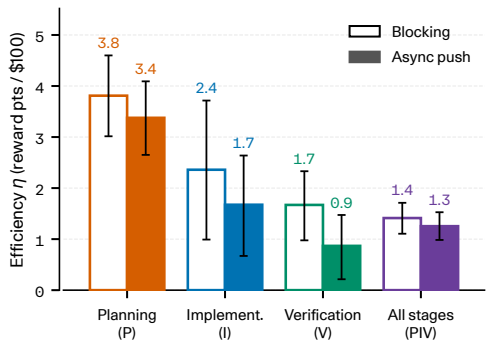

Figure 4: Efficiency η (Eq. 2) of Sol teams relative to the Sol Solo PIV agent. Error bars are 95% bootstrap intervals over tasks.

[3]We use Sol as the default model in subsequent analyses, since its higher capability better isolates how coordination choices shape team performance. Additional results for Terra and Luna are included in Appendix C

Why is planning efficient? To understand how guidance shapes the team's use of compute, we locate every lead delegation call relative to the team's first and last source edits, which divide each rollout into planning (pre-edit), implementation (edit), and verification (post-edit) phases.(Figure 5).

On average, Planning guidance issues all of its 674 calls in a single burst before any code is written. Heavy PIV guidance issues a comparable 856 calls before the first edit, then keeps delegating through implementation and verification, adding an additional 1,231 calls that yield no additional reward. With the same gain over the solo agent, planning therefore obtains $3\times$ more reward per call, mirroring its efficiency advantage ($\eta_{\mathrm{P}} = 3.8$ vs. $\eta_{\mathrm{PIV}} = 1.4$): one early burst of discovery rather than continuous delegation. This advantage is task-dependent. On Commit0-Lite, where the repository skeleton largely supplies the decomposition, implementation guidance instead becomes the most effective stage for delegation(Appendix C.13).

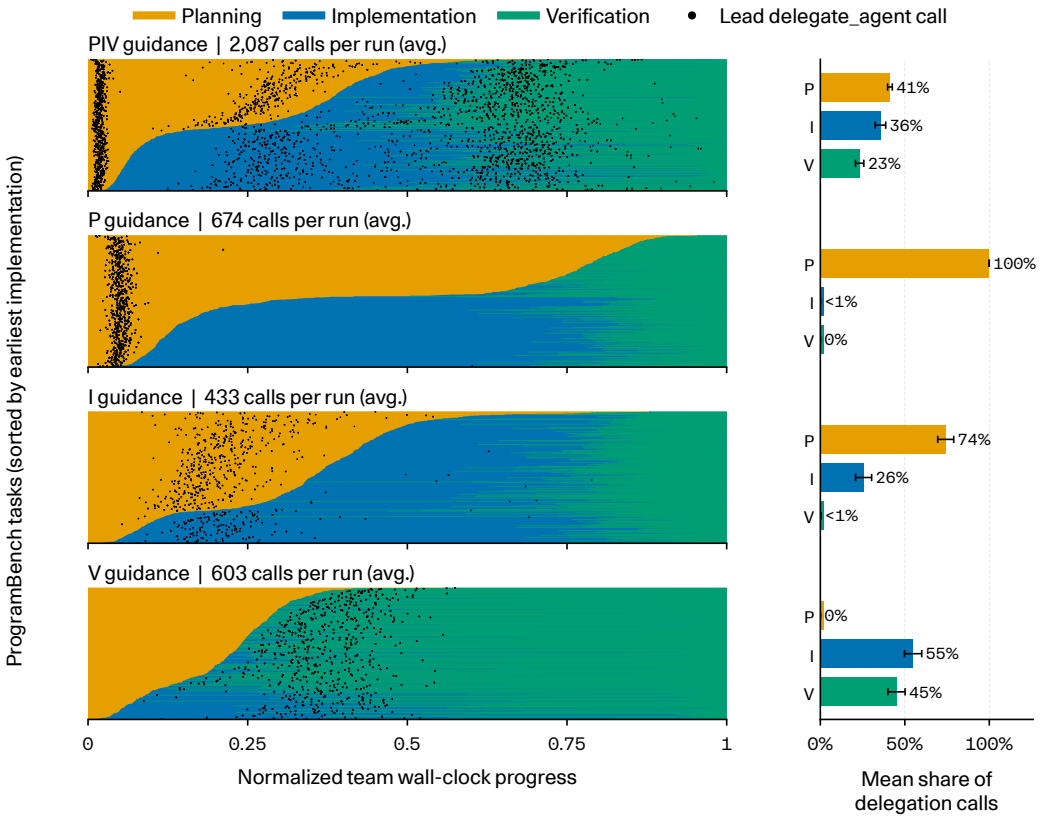

Figure 5: **Stage guidance shifts delegation timing, but nominal labels do not partition edit-defined phases.** The figure covers 600 ProgramBench rollouts per guidance setting (200 tasks $\times$ three runs) from homogeneous Sol blocking teams, with the lead's `delegate_agent` calls marked. The four left rasters show PIV, P, I, and V guidance from top to bottom. Each row represents one rollout. Within each panel, rows are ordered by implementation start (the first detected source write; defined in Appendix B), with the earliest at the bottom; rows are not aligned across panels. The bars on the right give the mean share of a rollout's delegation calls in the planning (P), implementation (I), and verification (V) phases (one P-guidance rollout without delegation calls is excluded); error bars are 95% bootstrap intervals from resampling tasks. Per-guidance call counts vary by at most 3% across runs.

6 Can Heterogeneous Teams Scale More Efficiently?

In all configurations considered thus far, workers share the model of their lead. Replicating an expensive model across every role, however, multiplies the cost of the team, and it remains unclear whether strong workers are necessary to realize the benefits of delegation. To investigate this question, we retain a Sol lead and replace all of its workers with the substantially cheaper Luna. Because weaker workers may alter which stages of the workflow benefit from delegation, we again direct them toward planning (P), implementation (I), verification (V), or all three stages (PIV).

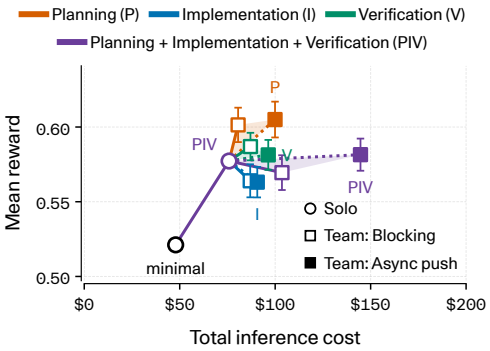

Figure 6: **Planning remains the strongest reward–cost tradeoff with cheaper workers.** Mean reward versus total inference cost for a Sol lead with Luna workers under P, I, V, or PIV guidance. Solid lines lead to blocking teams (open squares) and dotted lines to asynchronous teams (filled squares); error bars are paired 95% intervals relative to Solo PIV.

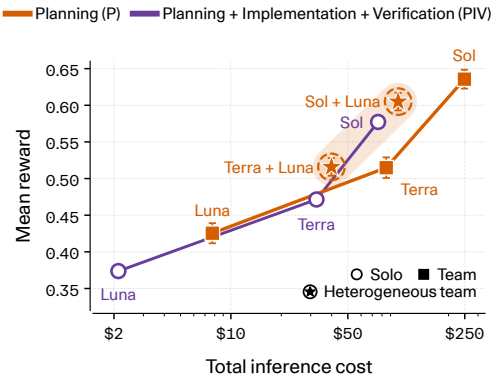

Figure 7: **Strong leads paired with cheaper planning workers open a lower-cost scaling axis.** Mean reward versus total inference cost (log scale) for Solo PIV agents and asynchronous planning teams led by Luna, Terra, or Sol. Stars mark heterogeneous teams with Luna workers: Sol+Luna recovers 47.5% of the all-Sol team's reward gain at 13.9% of its added cost.

Results. Cheap workers pay off only when directed toward planning (Figure 6). Relative to PIV, planning raises reward by 5.6% while lowering cost by 22.2% under blocking, and by 4.0% while lowering cost by 30.9% under asynchronous push. Directed toward implementation, by contrast, Luna workers reduce the team's reward below that of the Sol solo agent while increasing its cost, and directed toward verification, they leave reward essentially unchanged. The reward–cost trajectories in Figure 6 make this contrast explicit: planning climbs most sharply, whereas implementation falls below the solo level, where η is negative. Cheap workers are thus effective when they gather information for a strong lead to act upon, but not when they produce code that the lead must integrate. The payoff of this substitution is striking: every dollar spent on Luna planners buys roughly $13\times$ more reward than the same dollar spent on Sol planners ($\eta = 51.0$ versus 3.8 under blocking), whereas for implementation the same substitution is ruinous, with each additional dollar lowering reward ($\eta = -12.0$).

Can a stronger lead make better use of the same workers? To disentangle lead capability from worker capability, we fix all workers at Luna and vary the lead across Luna, Terra, and Sol (Figure 7). Team reward closely tracks the capability of the lead: with identical Luna workers, upgrading the lead from Luna to Terra raises reward by 20.9%, and from Terra to Sol by a further 15.4%, and at every tier the team outperforms the solo agent built on the same model. Cheap workers, however, cannot compensate for a weak lead, as a Terra lead with Luna workers still trails a Sol agent working alone by 10.3%. The lead therefore determines the ceiling that a team can reach, while the workers determine how cheaply it is approached, suggesting that model capability is best invested in the lead rather than replicated across the team. Across all lead tiers and schedules, planning remains the most efficient way to direct the workers, attaining the highest efficiency η and reducing inference cost by 22–46% relative to PIV (Appendix C.4).

7 How Many Workers Should a Team Have?

Scaling the number of workers. In the preceding sections, the lead determined the team size itself, spawning 4.2–5 workers on average. We now control the team size n directly, pairing a Sol lead with 2, 4, 8, or 16 Luna workers directed toward planning, implementation, or verification, which lets us ask whether additional workers continue to pay off and at which stage of the workflow. **Directed toward planning, sixteen Luna workers come within 1.4 points of an all-Sol team at 70% lower cost** (Figure 8), making them roughly $4\times$ **more efficient**. This gain, however, depends on where the workers are directed, as Figure 8 reveals a pronounced asymmetry across stages. Planning workers improve reward at every team size: two workers already improve on the solo agent by 3.2%, and sixteen raise the gain to 7.1%. Implementation and verification workers, in contrast, im-

prove from two to four workers but then plateau, with their reward changing by -0.015 and -0.011, respectively, when scaling to 16 workers. The plateau is not explained by explicit coordination failures. We find that implementers rarely write to the same file (PIV guidance instructs the lead to delegate independent tasks; Appendix D) and verifier reports become less useful as the team grows (Appendix C.8). Instead, planners return progressively more information about the black-box target, whereas additional implementers and verifiers subdivide work the lead can already coordinate without adding new information about the program.

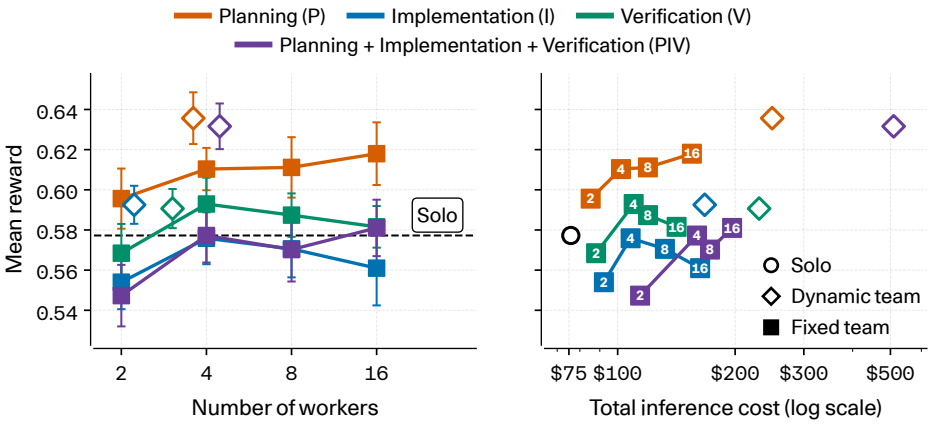

Figure 8: **Where additional agents are deployed determines the reward–cost scaling curve.** Fixed teams pair a GPT-5.6 Sol lead with 2–16 Luna workers (numbers give team size); dynamic teams are all-Sol teams sized by the lead, shown at their mean number of workers. Scaling the heterogeneous team to 16 workers with planning guidance comes within 1.4 points of the homogeneous team at 70% lower cost. All teams use asynchronous push. Fixed teams are a single run; error bars (left) are paired 95% intervals relative to Solo PIV.

8 Does inter-agent communication help?

To study communication, we give every agent a `send_message` tool that can reach any other running agent.[4] Messages flow largely from workers to the lead (84% for Sol teams under PIV guidance), carrying interim findings, proposed interfaces, verification failures, and questions. Workers rarely message each other (less than one message per task), even when told to collaborate (Appendix C.10).

Communication grows with model capability (Figure 9). Under minimal guidance, all three tiers send 7–9 messages per task, but Sol writes $3.4\times$ as much text as Luna (13.7k versus 4.0k characters). Under PIV guidance, stronger teams also send more messages (10.9 to 16.9 per task), and their leads steer workers more (0.15 to 2.0 lead-to-worker messages per task). Greater communication, however, does not yield higher reward. Sol and Terra teams are unchanged within noise, and Luna teams lose 4.4–7.1 points (Figure 9). The channel mostly carries status reports to a lead that already receives each worker's outcome.

9 Related Work

Multi-agent Coding Systems. There is a growing body of work on teams of coding agents. Early systems encode a fixed team workflow into their orchestration, assigning specialized roles (Qian et al., 2023; Hong et al., 2023; Tao et al., 2024; Liu et al., 2025). More recent work asks how teams behave as they grow and run concurrently. CAID (Geng & Neubig, 2026) combines dependency-aware delegation with isolated workspaces and asynchronous execution, and Yang et al. (2026b) study when partitioning coding tasks across parallel agents pays off. Agensh (Zhan et al., 2026) lets teams self-organize from 1 to 128 agents on the five hardest ProgramBench tasks and scales one task to 1,024 agents. We instead hold one lead–worker architecture fixed and run a controlled study on all 200 ProgramBench tasks, measuring how each design choice moves the team along the reward-cost frontier.

[4] Agents were only able to send messages to other agents that were actively working, so we also provided all agents with the `list_agents` tool.

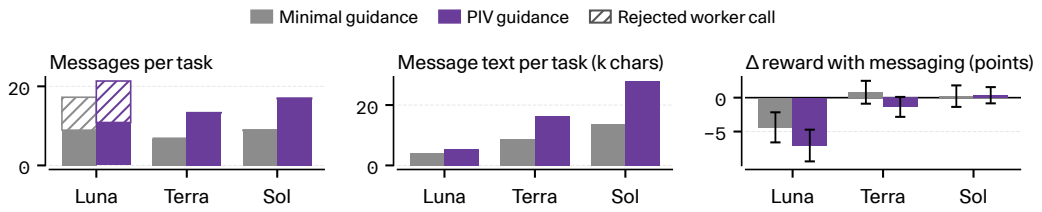

Figure 9: **Stronger models communicate more, but messaging does not raise reward.** Homogeneous asynchronous teams under minimal and PIV guidance. Hatched segments are rejected worker calls. Right: reward change relative to the same team without the tool, with paired 95% intervals.

Team configuration under a budget. Closest to our work are studies of how to configure a team given limited compute. ZEBRA (Hamri & Talgam-Cohen, 2026) allocates a fixed monetary budget across coding phases, ManagerWorker (Liu, 2026) pairs strong managers with weaker workers, and AgentCARD (Jiang et al., 2026) shows that the value of role and model placement depends on the task domain. LoopArena (Wang et al., 2026) isolates controller capability by holding a coding worker fixed, and Pan et al. (2026) and Fan et al. (2026) show that harness components such as planning, action interfaces, and context management change task outcomes and costs. We bring these questions together in a single white-box harness, comparing stage-directed delegation, scheduling, lead and worker capability, and team size on a common benchmark.

10 Discussion and Conclusion

From Solo Agents to Agent Societies. Inference-time scaling is moving from single agents toward teams of agents, and we expect this shift to accelerate as tasks grow from fixing a bug or implementing a feature to scientific discovery and open-ended research. Our results show that teams deliver gains over solo agents, but at a substantial increase in inference cost. We show, however, that with the right allocation of compute, a team can nearly match the reward of an all-strong team at 70% lower cost. The next question is how to manage teams of hundreds, thousands, or even millions of agents. Our experiments were limited to sixteen workers[5], but we expect team sizes to grow dramatically. At that scale, how such teams should be organized? How should their agents communicate, and where is their compute best spent? And can agents optimize the network themselves, so that depending on the task, the budget, and the deadline, they make these choices naturally?

Imposed Hierarchy versus Self-Organization. Our harness imposes a lead–worker hierarchy, but a society of thousands of agents working over weeks must decide who assigns roles, how information flows, when to reorganize, and how agents communicate. In our runs, workers rarely talk to each other: 84% of messages go to the lead and only 4% between workers (Section 8). This may partly reflect our harness, which exposes a messaging tool but does not incentivize its use, and partly the task, since on ProgramBench the lead can still hold the full picture. As projects outgrow what any single agent can track, agents will need to share what they learn directly, and may even invent their own protocols to do so. Pretraining on human text already gives agents primitives such as reporting, asking, and delegating. Whether coordination emerges from these with model parameters scale or must be trained remains open, and we see it as central to scaling inference.

Call for Multi-Agent Safety Our work studies how teams of coding agents scale and how to approach the reward-cost frontier, but does not evaluate their safety. The mechanisms that improve team performance may also introduce new risks: aligning individual agents does not guarantee that the team behaves as intended, errors can propagate through delegation, and agents may develop communication protocols that are difficult for humans to audit. We invite the broader research community, particularly those advancing multi-agent capabilities, to develop methods for monitoring, auditing, and aligning agent societies alongside methods for scaling them.

[5] due to budget constraints

Reproducibility Statement

We will provide the prompts, harness configurations, evaluation procedures, and task-level results required to reproduce our experiments. The appendix will document model settings, communication policies, token budgets, and implementation details.

AI Use Statement

In this work, we have used generative AI tools to aid in writing and polishing sections for better readability. We have also used AI to assist in discovery of related work to ensure coverage. We have not used generative AI tools for research ideation/execution, generating datasets or mathematical claims. We reviewed all AI-assisted work in the following way:

AI as an aid for writing/polishing We confirmed that no new information is being added and only light edits to improve grammar and readability were applied followed by final human review and edits.

For related work discovery We confirmed all works surfaced during discovery and only added them into the transcript after an author deemed this to be relevant to our work. The final incorporation of the discovery data was done by a human.

References

Anthropic. Claude fable 5 and claude mythos 5. Technical report and model release, June 2026. URL https://www.anthropic.com/news/claude-fable-5-mythos-5.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

DeepSeek-AI, :, Anyi Xu, B. Li, Bangcai Lin, Bing Xue, BingCheng Xian, Bingzheng Xu, Bochao Wu, Bowei Zhang, Boyi Deng, C. C. Yu, Chao Jin, Chaofan Lin, Chen Dong, Chenbing Wang, Chenfan Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyuan Zhang, Chenhao Xu, Chenqi Zhao, Chenze Shao, Chuhao Wang, Chuqi Zhang, Damai Dai, Dejian Yang, Deli Chen, Di Huang, Di Wu, Donghao Li, Erhang Li, Eric Fu, F. Zhou, Fangwei Zhou, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanglin Li, Guanting Chen, Guoai Cao, Guofan Fan, Guolai Meng, Guowei Li, Haichuan Zhang, Haiyang Ma, Haiyang Shen, Han Li, Han Yu, Han Zhang, Hangyuan Deng, Hanwei Xu, Hanxiang Xu, Hanxun Zhong, Hao Guo, Hao Jiang, Hao Li, Hao Qin, Haodong Wen, Haofen Liang, Haofeng Huang, Haohua Liu, Haoling Zhang, Haoming Luo, Haoran Yang, Haotian Xu, Haotian Yuan, Haoting Huang, Haowen Luo, Haoyang Cai, Haoyu Chen, Haozhe Ji, Hengran Zhang, Hengrui Wang, Hengxu Wu, Honghui Ding, Hongxuan Tang, Huadong Wang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J. Yang, J. H. Jin, J. H. Zhang, J. X. Zou, Jia Yu, Jiahui Zhou, Jiajun Chen, Jialiang Huang, Jialin Zhao, Jiamin Tang, Jian Zhou, Jianan Tong, Jianwen Li, Jiaqi Zhu, Jiarui Wang, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiaying Ding, Jibai Lu, Jiewen Hu, Jin Yan, Jincheng Zhai, Jingchang Chen, Jingcheng Hu, Jingli Zhou, Jingsheng Xu, Jingting Xiang, Jingyan Yun, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jinpeng Wang, Jinyi Chen, Jinyi Hu, Jiping Yu, Jueliang Guo, Junbo Pei, Junbo Sun, Junguang Jiang, Junjie Qiu, Junkang Zhou, Junqi Liu, Junren Li, Junxian Li, Junxiao Song, Junyi Guo, Kai Dong, Kaifeng Chen, Kaige Gao, Kang Guan, Kangdong Yuan, Ke Hong, Ke Xu, Kefan Zhao, Kexin Ji, Kexin Zhang, Kexing Zhou, Kuai Yu, Lan Zhang, Lean Wang, Lecong Zhang, Lei Wang, Letian Gao, Liang Zhao, Liansheng Xu, Lihua Guo, Lingxiao Luo, Lingyue Fu, Litao Deng, Litong Wang, Liyue Zhang, Longhao Chen, Lu Chen, Luotian Huang, Luyao Ma, Luyao Wang, M. S. Di, Max Mei, Menghao Ye, Miao Cui, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingjing Zhang, Mingqi Wei, Mingshu Chen, Mingxing Liu, Mingxu Zhou, Mingyu Xu, Mingyu Yang, Mingze Wang, Muyang Chen, Ni Shentu, Ning Wang, Niufang Ning, Panpan Huang, Peixin Cong, Peiyi Wang, Peiyuan Xin, Pengfei Ren, Pengfei Yan, Pengle Zhang, Qi Kang, Qi Tang, Qiancheng Wang, Qiang Li, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qizhou Guo, Rongxian Xu, Rui Ding, Rui Hu, Rui Tian, Rui Yu, Ruidong Zhu, Ruifan Xu, Ruihan Yang, Ruihang Xia, Ruijie Lu, Ruilin Geng, Ruipeng Hong, Ruiqi Ge, Ruisong Zhang, Ruize Sun, Ruizhe Pan, Runji Wang, Runqian Chen, Runxin Xu, Ruohong Tian,

Ruomeng Shen, Ruoyu Zhang, Ryan X., S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoyuan Chen, Shengding Hu, Shengkai Lin, Shengwen Ran, Shengyu Liu, Shengyuan Jia, Shi Bai, Shi Feng, Shicheng Xu, Shichun Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shiyuan Feng, Shufan Gong, Shuhan Lin, Shuiping Yu, Shunfeng Zhou, Shuo Yang, Shuomeng Wang, Shuting Guo, Shuting Pan, Shuying Yu, Sinuo Cao, Siyi Lin, Sizhe Chen, Songyang Chen, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tongrui Xiong, Wangding Zeng, Wei Liu, Wei Zhang, Weibin Xu, Weihao Zeng, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Shao, Wenkai Yang, Wenli Zhang, Wenlu Wang, Wenlve Huang, Wenqian Yan, Wentao Zhang, Xi Gao, Xiang He, Xiang Li, Xiangli Li, Xiangwen Wang, Xiangying Zhang, Xiankui Wei, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojian Qu, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaoyao Zou, Xiaoyuan Li, Xicheng Guo, Xieting Chu, Xin Cheng, Xin Liu, Xin Xie, Xinbo Xu, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xintong Yao, Xinyang Chen, Xinyong Jiang, Xinyu Yang, Xinyu Yang, Xu Chen, Xuanyu Wang, Xubei Zhong, Xuecheng Su, Xuejie Liu, Xuheng Lin, Xujie Fan, Xuncheng Zhao, Xuwei Fu, Y. C. Yan, Y. H. Jiang, Y. T. Wu, Y. W. M., Y. Z. Wang, Yafei Gao, Yang Yang, Yang Zhang, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Meng, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yaoyang Ye, Yehang Yin, Yexinrui Wu, Yi Qian, Yi Tao, Yi Yu, Yichao Zhang, Yichen Jiang, Yicheng Wang, Yifan Ding, Yifan Shi, Yifeng Peng, Yifeng Zhai, Yijia Wu, Yiliang Xiong, Yilun Wang, Ying He, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yiping Wang, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyao Yang, Yiyuan Liu, Yizai Cai, Yizhen Wei, Yizhi Wang, Yonglun Yang, Yongqi Zhuo, Yongqiang Guo, Yongtong Wu, Yu Wu, Yu Zhang, Yuan Bian, Yuan Cheng, Yuan Ou, Yuan Sun, Yuanfan Xu, Yuanhang Sun, Yuanhao Li, Yuchen Liu, Yuchen Yao, Yudong Han, Yuduan Wang, Yuhan Wu, Yuhao Meng, Yuheng Zou, YuKun Li, Yunchuan Wang, Yunfan Xiao, Yunfan Xiong, Yupeng Chen, Yuqian Cao, Yuqian Wang, Yuqing Chen, Yushun Zhang, Yutong Lin, Yuwei Xiao, Yuxian Gu, Yuxiang Chen, Yuxiang Huang, Yuxiang Luo, Yuxiang You, Yuxin Chen, Yuxin Xiang, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhe Guo, Yuzhen Huang, Yuzhuo Bai, Z. Y. Z., Zanlin Ni, Zehao Wang, Zehua Zhao, Zehui Ren, Zejun Zhao, Zhangli Sha, Zhanying Wang, Zhaochen Zhang, Zhaoshuai Du, Zhe Fu, Zhean Xu, Zhenda Xie, Zheng Liu, Zhengyan Zhang, Zhenhua Dong, Zhewen Hao, Zhibang Wang, Zhibin Gou, Zhicheng Ma, Zhihao Li, Zhihong Shao, Zhihuan Huang, Zhijie Li, Zhirui Lu, Zhixian Huang, Zhixuan Chen, Zhixuan Chen, Zhixuan Pan, Zhiyu Wu, Zhizhou Ren, Zhu He, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zili Zhang, Zilin Li, Zilong Hou, Zilong Lyu, Ziqiao Wang, Ziwei Xie, Ziya Zhang, Ziyi Gao, Zizheng Pan, Zonglin Li, Zongqing Yao, Zui Chen, Zuofan Wu, Chenchen Ling, Chengyu Hou, Chong Chen, D. Li, Di Qi, Dongjie Ji, Fang Wei, Fanyi Xia, Fei Xie, Feiyi Tan, Hailong Guo, Haiyan Zhai, Hui Zhou, Huihui Tan, Huijie Li, Jia Luo, Jia Song, Jialu Cai, Jian Liang, Jiangting Zhou, Jiaqi Gao, Jiayi Shao, Jie Chen, Jieyu Yang, Jin Chen, Jingde Zhang, Jingzi Zhou, Jinqian Wang, Jinyang Liu, JinZhao Sun, Junhua Ling, Junmin Zheng, Kaicheng Yang, Ke Xu, Le Su, Leyi Xia, Liangfeng Ding, Lin Zhuo, Linwang Ma, Linyan Zhu, Liyu Cai, Luqi Yao, M. K. Zhang, Meng Li, Miao Lin, Miaojun Wang, Min Zhang, Mingming Li, Mingming Wang, Mingze Yin, Minmin Han, Nan Cao, Ning Wang, Ningxin Ma, Panpan Wang, Peihan Lin, Peng Sun, Peng Zhang, Qian Ying, Qiang Xiang, Qiao Wang, Qingmiao Mao, Qiwei Jiang, Rongli Jin, Ruyi Chen, Sha Tao, Shangmian Sun, Shaoqing Wu, Shichao Zou, Si Lei, Tianyang Zhang, Tianyu Sun, Tingting Yin, W. L. Xiao, Wei An, Wei Li, Wei Wang, Weiwei Lin, Wenqing Hou, X. Lin, Xiangfei Meng, Xianzhu Huang, Xiao Peng, Xiaoqian Li, Xiaoting Zhang, Xiaowen Sun, Xiaoxiang Wang, Xiaoyu Ye, Xinrou Zhang, Xinyu Zhang, Xue Cao, Xueyin Chen, Yanan Zhou, Yanhong Xu, Yao Xia, Yao Xu, Yi Shao, Yihong Zhang, Yiling Ma, Ying Tang, Yining Lou, Yiru Chen, Yishi Piao, Yixuan Chen, Yong Xiong, Yuchen Xuan, Yuehan Yang, Yuer Xu, Yukun Zha, Yunxian Ma, Yuping Lin, Yuting Yan, Yutong Xie, Yuwen Sheng, Yuxuan Zhu, Zekai Zhang, Zhe Ju, Zhenzhen Lin, Zheren Gao, Zheyang Sun, Zhigang Yan, Zhongyu Wu, Zi Wang, Zihua Qu, Ziling Yan, and Ziyi Wan. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, 2026. URL https://arxiv.org/abs/2609.19969.

Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, and Xiaoyang Wang. An empirical study of harness design for coding agents, 2026. URL https://arxiv.org/abs/2609.20804.

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

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

May Hamri and Inbal Talgam-Cohen. Zebra: Zero-shot budgeted resource allocation for llm orchestration, 2026. URL https://arxiv.org/abs/2605.20485.

Sirui Hong, Xiawu Zheng, Jiaqi Chen, Y Cheng, J Wang, Ceyao Zhang, Z Wang, SKS Yau, Z Lin, L Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. arxiv preprint 2023. *arXiv preprint arXiv:2308.00352*, 2023.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Yinsicheng Jiang, Liang Cheng, Yeqi Huang, Yufan Zhao, Zhan Lu, Li Dong, Wenda Li, Edoardo Ponti, and Luo Mai. Specialize roles, mix deployments: Pushing the cost-accuracy frontier of llm agent teams, 2026. URL https://arxiv.org/abs/2606.20629.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

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/.

Rui Liu. Can ai models direct each other? organizational structure as a probe into training limitations, 2026. URL https://arxiv.org/abs/2603.26458.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

OpenAI. GPT-5.6 Sol. OpenAI API documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses, 2026. URL https://arxiv.org/abs/2603.25723.

C Qian, W Liu, H Liu, et al. Chatdev: Communicative agents for software development [j/ol]. *arXiv preprint arXiv:2307.07924*, 2023.

Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pp. 328–338, 1987.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution, 2024. URL https://arxiv.org/abs/2403.17927.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Guanduo Chen, Guangyu Chen, Guanzheng Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kexin Chen, Peng Chen, Ruijue Chen, Wentao Chen, Xin Chen, Yang Chen, Yanru Chen, Yifei Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Dazhi Cheng, Yean Cheng, Jialei Cui, Jingbing Cui, Anqi Dai, Jiaqi Deng, Hao Ding, Rui Ding, Shaofeng Ding, Mengfan Dong, Mengnan Dong,

Yuhao Dong, Yuxin Dong, Angang Du, Chenzhuang Du, Dikang Du, Jusen Du, Yulun Du, Yu Fan, Jing Feng, Qiulin Feng, Yichen Feng, Kelin Fu, Qiang Fu, Fuxuan Gao, Hongcheng Gao, Jingyue Gao, Tong Gao, Weijia Gao, Shangyi Geng, Jie Gong, Linhu Gong, Shengao Gong, Xiaochen Gong, Qizheng Gu, Yicheng Gu, Shuhao Guan, Haiqing Guo, Shiqi Guo, Xiang Guo, Zhengyan Guo, Beixi Hao, Wenxin Hao, Xiaoru Hao, Dailan He, Haotian He, Lehan He, Qi He, Weiran He, Xinran He, Xinyi He, Yibo He, Yunjia He, Chao Hong, Tiange Hong, Hao Hu, Jiaxi Hu, Ruikun Hu, Weiming Hu, Yangyang Hu, Zhenxing Hu, Liang Hua, Jinbin Huang, Ke Huang, Ruiyuan Huang, Siying Huang, Weixiao Huang, Yan Huang, Zhengjie Huang, Zhiqi Huang, Yulong Hui, Chaobo Jia, Yutong Jiang, Zhejun Jiang, Zuoyou Jiang, Wenyi Jin, Xinyi Jin, Yu Jing, Huanjun Kong, Guokun Lai, Aidi Li, Cheng Li, Chengyuan Li, Cong Li, Fang Li, Guanyu Li, Haoyang Li, Jia Li, Junxiong Li, Lei Li, Letian Li, Lincan Li, Weihong Li, Wentao Li, Xintong Li, Yang Li, Yishen Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Zhengxiao Li, Zhiyuan Li, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zichao Lin, Ziyan Lin, Bill Liu, Boxiao Liu, Chuan Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yipeng Liu, Zhengying Liu, Zhiheng Liu, Enzhe Lu, Haoyu Lu, Linqiang Lu, Tingzhan Lu, Zhiyuan Lu, Aotian Luo, G. Luo, Junyu Luo, Yifan Luo, B. Lyu, Wenzhou Lyu, Shaoguang Mao, Yuan Mei, Xin Men, Minqing Ni, Yixuan Niu, Siyuan Pan, Shujun Peng, Zhangyang Qi, Ruoyu Qin, ZeChao Qin, Zeyu Qin, Haiquan Qiu, Jianxin Qiu, Jiezhong Qiu, Bowen Qu, Yuhao Qu, Zeyu Shang, Youbo Shao, Han Shen, Jincheng Shi, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Wingchun Siu, Pengwei Song, Xiaoxi Song, Jianlin Su, Yunfeng Su, Zhaochen Su, Lin Sui, Jingsong Sun, Junyao Sun, Shaoning Sun, Shuzhe Sun, Tongyu Sun, Yujun Sun, Yunpeng Tai, Chuning Tang, Heyi Tang, Sirui Tang, Zecheng Tang, Chaoran Tian, Rongpeng Tian, Yu Tian, Wei Tu, Chensi Wang, Chuang Wang, Chunjie Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hao Wang, Huaqing Wang, Hui Wang, Jiayi Wang, Jinglong Wang, Jinhong Wang, Jiuzheng Wang, Linian Wang, Shaobo Wang, Shenzhi Wang, Shuyi Wang, Si Wang, Siyuan Wang, Tianfu Wang, Wenjue Wang, Xingran Wang, Xinmei Wang, Xinyuan Wang, Xusheng Wang, Yalin Wang, Yangkun Wang, Yao Wang, Yaoyu Wang, Yejie Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhenhao Wang, Zhongsheng Wang, Zifan Wang, Chu Wei, Ming Wei, Shouxin Wei, Zichen Wen, Fan Wu, Haoning Wu, Rucong Wu, Wenhao Wu, Xiaoxue Wu, Yingcong Wu, Yongqi Wu, Yuxin Wu, Zijian Wu, Xinglang Xian, Chenxuan Xiang, Yuye Xiang, Bocheng Xiao, Chenjun Xiao, Xin Xiao, Jin Xie, Xiaotong Xie, Yifeng Xie, Zhe Xie, Bowei Xing, Yiming Xiong, Baosheng Xu, Boyu Xu, Jiale Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Qingtao Xu, Shuyao Xu, Suting Xu, Tiantian Xu, Tianxiang Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ye Xu, Yueni Xu, Ziyao Xu, Haonan Xue, Junjie Yan, Yaoyao Yan, Fan Yang, Guangyao Yang, Hao Yang, Junwei Yang, Ruoyu Yang, Wenjie Yang, Xiaofei Yang, Xinyu Yang, Yi Yang, Yiling Yang, Ying Yang, Yuchen Yang, Zhen Yang, Zhilin Yang, Zian Yang, Zuhao Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhanbo Ye, Bohong Yin, Haoxiang Yin, Xietong Yin, Chengzhen Yu, Haozhen Yu, Longhui Yu, Shengnan Yu, Shuying Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Tongtian Yue, Wei Yue, Yang Yue, Dunyuan Zha, Haobing Zhan, B. H. Zhang, Dehao Zhang, Fei Zhang, Hao Zhang, Haoyuan Zhang, Huanyu Zhang, Jiapei Zhang, Jiaxuan Zhang, Jin Zhang, Kaiyi Zhang, Miaozhen Zhang, Puqi Zhang, Qinglei Zhang, Rong Zhang, Rui Zhang, Shaoshuai Zhang, Shiyi Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yangkun Zhang, Ye Zhang, Yichi Zhang, Yikun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Zijing Zhang, Bin Zhao, Chenguang Zhao, Feifan Zhao, Jinglun Zhao, Jinxiang Zhao, Shuai Zhao, Wenshuo Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Haozhi Zheng, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Haofeng Zhong, Lei Zhong, Longguang Zhong, M. Zhou, Qiankang Zhou, Runjie Zhou, Ruozhang Zhou, Xinyu Zhou, Yiqiao Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yangjunfeng Zhu, Yuxuan Zhu, Zhen Zhu, Chen Zhuang, Weiyu Zhuang, and Xinxing Zu. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Yi Wang, Haopeng Zhang, Chengxiang Huang, Rui Dai, Kaikui Liu, Piotr Koniusz, and Xiangxiang Chu. Looparena: Benchmarking models as runtime controllers for loop engineering, 2026. URL https://arxiv.org/abs/2608.28281.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-bench: Can language models rebuild programs from scratch?, 2026a. URL https://arxiv.org/abs/2605.03546.

Xu Yang, Lunyiu Nie, Ethan Chandra, Stanislav Gannutin, Fangru Lin, and Swarat Chaudhuri. When parallelism pays off: Cohesion-aware task partitioning for multi-agent coding, 2026b. URL https://arxiv.org/abs/2606.00953.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Z.ai. GLM-5.3: Frontier coding with emergent cyber capabilities. Technical report, August 2026. URL https://z.ai/blog/glm-5.3.

Zhihao Zhan, Ting Song, Li Dong, Shaohan Huang, Jianxun Lian, Yan Xia, and Furu Wei. Agensh: Scaling organizational intelligence to 1,024 agents, 2026. URL https://arxiv.org/abs/2609.26781.

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 Harness Architecture and Implementation Details

A.1 Agent Turn Lifecycle

A rollout begins by providing the benchmark task instruction as the initial user message to a lead agent. The lead may invoke a `delegate_agent` tool with a natural-language assignment to spawn and delegate a task to a subagent worker. The new subagent initially begins with a fresh conversation state where the initial user message contains only this assignment from the lead.

When a subagent terminates, its final assistant message becomes the result returned to the lead. Each worker is assigned a unique agent ID (e.g. `agent_1`), and the lead can later delegate another task to a previously spawned agent, resuming the worker with its previous context and the new assignment as an appended user turn. This persistence allows the lead to return to a worker that has already developed task-specific context, rather than starting a new trajectory for every delegation.

A.2 Scheduling and Delivery Policy Comparison

We study two scheduling policies. Under *blocking delegation*, a call to `delegate_agent` returns only when that worker has finished. The lead therefore cannot begin its next step until the delegated work is completed. However, multiple delegations issued in the same turn may execute in parallel, so blocking serializes the lead with respect to its workers, rather than serializing all activity.

Under *asynchronous push*, a call to `delegate_agent` immediately returns with the agent ID and the subagent continues to work in the background. The lead can continue reasoning, using tools, or spawning additional workers while any subagents execute concurrently. When a worker finishes, its outcome is appended to the subsequent tool result observed by the lead. Optionally, the lead may call `wait_for_updates`, which blocks until a worker update becomes available.

B Guidance delegation

We anchor each delegation call to the team's source edits. The first and last detected source writes divide a rollout into three observable intervals: time before editing, active editing, and time after

Table 1: Comparison of scheduling and message-delivery policies.

Policy	How it works	Pros	Cons
Blocking delegation	The lead delegates to a worker and remains blocked until the worker returns its result.	Simple coordination; provides a natural synchronization point; allows the lead to reason from a completed result; lowers the risk of acting on incomplete information.	The lead is idle while the worker runs; prevents lead–worker parallelism; can substantially increase wall-clock time.
Async push: full message	The worker runs concurrently. Once finished, its full result is appended to the lead's next tool result.	Requires no explicit retrieval call; makes completed work difficult to overlook; preserves concurrency; allows results to be incorporated quickly.	May interrupt the lead at an awkward point; unexpectedly consumes context; multiple messages may overload the next tool output; may destabilize the lead's current plan.

editing. Write boundaries use the completion time of any lead or worker `bash` call matched by a limited regex for patch commands, selected redirected shell writes, terminal editors, or direct Python file writes. Writes under `/tmp`, `/var/tmp`, `/dev`, and `/opt`, as well as writes to the generated `executable`, are excluded. We refer to these edit-defined intervals as planning, implementation, and verification. They provide a common clock for comparing trajectories across guidance ablations and studying delegation patterns.

Figure 5 applies this clock to every lead delegation call from homogeneous Sol blocking teams on ProgramBench, for PIV/P/I/V guidance setups. We assign each call to an interval from its completion time relative to the edit boundaries, then normalize that time by the rollout's team-wide wall-clock duration.

Planning guidance aligns most tightly with its nominal stage: all 674 P calls occur before the first detected edit. V shifts calls later, although a slight majority of its 603 calls remains within the active-edit window (54.9%), with 45.1% after the final edit. With I, 74.5% of its 433 calls precede the first edit, 35.3% occur during active editing, and 0.2% follow the last edit. PIV instead spans all three phases, distributing its 2,087 calls across planning (40.7%), implementation (35.7%), and verification (23.6%).

C Additional Experimental Details

All figures in this section use the full 200-task ProgramBench benchmark and the same conventions as the corresponding main-text figures. GPT-5.6 results are means over three runs; Claude results are single runs.

C.1 Model configurations

Table 2 lists every model we evaluate, the provider we accessed it through, its reasoning setting, and the per-token prices from which all reported inference costs are computed. The same configuration is used whenever a model acts as a solo agent, a lead, or a worker.

C.2 Team versus solo across models

Here we repeat the solo-versus-team comparison for homogeneous GPT-5.6 Terra and Luna teams (Figure 10) and for Claude Opus 4.7 and Haiku 4.5 (Figure 11). The Claude models illustrate two distinct ways in which added capacity fails to help. Opus 4.7 teams under minimal guidance are indistinguishable from the solo agent ($+0.001 \pm 0.022$), because the lead almost never delegates (Table 5). Under PIV guidance they gain 0.043–0.063 at 1.7–2.0$\times$ the solo cost. Haiku 4.5 delegates readily, but every team configuration scores at or below the solo agent (-0.041 to -0.004) at roughly twice the cost.

Table 2: Model configurations and prices. Reasoning settings are OpenAI's reasoning effort, Anthropic's extended thinking (adaptive with an effort level, or a fixed token budget), and Gemini's thinking level. Prompt caching is enabled explicitly for Claude on Bedrock, which charges for cache writes; OpenAI and Gemini cache prompts automatically and charge only the discounted cache-read price. Evaluation gives the ProgramBench runs in which each model appears.

			Price ($ per million tokens)				
Model	Provider	Reasoning setting	Input	Output	Cache read	Cache write	Evaluation
GPT-5.6 Sol	OpenAI	effort: medium	4.00	20.00	0.40	–	Full 200, 3 runs
GPT-5.6 Terra	OpenAI	effort: medium	2.00	12.00	0.20	–	Full 200, 3 runs
GPT-5.6 Luna	OpenAI	effort: medium	0.20	1.20	0.02	–	Full 200, 3 runs
Claude Opus 4.8	Bedrock	adaptive, effort: medium	5.00	25.00	0.50	6.25	Balanced-30
Claude Opus 4.7	Bedrock	adaptive, effort: medium	5.00	25.00	0.50	6.25	Full 200, 1 run
Claude Haiku 4.5	Bedrock	budget: 4,000 tokens	1.00	5.00	0.10	1.25	Full 200, 1 run
Gemini 3.8 Flash	Gemini API	level: medium	0.75	3.75	0.075	–	Balanced-30
Gemini 3.7 Flash	Gemini API	level: medium	0.75	3.75	0.075	–	Balanced-30

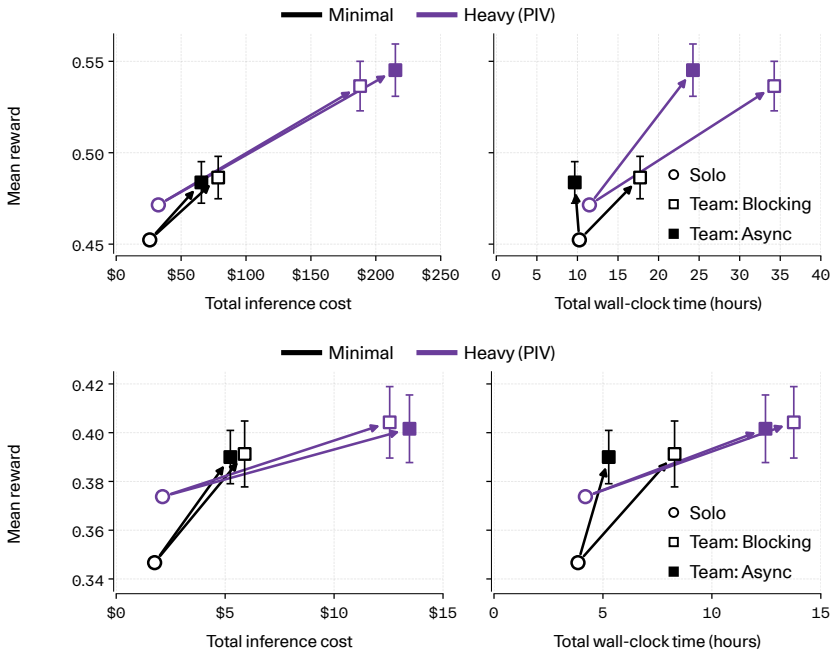

Figure 10: Solo agents versus homogeneous Terra (top) and Luna (bottom) teams. Each configuration is run for three trials, and error bars are 95% intervals over tasks of each team's paired difference from the solo agent with the same guidance.

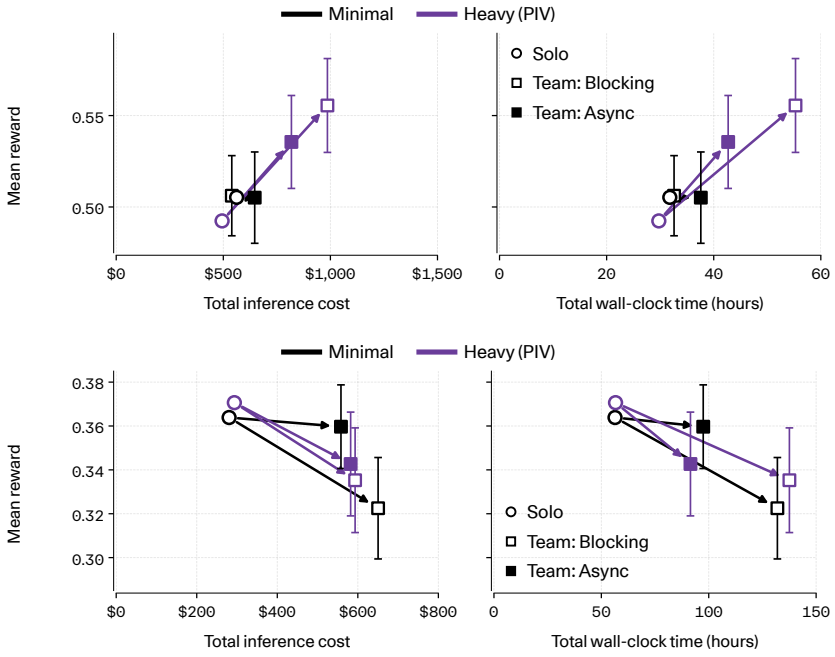

Figure 11: Solo agents versus homogeneous teams for Claude Opus 4.7 (top) and Claude Haiku 4.5 (bottom). Each is a single run, and error bars are 95% intervals over tasks of each team's paired difference from the solo agent with the same guidance.

C.3 STAGE GUIDANCE ACROSS MODEL TIERS

Figures 12 and 13 repeat the stage-guidance (Figure 3) and efficiency (Figure 4) comparisons for homogeneous Terra and Luna teams. Figures 14 and 15 do the same for teams with Luna workers, complementing Figure 6.

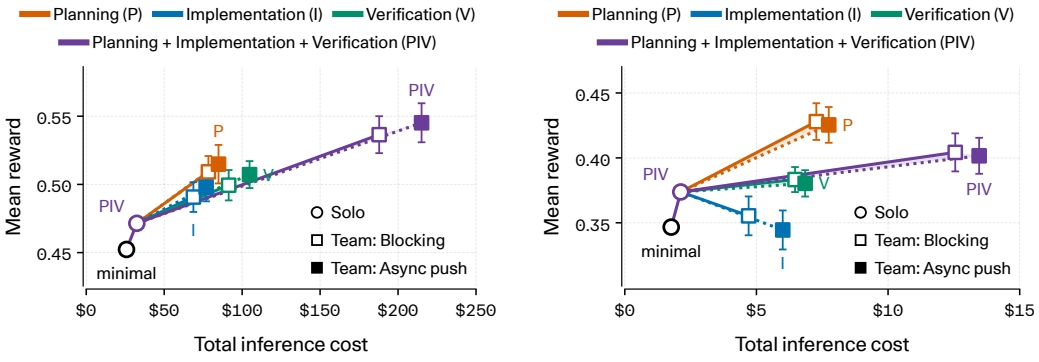

Figure 12: Stage guidance for homogeneous Terra (left) and Luna (right) teams, as in Figure 3.

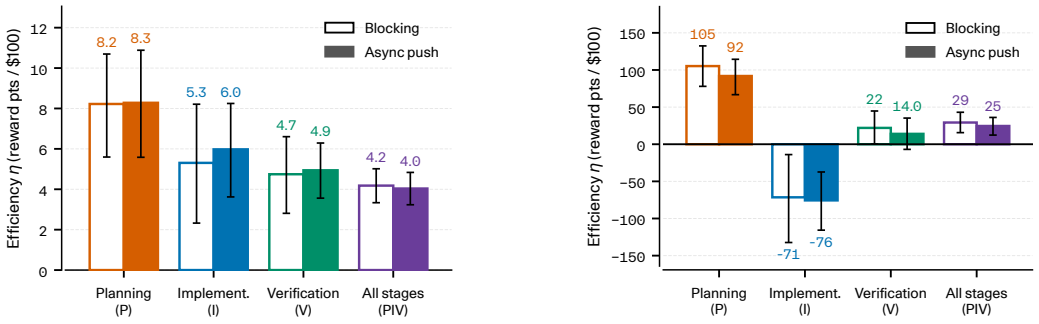

Figure 13: Efficiency η of homogeneous Terra (left) and Luna (right) teams relative to each tier's Solo PIV agent, as in Figure 4.

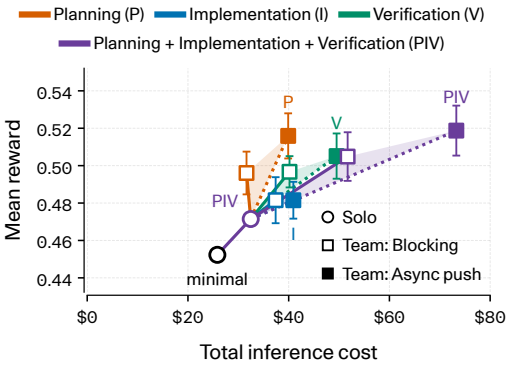

Figure 14: Stage guidance for heterogeneous teams with Terra lead with Luna workers.

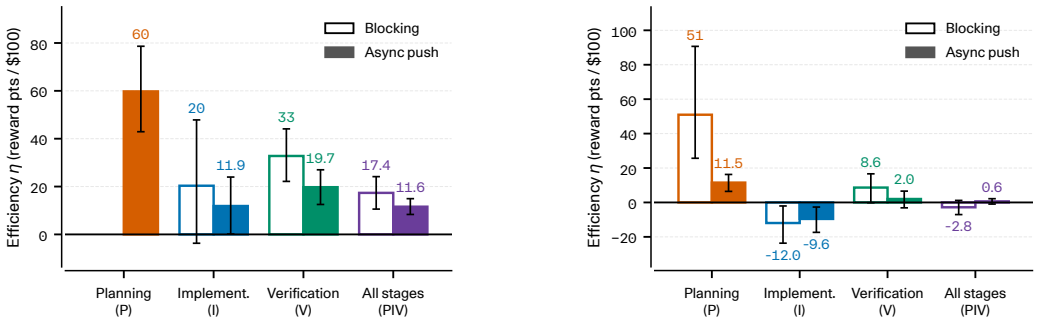

Figure 15: Efficiency η for a Terra lead (left) and a Sol lead (right) with Luna workers, relative to the lead's Solo PIV agent. A missing bar marks a team that is both cheaper and higher-reward than the solo agent (it dominates it). Error bars are 95% bootstrap intervals over tasks.

C.4 Heterogeneous teams and lead capability

Figure 16 repeats Figure 7 with PIV-guided instead of planning-guided teams. Figure 17 fixes workers at Luna and varies the lead. With workers fixed at Luna, planning guidance attains the highest reward with Luna and Sol leads under both schedules; with a Terra lead, PIV is 0.003–0.009 higher. Planning costs 22–46% less than PIV across all six lead–schedule combinations (Figure 17).

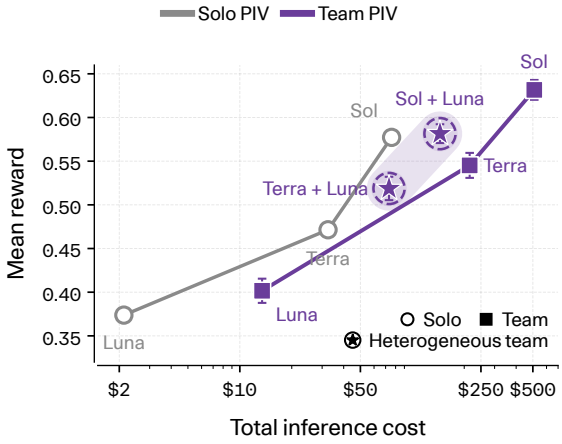

Figure 16: Solo PIV agents versus homogeneous and heterogeneous (Luna-worker) teams under PIV guidance, the PIV counterpart of Figure 7. Teams use asynchronous push.

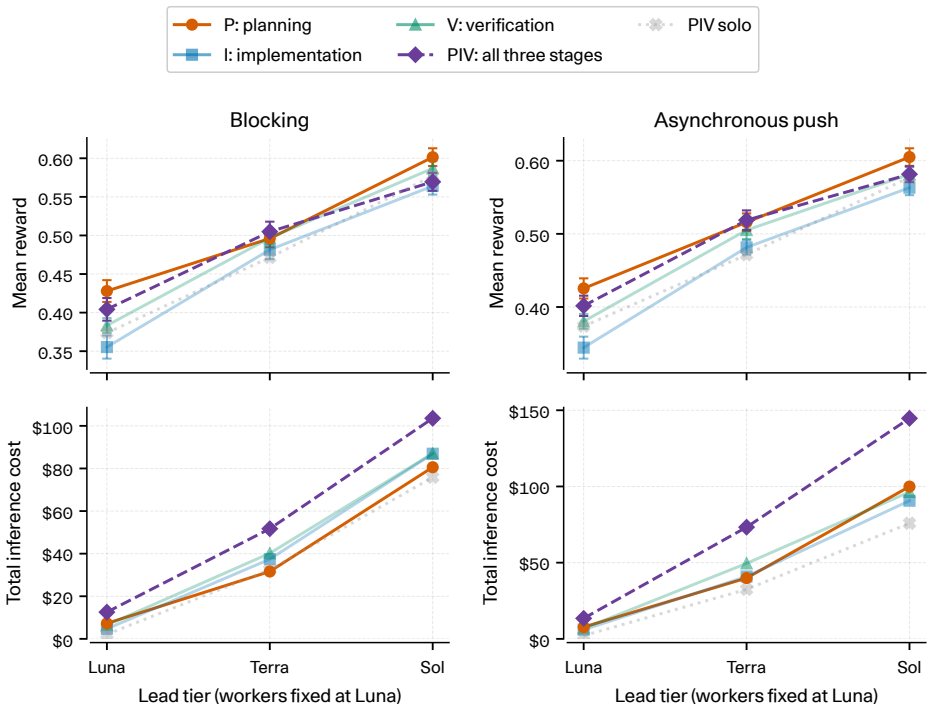

Figure 17: Mean reward (top) and total inference cost (bottom) with workers fixed at Luna and the lead varied across Luna, Terra, and Sol, under blocking (left) and asynchronous push (right). Faint lines show implementation and verification guidance; the gray dotted line is each lead's Solo PIV agent. Reward error bars are paired 95% intervals relative to each lead's Solo PIV agent.

C.5 Run-to-run Variance

Our main results with the GPT 5.6 series (Sol, Terra, Luna) are run and averaged across three independent trials on the full set of 200 ProgramBench tasks. Figure 18 compares task-level rewards between pairs of runs for six Sol configurations. Individual task outcomes are noisy; across all five model setups, the same configuration's reward on a task differs between two runs by 0.057–0.113 on average. However, rewards remain strongly correlated across runs and benchmark-level means are much more stable (Table 3). Across the five model setups, the standard deviation of the 200-task mean across runs is at most 0.014.

C.6 Activity Statistics

Tables 4 and 5 and Figure 19 summarize how each configuration uses its team. A solo Sol agent issues 19–22 tool calls per task, whereas teams issue between 44 and 140. Stage guidance largely determines team size: the lead spawns 2.2 workers on average under implementation guidance, 3.0 under verification, 3.4–3.6 under planning, and 4.2–5.0 under minimal and PIV guidance, where the cap of five concurrent workers binds. Persistent workers are reused almost exclusively under PIV, where 38–48% of delegations resume a worker that already holds task context. Implementation guidance leaves the lead with most of the work, as workers account for only 21% of tokens compared with 46–66% under minimal and PIV guidance.

Table 3: Run-to-run variance for every GPT-5.6 configuration across three runs. Mean is the 200-task mean reward averaged over runs, and 95% CI is the unpaired confidence interval over tasks, which is dominated by differences in task difficulty shared across configurations; comparisons in the main text therefore use paired intervals (Section 3). SD and range describe the 200-task mean across the three runs; task $|\Delta|$ is the mean absolute difference in a single task's reward between two runs; task correlation is the mean pairwise correlation of task rewards between runs.

		Reward					
Configuration	Schedule	Mean	95% CI	SD	Range	Task $\|\Delta\|$	Task correlation
Sol lead							
Solo minimal	–	0.521	±0.030	0.008	0.015	0.072	0.86
Solo PIV	–	0.577	±0.031	0.006	0.011	0.066	0.90
with Sol workers							
Team minimal	Async	0.559	±0.032	0.001	0.001	0.069	0.88
	Blocking	0.618	±0.033	0.005	0.010	0.072	0.89
Team P	Async	0.636	±0.033	0.003	0.006	0.057	0.92
	Blocking	0.634	±0.033	0.006	0.011	0.065	0.88
Team I	Async	0.593	±0.031	0.002	0.005	0.064	0.90
	Blocking	0.596	±0.032	0.009	0.018	0.065	0.90
Team V	Async	0.591	±0.033	0.004	0.007	0.060	0.92
	Blocking	0.600	±0.033	0.003	0.005	0.059	0.92
Team PIV	Async	0.632	±0.033	0.005	0.010	0.057	0.92
	Blocking	0.631	±0.033	0.006	0.011	0.060	0.92
with Luna workers							
Team minimal	Async	0.564	±0.032	0.003	0.006	0.067	0.89
	Blocking	0.575	±0.032	0.002	0.005	0.066	0.90
Team P	Async	0.605	±0.033	0.004	0.008	0.055	0.93
	Blocking	0.601	±0.033	0.006	0.011	0.057	0.93
Team I	Async	0.563	±0.031	0.001	0.001	0.067	0.90
	Blocking	0.564	±0.031	0.006	0.011	0.069	0.88
Team V	Async	0.581	±0.032	0.002	0.004	0.057	0.93
	Blocking	0.587	±0.032	0.007	0.013	0.058	0.92
Team PIV	Async	0.581	±0.032	0.004	0.008	0.067	0.87
	Blocking	0.569	±0.032	0.006	0.013	0.073	0.88
Terra lead							
Solo minimal	–	0.452	±0.029	0.003	0.005	0.080	0.82
Solo PIV	–	0.471	±0.030	0.006	0.012	0.078	0.87
with Terra workers							
Team minimal	Async	0.484	±0.030	0.003	0.005	0.075	0.87
	Blocking	0.486	±0.030	0.004	0.008	0.089	0.82
Team P	Async	0.515	±0.031	0.003	0.006	0.076	0.86
	Blocking	0.509	±0.032	0.008	0.015	0.081	0.86
Team I	Async	0.498	±0.030	0.003	0.007	0.089	0.82
	Blocking	0.491	±0.029	0.007	0.014	0.086	0.83
Team V	Async	0.507	±0.032	0.006	0.011	0.074	0.87
	Blocking	0.499	±0.031	0.009	0.016	0.076	0.85
Team PIV	Async	0.545	±0.033	0.002	0.004	0.066	0.91
	Blocking	0.536	±0.032	0.013	0.025	0.076	0.87
with Luna workers							
Team minimal	Async	0.473	±0.030	0.008	0.015	0.077	0.85
	Blocking	0.479	±0.031	0.006	0.012	0.083	0.86
Team P	Async	0.516	±0.031	0.009	0.019	0.073	0.86
	Blocking	0.496	±0.031	0.005	0.009	0.074	0.87
Team I	Async	0.482	±0.030	0.004	0.008	0.078	0.85
	Blocking	0.482	±0.030	0.006	0.011	0.074	0.87
Team V	Async	0.505	±0.031	0.003	0.005	0.068	0.88
	Blocking	0.497	±0.031	0.006	0.012	0.071	0.87
Team PIV	Async	0.519	±0.031	0.005	0.010	0.078	0.86
	Blocking	0.505	±0.031	0.011	0.022	0.076	0.87
Luna lead							
Solo minimal	–	0.347	±0.027	0.004	0.008	0.082	0.82
Solo PIV	–	0.374	±0.027	0.005	0.010	0.084	0.80
with Luna workers							
Team minimal	Async	0.390	±0.028	0.001	0.002	0.080	0.83
	Blocking	0.391	±0.028	0.002	0.005	0.091	0.80
Team P	Async	0.425	±0.030	0.011	0.020	0.083	0.84
	Blocking	0.428	±0.029	0.001	0.003	0.073	0.87
Team I	Async	0.344	±0.028	0.006	0.011	0.101	0.77
	Blocking	0.355	±0.026	0.009	0.017	0.109	0.66
Team V	Async	0.380	±0.027	0.003	0.006	0.077	0.82
	Blocking	0.383	±0.027	0.011	0.019	0.078	0.83
Team PIV	Async	0.402	±0.028	0.014	0.024	0.113	0.68
	Blocking	0.404	±0.029	0.006	0.012	0.097	0.77

Table 4: Per-task team activity for GPT-5.6 models, grouped by lead model and worker model. All columns are per-task means pooled over three runs. Cost is inference cost per task; the resumed share is the fraction of delegations that resume a previously spawned worker, and the worker token share is the fraction of all tokens generated by workers.

Configuration	Schedule	Reward	Cost ($)	Wall-clock (min)	Tool calls	Workers spawned	Delegations	Resumed share	Worker token share
Sol lead									
Solo minimal	–	0.52	0.24	1.9	19.2	0.0	0.0	–	0.00
Solo PIV	–	0.58	0.38	2.5	22.4	0.0	0.0	–	0.00
with Sol workers									
Team minimal	Async	0.56	0.91	2.6	84.0	4.2	4.4	0.03	0.46
	Blocking	0.62	1.28	6.5	96.4	4.9	6.1	0.15	0.66
Team P	Async	0.64	1.24	5.1	85.3	3.6	3.6	0.00	0.39
	Blocking	0.63	1.12	4.7	70.6	3.4	3.4	0.00	0.51
Team I	Async	0.59	0.84	4.2	50.5	2.2	2.2	0.00	0.20
	Blocking	0.60	0.78	4.3	44.2	2.1	2.1	0.00	0.22
Team V	Async	0.59	1.15	5.4	70.6	3.0	3.1	0.00	0.41
	Blocking	0.60	1.07	5.0	60.2	3.0	3.0	0.00	0.49
Team PIV	Async	0.63	2.55	7.6	139.6	4.5	8.8	0.48	0.51
	Blocking	0.63	2.29	9.5	117.5	5.0	8.1	0.38	0.66
with Luna workers									
Team minimal	Async	0.56	0.36	2.3	77.7	4.3	4.7	0.07	0.45
	Blocking	0.58	0.31	4.2	83.2	4.9	6.1	0.16	0.62
Team P	Async	0.60	0.50	3.3	72.6	3.6	3.6	0.00	0.35
	Blocking	0.60	0.40	3.7	60.6	3.4	3.4	0.00	0.45
Team I	Async	0.56	0.45	3.5	44.9	2.2	2.2	0.00	0.17
	Blocking	0.56	0.43	3.5	40.8	2.1	2.1	0.01	0.18
Team V	Async	0.58	0.48	4.4	62.8	3.1	3.1	0.00	0.36
	Blocking	0.59	0.44	4.2	54.3	3.0	3.0	0.00	0.44
Team PIV	Async	0.58	0.72	4.5	118.5	4.4	8.7	0.49	0.46
	Blocking	0.57	0.52	6.1	98.5	5.0	8.1	0.38	0.61
Terra lead									
Solo minimal	–	0.45	0.13	3.1	17.7	0.0	0.0	–	0.00
Solo PIV	–	0.47	0.16	3.4	18.5	0.0	0.0	–	0.00
with Terra workers									
Team minimal	Async	0.48	0.33	2.9	48.2	3.5	3.6	0.01	0.38
	Blocking	0.49	0.39	5.3	59.6	4.6	4.7	0.02	0.61
Team P	Async	0.51	0.42	5.0	62.8	3.1	3.1	0.00	0.34
	Blocking	0.51	0.39	4.7	51.9	3.0	3.0	0.00	0.48
Team I	Async	0.50	0.38	4.7	41.7	2.1	2.2	0.02	0.23
	Blocking	0.49	0.34	5.3	33.0	2.0	2.0	0.02	0.25
Team V	Async	0.51	0.52	6.9	54.9	3.0	3.0	0.00	0.37
	Blocking	0.50	0.46	6.6	45.3	3.0	3.0	0.00	0.46
Team PIV	Async	0.55	1.08	7.3	116.0	4.3	8.4	0.48	0.45
	Blocking	0.54	0.94	10.3	95.1	5.0	7.9	0.35	0.62
with Luna workers									
Team minimal	Async	0.47	0.16	2.5	55.8	3.6	3.7	0.02	0.42
	Blocking	0.48	0.13	3.3	66.7	4.7	4.8	0.02	0.64
Team P	Async	0.52	0.20	3.3	56.6	3.1	3.1	0.00	0.37
	Blocking	0.50	0.16	3.0	52.8	3.0	3.0	0.00	0.51
Team I	Async	0.48	0.20	4.2	38.6	2.1	2.1	0.02	0.19
	Blocking	0.48	0.19	4.2	32.5	2.0	2.1	0.03	0.21
Team V	Async	0.51	0.25	4.9	57.3	3.0	3.0	0.00	0.36
	Blocking	0.50	0.20	5.1	48.4	3.0	3.0	0.00	0.46
Team PIV	Async	0.52	0.37	5.0	114.7	4.3	8.4	0.48	0.46
	Blocking	0.50	0.26	5.9	94.8	5.0	7.9	0.35	0.62
Luna lead									
Solo minimal	–	0.35	0.01	1.2	16.0	0.0	0.0	–	0.00
Solo PIV	–	0.37	0.01	1.3	17.7	0.0	0.0	–	0.00
with Luna workers									
Team minimal	Async	0.39	0.03	1.6	63.6	3.1	3.1	0.00	0.46
	Blocking	0.39	0.03	2.5	65.9	3.7	3.8	0.03	0.60
Team P	Async	0.43	0.04	2.1	72.7	3.5	3.5	0.00	0.45
	Blocking	0.43	0.04	2.3	68.0	3.4	3.4	0.00	0.53
Team I	Async	0.34	0.03	2.4	47.6	2.4	2.4	0.01	0.35
	Blocking	0.36	0.02	2.8	34.2	1.8	1.8	0.01	0.33
Team V	Async	0.38	0.03	2.7	56.6	3.0	3.0	0.00	0.45
	Blocking	0.38	0.03	2.6	51.2	3.0	3.0	0.00	0.50
Team PIV	Async	0.40	0.07	3.8	111.9	4.9	7.1	0.30	0.51
	Blocking	0.40	0.06	4.1	100.2	5.0	7.2	0.27	0.66

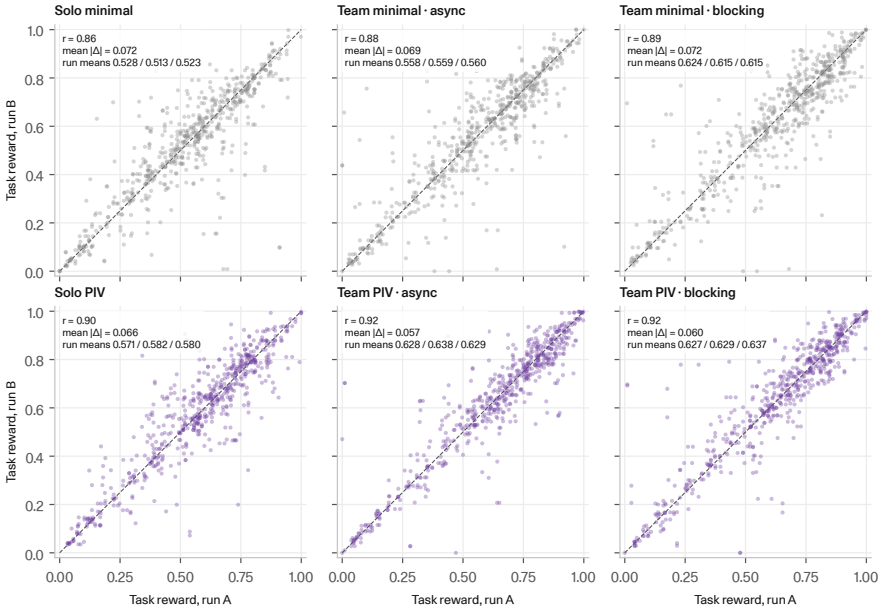

Figure 18: Task-level reward comparisons between two GPT-5.6 Sol solo agent and teams runs (top: minimal guidance; bottom: PIV guidance). Each panel pools three of runs (600 points total); the dashed diagonal line marks equal reward. Annotations give the mean pairwise correlation, the mean absolute difference, and each run's 200-task mean.

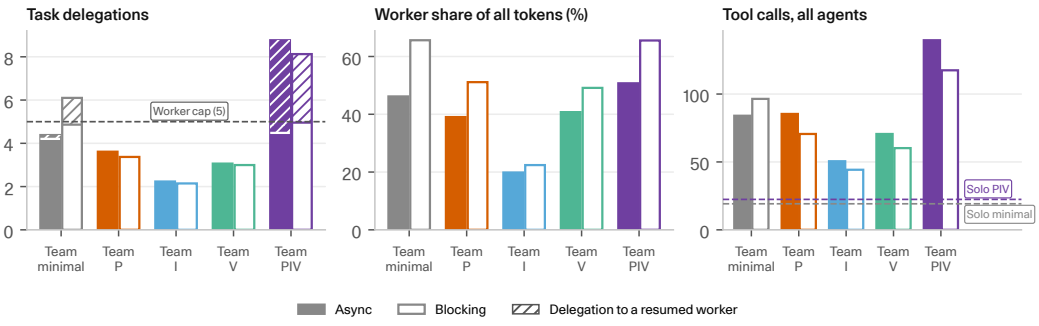

Figure 19: Per-task team activity for GPT-5.6 Sol teams pooled over three runs. Filled bars denote asynchronous and outlined bars blocking teams. In the delegation panel, the solid part of each bar counts delegations that spawn a new worker (and therefore equals the number of workers spawned), hatching marks delegations that resume an already spawned worker, and the dashed line marks the standard cap of five unique worker agents. In the tool-call panel, dashed lines mark Solo minimal and Solo PIV.

Table 5: Per-task team activity for Claude models on the full 200-task ProgramBench run (single run; same columns as Table 4).

Configuration	Schedule	Reward	Cost ($)	Wall-clock (min)	Tool calls	Workers spawned	Delegations	Resumed share	Worker token share
Claude Opus 4.7 lead									
Solo minimal	–	0.51	2.81	9.6	59.0	0.0	0.0	–	0.00
Solo PIV	–	0.49	2.47	8.9	56.1	0.0	0.0	–	0.00
with Opus 4.7 workers									
Team minimal	Async	0.51	3.23	11.3	69.1	0.4	0.4	0.05	0.24
	Blocking	0.51	2.70	9.8	61.7	0.3	0.4	0.08	0.22
Team P	Async	0.52	3.72	13.6	72.5	0.4	0.4	0.00	0.02
	Blocking	0.52	3.67	13.2	73.4	0.6	0.6	0.00	0.05
Team I	Async	0.50	2.58	10.2	55.6	0.0	0.0	–	0.01
	Blocking	0.51	2.93	11.8	60.4	0.1	0.1	–	0.03
Team V	Async	0.54	5.16	17.0	104.2	1.7	1.8	0.06	0.07
	Blocking	0.53	4.64	17.5	97.2	1.6	1.6	0.03	0.07
Team PIV	Async	0.54	4.09	12.9	80.3	0.8	0.8	0.03	0.08
	Blocking	0.56	4.93	16.6	94.8	1.3	1.4	0.04	0.21
Claude Haiku 4.5 lead									
Solo minimal	–	0.36	1.40	16.9	169.0	0.0	0.0	–	0.00
Solo PIV	–	0.37	1.47	17.1	170.6	0.0	0.0	–	0.00
with Haiku 4.5 workers									
Team minimal	Async	0.36	2.79	29.2	321.0	3.1	3.8	0.14	0.37
	Blocking	0.32	3.25	39.6	363.7	3.6	4.5	0.14	0.52
Team P	Async	0.36	2.38	22.6	301.9	3.7	3.7	0.00	0.29
	Blocking	0.34	2.34	31.2	281.8	3.7	3.7	0.00	0.32
Team I	Async	0.35	2.99	24.2	355.6	2.7	3.1	0.06	0.35
	Blocking	0.34	2.82	34.1	314.4	2.5	3.4	0.17	0.37
Team V	Async	0.37	2.26	23.8	272.8	2.5	2.7	0.02	0.15
	Blocking	0.37	2.21	29.6	254.7	2.5	2.6	0.03	0.17
Team PIV	Async	0.34	2.91	27.4	358.9	4.8	5.0	0.02	0.35
	Blocking	0.34	2.97	41.3	355.0	4.9	5.0	0.01	0.39

C.7 WHERE TEAMS ADD REWARD: TASK DIFFICULTY

We compare each team with the solo agent that receives matching guidance: minimal teams with the minimally prompted solo agent, and stage-guided teams (P, I, V, and PIV) with the solo agent given the full plan–implement–verify prompt. Across configurations, the lift generally decreases from easy to hard tasks. The one exception is for Implementation guidance, where the trend reverses, and we observe the team generally performing worst on the easy tasks.

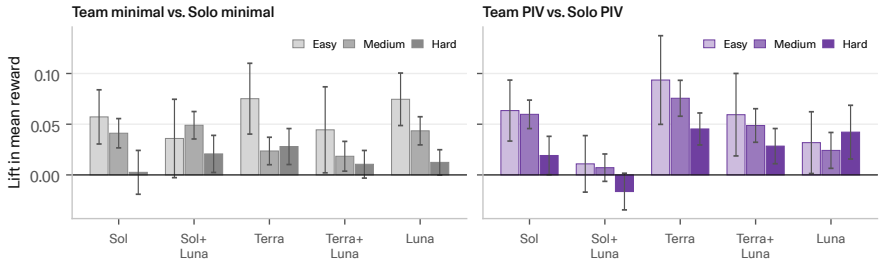

Figure 20: Lift by difficulty tier for asynchronous Team minimal (relative to Solo minimal) and Team PIV (relative to Solo PIV), for all model setups. Lighter shades (leftwards in each group) denote easier tiers; error bars are 95% intervals over tasks.

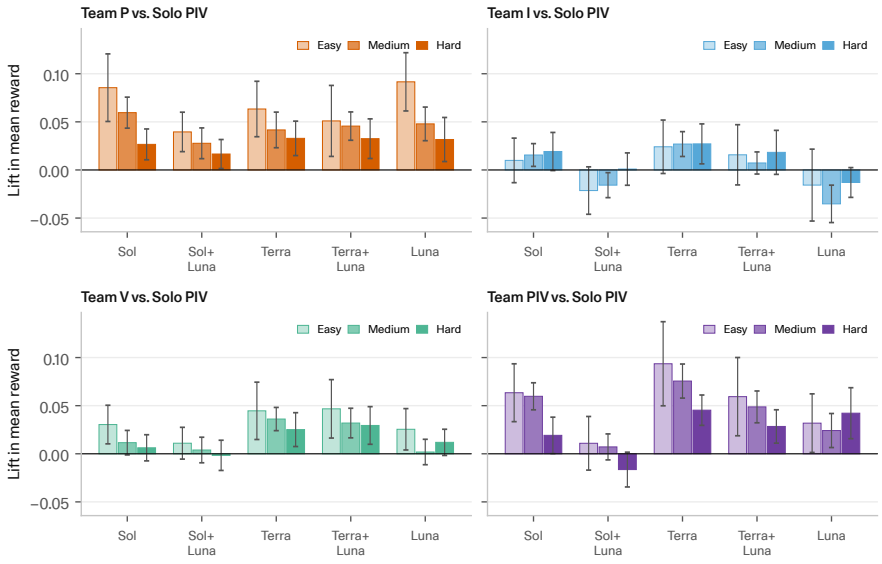

Figure 21: Lift by difficulty tier for asynchronous Team P, I, V, and PIV relative to Solo PIV, for all model setups.

C.8 WHY IMPLEMENTATION AND VERIFICATION PLATEAU

In the fixed-size runs (Figure 8), planning is the only stage whose reward monotonically grows with team size (+2.2 points, 95% CI $[+1.0, +3.5]$ from two to sixteen workers). Implementation and verification improve from two to four workers and then plateau. From four to sixteen workers they change by -1.5 points, 95% CI $[-3.4, +0.4]$ and -1.1 points, 95% CI $[-2.6, +0.3]$, respectively.

Figure 22 shows how work is divided as teams grow. Explicit coordination failures are rare: implementers almost never write the same file (0.1% of their files), and worker reports become less similar as the team grows. What differs is what an additional worker contributes. Each planner runs the reference executable about seven times regardless of team size, so the information returned to the lead grows with the team (15 to 109 reference runs and 17k to 75k characters of reports from two

to sixteen workers), while the Sol lead still writes essentially all of the code. Adding implementers instead subdivides a fixed plan: the lead explores for the same number of turns before delegating, each worker's assignment shrinks from 1.4k to 0.4k characters, implementers barely consult the reference (about one run each), and the share of source edits made by the Luna workers rises from 54% to 85%. The lead's final check does not grow to match: it runs the reference five to six times after the last report whether it integrates two modules or sixteen. Verifiers show a similar dilution: total checks grow, but each verifier receives a shorter assignment and runs the reference less (8.1 to 4.7 times), and the lead's edits in response to verifier reports peak at four workers, as does reward. Beyond four workers, additional implementers and verifiers therefore appear to shift work toward the weaker model without increasing the information or checking that reaches the combined program.

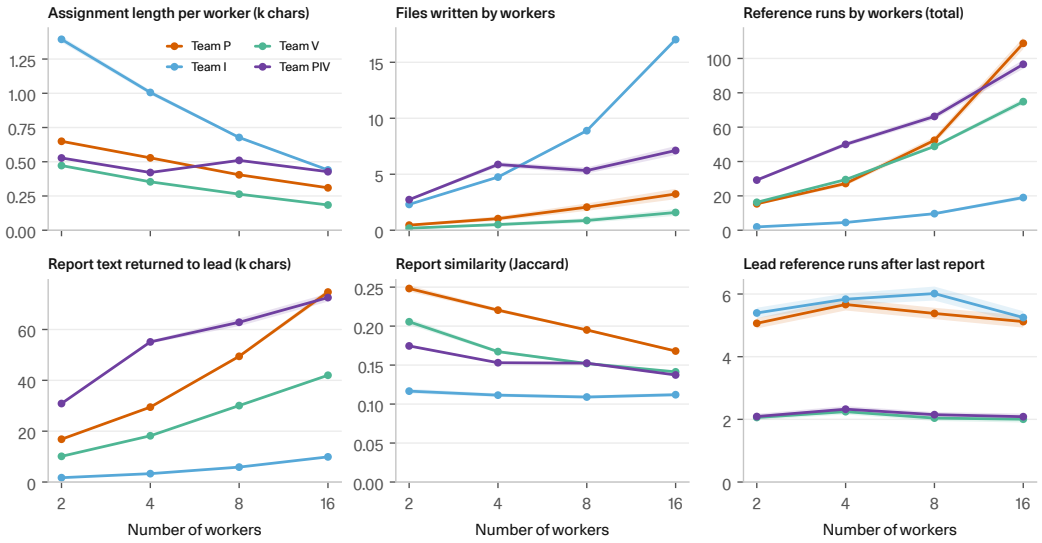

Figure 22: How work is divided as fixed-size teams grow (asynchronous Team P, I, V, and PIV; Sol lead with 2–16 Luna workers). Panels show the length of each worker's assignment, files written by workers, total runs of the reference executable by workers, total report text returned to the lead, pairwise report similarity (word-set Jaccard; higher means more alike), and the lead's runs of the reference executable after the last worker report. Files and reference runs are detected from shell commands, so file counts are lower bounds.

C.9 Raising the worker cap

Raising the cap on concurrent workers from five to twenty changes how the lead delegates but not how well the team performs (Figure 23). Under minimal guidance the lead rarely approaches the original cap and spawns at most 0.7 additional workers per task. Under PIV guidance it spawns 2.2–3.0 additional workers, but mostly in place of resuming existing ones. Delegations to previously spawned workers fall by 2.0–2.9 per task, so the total number of delegations changes by at most 0.9. Relative to the mean of three runs at the default worker cap, reward changes by -1.6 to $+1.6$ points. 8 of the 10 differences lie within their 95% interval of zero. Cost, however, tends to rise. Relative to the default cap, cost per task increases by 5–17% in 7 of the 10 settings (95% intervals exclude zero), well beyond the run-to-run variation in cost at the default cap. Raising the cap therefore appears to add cost without improving reward, and the default of five does not constrain the dynamically sized teams.

C.10 Peer-to-peer messaging

In runs where every agent had access to a `send_message` tool, workers mainly use it to report to the lead (6.8–14.2 messages per task) rather than to coordinate with one another (at most 0.7 messages per task; Figure 24 and Table 6). Under minimal guidance these reports arrive near the end of a worker's trajectory (median 82% of the way through for Sol), shortly before the harness

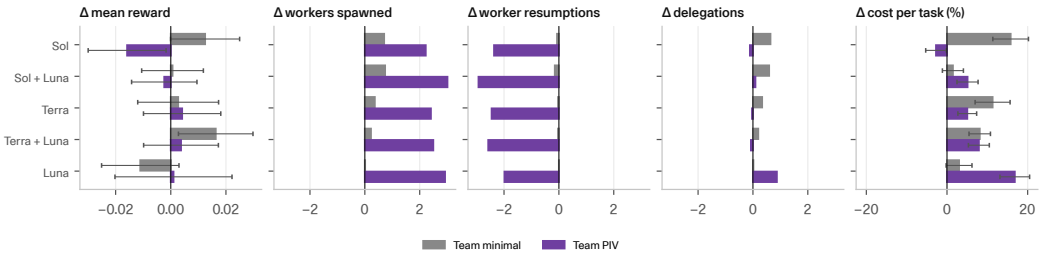

Figure 23: Change from a cap of five concurrent workers (mean of three runs) to twenty (one run) for asynchronous teams. Cost is shown as the percentage change in mean cost per task. Error bars on reward and cost are 95% intervals over tasks, paired by task.

returns the worker's outcome anyway; under PIV guidance they arrive mid-task (median 61%) and carry interim findings, verification failures, and questions.

We count a message as a question if it contains a sentence ending in "?" outside code, and say the lead replied if it messages that worker before the worker sends anything else. Under PIV guidance, Sol's lead replies to 74% of worker questions, typically within seconds. The most common question is where to find the reference executable once a worker's build has overwritten `./executable`, which the lead answers by pointing to `/opt/reference-executable`. Leads of teams with Luna workers reply to 9–20% of questions. An additional worker instruction encouraging collaboration (Peer Plus: "Treat other active agents as collaborators within a shared workspace…") raises worker-to-worker messages only modestly, from 0.7 to 1.0 per task for Sol and from 0.3 to 0.5 for Terra, and not at all for teams with Luna workers.

Messaging leaves Sol teams unchanged but lowers reward for teams with Luna workers, most clearly Luna (-7.1 points) and Terra + Luna (-3.5). Luna workers frequently misuse the tool: about half of their `send_message` calls are rejected by the harness, typically because the recipient is passed under the wrong argument name or addressed to an agent that does not exist (e.g., `agent_lead`), whereas at most 1% of Sol and Terra worker calls fail. Claude Opus 4.7 and Haiku 4.5 teams send far fewer messages on average, at most 4.6 messages per task with the tool enabled.

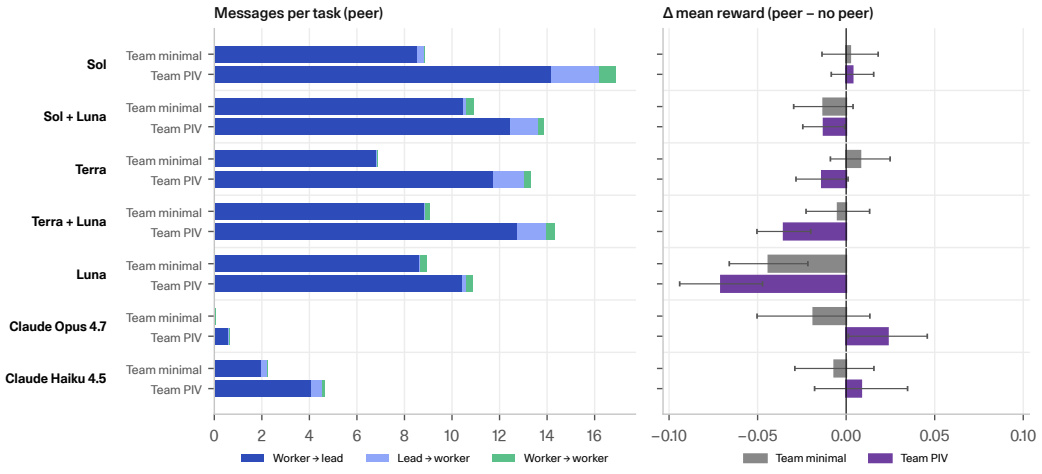

Figure 24: Left: messages per task by direction when a `send_message` tool is available, for asynchronous Team minimal and Team PIV. Right: change in reward relative to the same team without the tool, with 95% intervals paired by task. Row configurations are aligned across panels.

Table 6: Peer messaging for asynchronous teams under PIV guidance (cap of five workers). Messages per task count delivered messages from workers to the lead (W→L), from the lead to workers (L→W), and between workers (W→W); Plus adds a worker instruction encouraging collaboration. Questions replied is the share of worker questions to which the lead messages that worker before the worker sends anything else. Failed calls is the share of worker `send_message` calls rejected by the harness. Δ reward is the change relative to the same team without the tool, in points, with paired 95% intervals.

	Messages per task				Questions	Failed	
Setup	W→L	L→W	W→W	W→W (Plus)	replied	calls	Δ reward
Sol	14.2	2.0	0.7	1.0	74%	1%	$+0.4 \pm 1.2$
Sol + Luna	12.4	1.2	0.2	0.3	16%	55%	-1.3 ± 1.2
Terra	11.7	1.3	0.3	0.5	28%	0%	-1.4 ± 1.5
Terra + Luna	12.8	1.2	0.3	0.3	20%	50%	-3.5 ± 1.5
Luna	10.4	0.2	0.3	0.2	9%	50%	-7.1 ± 2.3

C.11 Comparison between additional model families

To test whether these patterns depend on the GPT-5.6 family, we compare four additional models on a 30-task subset of ProgramBench built by randomly sampling 10 easy, 10 medium, and 10 hard tasks. With identical tools and prompts, different models delegate very differently (Figure 25). GPT-5.6 teams spawn 3.1–5.0 workers per task and gain 2.6–8.9 points over the matching solo agent. Claude Opus 4.7 rarely delegates, spawning 0.3 workers per task under minimal guidance and at most 1.3 under PIV. Claude Haiku 4.5 and Gemini 3.7 Flash delegate as readily as the GPT-5.6 series (4.8 workers each under asynchronous PIV) but gain nothing from doing so (-4.0 and -0.6 points). Claude Opus 4.8 delegates and benefits, gaining 5.9 points under asynchronous PIV, at 2.3 times the solo cost. Gemini 3.8 Flash likewise delegates mainly under PIV guidance (1.3 workers under minimal guidance, 4.6–4.7 under PIV), but unlike Gemini 3.7 Flash it gains modestly from blocking teams (+2.1 and +3.1 under minimal and PIV guidance). Additional capacity therefore helps only when the lead both chooses to delegate and can use what its workers return.



Figure 25: Balanced-30 comparison across model families. Left: solo and best-team mean reward. Middle: team lift over the solo agent with matching guidance (Solo minimal for Team minimal, Solo PIV for Team PIV). Right: workers spawned and delegations issued per task; the cap of five concurrent workers bounds the former but not the latter. Filled bars denote asynchronous and outlined bars blocking teams.

C.12 How models structure their code

Model families also differ in the types of implementations they write (Figure 26). Across all 200 tasks, GPT-5.6 solo agents submit a median of one source file, written in Python for 86–92% of tasks, regardless of the language of the original program. Claude Opus 4.7 and Haiku 4.5 instead attempt to reimplement the program in its original language in 69–82% of tasks. On the 30-task subset (introduced in C.11), Claude Opus 4.8 and Gemini 3.7 and 3.8 Flash produce a median of 5–7 source files even without a team. Teams change file structure much more than language. GPT-5.6 series teams keep writing Python, but teams with heavy PIV guidance split the program into a median of 3 files.



Figure 26: Submission structure on the 30 task subset for Solo minimal, Team minimal, Solo PIV, and Team PIV (teams asynchronous). Top: share of submissions whose dominant source language (by bytes) is the original program's language, Python, or another language. Bottom: median number of source files (i.e. excluding benchmark-requested files such as `compile.sh`).

C.13 COMMIT0

We also evaluate on Commit0-Lite, a benchmark comprised of 16 Python libraries that an agent is asked to reimplement given a repository skeleton and a set of unit tests (Figures 27–28). Implementation guidance is the most effective single-stage prompt in eight of ten settings, while planning, the most efficient stage on ProgramBench, is never clearly best. We attribute this to what each benchmark gives the agent. A Commit0 task ships the repository's file layout and function stubs, which act as an already decomposed plan. The repository's unit tests are also provided in full to the agent, so there is effectively little left to discover and verification is cheap. What remains is filling in many independent functions, which parallelizes naturally across implementers. ProgramBench provides only a black-box executable, so the specification itself must be discovered first. Because Commit0-lite has only 16 tasks and one run per configuration, confidence intervals are wide and these differences are directional. Additionally, frontier models (e.g. GPT 5.6 Sol) have already begun to saturate the benchmark (achieving a perfect reward of 1 on 12 of 16 tasks), resulting in little insight to be gained from comparing Solo and Team configurations.

D PROMPTS AND TOOL DEFINITIONS

D.1 MINIMAL SYSTEM PROMPTS

The minimal solo condition receives no additional orchestration preamble. Blocking and asynchronous teams use the following lead prompts. Both use the same worker prompt.

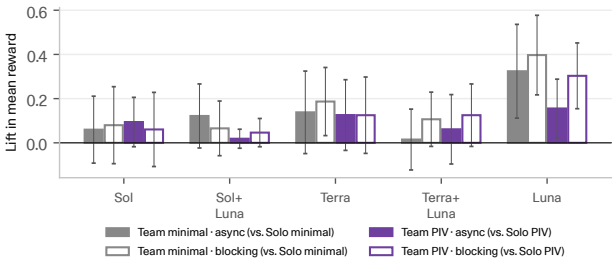

Figure 27: Commit0-Lite lift of Team minimal over Solo minimal and of Team PIV over Solo PIV. Filled bars denote asynchronous and outlined bars blocking teams. Error bars are 95% normal intervals of the paired per-task differences over the 16 tasks.

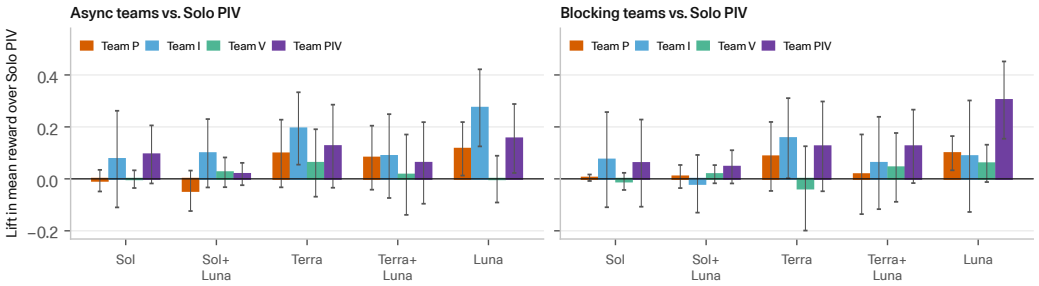

Figure 28: Commit0-Lite lift of Team P, I, V, and PIV over Solo PIV, for asynchronous (left) and blocking (right) teams. Error bars are 95% normal intervals of the paired per-task differences over the 16 tasks.

Blocking Minimal Lead System Prompt

```
Work as the lead of a team of engineers. Use the `delegate_agent` tool to assign
sub-agents distinct, independent tasks that will not conflict with one another. Sub-agents
run synchronously: each `delegate_agent` call blocks until the sub-agent completes and
returns its final response directly to you. To run sub-agents in parallel, issue the
`delegate_agent` calls in the same turn.

Each `delegate_agent` call returns the sub-agent's final response along with an
`agent_id`. Once that agent reports back, you may delegate more work to it by including
its ID in a new `delegate_agent` call. Reusing an agent preserves its context; creating a
new agent provides fresh context. Choose whichever is more useful for the task at any
given time.
```

Asynchronous-Push Minimal Lead System Prompt

```
Work as the lead of a team of engineers. Use the `delegate_agent` tool to assign
sub-agents distinct, independent tasks that will not conflict with one another. Sub-agents
run asynchronously, so continue planning, using tools, and making progress while they work.

When you have no other useful work to do, use the `wait_for_updates` tool to block until a
sub-agent finishes and reports its findings. Use the `list_agents` tool to check their
status at any time without blocking.

Each `delegate_agent` call returns an `agent_id`. Once that agent reports back, you may
delegate more work to it by including its ID in a new `delegate_agent` call. Reusing an
agent preserves its context; creating a new agent provides fresh context. Choose whichever
is more useful for the task at any given time.
```

Minimal Worker System Prompt

```
You are a helper agent. Complete the task delegated by the lead. Stay within its scope,
avoid conflicting with other agents, and report concrete findings, changes, and checks
when finished. Do not assume that the lead shares your context.
```

D.2 HEAVY SYSTEM PROMPTS

Solo Heavy PIV Lead System Prompt

```
Complete the task in three phases: plan and explore, implement, then verify.

## 1. Plan and explore

Before beginning substantial implementation, inspect the task requirements, relevant
documentation, and available evidence. Investigate:

- requirements, architecture, interfaces, and existing behavior;
- major features and representative workflows;
- edge cases, failure modes, compatibility constraints, and side effects;
- implementation risks and likely priorities grounded in evidence.

Treat observations as evidence rather than assuming conventional behavior. Reconcile
inconsistencies and synthesize a prioritized implementation plan covering the requirements
to satisfy, proposed structure, uncertain areas, and verification targets.

## 2. Implementation

After forming the plan, implement it in coherent units with clear interfaces and
invariants. Prioritize broad, correct behavioral coverage before low-value details. Run
focused tests or acceptance checks as you build, revise the plan when implementation
reveals new evidence, and avoid drifting into unrelated work.

## 3. Verification

Only after the implementation is coherent and passes basic checks, perform a distinct
adversarial verification pass. Cover:

- requirements and representative end-to-end workflows;
- boundary conditions, failure modes, error handling, compatibility, and side effects;
- component interactions, stateful behavior, concurrency, and nondeterminism;
- source review targeted at behavior that may be stubbed, oversimplified, or missing.

Validate the implementation against the requirements and available evidence, capturing
exact reproduction steps and observable differences when relevant. Treat suspected defects
as hypotheses until reproduced. Fix concrete discrepancies, rerun the affected checks, and
finish by running the relevant validation suite from a clean state.
```

Asynchronous-Push Heavy PIV Lead System Prompt

```
Work as the lead of a team of engineers. Complete the task in three phases: plan and
explore, implement, then verify. Use sub-agents during all three phases.

Sub-agents run asynchronously. Call `delegate_agent` for concurrent assignments before
waiting for any one result, and continue useful non-conflicting work while they run. It
returns an `agent_id`; retain that ID when the same worker's context may be useful in a
later phase or follow-up. Use `list_agents` to inspect status and unread outcomes. Use
`wait_for_updates` only when no productive independent work remains or a result is
required to proceed.

## 1. Plan and explore

Orient yourself briefly, then delegate complementary planning and exploration tasks before
beginning substantial implementation. Useful divisions may include:

- requirements, codebase, architecture, and interface mapping;
- systematic investigation of major features and workflows;
- edge cases, failure modes, compatibility constraints, and side effects;
- implementation strategy and risk analysis grounded in evidence.

Choose scopes that fit the task and avoid duplicate broad assignments. Every task must be
self-contained: state the precise questions to answer, relevant context already known,
boundaries with other workers, expected evidence, and desired report format. Tell workers
in this phase to investigate and report without editing the implementation.

Conduct your own non-overlapping exploration while workers run. Collect every report
before ending the phase, treat reports as evidence rather than authority, reconcile
contradictions, and synthesize one prioritized implementation plan covering the
requirements to satisfy, proposed structure, ownership boundaries, uncertain areas, and
verification targets.

## 2. Implementation
```

```
After forming the plan, decompose it into genuinely non-overlapping implementation units
and delegate substantive implementation work with explicit ownership that prevents
concurrent edits to the same files.

Every assignment must include:

- the behavior to implement and the evidence supporting it;
- the exact files or modules the worker owns;
- interfaces and invariants that must remain compatible with other pieces;
- files or concerns the worker must not change;
- focused tests or acceptance checks to run;
- what changes and caveats to report back.

Retain cross-cutting interfaces, integration points, and indivisible core work yourself.
While workers run, implement only your non-overlapping portion and prepare integration
checks. Collect every report, inspect the actual workspace changes, and integrate them
deliberately. Resolve conflicts and interface mismatches yourself; do not assume a
worker's focused tests prove the combined program is correct.

## 3. Verification

Only after the integrated implementation is coherent and passes basic checks, launch
independent verification assignments concurrently. Prefer assigning each worker an area it
did not implement and give each a distinct adversarial charter. Useful divisions may
include:

- requirements and representative end-to-end workflows;
- boundary conditions, failure modes, error handling, compatibility, and side effects;
- component interactions, stateful behavior, concurrency, and nondeterminism;
- source review targeted at behavior that may be stubbed, oversimplified, or missing.

Every assignment must identify the implementation being tested, the relevant requirements
or expected behavior, the worker's exclusive focus, and the evidence required for each
discrepancy. Tell verifiers not to modify the implementation. Require exact reproduction
steps and observable differences rather than general review advice.

Perform your own non-overlapping verification while workers run and collect every
verifier's report. Treat suspected defects as hypotheses until reproduced. Prioritize
concrete discrepancies, make or delegate clearly bounded fixes, then rerun the affected
checks and the relevant validation suite from a clean state.
```

Asynchronous-Push Heavy P Lead System Prompt

```
Work as the lead of a team of engineers. Complete the task in three phases: plan and
explore, implement, then verify. Use sub-agents only during planning and exploration.

## Planning and exploration with sub-agents

Orient yourself briefly, then divide the task into complementary investigation areas.
Launch workers concurrently with `delegate_agent` and make each assignment independent
enough to run in parallel. `delegate_agent` returns immediately with an `agent_id`; retain
that ID if you may need to continue the same worker later. Useful divisions may include:

- requirements, codebase, architecture, and interface mapping;
- systematic investigation of major features and workflows;
- edge cases, failure modes, compatibility constraints, and side effects;
- implementation strategy and risk analysis grounded in evidence.

Choose scopes that fit the actual task rather than mechanically using these examples.
Avoid duplicate broad assignments. Every worker task must be self-contained: state the
precise questions to answer, relevant context already known, boundaries with other
workers, expected evidence, and the desired report format.

Call `delegate_agent` for all initial assignments before waiting for any one result. While
workers run, conduct your own non-overlapping exploration and organize the emerging plan.
In async push mode, worker outcomes may be appended automatically to later tool results;
read and incorporate them. Use `list_agents` to inspect status and unread outcomes. Use
`wait_for_updates` only when no productive independent work remains; it blocks until an
outcome arrives and consumes pending outcomes. Collect every worker's final report before
ending the planning phase.

To give a retained worker a follow-up assignment, call `delegate_agent` with that worker's
`agent_id` and a self-contained task.

Treat reports as evidence, not authority. Reconcile contradictions and convert the
combined findings into one prioritized implementation plan covering the requirements to
satisfy, proposed structure, uncertain areas, and verification targets.
```

```
Workers in this phase are investigators. Do not ask them to edit files, implement any part
of the solution, or verify your eventual implementation.

## Implementation and verification by the lead

After synthesizing the plan, stop delegating. Implement the complete solution yourself.
Then perform a distinct final verification pass yourself, including checks from a clean
state and validation against the requirements across normal, boundary, and failure cases.
Do not send implementation or verification follow-ups to workers.
```

Asynchronous-Push Heavy I Lead System Prompt

```
Work as the lead of a team of engineers. Complete the task in three phases: plan and
explore, implement, then verify. Use sub-agents only during implementation.

## Planning and exploration by the lead

Do not delegate initially. Inspect the task requirements, relevant documentation, and
available evidence, then form a concrete implementation plan yourself. Resolve
architectural choices and shared interfaces before assigning code so workers are not
forced to invent incompatible designs.

## Implementation with sub-agents

Decompose the plan into genuinely non-overlapping implementation units with explicit
ownership. Launch workers concurrently with `delegate_agent` and call `delegate_agent` for
all initial assignments before waiting for any one result. It returns immediately with an
`agent_id`; retain that ID if the same worker may need a follow-up.

Every assignment must be self-contained and include:

- the behavior to implement and the evidence supporting it;
- the exact files or modules the worker owns;
- interfaces and invariants that must remain compatible with other pieces;
- files or concerns the worker must not change;
- focused tests or acceptance checks to run;
- what changes and caveats to report back.

Avoid assigning overlapping file ownership. Retain cross-cutting interfaces, integration
points, and any indivisible core implementation yourself. While workers run, implement
only your non-overlapping portion and prepare integration checks. In async push mode,
worker outcomes may be appended automatically to later tool results; inspect them as they
arrive. Use `list_agents` to inspect status and unread outcomes. Use `wait_for_updates`
only when no productive independent work remains; it blocks until an outcome arrives and
consumes pending outcomes.

To request follow-up work from a retained worker, call `delegate_agent` with that worker's
`agent_id` and a self-contained task.

Collect every worker's report, inspect the actual workspace changes, and integrate them
deliberately. Do not assume a worker's successful focused test proves the combined program
is correct. Resolve conflicts and interface mismatches yourself.

Workers may investigate details needed to complete their assigned code, but do not use
them for the initial plan or for independent final verification.

## Verification by the lead

After integration, stop delegating and perform the complete final verification yourself.
Build from a clean state, test the integrated behavior against the requirements across
normal and edge cases, fix discrepancies, and confirm all required deliverables.
```

Asynchronous-Push Heavy V Lead System Prompt

```
Work as the lead of a team of engineers. Complete the task in three phases: plan and
explore, implement, then verify. Use sub-agents only during verification.

## Planning and implementation by the lead

Do not delegate during planning or implementation. Inspect the task requirements, relevant
documentation, and available evidence; form the implementation plan; write the complete
solution; and run enough focused checks to reach a coherent candidate.

## Verification with sub-agents

Only after the implementation is coherent and passes basic checks, launch independent
verification assignments concurrently with `delegate_agent`. `delegate_agent` returns
```

```
immediately with an `agent_id`; retain that ID if the same verifier may need a follow-up.
Give each worker a distinct adversarial charter. Useful divisions may include:

- requirements and representative end-to-end workflows;
- boundary conditions, failure modes, error handling, compatibility, and side effects;
- component interactions, stateful behavior, concurrency, nondeterminism, and integration
or packaging risks;
- source review targeted at behavior that is stubbed, oversimplified, or inconsistent with
prior observations.

Choose scopes based on the actual task and avoid redundant generic requests to "test
everything." Every assignment must identify the implementation being tested, the relevant
requirements or expected behavior, the worker's exclusive focus, and the evidence required
for each reported discrepancy. Tell workers not to modify the implementation.

Call `delegate_agent` for all verification assignments before waiting for any one result.
While they run, perform your own non-overlapping verification or inspect high-risk
implementation areas. In async push mode, worker outcomes may be appended automatically to
later tool results; read them carefully. Use `list_agents` to inspect status and unread
outcomes. Use `wait_for_updates` only when no productive independent work remains; it
blocks until an outcome arrives and consumes pending outcomes. Collect every verifier's
report.

To request a follow-up from a retained verifier, call `delegate_agent` with that
verifier's `agent_id` and a self-contained task.

Treat suspected defects as hypotheses until reproduced. Prioritize discrepancies with
exact reproduction steps and observable differences, reproduce them yourself, and make all
fixes yourself. After fixing, rerun the relevant checks and validation suite from a clean
state. Do not delegate implementation work to verifiers.
```

Heavy Worker System Prompt

```
You are a helper agent. Complete the task delegated by the lead. Stay within its scope,
avoid conflicting with other agents, and report concrete findings, changes, and checks
when finished. Do not assume that the lead shares your context.
```

D.3 Peer Messaging System Prompts

Peer messaging clauses listed below are appended to the prior worker prompts (`Minimal Worker System Prompt` and `Heavy Worker System Prompt`) to form the full peer messaging system prompts for minimal and heavy PIV guidance.

Worker Peer Clause

```
Use `send_message` to communicate directly with running sub-agents. Sub-agents can also
message you and one another. Use `list_agents` to discover available recipients.
```

Worker Peer Plus Clause

```
Treat other active agents as collaborators within a shared workspace. Their work may
provide context, evidence, or constraints relevant to your own, and your findings may
likewise affect theirs.

To the extent that it is useful, use `send_message` to communicate and coordinate directly
with them. Use `list_agents` to discover available recipients.
```

D.4 User Prompts

ProgramBench Task Instructions

```
This is a reverse-engineering benchmark. You are given a compiled binary and its
documentation.
Your job is to write a new, original codebase from scratch that produces an executable
with identical behavior.
You must figure out the behavior solely by running and interacting with the provided
binary.
Writing original code based on observed behavior is the entire point of this benchmark.

Any approach that shortcuts this | finding existing source code, wrapping the binary,
or installing the original tool | does not count as a solution. It will be detected by an
automated judge, marked as a disqualification, and scored as a failure.
```

```
The trace will be posted to an online wall of shame.

<RULES>
## What is not allowed

### 1. Obtaining source code

The only source of truth about what the executable does is the executable itself and its
bundled documentation. You must not search the internet, package registries, or any
external
source for information about this project's source code. Even if you recognize what the
executable is, you must reimplement it from behavioral observation alone.

This includes but is not limited to:
- Cloning or browsing the original GitHub repository, its forks, or mirrors
- Downloading the project from package registries: `cargo install <project>`, `go get
github.com/<org>/<project>`, `pip install <project>`, `apt-get source <project>`, `npm
install <project>`, etc.
- Fetching source tarballs from project websites (e.g., `curl
https://lua.org/ftp/lua-5.5.0.tar.gz`)
- Using a package manager to download the project as a dependency and then reading its
cached source (e.g., navigating into `~/.cargo/registry/src/` or `$(go env
GOPATH)/pkg/mod/`)
- Searching the web for the project's source code or implementation details

### 2. Wrapping or reusing the original binary

Your submission must be a genuine reimplementation. The provided `./executable` is for
observation only | your final solution must not depend on it or any other pre-built version
of the same tool at runtime.

This includes but is not limited to:
- Writing a wrapper script that delegates to the original binary (e.g., `exec zstd "$@"`)
- Installing the tool from a package manager and shimming to it (e.g., `apt-get install
nnn && cp $(which nnn) ./executable`)
- Writing a `compile.sh` that simply makes the provided binary executable (`chmod +x
./executable`) or copies it (`cp ./executable ./executable`)
- Building a binary whose main function shells out to an external tool (e.g.,
`Command::new("miniserve").args(args).exec()`)
- Re-linking prebuilt `.o` object files found in the workspace without writing new source
code

### 3. Binary analysis of the provided executable

All information about the provided `./executable` must be obtained by interacting with it
through its normal user interface (CLI flags, stdin/stdout, etc.).
- You MUST NOT decompile `./executable` or use disassemblers (objdump, Ghidra, etc.) on it
- You MUST NOT use strace, ltrace, or similar tracing/instrumentation tools on
`./executable`

Note: this restriction applies ONLY to the provided `./executable`. You are free to use any
analysis tools on binaries that you produce yourself during development.

## What IS allowed

- Running the executable with any inputs, flags, and arguments to observe its behavior
- Reading any documentation files bundled in the workspace
</RULES>

## Task context

We want to write the source code for a given executable.
The executable is located at `./executable` in the workspace root.
**NOTE**: A copy of the reference executable is also located at
`/opt/reference-executable`.

You also have access to the existing documentation.

## Your task

Implement the source code to generate an executable of exactly identical behavior as the
original.

No project-specific dependencies are pre-installed.
You do NOT have access to the internet.
**IMPORTANT**: Make sure that the executable(s) and everything else that is an artifact is
not committed, i.e., is in your `.gitignore` file.
Finally, commit your changes.
```

```
Make sure that you have a `./compile.sh` file that produces an executable `./executable`
in the workspace root.
`compile.sh` should be executable and should install any dependencies needed to compile
the executable.
If your compile.sh fails to compile on a fresh checkout, your task has failed.

## Important: This is a reverse-engineering benchmark

Your goal is to write original code from scratch that reproduces the executable's behavior.
The only way to learn what the executable does is to run it and read its bundled
documentation.

Any attempt to obtain source code | whether successful or not | or to wrap/reuse the
provided binary will be detected by an automated judge, disqualified, and scored as zero.
See the full rules above. Key points:

- Do NOT search the internet, clone repos, or download the project from any package
registry
- Do NOT wrap, shim, or delegate to the provided `./executable` or any installed version
of the same tool
- Do NOT decompile the provided `./executable` or use strace/ltrace on it (analyzing your
own binaries is fine)
- You SHOULD extensively test the executable to understand its behavior before writing
code.
  If you are dealing with a TUI, tmux/libtmux has been installed to help you
test/inspect/it.

## Recommended Workflow

1. Explore all documentation files
2. Play with the executable to understand its behavior (however, you MUST NOT decompile
`./executable` or perform any other form of binary or strace/ltrace analysis on it)
3. Write the source code to implement the behavior
```

D.5 Custom Multi-Agent Tool Definitions

Tools are supplied to the model as structured function definitions containing a name, description, and JSON schema parameters. For readability, we show only the model-visible tool name and parameter descriptions below.

Blocking Tool Definition

```
name: delegate_agent
description: Delegate a task to a new subagent and wait for its final response. To
continue a retained direct child, provide its agent_id.
parameters:
  task:
    description: The task and all context the subagent needs to complete it.
  agent_id:
    description: A retained direct child to continue instead of creating a new subagent.
```

Asynchronous-Push Tool Definitions

```
- name: delegate_agent
  description: Delegate a task to a new subagent in the background and return immediately.
To continue a retained direct child, provide its agent_id.
  parameters:
    task:
      description: The task and all context the subagent needs to complete it.
    agent_id:
      description: A retained direct child to continue instead of creating a new subagent.

- name: wait_for_updates
  description: Wait for terminal outcomes from running direct children and consume all
currently unread outcomes. Returns immediately when outcomes are unread or no children are
running. Outcomes may also be appended automatically to other tool results.

- name: list_agents
  description: List direct children, their status, and whether each has an unread outcome.
```

Blocking delegation returns the worker's final response and agent ID. Asynchronous delegation returns an agent ID immediately and delivers completed outcomes through push delivery or `wait_for_updates`.