MACE-BENCH: MEASURING AVAILABLE AND REALIZED PARALLELISM IN MULTI-AGENT SOFTWARE ENGINEERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Current multi-agent evaluations report whether a task was solved, not whether it offered parallel work that a multi-agent system (MAS) could have used. We introduce MACE-Bench, which makes that work executable and observable. Each of its 104 long-horizon software tasks carries an executable reference DAG: one feasible decomposition of the work, recovered from a successful trajectory and tested by replay under the task's original verifier. Frontier model–harness systems run each task as a single agent, as the harness's own subagents, as those subagents given the reference plan, and as a deterministic scheduler that launches one agent per ready unit. When, without a plan, the model decides whether to delegate, cost rises up to 2.06× with no detected speedup on any system. When the runtime launches ready units, tasks solved by both conditions finish 1.77× to 2.02× faster at 1.00× to 1.32× the single agent's cost, without solving fewer tasks; the scheduler's plan is solution-derived and the scheduler verifies less, so part of this gain is not parallelism. Trace analysis finds that launching workers is cheap and locates the lost time in parent reasoning before delegation, retained critical work, and serial verification after the workers finish.

1 INTRODUCTION

Evaluations of multi-agent coding systems conflate three questions: whether a task contains independent work, whether a system finds a decomposition that exposes it (*planning*), and whether it runs that decomposition efficiently (*orchestration*). A multi-agent run that is no faster than one agent is hard to read without a reference for the first, and agent behaviour does not reveal it: one agent does one thing at a time, so a long sequential trajectory can conceal a shallow, wide work graph.

Our tasks are long-horizon and repository-level: a single agent needs a median of 7 to 12 min and 22 to 48 tool calls per task, with a long tail: the 90th percentile reaches 22 min (§2). A multi-agent system (MAS) promises to finish them sooner by running independent work concurrently, but long-horizon software benchmarks (Jimenez et al., 2024; Deng et al., 2025; Badertdinov et al., 2025; Merrill et al., 2026) score only the final repository state. Prior work shows that multi-agent gains depend on task structure and the single-agent baseline (Kim et al., 2025), that coordination adds failure modes (Cemri et al., 2025; Zhang et al., 2025b) and communication overhead (Zhang et al., 2025a), and that accuracy-only evaluation rewards needless complexity (Kapoor et al., 2025). None of it says how much parallel work a real software task offers.

MACE-Bench supplies an executable reference for task parallelism. For each task it provides a *reference work DAG*: semantic work units with declared inputs, outputs, file scopes and local checks, and edges that claim work dependencies. The reference is proposed from a successful trajectory and has no standing until executable intervention tests it: fresh workers replay each unit in the frozen task environment, the task's original verifier scores recomposed sibling outputs, withholding a producer's output tests a dependency claim, and omitting a unit tests a necessity claim. The result is an *executable witness*: evidence that one feasible decomposition works, with every structural claim labelled validated, refuted, inconclusive or untested (§2). Figure 1 shows one task.

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.

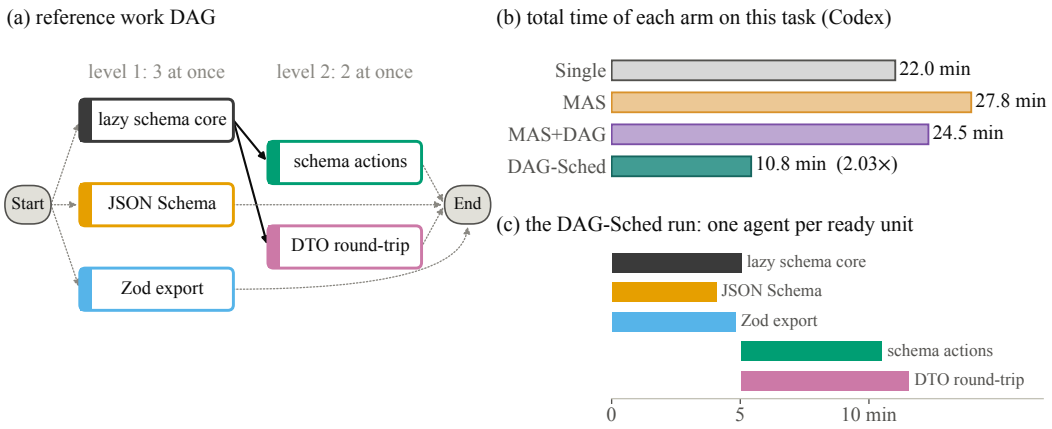


Figure 1: One task, all four arms (Codex × GPT-5.6 Sol, DeepSWE `dynamodb-toolbox` lazy recursive schemas; every arm passes). (a) The reference work DAG: three units can start at once, and two more follow the lazy-schema core. (b) Total time of each arm; DAG-Sched is 2.03× faster than Single (the Codex average is 2.02×). (c) The DAG-Sched run: one agent per ready unit, on the scheduler's clock (including process start-up, so slightly longer than in (b)). One run per cell.

A four-arm protocol (§3) uses the reference to make available parallelism, planning and orchestration separately inspectable: Single is a single agent; MAS is a MAS that decides for itself whether to delegate; MAS+DAG is the same MAS given the reference plan as text; and DAG-Sched is a deterministic scheduler that launches a fresh worker for every ready unit. The protocol partially factors planning from orchestration; it does not separate them causally (Appendix G.1).

On frontier model–harness systems and the 100 PARALLEL tasks, we report success, latency and cost separately (Table 1). The central result is a contrast in who decides to spawn: model-controlled MAS spawns real workers and spends more, while a runtime that launches ready units (DAG-Sched) is the fastest arm on every system. On Codex × GPT-5.6 Sol, MAS reaches 1.00× Single's total-time speed at 2.06× its cost, while DAG-Sched reaches 2.02× (trial time 1.73×) at 1.14× its cost. The reference plan (MAS+DAG) improves latency on the three original systems but not on Pi × Kimi K3. Across the systems we test, on tasks with verified parallel work, model-controlled delegation without a plan shows no detected latency gain, while immediate runtime execution of solution-derived units is much faster; the design localizes, but does not causally isolate, why.

Traces of the three original systems locate the lost time (§5). Spawning is not slow (median dispatch at most 0.02 s over 1437 spawns); the parent-only phases are long. The parent works alone before delegating, often keeps completion-critical work in its own lane (131 of 491 delegated cells), and runs a serial verification tail after the workers finish. These patterns are observed, not shown to cause the latency gap. The runtime scheduler removes both serial phases, but also the warm parent that could catch a unit's missed edge case.

Contributions. (1) *Measurement*: an executable reference for a task's available parallelism, with every structural claim tested by executable intervention (§2). (2) *Benchmark*: MACE-Bench, 104 long-horizon repository tasks from 4 upstream benchmarks (100 with parallel structure, 4 width-one controls), built with a measured yield from 1016 attempts (§2.1). (3) *Finding*: across the systems we test, model-initiated delegation adds cost without a detected latency gain, while runtime-initiated execution is the fastest arm on every system (§4). (4) *Diagnosis*: trace forensics that locate the lost opportunity in parent serialization, retained critical work and parent-owned verification, and document what runtime scheduling gives up (§5).

Table 1: What each multi-agent arm buys over the single agent, Single, on the 100 PARALLEL tasks. *Keeps*: share of the tasks Single solved that the arm also solves. *Adds*: tasks Single failed that the arm solves. *Speedup*: Single's total time over the arm's; *cost*: the arm's dollar cost over Single's; both geometric means over tasks both solved. †: the 95% interval excludes 1 (intervals in Appendix Table 21). One run per cell.

	keeps	adds	speedup	cost
Codex × *GPT-5.6 Sol*: Single solves 83 tasks				
MAS	92%	+8	1.00×	2.06×†
MAS+DAG	95%	+10	1.30×†	1.80×†
DAG-Sched	96%	+12	2.02×†	1.14×†
Claude Code × *Fable 5*: Single solves 81 tasks				
MAS	91%	+7	0.97×	1.14×†
MAS+DAG	93%	+14	1.17×†	1.02×
DAG-Sched	94%	+17	1.94×†	1.10×†
Claude Code × *Opus 5.5*: Single solves 83 tasks				
MAS	94%	+7	0.95×	1.23×†
MAS+DAG	95%	+13	1.26×†	0.96×
DAG-Sched	96%	+10	1.77×†	1.32×†
Pi × *Kimi K3*: Single solves 82 tasks				
MAS	95%	+4	0.95×	1.06×
MAS+DAG	90%	+13	1.01×	0.97×
DAG-Sched	94%	+15	1.84×†	1.00×

2 MACE-BENCH: A TESTED REFERENCE FOR HOW MUCH OF A TASK CAN RUN IN PARALLEL

Every downstream conclusion in this paper is read against a reference decomposition of each task, so the reference is a measurement instrument and we test it before using it. Appendix Table 20 maps each requirement of the measurement to the mechanism of MACE-Bench that meets it.

Tasks, units and the reference work DAG. A *task* is a frozen repository environment, an instruction and a hermetic verifier V taken unchanged from an existing benchmark. Tasks are long-horizon in the repository-level sense, many actions on a real codebase (median Single trial time 7 to 12 min and 22 to 48 tool calls across the three original systems, 90th percentile up to 22 min; Appendix C.3), not a deep work graph. A *semantic work unit* is a worker-sized objective that a fresh agent can complete from the base state, its declared inputs and shared *contract facts* (interfaces siblings agree on). The *reference work DAG* $D = (N, E)$ has such units as nodes, each with an objective, expected paths and a runnable local check. An edge $u \to v$ claims a *work dependency*: v cannot be done without state that u produces; trajectory order, shared files and verification order are not dependencies. A node is *product* work when the verifier depends on its output and *support* work otherwise.

Available parallelism. Let $w(n)$ be the duration of node n when a fresh worker executes it in isolation during replay. Total product work is $W = \sum_{n \in N_{\text{prod}}} w(n)$; the span S is the heaviest dependency path over product nodes; *product width* is the largest set of product nodes that can run at once, and *plan width* also counts support units. W/S is the idealized work–span parallelism of this decomposition (Brent, 1974; Blumofe & Leiserson, 1999): the speedup if every unit took its replay time and coordination were free. It describes one tested decomposition under single-sample durations, not an upper bound on what any system can achieve.

What the reference certifies. A MACE-Bench reference DAG is an executable witness of one feasible semantic decomposition of the task under declared inputs, outputs, interfaces and checks. Replay tests whether each unit executes under its information boundary, whether sibling outputs compose and pass the original verifier, and whether selected dependency and necessity claims survive intervention. Each claim carries one of four labels: validated, refuted, inconclusive or untested. The graph is not claimed to be the true task graph, nor unique, optimal or maximally parallel; another decomposition may expose more or less parallelism.

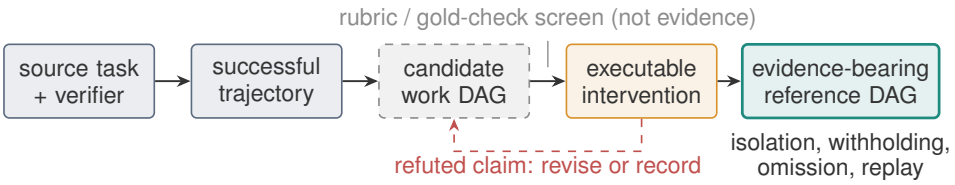


Figure 2: Construction. A candidate structure proposed from a successful trajectory has no standing until executable intervention tests it. Refuted claims are revised in bounded repair rounds or recorded in the released artifact. The rubric and gold-check probe only screen candidates; they are not evidence. Full eight-stage pipeline: Appendix A.

2.1 Construction and executable validation

A constructor model reads a successful trajectory of a frozen single-agent baseline and its verified solution and proposes a candidate DAG (Figure 2). A semantic rubric and a gold-check probe screen candidates for replay budget but are not evidence (Appendix B.4). The evidence comes from five intervention tests, each run by a fresh worker under the unchanged verifier:

(1) *Are units independently executable?* Isolation: each unit runs from the base state plus its ancestors' outputs and must pass its local check.
(2) *Do sibling outputs compose into a solution?* Recomposition: sibling deltas are merged mechanically and the union must pass V.
(3) *Is a claimed edge causal?* Withholding: the consumer reruns without its producer's output; if it still succeeds, the edge is refuted.
(4) *Is it a work dependency or only an interface?* Contract-first control: the consumer reruns with only the producer's interface.
(5) *Does the verifier need each unit?* Omission: the graph is recomposed without the unit.

Refuted claims enter at most two repair rounds before replay; infrastructure failures are labelled inconclusive, never counted as evidence (Appendix G).

Is the ruler reliable? For the structure the release relies on, yes: over the 104 released graphs, 411 unit isolation claims are validated (26 refuted) and 103 of 104 whole-graph recompositions pass the verifier. Dependency edges are the weak point (17 of 57 validated, most others inconclusive), which matters little here: 80 of 100 PARALLEL plans are depth-one fan-outs and 20 are deeper. Fan-out is the most basic multi-agent pattern, and model-initiated delegation still fails to exploit it, so the release tests parallel-work exposure, the simplest necessary test of coordination efficiency, not general dependency management (full counts and construction yield: Appendices B.3 and G, Table 19).

Are DAGs an artifact of one prompting recipe? Partly. The constructor comparison (Appendix N) finds different widths and W/S across strategies, so each reference is one witness among several. The pipeline also needs a solved seed trajectory and cannot cover tasks no agent has solved.

2.2 Selection, scope and release

MACE-Bench is a diagnostic benchmark, not a census: it selects long-horizon tasks with replay-supported parallel structure and says nothing about how common parallel work is. The final cut removed 7 graphs whose compiled plan run exposed a hidden dependency and 6 FrontierSWE tasks by a curation decision (Appendix G). The evaluation set was frozen after a first Codex pass, so it is not pre-registered. The evaluation set is 104 of the released tasks, from 4 upstream benchmarks, 100 PARALLEL (product width at least two, median 3, median W/S 1.9) and 4 width-one controls that serve as sanity checks and support no general claim (Appendix B.1).

3 PROTOCOL: FOUR ARMS THAT MOVE THE SPAWN DECISION

Giving a system the reference plan, or running the plan with a runtime, partially factors planning from orchestration (§1); the confounds below keep this comparison diagnostic, not causal.

Four arms. Each arm appends a fixed prompt to the same task instruction (verbatim in Appendix C.2); each step answers one question. Figure 3 shows who decides to spawn in each arm.

- **Single**, a *single agent*: one agent, no delegation tool, told to verify its implementation.
- **MAS**, an *autonomous multi-agent system*, asks whether model-initiated delegation pays. It adds the harness's native subagent tool (Codex `spawn_agent`, Claude Code `Agent`, Pi `delegate`); the parent decides whether, when and what to delegate to same-model workers with disjoint files, then integrates one verified solution.
- **MAS+DAG**, an *autonomous MAS with the DAG plan*, asks whether the bottleneck is planning. It adds the frozen plan text rendered from the reference DAG (units, dependencies, file scopes, contract facts), and its prompt strongly prefers concurrent launch of ready units while the parent keeps critical-path work and runs the checks. The parent still makes every spawn decision.
- **DAG-Sched**, a *deterministic DAG scheduler*, asks what changes when the runtime launches ready work. With MAS+DAG's plan, container and checkout but no parent model, a deterministic scheduler starts a fresh-context CLI process without a delegation tool for every unit whose dependencies have succeeded, immediately and uncapped. Units are told to stop when their own check passes; failures block descendants, nothing is retried, and the verifier scores the resulting tree. DAG-Sched is a runtime executing a solution-derived plan, neither a deployable self-planning system nor the compiled executor used in certification.

Systems. A system is one harness with one model: Codex × GPT-5.6 Sol (Codex CLI 0.156.1, reasoning effort high), Claude Code × Fable 5 and Claude Code × Opus 5.5 (Claude Code 2.1.280), and Pi × Kimi K3 (Pi 0.85.1, open-weights Kimi K3; Appendix D). All arms of a system share the model, flags, base image and limits; worker caps, host placement and integrity reruns are in Appendix G.3.

Timing. *Trial time* is the Harbor trial boundary: environment setup, agent execution and the verifier. *Total time*, the primary metric, is the model-active span from the first to the last model output over all agents; it excludes setup, the verifier and DAG-Sched's scheduler start-up, so it isolates agent execution. Appendix Table 21 reports trial time beside it. The *first spawn* is the first worker's first model call minus the parent's first model call; the *integration tail* is the parent's last model output minus the last worker's.

Outcomes and estimator. A cell passes when the verifier reward reaches the task's threshold; a cell with no verdict (one Codex MAS safety refusal and 3 Pi × Kimi K3 agent timeouts) counts as a fail. Every arm is scored on all 100 PARALLEL tasks and pass rates are compared by paired task bootstrap. A speedup of arm X over Single is the geometric mean of t_{Single}/t_X, and a cost ratio that of c_X/c_{Single}, both over task pairs in which *both* cells passed, so latency and cost are conditional on

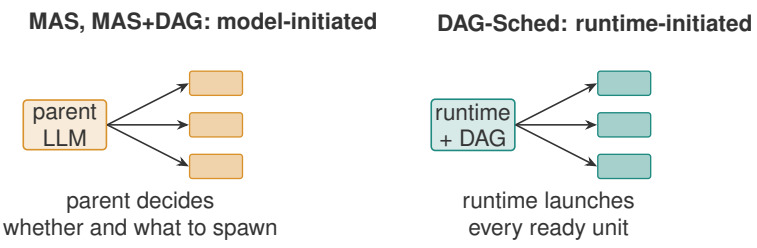


Figure 3: Who decides to spawn. In MAS and MAS+DAG the parent model decides whether, when and what to delegate, then integrates and verifies. In DAG-Sched the runtime launches every ready unit of the reference plan; no parent integrates; a composed check runs only where the plan has an integration unit, and units are told to check their own scope (§5.4).

Table 2: Explanations for a speedup other than parallel execution, and how this paper addresses each. "Not run" marks a control that is a limitation.

alternative explanation	what this paper does
more inference compute	cost reported beside every speedup; compute-matched control not run
independent parallel sampling (best-of-N)	one run per cell, no racing; racing control not run
solution information in the plan	first-edit proxy on non-delegating MAS+DAG cells (§4); Single with the plan not run
less verification in DAG-Sched	full-suite split of DAG-Sched cells (§4)
MAS+DAG vs DAG-Sched differ in parent work, context, process model, failure handling	not a pure scheduler ablation (§3)
host load; shallow graphs	same-host subset (Appendix L); deeper-plan subgroup (§4)

success. Intervals are 95% percentile bootstraps over tasks. Each system–task–arm cell has one run ($N = 1$), so the intervals describe task sampling, not run-to-run variation.

What the contrasts do not isolate. The plan is solution-derived, a partial oracle (localization, decomposition, file scopes, interface and sometimes test or toolchain facts), and MAS+DAG's prompt also changes; without a Single-plus-plan control we cannot split MAS+DAG's gain into information, wording and parallel execution. MAS+DAG → DAG-Sched also changes parent work, worker context, verification and failure handling, so it is not a pure scheduler ablation. Table 2 lists the remaining explanations and what this paper does about each.

4 Delegation chosen by the model does not buy speed; scheduled execution does

Table 1 gives headline results and Appendix Table 21 full statistics on the 100 PARALLEL tasks. Values are in the order Codex, Fable 5, Opus 5.5, Kimi K3 unless stated otherwise. Table 1 shows that the multi-agent arms rarely lose what Single solves: each keeps 90% to 96% of Single's solved tasks. They differ in what they add: MAS solves 4 to 8 tasks Single failed, DAG-Sched 10 to 17.

Does model-initiated delegation pay? Not detectably. Every MAS/Single total-time interval contains one: 1.00×, 0.97×, 0.95× and 0.95×. Nor did MAS fail to delegate: MAS spawns workers in 100%, 81%, 69% and 53% of cells, and on Codex in every cell. MAS costs 2.06×, 1.14×, 1.23× and 1.06× Single's cost (the Pi × Kimi K3 interval, [0.98, 1.14], includes one), and its pass rate does not differ detectably from Single's. We read this as no detected latency gain, not as equal latency.

Is the bottleneck planning? Partly, on three of four systems. With the plan, MAS+DAG is faster than Single on trial time on Codex × GPT-5.6 Sol, Claude Code × Fable 5 and Claude Code × Opus 5.5 (1.26×, 1.18×, 1.20×; every interval excludes one; cost 1.80×, 1.02×, 0.96×), but not on Pi × Kimi K3 (1.02× [0.93, 1.11]), although Pi × Kimi K3 delegates in 87% of MAS+DAG cells. The plan moves delegation earlier: on the 227 tasks where both arms delegated (three original systems), MAS+DAG spawned first in 92.5% of pairs. Part of the gain is plan information rather than parallelism: where Fable 5 and Opus 5.5 MAS+DAG did not delegate, a single agent holding the plan reached its first edit sooner than Single (median ratio 1.23 and 1.57; $n = 9$ and 40 self-selected tasks; first edit is a proxy).

What does runtime launch of ready work change? DAG-Sched is the fastest arm on every system, with every interval above one. Its total-time speedup over Single is 2.02×, 1.94×, 1.77× and 1.84× (trial time 1.73×, 1.78×, 1.58×, 1.78×; trial time adds setup and the verifier, which DAG-Sched does not shorten). Against MAS+DAG it is 1.38×, 1.47×, 1.30× and 1.72× faster on trial time, at 1.14×, 1.10×, 1.32× and 1.00× Single's cost. With Kimi K3 fixed, Pi and Claude Code reproduce the arm ordering (DAG-Sched/Single trial 1.78× and 1.54×); magnitudes differ by harness (Appendices D.1 and G.4).

What the DAG-Sched speedup is not. (i) *Not verification-neutral.* In the three original systems, no full test suite ran in 75%, 77% and 69% of DAG-Sched cells, against 39%, 43% and 45% of Single cells, partly by design (units stop at their own check). Splitting DAG-Sched cells by whether a full suite ran gives a larger DAG-Sched/Single speedup when it was skipped on Codex (2.09× against 1.80×) and Fable 5, but the split reverses on Opus 5.5; it is observational and confounded with task (Appendices F.1.5 and G.4). Skipped verification may inflate the speedup but does not explain most of it; the verifier still scores every cell (§5.4). (ii) *Not free of plan information*: the plan is solution-derived, and Single with the plan was not run, so this is not a pure ablation of scheduling. (iii) *Not selection-neutral*: the release excludes graphs whose compiled run exposed a hidden dependency (§2.2); the evaluation set was frozen after a first Codex pass; the plans come from a Fable 5 baseline; and Codex DAG-Sched uses 9 cells rerun after a provider quota failure (pre-rerun values in Appendix G.4). (iv) *Not only fan-out*: 80 of the 100 plans have no dependency, so DAG-Sched is mostly a parallel launch of solution-derived, file-scoped units; on the 20 deeper plans DAG-Sched/Single is still 1.51×, 1.49× and 1.50× on trial time (exploratory; both-pass n in Appendix G.4), every interval above one. (v) *Not only parallelism*: on the 4 width-one controls (no parallel work in the plan), DAG-Sched is still faster than Single on trial time on Codex (1.20×, $n = 3$) and Fable 5 (1.48×, $n = 4$), though not on Opus 5.5 (0.99×, $n = 4$), consistent with part of the gain not being parallelism; so few controls support no general claim.

Success conditioning. Speedups use only tasks both arms solved, while pass rates cover all tasks, so the result is not "twice as fast at the same accuracy". DAG-Sched passes more tasks than Single in point estimate on every system (paired difference +9.0, +12.0, +7.0 and +10.0 points); because DAG-Sched verifies less, the solution-derived plan is a likely contributor (Appendices E.2 and G.4).

Against the reference (exploratory). DAG-Sched's median ratio of speedup to the reference W/S is close to one, which parallel execution alone would not obviously achieve; this ratio, which is not a fraction of a bound, and a binning by plan width (Figure 21) are in Appendix G.4.

5 Where model-initiated runs lose time: the parent stays on the critical path

Why does MAS spawn real workers, spend more, and still finish no sooner than Single? We answer from traces of the three original systems (Codex × GPT-5.6 Sol, Claude Code × Fable 5, Claude Code × Opus 5.5) on the PARALLEL tasks; Pi × Kimi K3 received no mechanism forensics. The evidence has three layers: phase timings on every cell, labels from a post-hoc catalog (overlapping, not additive, and only observationally related to latency; Appendix F), and individual runs whose quoted lines were checked against the raw transcripts (Appendix F.5). 101 cells were also read by hand to check the catalog labels. By prevalence, the catalog flags an integration tail in 281 and retained critical work in 131 of 491 delegated MAS/MAS+DAG cells, and a composed tree that fails although every unit succeeded in 23 of 300 DAG-Sched cells. Owners, counterexamples and possible mitigations (none tested) for each mechanism are in Appendix G.5.

In a delegated model-initiated run, total time splits into the parent working alone before its first spawn, the worker window, and the parent working alone after its last worker (Figure 4a). The two parent-only phases take a mean 44%, 66% and 50% of total time in MAS and 40%, 57% and 51% in MAS+DAG. This is a partition, not a counterfactual saving: parent-only time includes real implementation work. DAG-Sched keeps at least two agents busy for a median 0.68, 0.67 and 0.74 of total time, against 0.46, 0.25 and 0.41 in delegated MAS cells (Appendix F.1.2). On tasks where MAS+DAG delegated and both arms passed, MAS+DAG's worker window alone is 0.89×, 0.63× and 0.67× of DAG-Sched's whole run (Figure 4b): on Fable 5 and Opus 5.5 most of DAG-Sched's lead over MAS+DAG is the parent-only phases, while on Codex the worker window alone is close to DAG-Sched's whole run.

5.1 Core mechanism 1: serial parent time before the first spawn

The first worker starts late: in MAS the median first spawn falls at 0.10, 0.44 and 0.22 of total time (Codex, Fable 5, Opus 5.5). The MAS prompt requests an upfront decomposition, so part of this prefix is prompt-induced. Spawning is not slow: over 1412 spawns (1437 with controls) the median

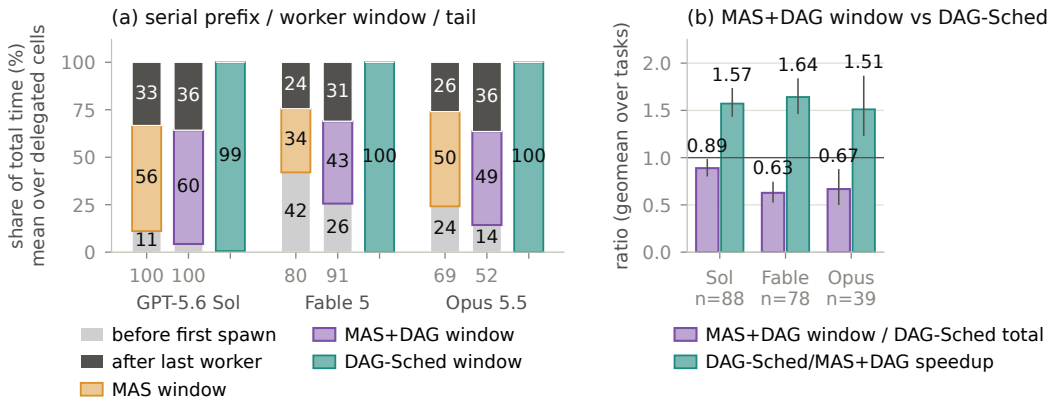


Figure 4: Phase decomposition of total time, three original systems. (a) Mean share before the first spawn, in the worker window, and after the last worker, over delegated MAS and MAS+DAG cells, with DAG-Sched for reference (no parent). A partition by construction, not a counterfactual saving: parent-only time includes real implementation work. (b) On tasks where MAS+DAG delegated and MAS+DAG and DAG-Sched both passed: MAS+DAG's worker window over DAG-Sched's whole run, and DAG-Sched/MAS+DAG total-time speedup. In (a), numbers under the bars are delegated cells.

dispatch takes 0.019, 0.003 and 0.004 s, and the parent's own model time is 77% to 94% of prefix seconds. In Fable 5 MAS on `go-git` (Appendix Figure 22a), the parent reads the repository, writes and builds a placeholder merge engine and its call site, and only then launches two workers, at 66% of the run. The parent thus uses part of the parallel window before it asks for workers. With the plan and MAS+DAG's wording it spawns earlier in 92.5% of the 227 paired tasks, a comparison confounded with plan information.

5.2 Core mechanism 2: the parent-owned verification tail

Both multi-agent prompts assign the phase after the last worker to the parent, so the tail exists by design; its length varies. The catalog flags an integration tail in 281 of 491 delegated MAS/MAS+DAG cells; in 36 randomly sampled tails it caught a real defect in 7 and rescued none of the 5 failing cells. On Fable 5 `dulwich` (Appendix Figure 22b) both arms pass and MAS+DAG's workers finish first, but MAS+DAG's parent runs the full suite twice and checks failures against the base tree (276 s) where MAS's runs the touched modules (48 s), so MAS+DAG ends at 481 s against 368 s. In Claude Code a timer can add dead time: in Opus 5.5 MAS on `koota` the parent starts `sleep 240` after the last completion is queued (272 s tail, 0.66× Single's speed); such overhang appears in 54 delegated Claude Code cells and in no Codex cell. The tail's length is a choice of verification scope and of waiting, and tail share is only weakly related to speedup.

5.3 Core mechanism 3: the parent retains critical work

Workers can exist while the parent's own lane holds the completion-critical product change. The catalog flags this in 131 of 491 delegated MAS/MAS+DAG cells, rarely in Codex MAS+DAG (3 of 100). In Opus 5.5 MAS on `go-genai`, the parent writes the accumulator and then starts three workers that each write a test file; the run reaches 0.64× Single's speed at 2.55× its cost. The traces state no motive, and we attribute none. A high worker count therefore does not imply critical-path parallelism, and headcount predicts speedup only weakly.

5.4 Core mechanism 4: what the runtime scheduler gives up

DAG-Sched removes both serial parent phases, consistent with its being fastest, but also warm context, adaptive reassignment, the parent's composed global check (only plans with an integration unit keep one) and any repair loop. In 23 of 300 DAG-Sched cells every unit reported success and the composed tree still failed. Of 26 audited DAG-Sched failures (one Opus 5.5 cell more than in Appendix Table 21), 15 are one unit missing an edge case in its own scope and only 1 is a cross-unit

interface disagreement. In Codex DAG-Sched on `textual-kitty`, the parser unit reports passing tests and the verifier fails on a key sequence it did not cover; the MAS+DAG parent on the same task runs an end-to-end check and patches the parser. A composed check is not a full remedy: MAS+DAG passes only 5 of the 15 tasks. The benchmark's DAG-Sched latency thus prices the removal of orchestration overhead against this lost checking. Appendix Table 22 summarizes the four mechanisms and lists five secondary mechanisms, described in Appendix G.5.2, with good-run controls in Appendix G.5.1.

6 Related work

Benchmarks score final state. SWE-bench (Jimenez et al., 2024), SWE-Bench Pro (Deng et al., 2025), SWE-rebench (Badertdinov et al., 2025; 2026), DeepSWE (Huang et al., 2026) and Terminal-Bench (Merrill et al., 2026) grade a repository's final state with tests. A pass shows that a task was solved, not what parallel work it contained or whether a system used it. MACE-Bench keeps their tasks and verifiers and adds a tested per-task reference for that work.

Task structure is given, not tested. Multi-agent benchmarks grade collaboration through designer-specified milestones or dependencies (Zhu et al., 2025; Geng & Chang, 2025); frameworks and DAG runtimes decompose work with their own planner (Hong et al., 2024; Qian et al., 2024; Wu et al., 2024; Kim et al., 2024; Yu et al., 2025); concurrent orchestration benchmarks score designer-built task structures (Ren et al., 2026; Xiong et al., 2026) and concurrent systems report their own parallel gains (Koh et al., 2026; Mao & Mirhoseini, 2026; Yang et al., 2026; Xu et al., 2026) (Appendix J). In each case the structure comes from the system under test or the benchmark designer and is not tested by intervention against the original verifier, so realized parallelism has no reference to be read against. Controlled comparisons and failure studies share this gap (Smit et al., 2024; Wang et al., 2024; Kim et al., 2025; Zhang et al., 2025a; Cemri et al., 2025; Zhang et al., 2025b); our forensics measure each loss against what the task permitted. W, S and W/S follow the work–span model (Brent, 1974; Blumofe & Leiserson, 1999), here measured for worker-sized units whose dependencies were tested by replay.

7 Limitations and conclusion

Limitations and future directions. MACE-Bench currently instantiates its construction protocol on repository-level coding tasks. The protocol needs only work units that can be run and composed against an executable success criterion, and we hope to apply it to other domains. Each reference is one feasible decomposition recovered from one successful solution, and most are fan-outs; richer dependency structure and alternative decompositions are future work. The scheduler arm executes a solution-derived plan, so it shows what ready-work execution achieves given a good plan, not a system that plans for itself. Appendix H discusses these points in full.

Conclusion. Whether a multi-agent system finishes sooner depends on work the task makes available and on what the system does with it, and final-state benchmarks see neither. MACE-Bench makes that work executable and observable. Across the systems we test, model-controlled delegation without a plan spends more with no detected latency gain, while runtime execution of the same ready units is the fastest arm on every system; on the three systems traced, the traces place the difference in when and what the parent delegates, not in launching workers. We hope MACE-Bench helps future multi-agent systems turn spawned workers into realized parallelism.

AI use statement

Generative AI tools are part of the object of study: the evaluated systems are LLM coding agents, and LLMs proposed and repaired candidate work DAGs under the replay protocol of §2.1, whose claims were then tested against the tasks' unchanged verifiers. We also used generative AI tools as coding assistants for analysis scripts, figure generation and trace extraction, and for drafting and editing parts of the text. Every statistic in the paper is generated from committed artifacts by scripts, and quoted trace excerpts were checked mechanically against the raw transcripts. We reviewed all

AI-assisted work and take responsibility for the final content of this paper, including text, claims and artifacts produced with the aid of generative AI.

Reproducibility statement

The release contains, for each of the 104 tasks, the reference DAG with every claim's replay label, the measured W, S and width, the rendered plan text and the scheduler used for the DAG-Sched arm. The appendix gives the construction and validation protocol (Appendices A, B.2 and B.3), the release composition and upstream provenance (Appendix B.1), the evaluation protocol with verbatim prompts, rerun rules and integrity conditions (Appendices C and C.2), additional results and sensitivity analyses (Appendix E), and the full post-hoc trace catalog (Appendix F). Every number in the paper is a macro generated from the released task records, the per-cell evaluation table and the trace catalog (Appendix S).

References

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 483–485, 1967. doi: 10.1145/1465482.1465560.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. arXiv:2505.20411.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.23866.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999. doi: 10.1145/324133.324234.

Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21 (2):201–206, 1974. doi: 10.1145/321812.321815.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems 38 (NeurIPS), Datasets and Benchmarks Track*, 2025. arXiv:2503.13657.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more LLM calls all you need? Towards the scaling properties of compound AI systems. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024a. arXiv:2403.02419.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations (ICLR)*, 2024b. arXiv:2308.10848.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 9780262033848. Chapter 27, Multithreaded Algorithms: work, span, parallelism.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989. doi: 10.1109/12.21127.

Longling Geng and Edward Y. Chang. REALM-Bench: A benchmark for evaluating multi-agent systems on real-world, dynamic planning and scheduling tasks. *arXiv preprint arXiv:2502.18836*, 2025.

Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. doi: 10.1137/0117039.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2308.00352.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Yucheng Jiang, Zora Zhiruo Wang, Ruishi Chen, and Diyi Yang. Inducing task models from computer-use traces. *arXiv preprint arXiv:2608.20319*, 2026.

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? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Transactions on Machine Learning Research*, 2025. arXiv:2407.01502.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024. arXiv:2312.04511.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025.

Jing Yu Koh, Ruslan Salakhutdinov, and Daniel Fried. Multi-agent computer use. *arXiv preprint arXiv:2606.01533*, 2026.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, Lichao Sun, Haotao Mi, and Leowei Liang. Long-Horizon-Terminal-Bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2308.03688.

Qiuyang Mang, Wenhao Chai, Zhifei Li, Huanzhi Mao, Shang Zhou, Alexander Du, Hanchen Li, Shu Liu, Edwin Chen, Yichuan Wang, Xieting Chu, Zerui Cheng, Yuan Xu, Tian Xia, Zirui Wang, Tianneng Shi, Jianzhu Yao, Yilong Zhao, Qizheng Zhang, Charlie Ruan, Zeyu Shen, Kaiyuan Liu, Runyuan He, Dong Xing, Zerui Li, Zirong Zeng, Yige Jiang, Lufeng Cheng, Ziyi Zhao, Youran Sun, Wesley Zheng, Meiyuwang Zhang, Ruyi Ji, Xuechang Tu, Zihan Zheng, Zexing Chen, Kangyang Zhou, Zhaozi Wang, Jingbang Chen, Aleksandra Korolova, Peter Henderson, Pramod Viswanath, Vijay Ganesh, Saining Xie, Zhuang Liu, Dawn Song, Sewon Min, Ion Stoica, Joseph E. Gonzalez, Jingbo Shang, and Alvin Cheung. FrontierCS: Evolving challenges for evolving intelligence. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2512.15699.

Yuzhen Mao and Azalia Mirhoseini. Decentralized multi-agent systems with shared context. *arXiv preprint arXiv:2606.10662*, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2601.11868.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. *arXiv preprint arXiv:2311.12983*, 2023.

OpenAI. Introducing SWE-bench Verified. `https://openai.com/index/introducing-swe-bench-verified/`, 2024. Blog post, 13 August 2024; 500 human-validated SWE-bench tasks.

Proximal Labs. FrontierSWE: an ultra long-horizon coding agent benchmark. `https://github.com/Proximal-Labs/frontier-swe`, 2026. No accompanying paper; task packages pinned by commit hash in the release provenance.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024. arXiv:2307.07924.

Zhenzhen Ren, Jiyan He, Xinpeng Zhang, Zhenxing Qian, Ke Han, Shuxin Zheng, GuoBiao Li, and Xiaoqing Zhang. OrchBench: Evaluating multi-agent orchestration plans in isolation via deterministic simulation. *arXiv preprint arXiv:2607.25656*, 2026.

Andries P. Smit, Nathan Grinsztajn, Paul Duckworth, Thomas D. Barrett, and Arnu Pretorius. Should we be going MAD? A look at multi-agent debate strategies for LLMs. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, pp. 45883–45905, 2024.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024. arXiv:2403.17927.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. Rethinking the bounds of LLM reasoning: Are multi-agent discussions the key? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6106–6131, 2024. doi: 10.18653/v1/2024.acl-long.331.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2407.16741; 24 authors.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *First Conference on Language Modeling (COLM)*, 2024. arXiv:2308.08155.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering (FSE 2025)*, 2025. doi: 10.1145/3715754. arXiv:2407.01489 ("Agentless").

Kaiwen Xiong, Haonian Ji, Shi Qiu, Zeyu Zheng, Cihang Xie, Xinyu Ye, and Huaxiu Yao. ClawArena-Team: Benchmarking subagent orchestration and dynamic workflows in language-model agents. *arXiv preprint arXiv:2606.31174*, 2026.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.

Zihan Xu, Haolin Tian, and Hai Jiang. A two-tier perspective on inference-time parallelism in multi-agent LLM systems. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2608.05791.

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. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024. arXiv:2405.15793.

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. *arXiv preprint arXiv:2606.00953*, 2026.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.12045.

Junwei Yu, Yepeng Ding, and Hiroyuki Sato. DynTaskMAS: A dynamic task graph-driven framework for asynchronous and parallel LLM-based multi-agent systems. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS)*, volume 35, pp. 288–296, 2025. doi: 10.1609/icaps.v35i1.36130. Vol. 35, No. 1.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *International Conference on Learning Representations (ICLR)*, 2025a. arXiv:2410.02506.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025b. arXiv:2505.00212.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You. MultiAgentBench: Evaluating the collaboration and competition of LLM agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8580–8622, 2025. doi: 10.18653/v1/2025.acl-long.421. arXiv:2503.01935.

A Benchmark definitions, construction and validation

This appendix gives the full definitions behind §2: the measured quantities, the construction and validation pipeline, the compiled execution plan and executor, and what the reference shows about the released tasks.

A.1 Measuring executable parallelism

We call parallelism *executable* when it is defined over units of work that isolated workers can perform from a shared starting state and whose outputs compose under the original verifier; the quantities below are all of that kind. Figure 5 shows how the pieces fit: a task and its verifier, the reference recovered from a successful trajectory, and the four evaluation arms that use it.

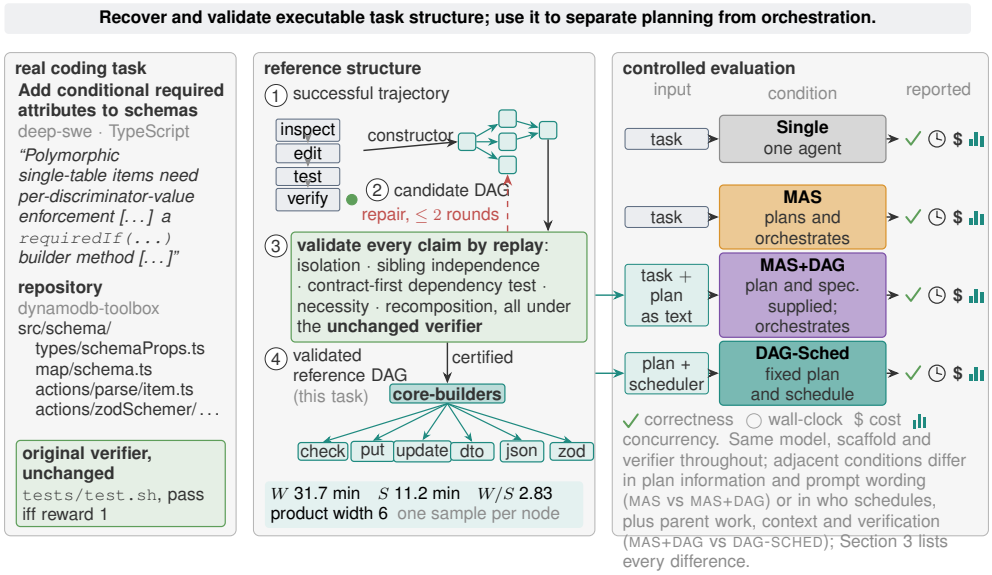


Figure 5: Overview (schematic). Left: a real repository task from an existing benchmark, with its verifier used unchanged. Center: from a successful trajectory a constructor proposes a semantic work DAG; its claims are tested by replay against the unchanged verifier where the replay reaches them and repaired if refuted (coverage in §A.2); the reference gives W, S, W/S and width (shown: a released task, one isolated-replay sample per node). Right: the four evaluation arms on one harness and model, each reporting success, time, cost and concurrency: Single (a single agent), MAS (an autonomous multi-agent system), MAS+DAG (the same system given the plan as text) and DAG-Sched (a deterministic scheduler on MAS+DAG's substrate) (§3). Recover and replay-test executable task structure, then use it to separate what the task offers from what a system realizes.

Tasks, trajectories, work units. A *task* is a frozen software environment, an instruction, and a hermetic verifier V from an existing single-agent benchmark, used unchanged. A *trajectory* is the tool-call record of an agent that solved it; its final workspace delta is the *gold* solution. A *semantic work unit* is a worker-sized objective that a fresh agent can perform given the task, the base state, its declared inputs and shared *contract facts*, the interfaces siblings agree on; one unit may change many files, and two independent units may change one file. A *semantic work DAG* $D = (N, E)$ has such units as nodes, each with an objective, inputs, outputs, expected paths and a runnable local check. An edge $u \to v$ is a *work dependency*: v cannot be performed without state that u produces. Nodes with no directed path between them are *siblings*, asserted independent from the same start.

What is not a dependency. Chronological order in the trajectory is evidence about the agent, not the work. A shared file is not a dependency. Verification order is not work order: an adapter written against a published interface does not depend on the implementation behind it. Conversely, a gold solution that is one dense unit is genuinely serial, and sharding it by file manufactures width, not parallelism.

Product and support work. A node is *product* work when the verifier depends on its output and *support* work otherwise (tests, harnesses, tooling). Headline quantities are computed over product nodes, because support width is concurrency the verifier cannot see.

Work, span, width. Let $w(n)$ be the measured duration of node n executed in isolation by a fresh worker. *Total work* is $W = \sum_{n \in N_{\text{prod}}} w(n)$; the *span* S is the weight of the heaviest dependency path over product nodes; *product width* is the largest number of product nodes that can be in flight at once (the largest antichain of product nodes; in a graph of depth one, the product-node count). *Plan width* counts every unit of the plan, support units included, in the same way; it is the number of leaves the DAG-Sched runtime of §3 can run at once, and we use it only for the evaluation subgroups of §4, never as a headline measure of the task. Durations are single samples from the validation replay and exclude scheduling, composition and verification.

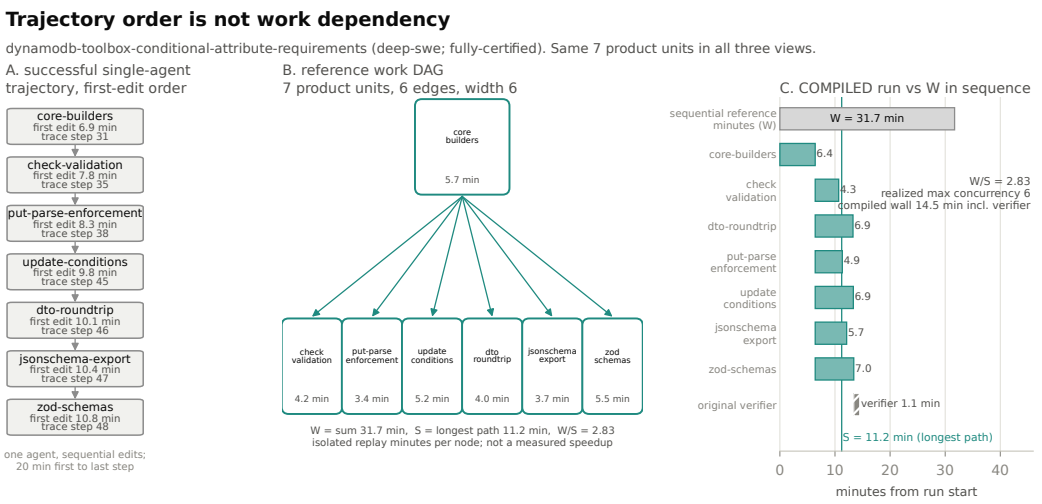


Figure 6: Trajectory order is not work dependency: one released task (`dynamodb-cond.`, DeepSWE, fully certified). (A) The successful single-agent trajectory touched its seven product units in one sequence. (B) The reference work DAG places six of them as siblings behind one producer; every edge survived the contract-first control. Single isolated-replay durations give $W = 31.7$ min, $S = 11.2$ min, $W/S = 2.83$, width 6. (C) The compiled reference run launched all six consumers when the producer finished (realized maximum concurrency 6) and passed the verifier in 14.5 min against the sequential reference minutes W. Reading dependencies off the trace would serialize work the tested graph places on separate branches.

Available parallelism. Two measures of the task, never interchanged, plus plan width for the evaluation. Product width is *instantaneous structural concurrency*: how many workers the graph can occupy at its widest point, whatever the branch lengths. Work over span, W/S, is the *idealized parallelism of this decomposition*: the speedup over serial execution that a scheduler would realize if every worker took exactly its measured replay time with zero coordination overhead. It is not a ceiling on a system's speedup over a single agent: W sums fresh-worker replays that a single agent does not match task by task, so a run of the same decomposition can exceed W/S (§E.1.3). Width is cheap to manufacture and silent about branch substance; W/S rewards balanced, substantive branches and is the natural reference for a speedup, although in the final evaluation its correlation with DAG-Sched's speedup is weaker than plan width's on two of three systems (§E.1.4). W, S and W/S are the work, span and parallelism of the multithreaded-computation model (Brent, 1974; Blumofe & Leiserson, 1999; Cormen et al., 2009), also called average parallelism (Eager et al., 1989); greedy list scheduling of a dependency graph is within a factor of two of optimal in general (Graham, 1969), and with as many worker slots as the graph can run at once, which the uncapped DAG-Sched runtime of §3 provides and the compiled executor provides for graphs of depth one (its slot count is the widest level, which equals the antichain width there; the audit found realized concurrency equal to the declared level width, §A.3), no ready node waits, so we read the compiled executor's distance from S as substrate overhead rather than scheduling loss; for DAG-Sched, whose leaves are fresh agents in one shared container, that distance also contains worker variation and contention. Both describe one decomposition under single-sample durations; neither bounds the task, which admits several valid decompositions.

Realized concurrency and realized speedup. For a system run, *realized concurrency* is what the system executed at once (workers started, and the share of time with at least two agents busy, parent included); *realized speedup* is the single agent's time over the system's time on the same task, under one stated timing definition (§C.1.3), aggregated as a geometric mean over tasks both solved, with success rates beside it and never folded in; realized speedup over W/S is the *converted fraction* of the available parallelism. *Horizon* is total product work W, with the single-agent wall as a secondary measure where it was run; neither is a claim that a task is long.

Figure 6 shows one released graph, the trajectory it was recovered from, and its compiled run.

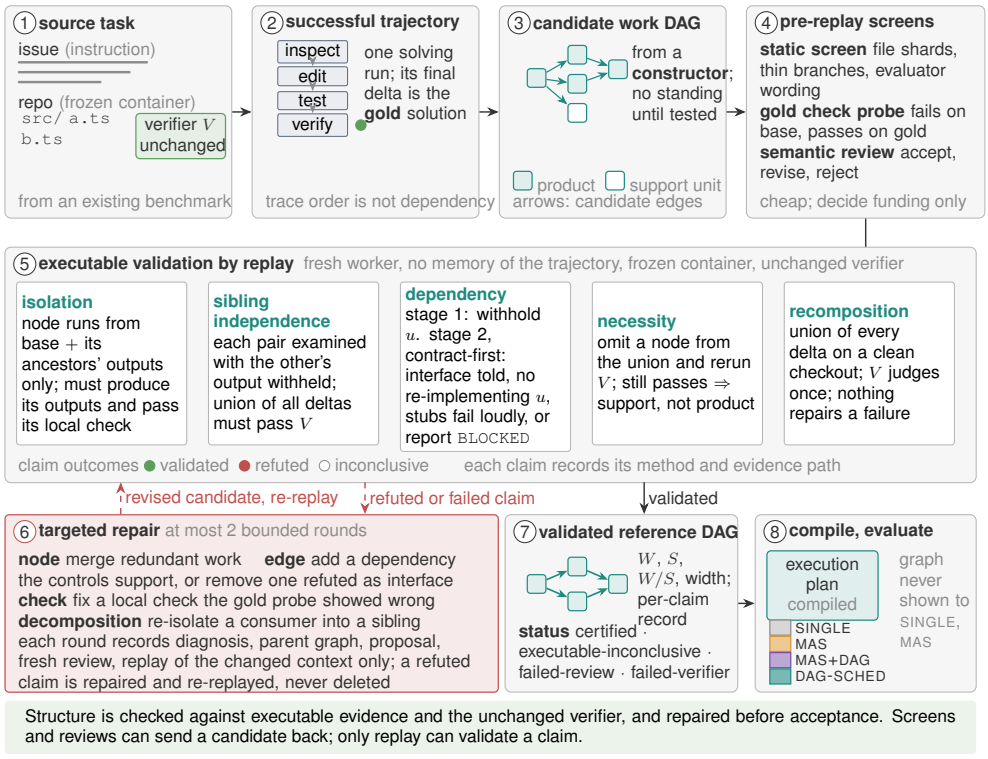


Figure 7: Construction, validation and repair, as implemented. Stages 1–4 produce and screen a candidate; screens decide funding only. Stage 5 tests every structural claim by replay with fresh workers in the frozen container under the unchanged verifier; a refuted or failed claim goes to at most two bounded repair rounds (6) and is re-replayed; the tested reference (7) is released with its per-claim record and status and compiled for evaluation (8). Structure is checked against executable evidence and repaired before acceptance; a plausible graph has no standing until it survives replay.

A.2 Building MACE-Bench: recovering and validating work DAGs

For each task we obtain a successful trajectory, propose a candidate work DAG from it, test every structural claim by intervention, repair what the tests refute within a bounded budget, and release the graph with its evidence. Evaluated systems see only the original task unless an arm is explicitly given the graph. Figure 7 shows the pipeline as implemented; this section walks through it and ends with the question a reader will ask first, whether the graphs are an artifact of one way of proposing them.

A.2.1 Sources, baselines and candidates

The public release draws its 104 tasks from exactly 4 upstream benchmarks: DeepSWE, SWE-rebench V2, SWE-Bench Pro and Terminal-Bench 2 (Table 7, Appendix B.1). The historical candidate corpus that construction processed was broader. It drew on 7 existing benchmarks, all software tasks with hermetic verifiers and GPU-free containers, used with their verifiers unchanged: DeepSWE (Huang et al., 2026), original repository feature tasks with hand-written verifiers; SWE-rebench (Badertdinov et al., 2025; 2026), monthly batches of repository tasks, run here under network-denied egress because live upstream fetching was the only observed leakage channel; the public slice of SWE-Bench Pro (Deng et al., 2025); Terminal-Bench 2.0 (Merrill et al., 2026); FrontierCS (Mang et al., 2026), competitive-programming tasks with graded verifiers; FrontierSWE (Proximal Labs, 2026), very long engineering tasks with graded verifiers; and Long-Horizon-Terminal-Bench (Li et al., 2026). Several other sources were scouted and excluded on evidence rather than taste: search tasks whose parallel branches were substitutes rather than complementary work (removing one did not change the outcome), tool-use tasks whose policy gates force admissible width to one, question suites whose omission controls passed from model memory or collapsed to

Table 3: Historical curated pre-release pool by source, pool and legacy analytic tier ($n = 122$ tasks; PARALLEL 112, CONTROL 10). This pool is not the public benchmark: the public release has 104 tasks from 4 sources (Table 7). Source labels in this historical table predate the correction that moved nine Ansible tasks from SWE-rebench to SWE-Bench Pro. CORE and EXTENDED partition the PARALLEL tasks that carry a legacy tier (Appendix P); the 6 manual-review admissions carry none. The PARALLEL pool is almost entirely repository feature tasks from three software-engineering sources; the longest-horizon sources contribute mainly controls.

Source	PARALLEL	CONTROL	Total	CORE	EXTENDED
DeepSWE	45	2	47	20	25
SWE-rebench	46	1	47	2	43
SWE-Bench Pro	15	1	16	6	9
FrontierSWE	5	1	6	0	0
Terminal-Bench 2	1	2	3	1	0
FrontierCS	0	2	2	0	0
LHTB	0	1	1	0	0
Total	112	10	122	29	77

width one once verification work was removed, and very long sources whose verifiers could not be run reproducibly (Appendix P). Table 3 gives the composition of the historical curated pre-release pool; it is superseded by the public release.

A frozen single-agent baseline runs on each candidate task (Harbor 0.22.0, Claude Code 2.1.247, Fable 5, the same model as one of the evaluated systems of §3); any verified success is eligible, and a broken verifier is the only hard exclusion. The successful trajectory is cleaned before construction: dead ends, failed hypotheses and debugging that did not contribute are dropped, low-level actions are grouped, and every original step is recorded as kept or dropped with a reason, so cleaning selects evidence and never rewrites it. Very long trajectories pass through a hierarchical semantic abstraction into episodes. A *constructor* then reads the task, the cleaned trajectory, a mechanical inventory of the artifacts the trajectory created and consumed, and the gold solution, and proposes a candidate graph. Construction is separated from validation by design: any method may propose a candidate, and nothing about how it was produced gives it standing. Five strategies share one target definition. *Direct* makes one pass over all the evidence with no prescribed analysis. *Forward* drafts requirements, work units and information flows from the task alone and then reconciles the draft against the evidence. *Backward* drafts the responsibilities the verified solution delivers and attributes them to the trajectory. *Hierarchical* groups trajectory episodes into behavioural responsibilities under an explicit granularity policy (a node is a coherent assignment with its own behaviour, sized to minutes not seconds; never cut a file by hunk; consumed-only declarations belong to the node that needs them) and then infers artifact dependencies. *Task-model induction* (TMI), adapted from Jiang et al. (2026), induces a procedural task model from trace activities and contracts its phases into responsibilities, recording observed sequence separately from declared dependency; an earlier version asserted that each phase depends on the one before, which made width one by construction and was withdrawn as circular. The production constructor is the hierarchical policy with direct as the fall-through when a screen rejects; §A.2.4 compares the strategies.

Pre-replay screens. Screens decide what is worth funding and establish nothing about a graph. A static screen rejects file-shard nodes, thin branches and evaluator wording in worker-facing text. A gold check probe runs every node's local check against the base tree and against the gold solution and gates out checks that pass on base (vacuous), fail on gold (invented internals) or need a file the node itself must write. A model review under a fixed protocol asks separately whether each producer is needed to *author* its consumer and whether it is needed to *verify* it. A frozen six-question rubric answers pass, fail or uncertain on grounding, semantic work units, plausible independence, plausible dependencies, granularity and omitted parallelism; calibrated blind on 29 stored candidates it produced no false rejects and five action-level false accepts, and its dependency question was wrong on seven of twelve edges it passed with a known outcome, which is why it is a sanity check and not a verifier.

A.2.2 VALIDATION BY INTERVENTION

A structural claim is `validated` only by an experiment that could have refuted it: a fresh worker with no memory of the trajectory executing in the frozen container, judged by the unchanged verifier or by the node's own local check. Every claim carries one of four outcomes, `validated`, `refuted`, `inconclusive` or `untested`, with the method that produced it and the evidence path, so a verdict can be re-derived offline from the released record. Five controls produce those outcomes.

Isolation. Each node is executed by a fresh worker that starts from the base commit with its ancestors' outputs pre-applied and nothing else, receives the objective, the contract facts and its declared inputs, and must produce its declared outputs and pass its local check. The captured output is the worker's own delta, and its duration is the single sample that W and S are built from.

Sibling independence. For each pair of nodes with no directed path, both isolation runs are examined with the other's output withheld, and the mechanical union of every node's delta must apply and pass V. Under the frozen rule a pair that touched a common file stays `inconclusive` even when the union composed cleanly and passed; a zero-spend audit found this the one repeated conservative rejection in the corpus (9 of 12 rejected graphs would be included by a human reviewer), and the relaxation was specified but not applied for this release.

Dependency, in two stages. A worker denied an upstream artifact usually rebuilds it, so a plain withholding test cannot establish an edge. In stage one the consumer v runs with the producer u's output withheld: a failure with no stand-in, where review names the produced state v needed, validates the edge; a pass with an independently verified alternative refutes it; and a run in which v *reconstructs* u's work (defines top-level symbols u defines, repeats a substantial fraction of u's added lines on a shared file, or writes real content into files reserved to u) is inconclusive and proceeds. In stage two, the *contract-first* control, v runs again under a stricter instruction: u's output is withheld, v is told u's published interface, it must not re-implement u, it may scaffold u's interface only with stubs that fail loudly with a fixed marker, and it must write a `BLOCKED` report if its work is impossible without u's produced state. The outcome is read from what the worker did. If v authored its implementation in full against the interface and its check failed only by executing a marked scaffold, the edge is *refuted as an interface dependency*: the work was authorable, and only verification needed u. If v wrote `BLOCKED`, or failed with no stand-in and review names the state it needed, the edge is *validated* as a work dependency. If v reconstructed u again, used a silent stand-in or failed for an unattributed reason, the edge is inconclusive with the reason recorded. Type declarations count as interface; a stub module whose header carries the marker is a whole-file scaffold; and a check failure that names a scaffold-only file is attributed to scaffolding even without a marker in the output, which matters for compile-time failures in typed languages.

Necessity by omission. A node's contribution is omitted from the union and the verifier rerun; a node whose omission leaves V passing has refuted product necessity and is support, not product, work. A failed patch application is never read as evidence of necessity.

Recomposition. The union of every node's own delta is applied to a clean checkout in topological-level order and judged once by V; a sibling-path overlap with conflicting hunks, or two creations of one path, refutes composition before the verifier runs, and nothing repairs a failure.

Interface dependencies are not work dependencies. The contract-first control is what lets the graph encode work dependencies only. In the 100 PARALLEL graphs of the public release, every one of the 57 edges had a dependency test attempted and 17 were validated (claim-type counts in Appendix B.3). The remainder of this paragraph describes the historical 112-task PARALLEL pool of the pre-release corpus, in which 61 of 135 edges were tested; of the 31 with a conclusive outcome, 20 are validated work dependencies and 11 were refuted as interface dependencies (30 inconclusive, 74 untested). An edge whose artifact-consumption claim is refuted is retained only when a separate work-contract review, a model review rather than an experiment, supports a work dependency; under that operative layer, counted over the 106 edges of the 110 PARALLEL graphs whose product units carry trace steps (§A.4.1), the pool has 72 review- or replay-supported, 10 inconclusive, 0 refuted and 24 untested edges, and the 11 refuted-claim edges are marked as such. Most refuted edges never reach a released graph, so the released fan-out shape is partly a property of this control; we claim only that about a third of the conclusively tested edge claims were interface dependencies,

which makes W/S a statement about authoring rather than about checking. The control was itself calibrated before production on a pilot in which 8 of 9 stage-two edges were refuted as interface and 1 was left inconclusive, while the pilot's one genuine produced-state dependency, on a serial control task, validated in a fresh stage-one control (Appendix Q).

A.2.3 REPAIR, STATUS AND ADMISSION

Bounded revision. After replay, at most two model-proposed revision rounds may repair a candidate: merge redundant work, fix a check the gold probe shows to be wrong, add a dependency the controls support, or re-isolate a consumer whose edge was refuted as interface into a sibling with a fake-based local check and a composition-time real check. Each round records its diagnosis, the parent graph, the proposal, a fresh review and the replay; pre-revision artifacts remain intact, and a rejected revision never replaces the last replay-bound deliverable. Stand-in files a sibling writes for an interface are recognized mechanically and stripped from the union, so they never become width. Unchanged nodes reuse evidence only when task bytes, image, model and scaffold, objective, check, predecessor contracts and patch hashes all match. A refuted claim is repaired and re-replayed; it is never deleted from a certified artifact.

Status. The graph's status is derived by one function and by nothing else: `certified` when recomposition and every required claim validate; `executable-inconclusive` when recomposition passes V but evidence is incomplete; `failed-review` when a required claim is refuted; `failed-verifier` when the recomposed union fails V; `unreplayed` when no execution evidence exists. Screens and reviews can send a candidate back to construction; they cannot certify, widen, narrow or skip a control.

Admission and the three per-task fields. A task's *pool* follows from structure alone: product width at least two is PARALLEL, width one is CONTROL. No single-agent or multi-agent evaluation result was used to select tasks, but every task needed one successful source trajectory from the frozen single-agent baseline (Fable 5 under Claude Code, §C.1.1), and the evaluation set of §3 further excludes the 7 graphs whose compiled reference run exposed a hidden dependency, a rule fixed before the final runs that favours the DAG-Sched arm, which executes the same units. Every released task then carries three independent fields that this paper keeps apart throughout: the pool (the admission decision), the *certification level* (the replay controls' verdict on the graph's claims), and the *compiled-execution status* (the outcome of one strict deterministic run where one exists, §A.3). Certification level describes evidence strength, not validity. In the public release, 99 of the 100 PARALLEL graphs are certified by replay at least partially (Appendix B.3). In the historical pre-release pool, 38 PARALLEL graphs were fully certified (recomposition and every required claim validated), 68 partially certified (recomposition passes V but at least one claim is inconclusive or refuted, with the record saying which; 22 of them carry a refuted local-check claim, where a node's own check failed in isolation while the union passed, so the check rather than the work is in question), and 6 carry a manual-review record. In all, 106 historical PARALLEL graphs (95%) were certified by replay at least partially. The manual-review admissions are candidates whose pipeline run was cut off before replay or failed for an infrastructure reason; they were admitted after a zero-spend static review only if the gold solution passes V, the reviewed graph has at least two substantive product branches with checks that passed the gold probe, every manual edit is recorded with its evidence, and a static consistency check passes. They carry `verifier_passed: gold-only` and the exact automated failure that prevented certification; manual review is not certification, and the trace-order analysis of §A.4.1 reports them separately. Of 345 tasks attempted under the frozen methodology, 278 produced a construction deliverable, 235 a verifier-passing recomposed union, 138 were fully certified, and 122 entered the historical curated pool after a curation that removed short-work and width-only graphs; the 10 controls were chosen from 37 certified width-one candidates by a deterministic rule covering every source, both serial and support-parallel structure and a spread of horizon, and the remaining 27 are archived with their runs intact (Appendix P). Replay over the historical pool's graphs cost $2,603 (median $15.7 per task). The cut from this pool to the 104-task public release (hidden-dependency exclusions, FrontierSWE removal and a reduced control set) is described in Appendix B.2.

Figure 8 shows one released item with every field a released record carries; Appendix M gives the full schema.

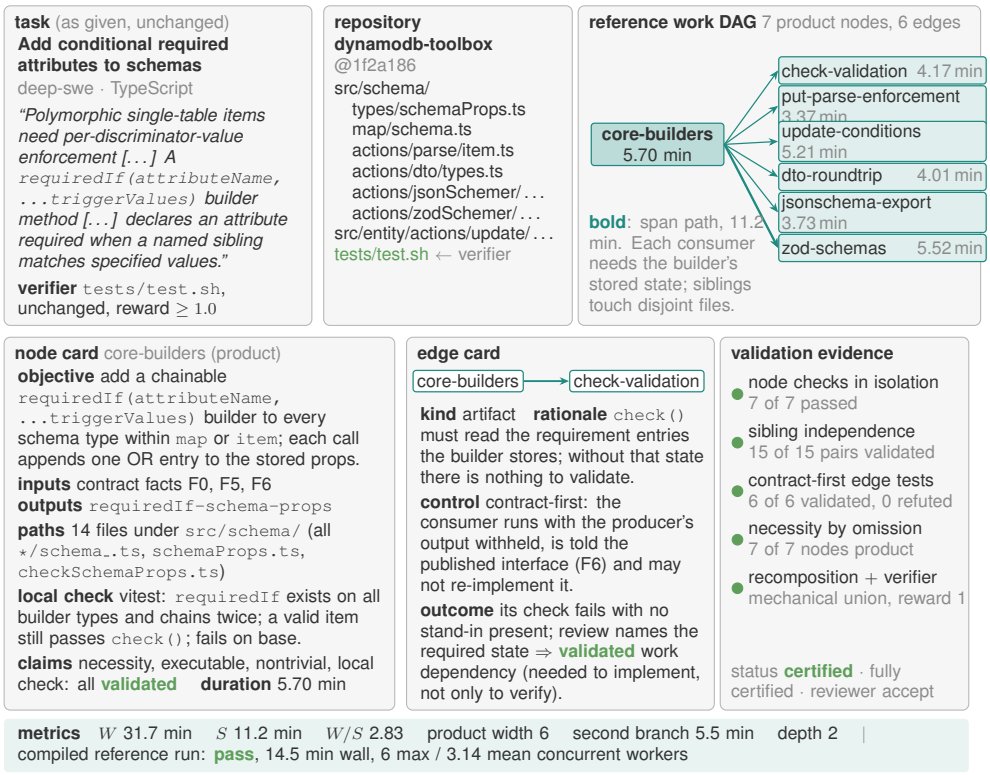


Figure 8: One MACE-Bench item, all values real (`dynamodb-toolbox-conditional-attribute-requirements`): task, repository, reference work DAG, one node card, one edge card with its validation method, the evidence strip and the structure metrics. The original verifier decides correctness; the DAG is one replay-tested reference decomposition, not a unique ground truth.

A.2.4 Does the constructor matter?

A reader will ask whether the graphs are an artifact of one way of proposing them. A preregistered comparison (`docs/constructor-comparison-20260920.md`, fixed before any arm ran) answered this on one fixed set of 21 tasks drawn by rule from the released corpus, so that every task has a successful trajectory and a verifier-passing graph and construction success is not confounded by broken tasks. Each arm ran one strategy with no fall-through, then the screens, review, replay with every control and at most two bounded repairs, under the production budgets; relaunches retried only infrastructure statuses, so no arm received a second construction draw. The forward arm produced no result: the provider's safety layer refused its stage-one draft on 21 of 21 tasks in both of 2 launches, a limit on how the inductive framing can be instructed on this provider, not a finding about the method. A cell is censored when its failure side is infrastructure or the per-task cost cap before a verdict existed; screened-out, rejected and composition-conflict outcomes are verdicts and stay in. Because censored cells are not evidence about graphs, the comparison uses the *paired common evaluable subset*: the 18 tasks on which every run arm reached a verdict (Figure 9, Table 4; the 3 excluded tasks and the full matrix are in Appendix N).

Direct and hierarchical are close on every outcome and their graphs align most closely across arms (node-path alignment 0.59 against 0.48 for direct versus backward and 0.21 for direct versus TMI). Backward recovers the widest graphs (mean width 5.0) on fewer tasks. TMI is weakest at recovering executable dependency: 5 of its 10 verifier-passing graphs are width one and no reviewer accepted one. A procedural model recovers what happened after what, and once observed order is recorded apart from dependency most of it is not dependency; this is the finding of §A.4.1 seen from the constructor's side. Median work over span of the verifier-passing graphs was 1.59 (direct), 1.50 (hierarchical), 2.67 (backward) and 1.00 (TMI). Three conclusions are supported; a ranking is not. The representation is recoverable under several strategies with different yield and width trade-offs. Order induction is not work decomposition. The production constructor is consistent with the comparison.

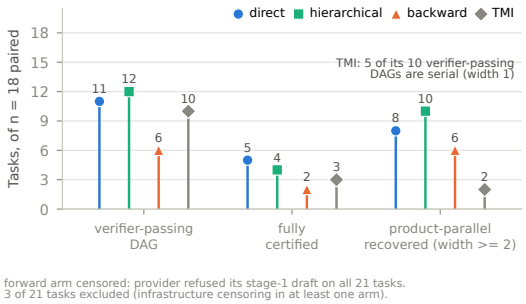


Figure 9: Paired constructor comparison on the common evaluable subset ($n = 18$ tasks, one denominator for every arm, one seed per cell): verifier-passing, fully certified and product-parallel graphs per arm; TMI's width-one graphs annotated; forward absent (provider refused every draft). Direct and hierarchical are similar, backward is wider on fewer tasks, TMI recovers few parallel graphs; no winner is claimed.

Table 4: Paired constructor comparison ($n = 18$ tasks per arm; every arm has a verdict on every task). Reviewer accept is over graphs that reached review; mean product width is over verifier-passing graphs; cost is the median per task in dollars (replay plus an estimated model-call cost; an estimate, not billed spend). Single seed per cell. Direct and hierarchical are similar; backward recovers wider graphs on fewer tasks; TMI recovers few parallel graphs; no winner is claimed.

Arm	Verifier pass	Fully certified	Product-parallel	Reviewer accept	Serial graphs	Mean width
Direct	11	5	8	11/15	0	3.5
Hierarchical	12	4	10	8/15	0	3.6
Backward	6	2	6	5/13	0	5.0
TMI	10	3	2	0/12	5	1.2
Median cost per task: direct $17, hierarchical $27, backward $31, TMI $37.						

What the study does not establish is a ranking: every cell is a single seed on 18 tasks, and the production constructor was chosen on yield before the experiment ran. Two earlier comparisons in the project did not survive audit (a shared prompt opener and an empty trajectory in one, self-censored evidence in the other) and are reported in Appendix N because several earlier conclusions rested on them.

A.3 From work DAGs to executable plans

A work DAG says what depends on what and nothing about workers, schedules or visibility, so MACE-Bench fixes one operational interpretation and ships it with every task. For the graph $A \to C$, $B \to C$ the plan reads, in effect: launch A and B in isolated workspaces; when both have finished, launch C with A's and B's outputs applied and their contract facts in its instruction; compose the three deltas mechanically; run the original verifier once (Figure 10). Nothing in the graph says how long to wait, what a worker may see, or what happens when two siblings edit one file; the plan does.

The plan. A deterministic compiler maps a reference work DAG to an execution plan in a fixed schema (`execution-plan-v1`); recompilation is byte-identical. Each plan node carries a worker instruction rendered from the objective and the allowed contract facts, its dependencies and readiness rule, its predecessor inputs with their delivery mode (workspace patch or text fact), its declared outputs with artifact types, and two label sets: `may_see`, the outputs of its ancestors, and `must_not_see`, the outputs of every node that is neither an ancestor nor itself. The plan fixes concurrency semantics (dependency-triggered launch, no level barriers), failure propagation (a failed node skips its descendants), a retry policy that retries infrastructure failures and never agent failures or deterministic predecessor conflicts, the composition strategy and the final verifier specification. A sentence-level sanitizer removes evaluator wording (gold, replay, certification, trace step) from worker-facing text and fails compilation if any survives.

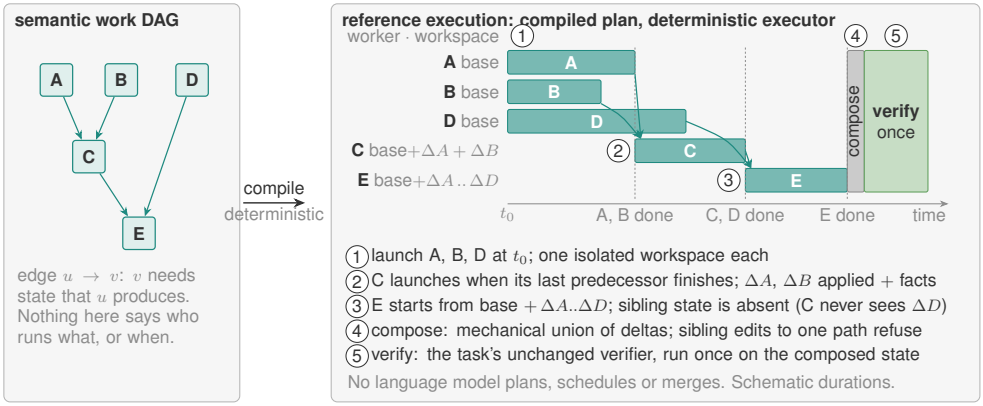


Figure 10: Compiled execution, schematic: a semantic DAG ($A, B \to C$; $D \to E$; $C \to E$) becomes an explicit schedule. Eligible nodes launch at once in isolated workspaces; a consumer launches when its declared predecessors finish, with their outputs applied; the deltas are composed mechanically and the unchanged verifier runs once. Durations are illustrative.

The executor. A deterministic executor runs the plan through a worker-adapter contract: frozen environment, instruction, allowed facts and predecessor artifacts in; patch or artifacts, status, timing, cost and logs out. One fresh worker per node, with delegation and subagents disabled. A node launches the moment its last predecessor finishes and a worker slot is free; the compiler sets the slot count to the graph's widest level, so no ready node waits. Each worker starts from the base commit plus its ancestors' deltas and nothing else, so sibling state is physically absent rather than merely hidden. Build artifacts a worker leaves behind (`target/`, byte-code caches and the like, an overridable default set) are stripped from its patch the moment it returns, so successors and the composition see only the node's work; the dropped paths are recorded per node. The executor performs no language-model repair, no fallback integration and no interface repair. The live adapter wraps the same container leaf adapter used by certification replay, so worker prompts are byte-identical to those of the replay controls.

Composition and the refusal policy. Composition is the mechanical union of the successful nodes' own deltas. Under the default *strict* policy an overlapping sibling edit on any shared path aborts composition before the verifier runs, because a policy that never merges silently is the right default for a reference. The audit then re-composes such runs *diagnostically* (a three-way merge of the same deltas) and reads the verifier on the result; only a verifier failure of the composed state, or a conflict on product files that does not compose, is a structural failure. This is why the paper never reads a strict refusal as evidence against a graph: it is a policy outcome, recorded as such.

Fidelity. A separate audit script checks the 42 executor runs that have full schedule capture for six properties: input provenance (each worker saw exactly its `may_see` set), sibling isolation (no worker saw a `must_not_see` artifact), dependency-ordered launch, realized concurrency within the graph's width, mechanical composition (no post-composition edits) and verifier provenance (the original verifier at its frozen hash). Over 42 audited runs from 5 sources (product widths 1 to 9, depth up to 3) there were 0 violations, and realized maximum concurrency equalled graph width in 42 of 42 runs (Appendix O).

Two roles, kept apart. *Role A, validity audit.* Does the reference graph survive isolated execution under its declared dependencies? The pairwise controls of §A.2.2 never execute a whole graph at once; compiled execution does. A graph whose composed output fails V after every node passed its own check, or whose siblings edit conflicting regions of a product file, has a *hidden dependency* that the pairwise controls did not surface. Because a raw pass rate mixes policy, stochastic worker failure, executor faults and real conflicts, every run is attributed to exactly one of five categories: (1) strict pass; (2) strict refusal whose diagnostic composition passes V, a policy outcome and not a failure; (3) hidden dependency; (4) worker failure, a node's own run failed; (5) executor or environment failure. Only category 3 is a structural failure, and only categories 1–3 form the denominator of

the structural rate (§A.4.3). *Role B, reference execution condition.* Executed directly, how much of the available parallelism becomes speedup? In the pilots this role was played by the compiled executor itself (Appendix U). Its isolated per-node containers, however, are a different substrate from the single container and shared checkout in which the agent arms run, so a compiled-versus-agent comparison mixes scheduling with substrate. The final evaluation therefore assigns role B to a different executor, DAG-Sched (§3): the same units and dependency rule, dispatched by a deterministic scheduler as leaf agents inside the orchestrating arm's own container and checkout. The compiled executor remains the validity audit (role A) and the certification record; no compiled run enters the final comparison. Either way the reference condition is not an autonomous planner and not an upper bound on the task.

A.4 What the reference shows about the released tasks

The public release holds 104 tasks from 4 sources: 100 PARALLEL (product width at least two) and 4 CONTROL (width one), each shipped with the original task and verifier, its reference graph and per-claim record, a compiled plan and metadata (composition: Appendix B.1). The statistics in this subsection were computed on the historical curated pre-release pool of 122 tasks (112 PARALLEL, 10 CONTROL), which contains the release; they describe that historical pool and were not recomputed on the release. Three findings follow from the graphs alone.

A.4.1 Successful trajectories overstate work dependency

Order each graph's product units by the step at which the trajectory first edited them. Every unordered pair of product units is then a pair the trajectory executed in sequence; the reference graph declares it dependent if a directed path connects the two units and independent otherwise. The unit of analysis is the pair, pooled across graphs, so a graph with k product units contributes $k(k-1)/2$ pairs and large graphs weigh more (the 10 largest hold 40% of the pooled pairs); we report per-graph medians beside every pooled share.

Across the 105 replay-evidenced PARALLEL graphs whose units all carry trace steps, 521 of 574 pairs (91%) are independent, and the median graph places every product unit on its own branch (84 of 105 graphs are pure fan-outs; per-graph distribution in Figure 11). The pooled share is dominated by those fan-outs. Within the 21 replayed graphs that declare at least one product dependency, 72% of 187 trace-ordered pairs are still independent (per-graph median 0.67), which is the measure of how much observed order was dependency. On the 38 fully certified graphs, whose sibling claims were all validated by isolation replay, the share is 96% of 200 pairs, but only 2 of those graphs declare any product dependency. Where dependencies are declared, trace order is not a reliable guide to their direction: of the product-dependency edges whose endpoints both carry trace steps (72 forward, 22 backward), 23% point backwards in trace time, in 10 graphs, the consumer having been touched before its producer. In duration terms the reference's critical path is about half its total work: median span/work is 0.53 ($n = 106$), and the critical path holds a median of 0.33 of a graph's product units. W is a sum of isolated single-worker replays, not a trajectory duration: on the worked example the recorded single-agent wall is 20.0 minutes against $W = 31.7$, and over the 53 tasks with both the ratio of W to single-agent wall has median 0.99 (IQR 0.78–1.31), so W is calibrated to single-agent wall at the median but not task by task (§H.1).

What "independent" rests on. Over all 110 usable graphs, including 5 manual-review graphs that were never replayed, 565 of 689 pairs (82%) are independent. Of those independent pairs, 81% are backed by a sibling claim validated by isolation replay, 12% by a claim left inconclusive because the two units edited a common file although the union passed the verifier, and 7% by no replay claim at all, every one of them in the manual-review graphs. Those 5 graphs hold 17% of the pooled pairs and 57% of the dependent pairs, which is why the replay-evidenced population is the one we quote. Two further caveats bound the claim. The graph encodes work dependencies only: interface and verification dependencies are removed by the contract-first control, so "independent" means independently *authorable*, and the fan-out shape partly reflects that design (§A.2). And the graph does not test dependencies the trajectory asserted; it tests whether work the trajectory did in sequence needed to be done in sequence. We therefore do not say that most dependencies in trajectories are spurious, nor that they are verification dependencies. We say that in our corpus sequential trajectory order substantially overstates semantic work dependency: the replay-evidenced graphs place most

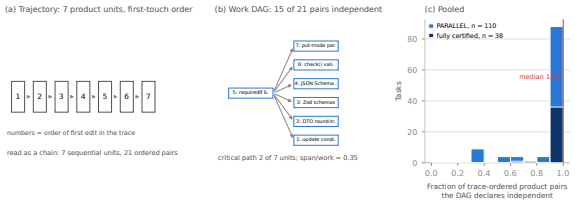


Figure 11: Trajectory order against work dependency. (a) The worked example's 7 product units in the order the successful trajectory first edited them. (b) The same units in the reference graph: 15 of 21 trace-ordered pairs are independent and 4 of 6 edges point against trace order. (c) Per-graph fraction of trace-ordered product-unit pairs that the graph declares independent, all PARALLEL graphs with trace steps ($n = 110$, of which 5 are unreplayed manual-review graphs) and the fully certified subset ($n = 38$). Reading dependencies off the trace would serialize work the replay-tested graphs place on separate branches.

Table 5: Horizon $\times$ available parallelism, PARALLEL tasks with replay durations ($n = 107$; the 5 manual-review tasks have none). *Longer*: product work at least 8 min; *parallel*: work/span at least 2.0; both thresholds fixed before counting. The regime where structural parallelism should pay holds 33 tasks, and 41 longer tasks of width two or more have a work-over-span ratio below two. Thresholds are illustrative; the counts move under nearby thresholds.

	work/span < 2.0	work/span ≥ 2.0
product work < 8 min	25	8
product work ≥ 8 min	41	33

trace-ordered product pairs on independent branches, and about a quarter of the dependencies they declare run against trace order, so a trajectory is not a measurement of available parallelism in either direction.

A.4.2 AVAILABLE PARALLELISM IS HETEROGENEOUS AND WEAKLY RELATED TO HORIZON

Over the 107 PARALLEL tasks with measured durations, total product work has median 10.1 min (interquartile range 6.3 to 13.3; single-agent wall, known for 58, median 9.3 min), and work over span has median 1.89 (1.50 to 2.42, maximum 5.62): W/S reaches two on 41 tasks, three on 6, and sits below 1.5 on 27. W/S shows little rank correlation with horizon in this subset: Spearman correlation between product work and work/span is 0.06 ($n = 107$; with single-agent wall as horizon, 0.02 over the 53 tasks that have both), and the pool is range-restricted to width at least two with most short or width-only graphs removed by curation, which attenuates any correlation; 41 of the 74 longer tasks (55%) have work/span below two, and 24% of the short tasks have work/span of two or more (Table 5, Figure 12). Product width tracks W/S (rank correlation 0.69) but is not substance: 6 of 65 graphs of width three or more have W/S below 1.5, and 8 graphs have a second-longest branch under one minute; the released pool under-represents this pattern because curation removed most width-only graphs. The second-longest product branch has median 2.7 min and reaches two minutes in 72 tasks, and 86 of 112 graphs are single-level fan-outs. How the evaluation's speedups vary with horizon and width is reported, as an exploratory subgroup analysis, in §E.1.4.

A.4.3 THE REFERENCE IS EXECUTABLE, AND ITS EXECUTION IS MEASURED

The pairwise tests of §A.2.2 never execute a whole graph at once. Compiled execution does (role A of §A.3; Figure 13, Table 6). Of the 111 graphs with a compiled run (102 released tasks and 9 archived controls; 20 evaluation tasks have none, and one task, `spice-circuit-simulator-in-rust`, was run twice with its repaired-plan rerun counted), 109 have an outcome attributable to the graph, and 100 of them (91.7%) are structurally executable: the isolated workers' outputs compose and pass the verifier, directly under the strict policy (72) or, for the 28 runs the fail-closed shared-path policy refused, under diagnostic composition. 9 runs (8.3%) hide a real dependency: 3 composed states fail the verifier although every node passed its own check, and 6 sibling pairs edited conflicting regions of a shared product or test file. 2 worker failures and 0 environment failures are reported apart and never charged to the graph, and the fidelity audit found 0 scheduler violations in the 42 audited runs with full schedule capture (the other runs

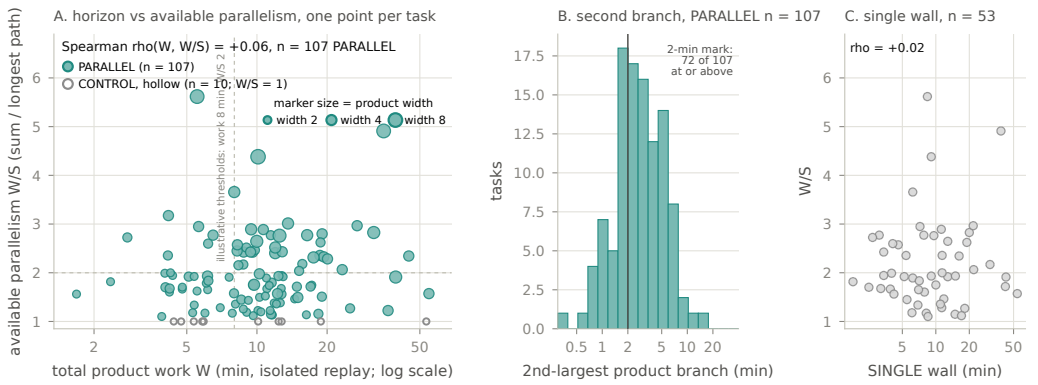


Figure 12: Horizon and available parallelism are distinct axes. (A) One point per PARALLEL task with replay durations ($n = 107$): total product work W (isolated-replay minutes, log scale) against W/S, marker size product width; CONTROL tasks hollow ($W/S = 1$). Dashed lines are the fixed, illustrative thresholds of Table 5. (B) The second-largest product branch in minutes, with the 2-minute mark. (C) W/S against single-agent wall for the tasks with a recorded wall and replay durations. In the measured subset total work and W/S show little rank correlation (Spearman 0.06, $n = 107$) and wide graphs with small W/S exist, so horizon is a poor guide to parallel opportunity.

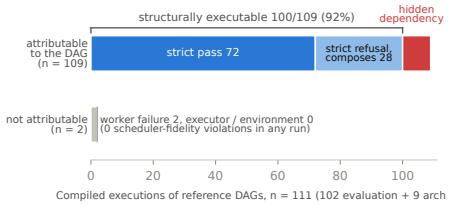


Figure 13: Compiled execution as a validity test over the 111 graphs with a compiled run (102 released tasks, 9 archived controls). Structurally executable (strict pass plus strict refusals whose diagnostic composition passes) against hidden dependency; worker failures apart; the structural rate is over the 109 attributable runs. 91.7% of the 109 adjudicable compiled runs execute under isolated scheduling once conservative refusals are re-composed diagnostically (64.9% of all runs pass under the strict executor); the 9 hidden dependencies are the structural failures, and a conservative refusal is not one.

were not audited). The raw strict pass rate (64.9% of 111) is lower mainly because of the conflict policy, and also because of the two worker failures.

What the hidden dependencies are. The 9 category-3 runs are of two kinds. In 3 the union of all node outputs applied cleanly and the verifier failed after every node had passed its own check; in 6 two siblings edited overlapping regions of a product file and no composition existed. In each, one node's correctness depended on a sibling's state that the pairwise controls did not surface, because each sibling passed its own check in isolation. A recurring shape, named in the compiled probe report for the spice and participle graphs, is a shared registration point: two siblings each add a module declaration to the same manifest or entry file, semantically independent and textually colliding. Of the 4 hidden-dependency graphs with a stored reviewer grade, 4 had been flagged by the reviewer for a possible compile or interface dependency. One category-3 run deserves its own sentence: `spice-circuit-simulator-in-rust`, a FrontierSWE task whose pre-repair compiled run (Opus 5 worker, 2026-09-22) refused composition on exactly such a registration collision, was repaired so that one root unit owns every registration file and rerun the same day with the same worker model; the 12 nodes composed without conflict and the graded verifier passed 67 of 99 suite decks against a threshold of 69, with 31 of the 32 failing decks covered by no node's own check. The mechanical rule says category 3; the honest reading is that the node checks were too narrow to certify their families, and the repair for that is stronger node verification, not a different decomposition.

Table 6: Compiled-execution outcomes over all 111 compiled runs (102 in the release, 9 archived controls). Categories 1–3 are attributable to the graph and form the denominator of the structural rate ($n = 109$); categories 4 and 5 are not charged to the graph. The raw strict pass rate counts category 1 only, over all runs. The gap between the strict rate and the structural rate is the conservative refusal policy and two worker failures.

Category	Definition	Runs
1	strict pass: mechanical composition passed the original verifier	72
2	strict composition refusal on a shared path; diagnostic composition passes (structurally executable, not a failure)	28
3	hidden dependency: every node succeeded but the composed state fails the verifier (3), or a shared-file conflict that does not compose (6)	9
4	worker stochastic failure (a node's own run failed)	2
5	executor or environment failure	0
Structurally executable (1 + 2) over attributable runs (1 + 2 + 3)		100/109 (91.7%)
Hidden dependency (3) over attributable runs		9/109 (8.3%)
Raw strict pass (1) over all runs		72/111 (64.9%)
Scheduler-fidelity violations (input provenance, isolation, ordering, width bound, mechanical composition), in the 42 audited runs only		0

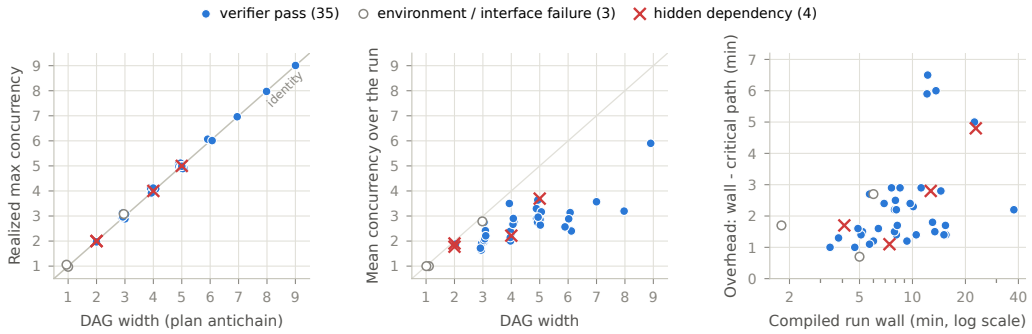


Figure 14: Run-level fidelity view of 42 executor runs (41 distinct graphs). Realized maximum concurrency equals the plan's declared level width in every audited run; mean concurrency reflects branch imbalance; overhead is wall minus critical path. Hidden-dependency failures are marked. The executor realizes the declared width; what it cannot recover is the graph's span.

That run is the one reference run not on the default worker model, and its $186 is included in the spend totals below. A hidden-dependency outcome is recorded as compiled-execution status; the graph keeps its certification record and is not edited, and the hidden dependencies concentrate in graphs whose siblings share product files, the case the shared-file independence rule is conservative about (per-task list: Appendix O).

Execution prices the reference. The 102 released reference runs spent a median of $5.7 on workers (agent-reported API-equivalent cost; interquartile range 3.8 to 8.3; total $877), finished within 1.25× of their executed critical path, with a median 20% of wall spent outside the realized agent-only critical path (per-node container setup and capture, queue waits, composition and verification; 24% over the 65 runs whose verifier ran and passed, the other 37 being strict refusals or failures) and verification alone a median 10% where it ran, and ran their own workers 1.69 times faster than serializing them would have (median over all 102 runs, refusals included). That last number compares a schedule with its own serialization, not with an independent single-agent run, and it is not the evaluation result of §4, whose DAG-Sched arm is a different executor (§C.1.2). Over the 42 runs with full schedule capture, mean concurrency ranged from 1.6 to 5.9 (median 2.65), orchestration overhead from 0.7 to 6.5 minutes (median 1.7), and the median ratio of compiled wall to critical path was 1.31 (Figure 14).

Table 7: Composition of the 104-task public release by upstream source and pool, after the source-label correction described below.

upstream source	citation	PARALLEL	CONTROL	total
DeepSWE	Huang et al. (2026)	41	1	42
SWE-rebench V2	Badertdinov et al. (2025; 2026)	36	1	37
SWE-Bench Pro (public set)	Deng et al. (2025)	22	1	23
Terminal-Bench 2	Merrill et al. (2026)	1	1	2
total		100	4	104

B THE PUBLIC RELEASE, CONSTRUCTION FUNNEL, CERTIFICATION COVERAGE AND A CONSTRUCTION CASE

B.1 THE PUBLIC RELEASE: 104 TASKS FROM FOUR SOURCES

The public MACE-Bench release is the evaluation set: 104 tasks, 100 PARALLEL tasks (product width at least two) and 4 width-one CONTROL tasks, drawn from exactly 4 upstream benchmarks (Table 7). Every task keeps its upstream instruction, frozen environment and verifier unchanged. Earlier construction material (a larger candidate corpus and a curated 122-task pre-release pool that also contained FrontierSWE and other sources) is historical and is described only in the funnel of Appendix B.2.

Source-label correction. Nine Ansible-repository tasks (`instance_ansible__ansible-*`) were labelled SWE-rebench V2 in the release manifest, the task metadata and the per-cell metrics. They come from the public set of SWE-Bench Pro. All nine are PARALLEL tasks; eight are in the release and one is a historical task excluded for an unrelated reason (its compiled run exposed a hidden dependency). The correction changes only the source label fields. No task instruction, test, verifier, reference DAG, execution plan, replay record, trajectory, timing or cost changed, and a script checks that every experimental metric is byte-identical before and after the relabel (`prep/W1/verify_relabel.py`). Source-grouped statistics in this paper use the corrected labels. Exact task lists per source, licences and URLs are in the release provenance file (`final-upstream-provenance.md`).

B.2 CONSTRUCTION FUNNEL AND RELEASE CURATION

Table 8 recomputes the construction funnel from the committed logs. The first rows describe the whole candidate pool that the construction pipeline processed before any release decision; they are historical candidate-corpus numbers, not properties of the release. The last row is the public release. The stages are not all nested: tiering (row 5) is a curation label assigned separately from the certification level (rows 3 and 4).

From the curated pool to the release. The cut from 122 to 104 tasks removed 18 tasks: 7 PARALLEL tasks whose compiled run exposed a hidden dependency (a rule fixed before the final evaluation runs; it can favour a plan executor and hence DAG-Sched), 6 FrontierSWE tasks (5 PARALLEL, one control; a project decision with no recorded task-level reason, files kept in the repository) and the remaining archived former controls, which reduced the control pool to 4. The arithmetic closes: $112 - 7 - 5 = 100$ PARALLEL tasks.

Human involvement and repair. The curation record lists 5 manual-curation admissions and 3 salvage admissions; all other released graphs entered through the automated pipeline and its fixed admission rule. Repair rounds are recorded per task in the provenance of each `reference_dag.json` (for example a `revision/round-2` replay), but the construction manifest has no structured repair-count field, so we do not report a repair frequency.

Replay spend. Replay across all construction attempts cost $13,955 (mean $13.74, median $5.33 per attempt). The replay records of the tasks that ended up in the release account for $2,067 ($2,603

Table 8: Construction funnel. Rows 0 to 6 describe the historical candidate pool and the pre-release curation; row 7 is the 104-task public release. Sources: `experiments/production/curation-20260921.json` (row 0), `release/dags-20260919/curation-20260921/release_manifest.json` (rows 1 to 5), `release/benchmark-20260920/manifest.json` and `evaluation_set.json` (rows 6 and 7).

	stage	N	definition
0	construction attempts	1016	every task-by-wave attempt across all construction waves (a task can appear in several waves); 525 reached a verifier-passing recomposition
1	per-task construction deliverables	345	one deduplicated production candidate row per task
1b	with a DAG artifact	278	the constructor produced a candidate graph
2	verifier-passing recomposition	235	the union of the fresh-worker unit outputs passed the unchanged task verifier
3	at least partly certified	225	certification level fully certified or verifier-passing/partial
4	fully certified	138	every tested claim validated
5	usable after tiering	146	curation tier other than REPLACE (includes controls)
6	curated pre-release pool	122	historical curated pool (112 PARALLEL plus controls)
7	public release	104	100 PARALLEL and 4 CONTROL tasks

Table 9: Claim outcomes over the 104-task release, by claim type. Sibling independence is counted per sibling pair and recomposition per graph. "Tested" means a fresh-worker experiment that could have refuted the claim was run.

claim type (unit of count)	validated	refuted	inconclusive	untested
executable in isolation (unit)	411	26	8	0
sibling independence (pair)	745	0	72	0
recomposition passes the verifier (graph)	103	0	0	1
dependency by withholding (edge)	0	3	1	0
dependency under contract-first control (edge)	19	8	29	0
necessity by omission (unit)	284	93	45	23

for the historical curated pool). These figures exclude constructor model calls and the evaluation runs.

B.3 CERTIFICATION COVERAGE OF THE PUBLIC RELEASE

Every structural claim in a released reference DAG carries one of four outcomes: `validated`, `refuted`, `inconclusive` or `untested`. Table 9 counts them over the 104 released graphs (442 work units, 60 edges). An infrastructure failure never counts as evidence for or against a dependency; it yields `inconclusive`.

How to read edge coverage. Every one of the 57 edges in the 100 PARALLEL graphs had a dependency test attempted, but only 17 were validated. Across all 60 released edges, 29 tests ended `inconclusive` under the contract-first control, which gives the consumer only the producer's interface and so separates an interface dependency from a work dependency. An inconclusive or refuted edge stays in the released graph with its outcome recorded. Edge coverage is also a small part of the evidence, because many released graphs have no edges at all (they are pure fan-outs whose units all run from the base state). For those graphs the load-bearing claims are unit executability, sibling independence and recomposition under the unchanged verifier. By the release's own certification level, 99 of the 100 PARALLEL graphs carry a verifier-passing recomposed replay; the remaining one was admitted by manual review on the gold solution only.

Historical comparison. The historical 112-task PARALLEL pool of the pre-release corpus had 61 of 135 edges tested (20 validated) and 106 of 112 graphs at least partly certified. These numbers are superseded. Every historical task that carried an untested edge was among the tasks removed in the

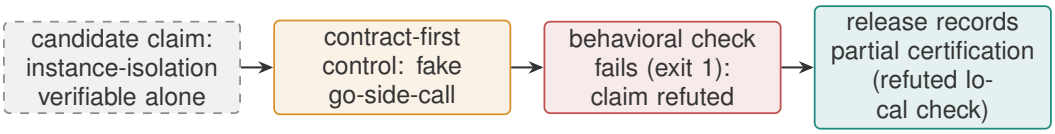


Figure 15: Claim lineage of one released unit of `tengo-callable-instance-isolation`. The candidate claims that `instance-isolation` can be verified on its own. A contract-first isolation replay substitutes a scaffold for its sibling `go-side-call`; the unit's declared behavioral check fails (exit 1), so the claim is recorded as refuted and the release marks the graph as partially certified. The figure shows only this claim; the earlier edge refutation is described in the text.

cut to the release, so coverage on the release is higher than on the historical pool; this is a property of the curated set, not a change in the protocol.

B.4 Pre-replay screens: the six-question rubric and the gold check probe

Construction uses two cheap checks before any replay money is spent. Neither is evidence about a graph, neither certifies anything, and neither acts as a verifier. They only decide whether a candidate is worth replaying or should be revised first. All structural evidence in the release comes from executable replay and intervention (Appendix B.3).

Gold check probe. Every unit's local check runs against the base tree and against the gold solution. A check that passes on the base tree is vacuous; a check that fails on gold tests invented internals; a check that needs a file the unit itself must write is invalid. Such checks are gated out or rewritten. The probe says nothing about whether units are independent or whether an edge is a work dependency.

Six-question rubric. A model reads only pre-replay inputs (the task instruction, the candidate graph with every replay-derived field removed, a cleaned trace summary and the gold footprint) and answers each of six questions with PASS, FAIL or UNCERTAIN: (1) are units grounded in the trajectory and gold hunks; (2) are they semantic work units rather than file shards; (3) are declared siblings plausibly independent; (4) are declared dependencies plausibly work dependencies rather than trajectory order or verification order; (5) is granularity sensible (minutes, not seconds, and no mega-node); (6) is parallel work omitted. The rubric outputs an action (replay, revise or reject) and a structured item list.

Calibration. Run blind on 29 stored candidates that had already passed the static screen and the probe, the rubric made no false reject and five action-level false accepts. Its dependency question passed edges that withholding controls later refuted: it was wrong on seven of twelve edges it passed with a known outcome. As a funding gate it would have sent almost every candidate to replay. We therefore froze it as an advisory screen whose item list feeds a constructor retry, and never read its answers as evidence. This weakness is the reason the benchmark's structural claims rest on replay rather than on model judgment.

B.5 A construction case: replay refutes a proposed claim

The DeepSWE task `tengo-callable-instance-isolation` shows how candidate structure gains or loses standing only through executable evidence (Figure 15). We report the lineage as the artifacts record it.

A superseded candidate failed recomposition. An earlier candidate from a backward constructor had 3 units and no integration unit. In replay, the mechanical union of the fresh-worker outputs failed the unchanged verifier, so its recomposition claim was recorded as refuted with the note that a substantive integration step might be required. The candidate was not admitted.

A proposed edge is refuted. The hierarchical constructor then proposed 4 work units (`go-side-call`, `instance-isolation`, `repro-harness`, `docs-go-side-call`)

with an edge `go-side-call` → `instance-isolation`. A contract-first control gave the consumer only the producer's declared interface, through a hand-written scaffold that records calls and returns values unchanged. The fresh worker authored `instance-isolation` in full against the scaffold, with no line shared with the producer's output, and its check failed only through the scaffold. The edge was therefore refuted as a work dependency: the dependency is on an interface and on verification, not on the producer's work. The edge was removed and the two units became siblings.

The unit's own executability claim is refuted too. Because the behavioral assertions of `instance-isolation` need the real `go-side-call`, the unit's declared verification command fails in isolation (exit 1). Its `executable` claim is recorded as `refuted`, while 3 of the four units have validated `executable` claims. The unit keeps a scaffold-based local check, and its behavioral end-to-end check is deferred to the recomposed union under the unchanged verifier. The scaffold and test code are stored inline in the released `reference_dag.json`.

Bounded repair and the released graph. A second repair round changed only the check of `repro-harness`: replay showed the unit produced its output but its command hard-coded an absolute path outside the captured workspace, and the reviewer flagged the check as compilation-only; the repair fixed the path and restored a base-failure assertion. The released graph has 0 edges, product width 2, plan width 4, $W = 17.2$ and $S = 9.8$ replay minutes. The union of the four unit outputs passes the verifier, and omitting any unit makes it fail, so all four necessity claims are validated. The independence claim between `go-side-call` and `instance-isolation` is `inconclusive`, because both units may write the same file.

What the release records. The refuted claims ship with their evidence. The manifest lists the task as `verifier-passing/partial (refuted local check)` with a reviewer grade of keep-with-caveat, and the graph file's own status field reads `failed-review`. No compiled execution result is claimed for this task. A reader can re-derive every step from the released file. The case does not show a bad graph repaired into a perfect one. It shows proposed claims being tested, some failing, and the failures being recorded.

C Full evaluation protocol

This appendix expands §3: systems and evaluation set, the four arms, concurrency limits, the differences between MAS+DAG and DAG-Sched, timing definitions, rerun rules and scoring, and the per-cell integrity conditions.

C.1 Evaluation design

The same task is run under four arms that share one harness, one model and one container, so that differences between arms are attributable to what each arm changes rather than to capability (Figure 16); the harness alone moves coding-agent results substantially (Yang et al., 2024; Wang et al., 2025; Xia et al., 2025), so it is held fixed within a system.

C.1.1 Systems and evaluation set

Three systems. Codex × GPT-5.6 Sol (Codex CLI 0.156.1, reasoning effort high), and Claude Code 2.1.280 with Fable 5 and with Opus 5.5. A system is one harness with one model; results are reported per system and pooled only where a figure says so. The recorded model identifier of every Fable 5 cell is `anthropic/claude-fable-5`.

Same-model provenance. The source trajectories were produced by the frozen baseline (Harbor 0.22.0, Claude Code 2.1.247, Fable 5; §A.2). The release records Fable 5 as the constructor model of 34 of its 122 graphs (the other 88 name a later repair pass without a model) and as the reviewer of 119; replay workers default to the same model; of the compiled reference-run records, 75 name Fable 5 as the worker model, 2 (the spice reruns) Opus 5, and 65 no model (the executor's default is Fable 5). Every task was therefore admitted because the frozen baseline, which runs Fable 5, solved it once, and the plan text that MAS+DAG and DAG-Sched receive was distilled from a Fable 5

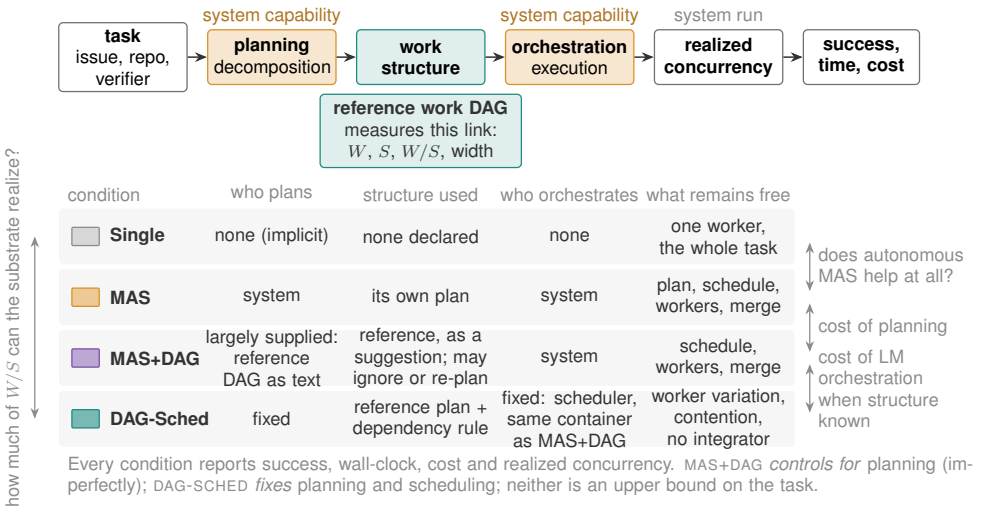


Figure 16: What each arm controls (schematic). Speedup is the end of a chain from the task's structure through planning, orchestration and the substrate; the reference work DAG measures the structure link. Per arm: who plans, what structure is used, who orchestrates and what remains free; the margin labels name the adjacent comparisons. MAS+DAG controls planning imperfectly (it may ignore or re-plan the plan text, and the text carries information as well as structure); DAG-Sched fixes plan and schedule but differs from MAS+DAG in more than who schedules (§C.1.2); neither is an upper bound on the task.

solution. The Claude Code × Fable 5 system may gain from this in every arm, most plausibly in MAS+DAG and DAG-Sched; we have no measurement that separates the effect, so cross-system comparisons of levels should be read with it in mind.

Evaluation set. The evaluation set holds 104 tasks, 100 PARALLEL and 4 CONTROL (`experiments/paper-eval/frozen-104.tasks.json`). It is the release (§A.2) minus the Long-Horizon-Terminal-Bench control, the 7 graphs whose compiled run exposed a hidden dependency (category 3 of §A.3) and the 6 FrontierSWE tasks (a project decision of 2026-09-23; the task-set file records no reason), which gives 108 tasks, and then minus the 2 network-denial controls and 2 archived controls. 92 of the 104 tasks have a compiled reference run. The category-3 rule was fixed before the final runs, but it is not neutral: those graphs failed exactly where DAG-Sched can fail, because the compiled executor runs the same units and every unit can succeed while the composed state fails. Removing them favours DAG-Sched on pass rates and possibly on speed. The set was frozen on 2026-09-24, after a first Codex pass on a larger provisional slate, so we do not call it pre-registered; the exclusions are by source, pool and compiled-audit category and are listed in the task-set files. Headline statistics use the 100 PARALLEL tasks; the 4 width-one controls are reported descriptively (Appendix L). The graph is never shown to Single or MAS.

Cells and compute. Three systems, 104 tasks and four arms give 1248 cells. Each cell has one canonical attempt ($N = 1$), chosen by each run's documented rule (first attempt with a verdict for the Claude Code runs, the record's trial otherwise); 157 cells (4 Codex, 84 Fable 5, 69 Opus 5.5) have more than one archived trial, and the attempt rule changes the verdict of a few of them (Appendix K). Every cell runs in one Harbor trial whose container is limited to 2 CPUs and 8 GiB, shared by the parent and every worker or unit of the cell.

C.1.2 FOUR ARMS, MODEL-INITIATED AND RUNTIME-INITIATED SPAWNING

- Single: one agent, the original task, no delegation tool.
- MAS: the harness's native subagent tool (Codex `spawn_agent`; Claude Code `Agent`), the task and no plan. The model must both plan and orchestrate.
- MAS+DAG: MAS plus the task's frozen plan text, rendered from the reference DAG (units, dependencies, file scopes; the version with the necessity tag removed after the integrity finding of

§C.1.4). The model still decides whether, when and how many workers to spawn, writes each worker's brief, and may implement, integrate, repair and verify itself.

- DAG-Sched: a deterministic scheduler on MAS+DAG's substrate: the same Harbor trial, container image and limits, shared checkout, model, flags and plan text. It dispatches one leaf agent process for every unit whose dependencies have all succeeded, immediately and with no cap. A leaf receives the task instruction, the plan and a fixed note naming its unit and the units already finished or running. A unit succeeds only on a clean exit, a completed turn and a non-empty final message; the descendants of a failed unit are blocked, independent units keep running, and nothing is retried. There is no language-model orchestrator and no integration step beyond the plan's own integration units; the verifier then scores the tree as it is. In 80 of the 100 PARALLEL tasks the plan has depth one, with no dependency between units, so DAG-Sched starts every unit at once; dependency-triggered launch matters only on the other 20 (§E.1.4). DAG-Sched is therefore mostly a parallel launch of solution-derived, file-scoped units.

Concurrency limits. The final policy has no benchmark concurrency cap: both harnesses are configured for 64 concurrent workers, above the widest DAG in the set (11 units). Not every kept cell ran under it. The 104 Opus 5.5 MAS cells, 44 Codex MAS cells and 23 Codex MAS+DAG cells come from runs whose prompt still said never to have more than 8 agents active at once (Codex: 7 worker threads). Cells where that cap visibly changed the plan or the worker allocation were rerun uncapped under a stated rule; the kept capped cells started at most 5 workers (MAS) and 7 (Codex MAS+DAG, on a task whose plan has that many units), so we expect the cap to matter little, but it is a difference between cells. It also confounds the MAS-to-MAS+DAG comparison on Opus 5.5, where every MAS cell had the cap sentence and no MAS+DAG cell did. Claude Code also runs at most 10 parallel-safe tool calls per batch by default; whether subagent calls count toward that limit was not verified, so it may be a hidden cap on MAS and MAS+DAG for the widest plans.

DAG-Sched is not the compiled plan executor. The executor of §A.3 (COMPILED) runs each unit in its own container and composes patches mechanically. It serves the certification audit and was the fourth arm of the pilots (Appendix U); none of its runs enter the final comparison, and no pilot number is presented as a result for DAG-Sched. Codex leaves run with multi-agent features off. Claude Code DAG-Sched is mixed-harness: only the 8 Fable 5 and 23 Opus 5.5 cells on the fixed version remove the delegation and messaging tools from the leaves, block tree-changing `git` commands in the shared checkout and record out-of-memory kills. The other Claude Code DAG-Sched cells ran the earlier version. A read-only scan of every Claude Code DAG-Sched leaf log (`paper/figures/trace-20260925/scan_cc_d_leaf_tools.py`) finds that no leaf, in either version, was offered or called the subagent tool (`Agent` or `Task`); earlier-version leaves were offered the messaging tools, and leaves in 1 Fable 5 and 3 Opus 5.5 cells called one of them (listing agents or stopping a task), with no spawn. One earlier-version Fable 5 leaf attempted to fetch, from the upstream repository, the `ansible` source file that its task changes, a listed leak channel; the fetch failed ("Socket is closed"). So "one agent per unit" holds in every cell. Every cell in which a leaf ran a tree-changing `git` command while another leaf was running was rerun on the fixed version (§C.1.4). The Claude Code port passed smoke tests but has not had an independent code review.

What still differs between MAS+DAG and DAG-Sched. Besides who decides what runs when, and how many workers run, the two arms differ in worker context (a MAS+DAG worker receives a brief from a parent that has explored the repository; a DAG-Sched leaf starts from the instruction, the plan and the fixed note), worker tools (MAS+DAG workers can message the parent; DAG-Sched leaves have no peers), parent work (MAS+DAG's parent may take the critical unit, integrate, repair and run the task's checks; DAG-Sched has no parent), failure handling (MAS+DAG's parent may retry or take over a failed unit; DAG-Sched blocks its descendants), process model (Codex MAS+DAG runs one process with threads, DAG-Sched one process per running unit) and cost structure (MAS+DAG pays for a growing parent context, DAG-Sched for fresh leaf contexts). These differences are part of what orchestration means here, and they are why DAG-Sched/MAS+DAG is not a pure scheduling effect. Every Codex spawn requests the full history for its worker (`fork_turns = all` in 100% of 724 MAS and MAS+DAG spawns), yet a worker's first model call carries a median 12035 input tokens (range 9219 to 19745) while parents hold up to 85243 at spawn time, and the worker rollouts contain the task and plan but not the parent's exploration; what a forked worker

actually receives is inferred from these token counts, not confirmed from the CLI source. Claude Code subagents receive only the prompt the parent writes.

Model-initiated and runtime-initiated spawning (definitions). We call spawning *model-initiated* when a model decides whether, when and how many workers to start (MAS, MAS+DAG), and *runtime-initiated* when a scheduler starts every ready unit, one agent per unit, with no parent (DAG-Sched). The adjacent comparisons then read as follows. Single to MAS: does autonomous delegation help at all, and at what cost? MAS to MAS+DAG: what does the plan add to an active orchestrator? MAS+DAG to DAG-Sched: what changes when the model-initiated orchestrator is replaced by a runtime scheduler on the same substrate? Single to DAG-Sched: how much of the reference structure becomes lower latency? Two limits apply throughout. The plan text supplies information (files, contracts, toolchain, mechanism) as well as structure, so a gain of MAS+DAG or DAG-Sched over Single need not be a parallel-execution gain; and DAG-Sched executes one solution-conditioned decomposition, so it is not an upper bound.

C.1.3 Timing, outcomes and statistics

Three timing definitions, never mixed in one comparison. *Trial time* is the Harbor trial's end minus its start (environment setup, agent and verifier); it is the protocol measure. *Agent time* is Harbor's agent-execution interval (harness lifecycle plus model and tool work). *Total time* runs from the first model interaction to the last model output over the parent and all workers (for DAG-Sched, all units), from rollout and transcript timestamps; DAG-Sched's scheduler start-up before its first unit lies outside it. Every ratio in this paper names its definition, and trial and total values are printed side by side wherever a headline uses total time. The *integration tail* is the parent's last model output minus the last worker's last model output; in 3 Opus 5.5 MAS cells the parent produced no output after its last worker (2 ended their turn while background workers ran, 1 was killed with exit code 137 during a parent tool call), and the tail is clamped to zero. The *first spawn* is the first worker's first model call minus the parent's first model call (MAS, MAS+DAG), or the first unit start minus the scheduler start (DAG-Sched); the case studies instead time the parent's spawn call, which can differ by a few seconds. An independent recomputation of total time with a different event choice moves no speedup ratio by more than 0.020.

Outcomes, reruns and scoring. A cell *passes* when the verifier reward reaches the task's acceptance threshold. A cell that produces no verdict (the one Codex MAS safety refusal and the 3 Pi $\times$ Kimi K3 agent timeouts) counts as a fail in every pass rate and pass difference, so every arm has 100 scored PARALLEL tasks; such cells cannot enter a both-pass timing or cost pair. In 9 Codex DAG-Sched cells of the integrity rerun, unit workers hit the API provider's quota limit. The event was infrastructure: all 9 started between 22:02 and 22:11 UTC on 2026-09-24 on 6 different pods, and 7 other Codex DAG-Sched cells started in that window had no quota error. All 9 were rerun on 2026-09-25 whatever their recorded outcome, each on its original pod with the same configuration (except for the one login change described below), and each rerun result replaces the original whatever it is, so the rule does not select on outcome. 8 rerun cells pass and 1 fails; no final rerun cell has a quota error or a scheduler worker error in any unit. One of them (`tengo-destructuring-bindings`) hit the quota again when the API credit ran out; an attempt that fell back to the exhausted API key failed at no cost and was not scored, and the final attempt ran through a Codex CLI login billed to a ChatGPT subscription, with two hosts added to its egress (`chatgpt.com`, which also serves the model calls, and `auth.openai.com`). The discarded API attempt had recorded a pass, so replacing it does not select on outcome. Its cost ($2.04) is therefore a list-price estimate, and its model calls went through the ChatGPT backend, whose latency may differ from the API's; without this cell, Codex DAG-Sched/Single is 2.03$\times$ on total time instead of 2.02$\times$ (Table 25). Before the rerun, Codex DAG-Sched passed 87/100 as recorded and 84/91 with the 9 cells censored (DAG-Sched/Single total time 2.00$\times$); with the rerun cells it passes 92/100. A refusal is not an invalid cell: the Codex harness records the one API safety refusal as no verdict, and we score it as a fail, as the Claude Code harness does. With it censored instead, Codex MAS passes 84/99 (paired difference vs Single +1.0). One task (`defi-wonderland__smock-134`) fails in every Codex arm on the same pass-to-pass test, whose expected JSON error text depends on the Node version: the Codex installation puts Node 22 on the path and the test expects the older message, while every Claude Code cell with a verifier report passes it. Each Codex pass count is therefore one task lower for a harness reason; within-Codex comparisons are unaffected. Agents

killed with exit code 137 (likely out of memory; there are no kernel logs) are scored at the verifier's reward on the tree they left: 13 agent-level cells, and 3 DAG-Sched cells with 9 killed units; Harbor labels the agent-level kills as rate-limit errors. Dropping every task with a killed cell moves each Claude Code ratio to Single by at most 0.02.

Pairing and estimator. A speedup X/Y is t_Y/t_X per task and a cost ratio X/Y is c_X/c_Y; both are aggregated as geometric means over task-matched pairs. The primary pairing is *both-pass*: tasks on which both cells passed, for speedups and cost ratios alike, so each point uses one task set. *All-verdict* pairs (every task where both cells have a verdict) are the sensitivity (Table 25). The choice of both-pass as primary was made during the analysis, after the workstreams had produced both, so both are always available. Pass rates are passes over all tasks, with no-verdict cells counted as fails, and arms are compared by the paired pass difference over all tasks. Cost is the dollar cost each harness records for the cell (API-equivalent; for Claude Code the last stream-json result event, which includes background subagents); a few Claude Code cells have no recorded cost, so a cost n can be one or two below the speedup n.

Uncertainty. Intervals are 95% percentile bootstraps over tasks (10000 draws, 5000 for Spearman correlations, seed 20260925). There is one run per cell, so an interval describes which tasks were sampled, not how a rerun of the same task would vary; run-to-run variation is real (two attempts of one Fable 5 DAG-Sched cell differ by 257 s) and no claim about it is made. Subgroup analyses by horizon, width and delegation are exploratory, with no multiplicity correction. Pooled estimates resample tasks with all systems of a task together.

C.1.4 Integrity conditions

Two leak channels were found during the evaluation: hosted web search, which runs on the provider's side and is not stopped by container egress rules, and future commits in task images (23 SWE-bench-Pro-lineage images held the gold fix in history newer than the base commit). A third finding was a necessity tag in an earlier plan-text version, which the final plan text removes. The affected cells were rerun with web tools disabled on history-truncated images whose verifier receives the needed commit separately; the gold solution passes and an empty solution fails every validated verifier. Each rerun set was chosen by a rule on exposure (a leak channel used, a harness bug, a broken verifier, an API quota error), not on the cell's outcome; the Claude Code sets were fixed before any of their rerun results was known, and every original result in a set was discarded whatever the rerun gave. The conditions still differ by cell, and no system has every cell under the clean conditions:

- *Claude Code.* The 43 rerun cells ran with web tools off, on history-cut images and (DAG-Sched) on the fixed harness. All other Claude Code cells ran with web tools *on*, on full-history images, and (DAG-Sched) on the earlier harness; the web tools had no successful use outside one Single cell, which was rerun. 2 Opus 5.5 cells read future history and keep their verdicts: one Single cell, which failed, and one MAS cell that saw changelog headings only.
- *Codex.* 267 cells were rerun with web off on truncated images (181 integrity, 9 quota-fix and 77 verifier-fix reruns). The other 149 cells ran with web on, on full-history images; an audit found no web search, no future-commit read and no successful external fetch in them.
- *Hosts.* Codex cells ran on 6 pods and each Claude Code system on 2 hosts; Opus 5.5 MAS+DAG cells all ran on one host under heavy load, and most Opus 5.5 DAG-Sched cells on the other. The same-host sensitivity is in Appendix L.
- *Days.* The arms of one system did not run concurrently. Codex Single and MAS ran partly on 2026-09-23 (53 and 41 PARALLEL cells) while no Codex MAS+DAG or DAG-Sched cell did; all Opus 5.5 Single and MAS cells ran on 2026-09-24 and 22 Opus 5.5 DAG-Sched cells on 2026-09-25. Provider latency can differ between days, which the task-bootstrap intervals do not cover; a same-day-pairs sensitivity is in Appendix L.

C.1.5 Reading fixed before the runs, and what changed

The 2026-09-22 draft, written after the pilots and before the final runs, fixed a reading for every outcome pattern (Appendix T): among them, an ordered triple of the three multi-agent arms means losses in both planning and orchestration; speed rising with cost means latency was bought with

inference; and the compiled arm losing to the orchestrator on a task means the reference graph was wrong there. The first and third were written for a triple whose fourth arm was the compiled executor, so on the final matrix they are readings carried over to a new arm, not pre-fixed tests. With that caveat and the confounds of §F.1.5, the ordering MAS $<$ MAS+DAG $<$ DAG-Sched holds on speed on each of the three original systems (on Pi $\times$ Kimi K3, MAS+DAG is not faster than Single), and MAS is the arm whose extra cost buys no detected speed. For the third, where MAS+DAG beats DAG-Sched, the traces show straggling units (DAG-Sched's primary label in 8 of the 15 Opus 5.5 cases), container contention and, in some cases, coupled units the graph declares independent, so the reading holds only for the last kind (§F.1.4). The final analysis departs from the rest of that plan in at least these ways: one run per cell instead of up to three; geometric means of per-task ratios over both-pass pairs instead of medians over successful replicates; paired bootstrap pass differences instead of Wilson intervals; no equal-success view over tasks every arm solved; no per-regime stratification or regime interaction (horizon and width appear only as exploratory subgroups); agent kills and verifier timeouts scored by the harnesses' rules instead of censored; the phase partition of total time instead of the planned overhead fractions; a final DAG-Sched, the deterministic scheduler on MAS+DAG's substrate rather than the compiled executor; an evaluation set frozen during the runs; and a taxonomy derived bottom-up from the final traces (post hoc, §F.1.6) instead of the plan's taxonomy, which was applied only to the pilots (Appendix U). Of the analysis choices, only these readings and the rerun rules of §C.1.4 were fixed before the results they concern; the pairing, the estimator, the subgroup bins and the taxonomy were chosen during the analysis.

C.2 Arm prompts, verbatim

Each prompt below is appended to the task instruction. Arms MAS+DAG and DAG-Sched also append the plan text (section "Provided work decomposition"). The prompts are those of the final uncapped policy; cells that ran under the older capped policy received the same text plus one sentence limiting active agents to eight (Appendix C.1.2). The prompts differ in their verification instructions: Single is told to verify its implementation, MAS to integrate into one verified solution, MAS+DAG to run the task's checks itself, and a DAG-Sched leaf to stop when its own unit check passes.

Single (single agent).

```
Solve the original task yourself.  Do not delegate work or
create subagents.  Use the normal task environment and
tools, then verify the final implementation.
```

MAS (autonomous MAS; the "plan-first" prompt).

```
Before editing, inspect enough of the repository to form
your own upfront decomposition of the work into independent
units and their dependencies.  Then delegate the independent
units to same-model general-purpose background workers,
assign each worker disjoint files or artifacts, and
integrate their outputs into one verified solution.  Decide
yourself how many workers, if any, the task warrants; do not
spawn redundant or trivial work; use only same-model
workers; do not ask for or infer any evaluator reference
decomposition.
```

MAS+DAG (MAS + plan).

```
The task instruction ends with a section titled 'Provided
work decomposition'.  It is a validated decomposition of
this task into work units with validated dependencies
between them; let it strongly inform how you execute,
without executing it mechanically.  When two or more
independent implementation units are ready at the same time
and each is non-trivial, strongly prefer delegating them
concurrently to same-model general-purpose background
workers launched together, one unit per worker, with
```

Table 10: Arm Single (single agent) duration and interaction length per task: median with the 10th and 90th percentiles in brackets, seconds. Trial time includes environment setup and the verifier; total time is the model-active span (§3). Pi × Kimi K3 runs under a different harness whose fields are only analogs (trial wall time and agent wall time), so it is reported separately and not pooled.

system	cells	trial time (s)	total time (s)	tool calls
Codex × GPT-5.6 Sol	104	524 [295, 863]	421 [223, 700]	37
Claude Code × Fable 5	103	694 [361, 1311]	634 [260, 1229]	48
Claude Code × Opus 5.5	104	421 [195, 894]	333 [122, 815]	22
Pi × Kimi K3 (analogs)	104	855 [374, 1619]	820 [298, 1595]	77

```
disjoint files or artifacts where possible.  Serialize such
units only for a concrete correctness, shared-state or
coordination reason.  Do not delegate purely serial or
critical-path work just to follow the decomposition:
implement it yourself.  While workers run, stay productive
on critical-path work, integration preparation, or other
work that does not conflict with theirs.  Give every worker
enough global context to understand how its unit fits the
whole task:  the whole decomposition, the contract facts,
what the other workers are doing, and how its output will be
integrated.  Correctness matters more than the number of
workers.  You may deviate from the decomposition if
execution reveals a better approach, but do not leave
clearly available validated parallelism unused without a
reason.  Integrate all work into one verified solution and
run the task's checks yourself.  Use only same-model
workers.
```

DAG-Sched (runtime scheduler). DAG-Sched has no orchestrator prompt. Each leaf receives the task instruction and the plan text, followed by the template below (fields in braces are filled by the scheduler). The Claude Code port uses the same template (checked in its leaf transcripts); Codex leaves run with multi-agent features off.

```
{instruction}
## Your assignment
You are one worker in a team executing the provided work
decomposition above.  A deterministic scheduler starts a
worker for each unit as soon as the units it depends on have
finished; no agent coordinates the team.  All workers share
this checkout and working directory, so edits by other
workers are visible to you immediately.
Your unit:  `{unit_id}` -- {title}
Implement only this unit, as its section in the provided
work decomposition describes.  The other units belong to
other workers:  do not implement them, and do not revert or
rewrite their changes.  Units already finished:  {finished}.
Units running now in parallel with you:  {running}.  When
your unit is complete and its unit check (if the
decomposition gives one) passes, end your turn with a short
summary of what you changed.
```

C.3 Task duration and interaction length

"Long-horizon" in this paper refers to the repository-level task and the extended agent interaction it requires. It does not mean that every reference DAG is deep: many released graphs are shallow fan-outs, and a long sequential-looking trajectory can conceal a shallow, wide work graph. Table 10 makes the interaction length concrete for the Single (single-agent) runs on the 104 released tasks (all verdicts, not only passes).

The duration of the seed successful trajectory from which each graph was constructed is recorded in the release manifest for only 52 of the 104 tasks (median 8.4 minutes, 10th to 90th percentile 3.5 to 21.9); we therefore use the evaluation's own Single runs above rather than claim full coverage for that field. We did not extract a separate count of repository interactions; tool calls are the closest available proxy.

D FOURTH SYSTEM: PI × KIMI K3

Setup. The harness is Pi 0.85.1; the model is the open-weights Kimi K3 served from a dedicated endpoint, with thinking set to high. MAS and MAS+DAG delegate through Pi's `delegate` tool, and DAG-Sched is the same scheduler on MAS+DAG's substrate as for the other systems. The run used the same 104-task slate, the same arm prompts (identical to Appendix C.2) and the same plan text. Every task ran on an image whose git history ends at the task's base commit, web tools were disabled, and container egress allowed only the model endpoint. Scoring follows the paper's rule: the tree is scored as it is, and an agent timeout counts as a fail. Total time runs from the first user message to the last assistant message over the parent and every worker or leaf; Pi stamps a message when it starts, so a run ends at the later of the last assistant message and the last worker or leaf exit. Cost is the list-price equivalent of the tokens used.

Results. Table 11 gives the full comparison with the paper's conventions. MAS delegated in 53% of PARALLEL cells (median 2 workers when it did), MAS+DAG in 87% (median 3), and DAG-Sched dispatched a median 4 leaves. MAS is not detectably faster than Single, MAS+DAG is not detectably faster either, and DAG-Sched is the fastest arm; DAG-Sched/MAS+DAG is 1.72× on trial time. By paired cost ratio MAS, MAS+DAG and DAG-Sched cost 1.06×, 0.97× and 1.00× Single's cost; by mean cost per cell over all 104 tasks, 1.02×, 0.88× and 0.88×. The 3 agent timeouts (2 in MAS+DAG, 1 in DAG-Sched) count as fails; the table caption gives the censored and reward-at-cutoff sensitivities, neither of which changes a conclusion.

Scope of the trace forensics. Pi × Kimi K3 enters every quantitative comparison of the paper, but the mechanism forensics (spawn latency, phase composition, the parallelism-loss catalog and the secondary mechanisms of Appendix F.4) cover only the three original systems; Pi × Kimi K3 was not audited at trace level.

Integrity audit. An offline audit of all 416 scored cells found no secret in any pulled file (0 signatures) and no MAS or MAS+DAG worker ending in an endpoint error (0). 150 cells attempted a network command; on review every attempt either failed at egress or was local (a repository clone from the task directory, a localhost server). 101 cells queried git history beyond `HEAD`; in all 77 unfiltered `git log --all` outputs the newest commit is the task's base commit (77 of 77). The DAG-Sched leaf guard blocked 9 tree-changing `git` commands. The run's infrastructure censoring (an empty endpoint key on the first launch, and a three-minute endpoint outage affecting three DAG-Sched cells) stopped cells before any model work; each affected cell was rerun once, and the rerun is the scored cell. Unlike the other systems, every Pi × Kimi K3 cell ran under the clean conditions, so its rows are not directly comparable in integrity terms, and the harness differs as well as the model.

D.1 SAME-MODEL ROBUSTNESS ACROSS HARNESSES

To check whether the ordering of the arms depends on the Pi harness, we ran Kimi K3 under Claude Code on the same dedicated endpoint, the same 100 PARALLEL tasks, the same arm prompts, plan text and DAG-Sched scheduler template, and the same agent time limits. Table 12 compares the two harnesses. Both reproduce the ordering: MAS stays near Single, MAS+DAG gives little speed gain (both intervals include one), and DAG-Sched is the fastest arm. The magnitude differs: the DAG-Sched/Single point estimate is lower under Claude Code, but the direct paired comparison on tasks where both cells passed in both harnesses includes one (0.90× [0.80, 1.03], $n = 69$), so this single-run evaluation does not rank the harnesses.

Several caveats apply. Run-to-run churn is large: 19 of the 78 Claude Code cells that were rerun for infrastructure reasons and had a valid first verdict changed verdict. The two harnesses ran at different

Table 11: Fourth system, Pi 0.85.1 × Kimi K3 (dedicated endpoint), on the 100 PARALLEL tasks, computed with the paper's conventions (both-pass geometric means, 95% task bootstrap; $N = 1$ per cell). Pass = passes / tasks; the 3 agent timeouts count as fails (§3). Sensitivity: with the timeouts censored instead, MAS+DAG passes 87/98 and DAG-Sched 92/99; scored by the reward at the cutoff, MAS+DAG passes 88/100 and DAG-Sched 92/100. Total time runs from the first user message to the last assistant message over the parent and every worker or leaf. Cost is the run's list-price equivalent.

arm	pass	Δ pass vs Single (pp)	speedup, trial	speedup, total	cost vs Single	n
Single	82/100	–	1	1	1	–
MAS	82/100	+0.0 [−6.0, +6.0]	0.96× [0.89, 1.03]	0.95× [0.88, 1.02]	1.06× [0.98, 1.14]	78
MAS+DAG	87/100	+5.0 [−4.0, +14.0]	1.02× [0.93, 1.11]	1.01× [0.91, 1.11]	0.97× [0.87, 1.07]	74
DAG-Sched	92/100	+10.0 [+2.0, +18.0]	1.78× [1.58, 2.00]	1.84× [1.62, 2.09]	1.00× [0.89, 1.13]	77
DAG-Sched vs MAS+DAG		+5.0 [−2.0, +12.0]	1.72× [1.58, 1.88]	1.79× [1.63, 1.97]	1.02× [0.96, 1.10]	83

Table 12: Same-model robustness across harnesses: Kimi K3 on one dedicated endpoint, 100 PARALLEL tasks, one run per cell. Pass: passes out of 100 tasks, no-verdict cells counted as fails. Speed: geometric mean over tasks where both cells passed of single-agent time over arm time, trial time unless noted; brackets: 95% task bootstrap; n: matched tasks. Cost: geometric mean over all tasks of arm cost over single-agent cost (list-price equivalent). Both columns come from the same robustness script, so the Pi intervals can differ from Table 21 in the second decimal.

	Claude Code × Kimi K3	Pi × Kimi K3
Pass Single / MAS / MAS+DAG / DAG-Sched	76 / 81 / 84 / 90	82 / 82 / 87[a] / 92[b]
MAS vs Single, trial	0.99× [0.92, 1.07], n=71	0.96× [0.89, 1.02], n=78
MAS+DAG vs Single, trial	1.09× [0.98, 1.21], n=68	1.02× [0.93, 1.11], n=74
DAG-Sched vs Single, trial	1.54× [1.38, 1.73], n=75	1.78× [1.57, 2.01], n=77
DAG-Sched vs Single, total time	1.61× [1.43, 1.83], n=75	1.84× [1.61, 2.10], n=77
DAG-Sched vs MAS+DAG, trial	1.46× [1.31, 1.63], n=80	1.72× [1.58, 1.87], n=83
Cost vs Single: MAS / MAS+DAG / DAG-Sched	1.09 / 0.91 / 1.03	1.05 / 0.97 / 0.95
MAS cells with no worker	50%	49%
MAS+DAG cells with no worker	20%	16%
DAG-Sched vs Single, trial, by reference work over span (exploratory)		
$W/S < 1.5$ (16 tasks)	1.21× [0.95, 1.46], n=8	1.43× [1.05, 2.00], n=6
$1.5 \leq W/S < 2.5$ (50)	1.43× [1.23, 1.64], n=39	1.63× [1.40, 1.90], n=40
$W/S \geq 2.5$ (34)	1.83× [1.50, 2.24], n=28	2.07× [1.71, 2.54], n=31

[a] Includes 2 agent timeouts counted as fails. [b] Includes 1 agent timeout counted as a fail.

times and endpoint loads, and Claude Code's per-call wall time is about 15% higher than Pi's in both Single and DAG-Sched, so absolute minutes are not comparable across harnesses. Claude Code needed an infrastructure rerun pass (96 cells, 23%, rerun once); every scored cell has no retry, but its DAG-Sched pass count depends on the attempt (85/100 on valid first attempts, 90/100 as scored). On one task (`rochacbruno__dynaconf-243`) the Claude Code MAS, MAS+DAG and DAG-Sched cells overlapped two watchdog outage windows; without that task every speed ratio moves by at most 0.01×. Finally, the harnesses differ in system prompt, tools and context handling; that difference is the manipulation, not a controlled factor, and the check says nothing about which harness feature matters. It did not test the trace mechanisms of §5.

E ADDITIONAL RESULTS

This appendix expands §4 with adjacent-arm comparisons, subgroups by horizon and width, and sensitivity analyses.

E.1 RESULTS

Table 21 gives every arm against Single and Table 13 the adjacent arms, on the 100 PARALLEL tasks, one run per cell, both-pass pairs, geometric means, with Codex DAG-Sched's 9 quota-affected cells replaced by their rerun (§C.1.3). Values are given in the order Codex, Fable 5, Opus 5.5. Success, time and cost are reported separately and never combined into one score.

Table 13: Adjacent arms on the PARALLEL tasks: what the plan text adds to the autonomous system (MAS+DAG/MAS) and what deterministic scheduling adds to the plan-informed system (DAG-Sched/MAS+DAG). Conventions as in Table 21: both-pass pairs, geometric means, 95% task-bootstrap intervals, $N = 1$ per cell. Δ = paired pass difference (pp) over all tasks, no-verdict cells counted as fails; n = speedup pairs (Claude Code cost pairs can be one lower).

comparison	Δ pass (pp)	speedup, trial	speedup, total	cost ratio	n
Codex × GPT-5.6 Sol					
MAS+DAG/MAS	+5.0 [−2.0, +13.0]	1.28× [1.19, 1.38]	1.31× [1.21, 1.43]	0.86× [0.80, 0.93]	79
DAG-Sched/MAS+DAG	+3.0 [−1.0, +8.0]	1.38× [1.29, 1.49]	1.57× [1.44, 1.73]	0.62× [0.59, 0.65]	88
Claude Code × Fable 5					
MAS+DAG/MAS	+8.0 [−1.0, +17.0]	1.21× [1.13, 1.30]	1.23× [1.14, 1.32]	0.88× [0.82, 0.95]	75
DAG-Sched/MAS+DAG	+4.0 [−3.0, +11.0]	1.47× [1.35, 1.60]	1.61× [1.45, 1.78]	1.06× [1.00, 1.13]	85
Claude Code × Opus 5.5					
MAS+DAG/MAS	+7.0 [+1.0, +13.0]	1.28× [1.18, 1.39]	1.31× [1.19, 1.44]	0.79× [0.72, 0.87]	83
DAG-Sched/MAS+DAG	−2.0 [−9.0, +5.0]	1.30× [1.18, 1.44]	1.37× [1.22, 1.56]	1.37× [1.26, 1.50]	85

E.1.1 AUTONOMOUS DELEGATION: HIGHER COST, NO DETECTED SPEEDUP

MAS is not detectably faster than Single on any system: MAS/Single is 0.98× [0.92, 1.05], 0.99× [0.93, 1.06] and 0.95× [0.88, 1.01] on trial time (n = 76, 74, 78) and 1.00×, 0.97× and 0.95× on total time. We read this as no speedup detected, not as equality. It costs more on every system, by an amount that depends on the harness: 2.06× [1.92, 2.21] Single's cost on Codex against 1.14× and 1.23× on the two Claude Code systems. The extra cost comes with extra uncached input (MAS/Single 2.32×, 1.51×, 1.39×); that the workers re-read the repository and that the Codex parent's context keeps growing is our inference from transcripts. MAS's pass rate does not differ detectably from Single's (+1.0, +0.0 and +2.0 points, every interval including zero). On Opus 5.5 the cells in which MAS delegated were slower than Single (0.85× on total time, 0.87× on trial time, $n = 50$) at 1.44× Single's cost, while those in which it did not were faster (1.16× total, 1.11× trial, $n = 28$); both groups are self-selected by the model.

E.1.2 THE PLAN AND ITS INSTRUCTIONS: FASTER THAN THE SINGLE AGENT ON THREE OF FOUR SYSTEMS

MAS+DAG differs from MAS in the plan text and in its orchestration instructions (concurrent launch, critical-path ownership, parent verification), so its gain is not attributable to the plan alone. On the three original systems MAS+DAG is faster than Single, on trial time (1.26× [1.18, 1.35], 1.18× [1.10, 1.26], 1.20× [1.12, 1.30]) and on total time (1.30×, 1.17×, 1.26×), and faster than MAS at lower cost (Table 13). On Pi × Kimi K3 it is not (MAS+DAG/Single 1.02× on trial time; Appendix D). Its cost relative to Single is system-specific: 1.80× on Codex, 1.02× and 0.96× on the two Claude Code systems. The success gain is positive on every system, but its interval excludes zero on one: MAS+DAG minus Single is +6.0 [−1.0, +13.0], +8.0 [−1.0, +17.0] and +9.0 [+1.0, +17.0] points; the Fable 5 lower bound is borderline and reaches zero under another bootstrap seed. Part of MAS+DAG's gain does not come from multi-agent execution at all (§F.1.5).

E.1.3 DETERMINISTIC SCHEDULING: THE FASTEST ARM, WITH TWO CONFOUNDS

DAG-Sched is the fastest arm on every system. Against Single it is 1.73× [1.60, 1.88], 1.78× [1.63, 1.96] and 1.58× [1.42, 1.76] on trial time and 2.02× [1.84, 2.23], 1.94× [1.75, 2.16] and 1.77× [1.56, 2.00] on total time (n = 80, 76, 80 both-pass pairs). The Codex total-time value is 2.02× over all-verdict pairs ($n = 100$) and 2.03× without the one cell served through a Codex CLI login ($n = 79$; §C.1.3); its trial-time value is 1.74× over all-verdict pairs, and the total and trial values are always printed together. Against MAS+DAG it is 1.38×, 1.47× and 1.30× on trial time and 1.57×, 1.61× and 1.37× on total time, every interval excluding one. Two confounds may inflate these speedups, and neither is resolved here. DAG-Sched has no parent, so apart from the plan's own units no agent runs the composed test suite: no full suite ran in 75%, 77% and 69% of DAG-Sched cells under the regular expression used for the taxonomy and in 66%, 67% and 62% under a broader detector, against 39%, 43% and 45% (regular expression) and 33%, 35% and 39% (broader detector) of Single cells. And the plan carries information that a single agent can use without any parallelism (§F.1.5). The benchmark verifier still scores every DAG-Sched cell, so the skipped suite affects

Table 14: Per-system subgroups (exploratory, no multiplicity correction; PARALLEL, both-pass pairs, $N = 1$ per cell). Cells: geometric mean (pairs). Plan width counts every plan unit, support units included (§A.1); it tracks the number of units (Spearman 0.90), so the plan-width trend is partly built into DAG-Sched. W is product work in minutes; the same three W bins are used for every horizon split. Depth one means no dependency between units. (a) DAG-Sched/Single on trial time. (b) DAG-Sched/MAS+DAG by W on trial and on total time, and on total time by reference work of the timed DAG (support units included) as a sensitivity. (c) Spearman correlations with 95% intervals; the timing of each column is named.

(a) DAG-Sched/Single, trial time

	by plan width				by W (min)			by plan depth	
system	≤ 2	3	4	5+	< 8	8–20	≥ 20	1	≥ 2
Codex	1.40× (11)	1.66× (19)	1.70× (26)	2.01× (24)	1.73× (27)	1.77× (45)	1.54× (8)	1.80× (63)	1.51× (17)
Fable 5	1.45× (9)	1.52× (20)	1.81× (24)	2.20× (23)	1.75× (29)	1.73× (40)	2.29× (7)	1.84× (64)	1.49× (12)
Opus 5.5	1.46× (12)	1.49× (19)	1.47× (24)	1.85× (25)	1.61× (27)	1.61× (46)	1.33× (7)	1.60× (65)	1.50× (15)

(b) DAG-Sched/MAS+DAG by horizon

	trial time, by W			total time, by W			total time, by reference work		
system	< 8	8–20	≥ 20	< 8	8–20	≥ 20	< 8	8–20	≥ 20
Codex	1.43× (31)	1.39× (49)	1.20× (8)	1.71× (31)	1.55× (49)	1.26× (8)	1.78× (22)	1.59× (47)	1.33× (19)
Fable 5	1.62× (31)	1.39× (48)	1.37× (6)	1.84× (31)	1.50× (48)	1.40× (6)	1.91× (21)	1.66× (47)	1.19× (17)
Opus 5.5	1.31× (30)	1.30× (48)	1.24× (7)	1.42× (30)	1.38× (48)	1.15× (7)	1.48× (23)	1.41× (43)	1.19× (19)

(c) Spearman ρ

system	DAG-Sched/Single trial, plan width	DAG-Sched/Single trial, W	DAG-Sched/Single trial, W/S	DAG-Sched/MAS+DAG trial, W	DAG-Sched/MAS+DAG total, W
Codex	0.24 [0.02, 0.45]	-0.10 [-0.32, 0.13]	0.22 [-0.00, 0.42]	-0.16 [-0.37, 0.06]	-0.24 [-0.43, -0.04]
Fable 5	0.38 [0.19, 0.56]	0.12 [-0.12, 0.34]	0.48 [0.27, 0.65]	-0.23 [-0.42, -0.02]	-0.28 [-0.46, -0.08]
Opus 5.5	0.19 [-0.02, 0.38]	0.06 [-0.17, 0.29]	0.15 [-0.08, 0.36]	-0.05 [-0.28, 0.17]	-0.10 [-0.32, 0.13]

the latency comparison, not the success measure: DAG-Sched's pass rate does not differ detectably from MAS+DAG's (DAG-Sched minus MAS+DAG +3.0 [−1.0, +8.0], +4.0 [−3.0, +11.0], −2.0 [−9.0, +5.0] points) and is above Single's in point estimate on every system (+9.0 [+2.0, +17.0], +12.0 [+3.0, +21.0], +7.0 [+0.0, +14.0] points); the Codex and Fable 5 intervals exclude zero and the Opus 5.5 interval includes zero. Codex DAG-Sched passes 92/100 with the rerun cells (§C.1.3). These pass comparisons also rest on an evaluation set from which the 7 graphs whose compiled run exposed a hidden dependency were removed, a rule that favours DAG-Sched (§C.1.1).

Cost. DAG-Sched's cost premium over Single is modest on two systems and not on the third: 1.14× [1.05, 1.24], 1.10× [1.01, 1.21] and 1.32× [1.21, 1.46]. Against MAS+DAG it depends on the harness: DAG-Sched costs 0.62× MAS+DAG's cost on Codex and 1.06× and 1.37× on the two Claude Code systems. Our reading is that Codex MAS+DAG pays for a growing parent context that DAG-Sched does not have, while each Claude Code DAG-Sched cell pays for several fresh leaf contexts. DAG-Sched uses no more input tokens than Single (0.90×, 0.78×, 0.95×) but more uncached input (1.75×, 1.66×, 1.76×); that cold leaves share no cached prefix is our inference.

How much structure becomes latency. The median ratio of a task's speedup to its reference W/S is 1.02, 0.94 and 0.92 for DAG-Sched on total time (0.89, 0.85 and 0.81 on trial time), 0.70, 0.62 and 0.65 for MAS+DAG (0.67, 0.63, 0.58) and 0.49, 0.51 and 0.49 for MAS (0.48, 0.50, 0.47). W/S describes one tested decomposition under single-sample durations and is not an upper bound: DAG-Sched's total-time speedup exceeds the task's W/S on 51%, 47% and 44% of both-pass tasks (40%, 42% and 36% on trial time), because W sums fresh-worker replays that a single agent does not match task by task. DAG-Sched's value may also include the two confounds above; the ordering, not the level, is the finding.

When MAS+DAG beats DAG-Sched. DAG-Sched's total time is more than 1.25× MAS+DAG's on 4, 4 and 15 tasks (all-verdict pairs). A straggling unit is DAG-Sched's primary label in 8 of the 15 Opus 5.5 tasks; the case studies of §F.1.4 show a unit whose own checks could not pass before a dependent unit landed, and exit-137 kills (likely out of memory) during concurrent builds.

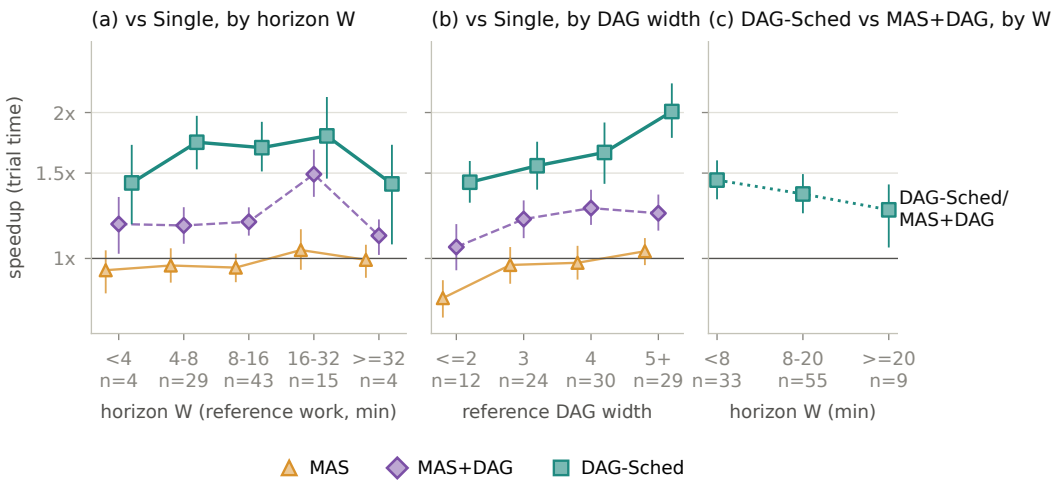


Figure 17: Trial-time speedup by task structure, pooled over the three systems (points: geometric means over both-pass system–task pairs; bars: 95% intervals resampling tasks with all systems of a task together; n = tasks per bin). (a) MAS, MAS+DAG, DAG-Sched against Single by horizon W (product work; the axis label "reference work" means W, and the five display bins are coarser than the three analysis bins of Table 14): flat. (b) Against Single by plan width (the axis label "reference DAG width"): the number of plan units, support units included, that can run at once, which is what DAG-Sched parallelizes; rising for DAG-Sched. Plan width follows the number of units (Spearman 0.90) and DAG-Sched runs one agent per unit, so part of this trend is built into DAG-Sched. (c) DAG-Sched against MAS+DAG by W: DAG-Sched's lead narrows on longer tasks, but is still above one in the top bin (1.26× pooled, n as printed). Exploratory subgroups, no multiplicity correction; per-system values in the figure's sidecar and Table 14. $N = 1$ run per cell. DAG-Sched's gain goes with plan width rather than horizon, and DAG-Sched's lead over MAS+DAG narrows on the longest tasks.

E.1.4 Horizon, width and the DAG-Sched/MAS+DAG gap

The per-system values behind Figure 17 are in Table 14. Width here is plan width, support units included (§A.1). DAG-Sched/Single rises from plan width two or less (1.40×, 1.45×, 1.46×) to plan width five or more (2.01×, 2.20×, 1.85×) on trial time, and its rank correlation with plan width (0.24 [0.02, 0.45], 0.38 [0.19, 0.56], 0.19 [-0.02, 0.38]) excludes zero on Codex and Fable 5 and includes it on Opus 5.5. With W/S the association is weaker on Codex and Opus 5.5 and stronger on Fable 5: the correlation of DAG-Sched/Single (trial) with W/S is 0.22 [-0.00, 0.42], 0.48 [0.27, 0.65] and 0.15 [-0.08, 0.36], and its interval excludes zero on Fable 5 only. By depth, DAG-Sched/Single on trial time is 1.80×, 1.84× and 1.60× on the 80 depth-one tasks and 1.51×, 1.49× and 1.50× on the 20 deeper ones ($n = 17, 12, 15$ pairs). Plan width also tracks work over span (Spearman 0.52 [0.37, 0.65]) and node count (0.90), and DAG-Sched starts one agent per unit, so the trend is exploratory and partly built into DAG-Sched. Across the three W bins DAG-Sched/Single is flat, and no correlation with W excludes zero (trial time -0.10 [-0.32, 0.13], 0.12 [-0.12, 0.34], 0.06 [-0.17, 0.29]); we do not read this as "horizon does not matter", because DAG-Sched's lead over MAS+DAG falls in the longest W bin on all three systems (Spearman with W on total time -0.24 [-0.43, -0.04], -0.28 [-0.46, -0.08] and -0.10 [-0.32, 0.13]; on trial time -0.16 [-0.37, 0.06], -0.23 [-0.42, -0.02] and -0.05 [-0.28, 0.17]; the interval excludes zero on Codex and Fable 5 on total time and on Fable 5 only on trial time; the same direction holds with reference work, support units included). Pooled over systems, DAG-Sched/MAS+DAG on trial time is 1.45×, 1.36× and 1.26× across the three W bins (Figure 17c; the top bin holds 21 system–task pairs). Measured reference work is moderate (median W 10.0 isolated-replay minutes), so these trends describe work graphs of minutes to tens of minutes. W is a property of one decomposition, not the task horizon: the tasks are long-horizon in the sense of repository-level work and extended agent interaction (Appendix C.3).

E.1.5 Sensitivity and controls

The speed ordering DAG-Sched > MAS+DAG > Single survives the pairing, host, killed-cell and CLI-login-cell sensitivities (Appendix L), and MAS/Single stays without a detected difference. All-verdict pairing changes no conclusion (Table 25). On the Opus 5.5 pairs that ran

Table 15: Outcomes of Single against each other arm on the 100 PARALLEL tasks: both pass / Single only / other arm only / neither.

system	Single vs MAS	Single vs MAS+DAG	Single vs DAG-Sched
Codex × GPT-5.6 Sol	76 / 7 / 8 / 9	79 / 4 / 10 / 7	80 / 3 / 12 / 5
Claude Code × Fable 5	74 / 7 / 7 / 12	75 / 6 / 14 / 5	76 / 5 / 17 / 2
Claude Code × Opus 5.5	78 / 5 / 7 / 10	79 / 4 / 13 / 4	80 / 3 / 10 / 7
Pi × Kimi K3	78 / 4 / 4 / 14	74 / 8 / 13 / 5	77 / 5 / 15 / 3

on one host, DAG-Sched/Single is 1.49× ($n = 38$), MAS+DAG/Single 1.28× ($n = 45$) and DAG-Sched/MAS+DAG 1.27× [1.03, 1.60] ($n = 19$) on trial time, against 1.58×, 1.20× and 1.30× on all hosts. On pairs whose two trials started on the same UTC day, DAG-Sched/Single on trial time is 1.68× ($n = 32$), 1.76× and 1.51× and MAS+DAG/Single 1.23×, 1.17× and 1.20×, each equal to or slightly below its all-pairs value (Appendix L). On the 4 width-one controls, reported descriptively, DAG-Sched/Single on trial time is 1.20× on Codex ($n = 3$), 1.48× on Fable 5 and 0.99× on Opus 5.5 ($n = 4$). These graphs have product width one (support units can still run beside the product unit), so this is not a claim about parallel product work.

E.2 PAIRED OUTCOME CONTINGENCY

Latency and cost ratios in the main text are computed over task pairs in which both compared arms passed; pass rates are computed over all tasks. Table 15 shows how the pairs split, so that a reader can see which tasks enter the conditional ratios. Each cell is one run ($N = 1$). As in Table 21, a cell with no verdict (the Codex MAS safety refusal and the 3 Pi × Kimi K3 agent timeouts) counts as a fail, so every row sums to 100.

In every system the other-arm-only count for DAG-Sched is at least as large as the Single-only count, so conditioning the DAG-Sched ratios on both-pass pairs does not drop a block of tasks that DAG-Sched failed and Single passed. The conditional ratios still describe efficiency on tasks both arms solved, not speed at equal accuracy.

F TRACE ANALYSIS: PHASES, MECHANISMS AND THE FULL CATALOG

This appendix expands §5. The catalog was derived after the traces had been read; it is exploratory, its buckets overlap and are not additive, and every association with latency is observational.

F.1 WHERE THE TIME GOES

This section explains the ordering of §4 from the traces. Every quantity is measured from transcript and scheduler timestamps on the total-time clock unless stated; case studies are single runs and are labelled as such; the taxonomy of §F.1.6 was derived after the traces had been read and is exploratory.

F.1.1 A MEASURED PARTITION OF MODEL-INITIATED RUNS

In a delegated run of a model-initiated arm, total time splits into three parts: the parent alone before its first spawn, the worker window, and the parent alone after its last worker (Figure 4a). The parent-only phases take a mean 44%, 66% and 50% of total time in MAS ($n = 100$, 80, 69 delegated cells) and 40%, 57% and 51% in MAS+DAG ($n = 100$, 91, 52). The split between the two bookends depends on the harness: Codex spawns early and keeps a long tail (MAS prefix 11%, tail 33%), while Fable 5 spends more before the first spawn (MAS prefix 42%, tail 24%); Opus 5.5 MAS splits more evenly (prefix 24%, tail 26%). This is a partition by construction, not a counterfactual saving: parent-only time also contains real implementation work, including cases where the parent keeps the largest unit for itself, and we never call it lost time.

F.1.2 MODEL-INITIATED AGAINST RUNTIME-INITIATED SPAWNING

Model-initiated spawning. MAS delegated in 100%, 81% and 69% of cells and MAS+DAG in 100%, 91% and 52% (Figure 18a). The median first spawn came at 41.6, 281.9 and 104.4 s in

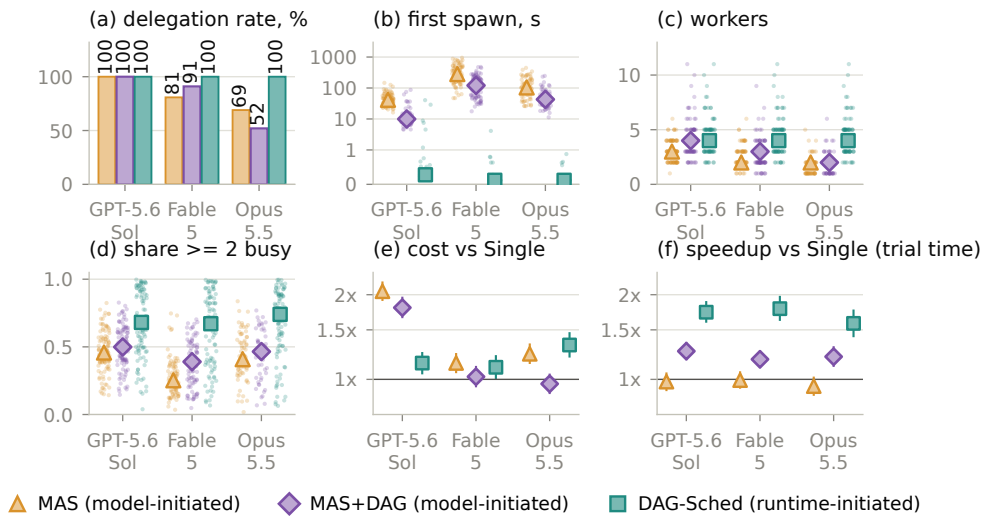


Figure 18: Model-initiated (MAS, MAS+DAG) against runtime-initiated (DAG-Sched) spawning, three original systems. (a) Share of cells with at least one worker. (b)–(d) Medians over delegated cells (dots: cells) of the first spawn, the number of workers or units, and the share of total time with at least two agents busy, parent included. (e)–(f) Cost and trial-time speedup against Single (geometric means over both-pass pairs, 95% intervals). For DAG-Sched, first spawn is the unit process launch; for MAS and MAS+DAG, the worker's first model call, so the two are not directly comparable. The runtime dispatches at once and keeps more agents busy; Codex delegates in every cell and early, the Claude Code systems selectively and late. $N = 1$ run per cell.

MAS and 9.9, 120.9 and 42.9 s in MAS+DAG. Active arms started fewer workers than the plan has units in most Claude Code cells. The median share of total time with at least two agents busy, parent included, was 0.46, 0.25 and 0.41 in MAS and 0.50, 0.39 and 0.46 in MAS+DAG. This metric counts the parent because the parent often works beside its workers; a workers-only share understates the model-initiated arms and is never set against DAG-Sched.

Runtime-initiated spawning. DAG-Sched's scheduler dispatched its first unit a median 0.3 s (Codex), 0.1 s (Fable 5) and 0.1 s (Opus 5.5) after it started; 7 of all 312 DAG-Sched cells (controls included) needed more than a second (at most 40.5 s). With no parent, the share of total time with at least two agents busy was 0.68, 0.67 and 0.74. The data support the statement that these tasks contain parallelism that the model-initiated orchestrators we tested do not expose efficiently, subject to the verification and plan-information confounds of §F.1.5. Runtime-initiated spawning exposes the whole width, and with it the coupling, oversized units, cold starts and contention of the plan's units, and no agent integrates the result (§F.1.4).

F.1.3 What the plan text changes for a model-initiated orchestrator

Given the plan, the model-initiated orchestrator spawns earlier: the median first spawn moves 4.2×, 2.3× and 2.4× earlier from MAS to MAS+DAG, and the parent-only prefix shrinks (median prefix share 10% to 3%, 44% to 25%, 22% to 11%). The tail's share of total time does not shrink: it is 36%, 24% and 22% in MAS against 34%, 26% and 32% in MAS+DAG (medians). In seconds, on tasks where both arms delegated, the median tail shortens on Codex (131 to 108 s, $n = 100$), is unchanged on Fable 5 (140 to 141 s) and grows on Opus 5.5 (72 to 131 s). The first-spawn ratios are ratios of medians over the delegated cells of each arm, not paired. Opus 5.5 delegated less often with the plan than without it (52% against 69%), but every Opus 5.5 MAS cell also carried the old eight-agent prompt sentence and no MAS+DAG cell did (§C.1.2), so this difference is not attributable to the plan alone. On one Codex task (`hydra-zen`, $N = 1$), MAS spawned at 80 s after a full test run and ended at 469 s (trial 498 s) with a 219 s tail, while MAS+DAG spawned at 10 s and ended at 192 s (trial 224 s) with a tail of 86 s.

F.1.4 Replacing the model-initiated orchestrator by a runtime scheduler

Where DAG-Sched's gain over MAS+DAG comes from. On tasks where MAS+DAG delegated and both arms passed, MAS+DAG's worker window alone is 0.89× [0.81, 0.98], 0.63× [0.53, 0.74] and 0.67× [0.51, 0.87] of DAG-Sched's whole run ($n = 88$, 78, 39; geometric mean), and DAG-Sched is 1.57×, 1.64× and 1.51× faster than those MAS+DAG runs on total time (1.38×, 1.50× and 1.44× on trial time; Figure 4b). The window ratio is defined on total time only. The geometric mean hides spread: the window is longer than DAG-Sched's whole run in 38%, 29% and 36% of the pairs. We infer from the partition, without a counterfactual run, that most of DAG-Sched's gain over MAS+DAG is the removal of the parent-only phases; DAG-Sched also verifies less often (§F.1.5).

What DAG-Sched pays. Among the 100 DAG-Sched cells per system (Codex, Fable 5, Opus 5.5), the post-hoc labels flag shared-tree interference in 28, 20 and 26, coupled units run as if independent in 15, 27 and 33, a straggling unit (usually the task's largest reference unit running near its replay time, Appendix F.4.2) in 22, 32 and 19, cold re-exploration in 3, 24 and 15, and a failed composition in which every unit had succeeded in 8, 6 and 10 cells; DAG-Sched never retries a unit. Most audited DAG-Sched failures (15 of 26) are one unit missing an edge case inside its own scope; only 1 is a cross-unit interface disagreement, and MAS+DAG, which has a composing parent, passes only 5 of those 15 tasks. The labels overlap and their recall is limited (§F.1.6).

What the model-initiated parent keeps (case studies, $N = 1$ each). On Fable 5 `kombu`, MAS+DAG's warm parent implemented two coupled units in sequence itself and finished in 492 s (trial 557 s). In DAG-Sched the upstream unit ran alone for most of the run because its own checks could not pass until the dependent unit, which the scheduler starts only afterwards, had landed; DAG-Sched took 960 s (trial 1026 s). The implementation edge is correct; the coupling is in verification order, which the plan cannot express, and the same shared-state hang also slows MAS's worker on this task (Appendix F.4.1). On Opus 5.5 `flipt`, the 4 concurrent units' Go builds in one 2-CPU, 8-GiB container were killed with exit code 137 (likely out of memory), and DAG-Sched took 1824 s against 715 s for MAS+DAG, which did not delegate (trial 2017 and 854 s). On Opus 5.5 `remeda`, 4 small units each ran the full test suite and DAG-Sched took 565 s against 183 s (trial 724 and 352 s). The trace analysis also records the parent of a MAS+DAG run implementing the critical unit faster than a cold DAG-Sched leaf, repairing support files that no unit owns, and running the composed suite. The opposite case exists too: on Opus 5.5 `testem`, MAS+DAG spent 341 s of its 588 s in the tail and DAG-Sched finished in 71 s (trial 632 and 111 s; DAG-Sched/Single 6.64× on total time and 4.63× on trial time for that task). Appendix K lists all ten annotated traces.

F.1.5 Two confounds of the scheduler's gain

Skipped self-verification. DAG-Sched runs the composed test suite less often than Single (§E.1.3), and the exact shares depend on the detector (a regular expression over commands, with known false negatives; we report two). Split by whether the DAG-Sched cell ran a full suite, DAG-Sched/Single on total time is 2.09× (skipped, $n = 62$) against 1.80× (ran, $n = 18$) on Codex, 2.01× against 1.72× on Fable 5, and 1.67× against 2.09× on Opus 5.5, where the split reverses; on trial time the same split is 1.75× against 1.66×, 1.83× against 1.64× and 1.48× against 1.89×. Under the broader detector the Codex gap nearly vanishes on total time and is gone on trial time (2.05× against 1.95× on total time, 1.73× against 1.74× on trial time). This is an observational split, confounded with the tasks on which DAG-Sched skips, not an effect estimate (Appendix K). Skipped verification may inflate DAG-Sched's speedup; it does not change DAG-Sched's success, which the benchmark verifier scores in every cell.

Plan information. The plan names files, contracts, the toolchain and often the mechanism, and a single agent can use that without delegating. The cleanest proxy we have is the first edit of MAS+DAG cells in which the model did not delegate: on the same tasks, the median over tasks of the ratio of Single's first edit to theirs is 1.23× on Fable 5 ($n = 9$) and 1.57× on Opus 5.5 ($n = 40$), against 1.88× and 2.05× for delegated MAS+DAG. First edit is a proxy, and the non-delegated groups are self-selected. DAG-Sched's own median first edit (15, 17 and 17 s) is the minimum over at least four agents started at once with narrow scopes, an order statistic, so we do not use it as plan-information evidence.

Table 16: Arm MAS+DAG split by whether the model delegated, each against Single on the same tasks (PARALLEL). Pass = MAS+DAG / Single pass rate on the group's tasks; speedup and cost: both-pass geometric means with 95% task-bootstrap intervals; n = both-pass pairs. The split is chosen by the model, so it is observational and self-selected: Opus 5.5 did not delegate on tasks that Single also found easier (Single passes 92% of them against 75% of the delegated ones). Codex delegated in every cell. $N = 1$ per cell.

MAS+DAG cells	cells	pass MAS+DAG / Single	speedup, trial	speedup, total	cost vs Single
Codex × GPT-5.6 Sol					
delegated	100	89% / 83%	1.26× [1.18, 1.35]	1.30× [1.21, 1.41]	1.80× [1.66, 1.95] (n=79)
not delegated	0	n/a	n/a	n/a	n/a
Claude Code × Fable 5					
delegated	91	89% / 80%	1.19× [1.11, 1.28]	1.17× [1.09, 1.26]	1.04× [0.95, 1.13] (n=67)
not delegated	9	89% / 89%	1.10× [0.94, 1.29]	1.17× [0.95, 1.44]	0.92× [0.73, 1.14] (n=8)
Claude Code × Opus 5.5					
delegated	52	87% / 75%	1.25× [1.09, 1.43]	1.29× [1.10, 1.50]	1.12× [0.99, 1.25] (n=35)
not delegated	48	98% / 92%	1.17× [1.08, 1.27]	1.25× [1.13, 1.38]	0.86× [0.79, 0.93] (n=44)

MAS+DAG delegated and not delegated. Table 16 splits MAS+DAG by whether the model delegated. Codex delegated in every cell; Fable 5 declined in 9 cells and Opus 5.5 in 48. Where Opus 5.5 MAS+DAG did not delegate it was 1.25× [1.13, 1.38] faster than Single on total time and 1.17× on trial time ($n = 44$) at 0.86× Single's cost, and passed 47 of those 48 tasks against 44 for Single (paired difference +6.2 [+0.0, +14.6] points, touching zero); DAG-Sched also passed 47 of them. These are easier, self-selected tasks (Single passes 92% of them against 75% of the delegated ones). We therefore claim only that not every gain of MAS+DAG is a multi-agent gain. We do not claim that this subgroup is Opus 5.5's best, nor that it is faster than delegated MAS+DAG: that order reverses between the both-pass and the all-verdict pairing.

The control that would separate information from parallelism (proposed, not run). An Single run given the same plan with delegation disabled (Single+PLAN) would split DAG-Sched/Single into an information part and a parallel part and give non-delegated MAS+DAG a matched control. The smallest useful version is 40 PARALLEL tasks on all three systems, stratified by Opus 5.5 MAS+DAG delegation, width and DAG-Sched/Single speedup, at an estimated $418 using Single's mean cost per cell ($1,045 for all tasks on all systems); the self-selected non-delegated Opus 5.5 subset alone would not do.

F.1.6 A POST-HOC TAXONOMY OF INEFFICIENCIES (EXPLORATORY)

To name what the partition does not, one workstream labelled every MAS, MAS+DAG and DAG-Sched cell bottom-up after reading the traces: rules over extracted events plus 101 cells read by hand (Table 24, Appendix K). The categories were not fixed before the traces were seen and are not the pilot taxonomy of the 2026-09-22 plan (Appendix T); they co-occur and overlap in time, so no share of any arm's time is attributed to a category, and flagged-against-unflagged contrasts are observational. The most frequent primary labels over the 900 multi-agent cells are the integration tail (218), the serial prefix (111) and no delegation (103), the last often a rational choice rather than a loss. Among the worst MAS runs (total time above 1.25× Single's), the dominant primary label depends on the harness: the integration tail in 16 of 23 Codex cells, the serial prefix in 15 of 32 Fable 5 cells, and on Opus 5.5 the serial prefix (8) and the integration tail (8) equally often among 25 cells.

Two starting hypotheses were not supported. *Duplicated context*: workers re-read the repository rather than inheriting a forked history (§C.1.2). *Contention as a general tax*: only 3%, 4% and 5% of heavy build or test calls in MAS, MAS+DAG and DAG-Sched overlap another heavy call ($n = 9328$ calls), although a heavy call is slower when it does (matched slowdown 1.11× with one concurrent heavy call, 1.59× with two, 2.25× with three or four), and a few DAG-Sched cells are catastrophic for this reason (cause inferred; no kernel logs).

F.1.7 PROPOSED FIXES (NONE RUN)

Table 17 maps each recurring inefficiency to an intervention and labels it as a harness fix (the harness misbehaves or mislabels), an orchestration-policy change (the model-initiated orchestrator decides

Table 17: Proposed interventions for the inefficiencies of §F.1.6 (none run). Type: *harness* = harness fix; *policy* = orchestration-policy change; *arm* = new experimental arm; *future* = future work on DAG construction. Test sizes and cost estimates are in the trace-analysis report (`docs/orchestration-trace-analysis-20260925.md`, section 10).

inefficiency	intervention	type
integration tail, duplicate verification, parent rework	scope-limited verification: targeted tests on touched modules, no repeated full-suite runs without new evidence, and waiting on completion events rather than a timer	policy
serial prefix, parent keeps the critical unit	spawn first: dispatch the ready frontier in the first turns and delegate the critical unit	policy
no integrator, failed unit, skipped verification	DAG-Sched plus an integrator unit that composes, runs the checks once and fixes; retry a failed unit once	arm
cold worker	a context digest (files read, key facts) with each spawn or unit	harness
fixed-sleep polling	notification-driven waiting	harness
shared-tree interference	per-agent worktrees with automatic merge; per-agent temporary directories	harness
false independence, straggler, DAG-Sched's narrowing lead on long tasks	certify unit independence, add missing edges, repartition a unit only where replay shows needless bundling, start the critical path first	future
resource contention	a per-container semaphore on heavy build and test commands, or larger DAG-Sched containers	harness / arm
harness artifacts	scan worker streams for API errors and rerun the affected cells (done for the 9 quota cells, §C.1.3), label exit-137 kills correctly	harness
parent exits before its background workers	block the final answer until background workers report	harness
plan information against parallelism	Single+PLAN, delegation disabled (§F.1.5)	arm
verification confound	a DAG-Sched arm with a mandatory final composed-suite unit, or DAG-Sched timed with a verifier-equivalent step	arm

differently), a new experimental arm, or future work on DAG construction. None has been run; each would be tuned on a declared diagnostic split and evaluated once on held-out tasks (Appendix T), and the trace analysis gives a test size and a cost estimate for each.

F.2 IDEAL MODEL-INITIATED AND RUNTIME-INITIATED SYSTEMS

None of the arms is ideal. An ideal model-initiated MAS would spawn the independent critical units right after minimal understanding, keep workers balanced and warm, and verify once (Figure 3); an ideal runtime-initiated MAS would run a validated graph with scoped context, respect resources, and end with an explicit integrator and one verifier-equivalent check. MAS and MAS+DAG implement the first family and DAG-Sched the second.

Cases in which a warm parent beats DAG-Sched and the contention cases are annotated in Appendix K; in Fable 5 `flipt`, concurrent Go builds were killed with exit code 137 (likely out of memory) and DAG-Sched took 819 s against 532 s for a non-delegating MAS+DAG.

F.3 THE FULL CATALOG OF PARALLELISM-LOSS BUCKETS

Table 18: The full post-hoc catalog of 114 parallelism-loss buckets (exploratory; derived after reading the final traces). Flagged/eligible counts pool the PARALLEL cells of all three systems; the eligible set is bucket-specific (for example, delegated MAS/MAS+DAG cells). Buckets overlap in time and are not additive; *audited* buckets count cells read by hand. Per-system counts, detector rules, overlaps and observational effects are in the machine-readable catalog.

bucket	type	definition	flagged/elig.
delegation			
`NO_DELEGATION`	symptom	A model-initiated (MAS or MAS+DAG) parent never spawns a worker; the cell runs as one agent.	107/598
`CORRECT_NO_DELEGATION`	task property	No delegation on a task where the evidence says little usable parallelism was available.	80/598
`MISSED_PARALLELISM`	root cause	No delegation although the same task's DAG-Sched cell converted the DAG into a large, successful speedup.	22/598
`UNDER_DELEGATION`	root cause	The parent delegates, but spawns clearly fewer workers than the DAG's width while the same task's DAG-Sched cell is much faster.	123/491
`PARENT_RETAINS_CRITICAL_WORK`	contributing cause	The delegating parent does most of the busy work itself (implementation on the parent's thread), so the run stays close to serial.	131/491
`DELEGATES_ONLY_SUPPORT_WORK`	root cause	Every worker gets a support unit (tests, docs, review, probes) while the parent keeps the implementation (wrong delegation target).	69/491

bucket	type	definition	flagged/elig.
OVER_DELEGATION	root cause	Several workers on a task that the single agent finishes quickly; coordination overhead exceeds any parallel gain.	10/491
SERIAL_PREFIX	symptom	Phase: the parent works alone for a long time before the first worker starts.	176/491
PREFIX_SCAFFOLDING	contributing cause	Part of the prefix is setup the workers need: the parent edits shared code/contract files before fanning out (likely necessary setup).	56/491
PREFIX_PRE_SPAWN_VERIFICATION	contributing cause	The parent runs builds/tests (often the full suite) before it spawns anyone.	18/491
AVOIDABLE_LATE_SPAWN	root cause	A long read-only exploration prefix although a cold plan-guided worker (same task, DAG-Sched) was already editing much earlier.	112/491
SERIAL_SPAWN_LATENCY	contributing cause	Workers of one fan-out start staggered: in Claude Code the Agent calls of a fan-out mostly sit in ONE assistant response, each call starts its worker when its block finishes streaming, and each block carries a long hand-written brief, so the starts are spaced by brief-generation time (612 of 688 Agent calls share a message id; spawn_messages.py).	83/491
SYNC_SPAWN_BLOCKING	harness property	A Claude Code parent uses a foreground (blocking) Agent call, so it cannot work or spawn more while that worker runs.	0/491
UNIT_BUNDLING	contributing cause	The parent gives one worker several plan units. Seen in 5/16 audited under-delegation cells; 0/32 audited stragglers were caused by it.	5/16
PREFIX_NECESSARY_SETUP	task property	Audited judgement that a serial prefix was needed to decompose / to build what the workers need.	9/29
PREFIX_AVOIDABLE_AUDITED	root cause	Audited judgement that a serial prefix was avoidable (e.g. the plan already gave the information).	8/29
coordination			
WORKTREE_ISOLATION	harness property	A Claude Code parent spawns workers with isolation: worktree, so they edit separate git worktrees instead of the shared checkout and the parent must merge their files back (the one exception to 'MAS and MAS+DAG share one checkout').	13/292
DUPLICATE_DEFINITION_COLLISION	root cause	Two agents define the same symbol (Go 'redeclared in this block' etc.) in the shared tree, breaking the build for everyone.	8/791
INTEGRATION_TAIL	symptom	Phase: the parent works alone for a long time after the last worker ends.	281/491
TAIL_VERIFICATION	contributing cause	Much of the tail is the parent's own builds/tests.	227/491
TAIL_REWORK	contributing cause	In the tail the parent re-edits files that workers produced.	114/491
TAIL_LATE_IMPLEMENTATION	contributing cause	In the tail the parent writes files that no worker touched (unowned support files, leftover units, extra tests).	140/491
TAIL_REVIEW_ONLY	contributing cause	The tail has no parent edit and little testing: it is diff review, reading and model time.	49/491
TAIL_SLEEP_OVERHANG	contributing cause	Part of the tail is the parent sitting in a fixed sleep after the last worker has already ended (a sleep that spans the worker's end, or one started after it before the parent noticed); added after the tail audit.	54/491
TAIL_BASELINE_COMPARISON	contributing cause	In the tail the parent checks the original code (git stash / checkout / baseline worktree) to show that failures pre-exist (added after the tail audit).	81/491
TAIL_MODEL_DOMINATED	contributing cause	Most of the tail is the parent's model time (reasoning / generation), not tools.	144/491
PARENT_REWORK	contributing cause	The parent re-edits a file after the worker that produced it has finished.	169/491
PARENT_REWORK_SUBSTANTIAL	contributing cause	Sub-case: rework touches >= 2 worker files or >= 3 parent edit events after the workers.	108/491
PARENT_WORKER_SAME_FILE_CONCURRENT	contributing cause	The parent edits a file while the worker that also edits it is still running (patch-collision risk).	12/491
MULTI_AGENT_SAME_FILE	contributing cause	Two or more workers/units edit the same repository file.	88/791
EDIT_APPLY_FAILURE	symptom	An edit fails because the file no longer matches what the agent read (Codex 'Failed to find expected lines', Claude Code 'modified since read').	68/1199
PATCH_COLLISION	contributing cause	Sub-case of EDIT_APPLY_FAILURE: the failing file had been edited by another thread earlier (a sibling or the parent changed it under the agent).	16/899
SHARED_TREE_DESTRUCTIVE_GIT	root cause	An agent runs git stash / checkout – / restore / reset –hard / clean on the shared checkout while another agent is running.	71/791
POLL_WAIT	symptom	The parent waits with fixed sleeps or notices finished workers late.	100/491
FIXED_SLEEP_OVERSHOOT	contributing cause	Sub-case: the parent's fixed sleep outlasts the last worker (the worker finished while the parent was sleeping, or the parent starts a fixed sleep after the last worker has finished).	67/491
PARENT_IDLE_WAIT_DOMINANT	symptom	The parent spends most of the worker window blocked in wait_agent / idle instead of working on the critical path.	96/491
PEER_MESSAGING	positive mechanism	Workers message each other directly (Codex send_message to a sibling) to negotiate contracts.	39/491
D_NO_INTEGRATOR	root cause	In DAG-Sched every unit succeeds but the combined tree fails the verifier; nobody reconciled or ran the composed checks.	23/300
D_UNIT_FAILURE	root cause	In DAG-Sched a unit exits non-zero / incomplete; its descendants are blocked and nothing retries.	3/300

bucket	type	definition	flagged/elig.
LEFTOVER_SCRATCH_TEST_COLLISION	root cause	An agent-written test file collides with the hidden test at verification (symbol redeclared / duplicate), failing the build.	4/1200
WORKER_SPEC_MISS	root cause	A unit implements one requirement inside its own scope wrongly or not at all; with no integrator nobody catches it (outcome-side cause of D_NO_INTEGRATOR failures).	16/26
CROSS_UNIT_INTERFACE_MISMATCH	root cause	Units' outputs do not fit together at a shared interface (DAG-Sched failure cause).	1/26
TAIL_PARENT_OWN_UNIT	contributing cause	The measured 'integration tail' is really the parent implementing the core unit it kept after its only (support) worker finished: serial implementation, not integration.	7/36
TAIL_REWORK_WORKER_BUG	contributing cause	In the tail the parent fixes a real bug in a worker's code.	7/36
TAIL_REWORK_INTERFACE_MISMATCH	contributing cause	In the tail the parent reconciles worker outputs that do not fit together.	2/36
TAIL_REWORK_STYLE_ONLY	symptom	Tail 'rework' that is only formatting or small cleanups (explains why TAIL_REWORK overstates substantive rework).	8/36
WORKER_OUTPUT_REDONE	root cause	The parent discards or rewrites a worker's work. NOT OBSERVED in the 36 audited tails.	0/36
SHARED_TREE_DESTRUCTIVE_GIT_HARMFUL	root cause	Audited version of SHARED_TREE_DESTRUCTIVE_GIT with visible harm (a sibling's edit reverted / an edit failing 'modified since read' / a stash-pop conflict).	3/18
scheduling			
STRAGGLER	symptom	One worker/unit finishes far later than the others, and the run waits for it.	123/721
STRAGGLER_CAUSE_TOOL_HANG	contributing cause	The straggler had a single tool call of >= 115 s or hit the Bash 2-minute cap.	21/721
STRAGGLER_CAUSE_SIBLING_WAIT	contributing cause	The straggler waited for a sibling's output (a polling loop on a file / process / check, or it said it was waiting for / blocked by another unit).	2/721
STRAGGLER_CAUSE_COLD_START	contributing cause	The straggler spent at least half its lifetime before its first edit (exploration).	52/721
STRAGGLER_CAUSE_VERIFICATION	contributing cause	The straggler spent >= 40% of its lifetime in test/build tool calls.	10/721
STRAGGLER_CAUSE_OVERSIZED_UNIT	contributing cause	The straggler carries most of the run's worker time (oversized unit).	76/721
STRAGGLER_CAUSE_INTERFERENCE_OR_CONTENTION	contributing cause	The straggler hit sibling interference or resource contention (errors in sibling files, kills, contention statements).	23/721
STRAGGLER_CAUSE_PARENT_FOLLOWUP	contributing cause	The parent re-engaged the finished straggler with a follow-up/audit task, so its end time reflects a second assignment (added after the straggler audit; Codex only, where worker turns are logged).	3/721
STRAGGLER_CAUSE_UNEXPLAINED	symptom	Straggler with none of the automatic cause signals.	19/721
LOAD_IMBALANCE	symptom	Worker/unit durations are very unequal (distributional), whether or not the longest ends alone.	78/542
D_OVERSIZED_UNIT	root cause	In DAG-Sched, one unit alone spans most of the run while the other units finish quickly: the plan's critical unit is large, so DAG-Sched cannot be faster than that unit.	45/300
WRONG_EDGE_DIRECTION	root cause	An upstream DAG-Sched unit cannot make its own build/check pass (or its tests hang) because the missing work is owned by one of its own descendants: the plan edge points the wrong way for this unit (or the two units should be one).	5/300
SHARED_TREE_OBSERVATION	contributing cause	A worker/unit reads a repository file that a concurrently running sibling edited (it sees sibling state through the shared checkout).	370/721
BUILD_TEST_INTERFERENCE	contributing cause	A worker/unit's build or test fails with an error located in a file that another live worker/unit (or the parent) is editing.	111/791
BUILD_TEST_INTERFERENCE_PARENT_WIP	contributing cause	Sub-case: the error is in a file the parent is editing concurrently (the parent's work-in-progress breaks a worker's checks).	28/491
SIBLING_POLL_WAIT	symptom	A worker/unit busy-waits for another unit's output (while/until/for-seq loops with sleep that test a file, process or grep result, or retry a build/check until it succeeds).	54/791
SEMANTIC_DEPENDENCY_SIGNAL	symptom	Candidate hidden dependency: a worker/unit says it is waiting for / blocked by / depends on another unit, or that a needed symbol does not exist yet.	85/791
INTERFERENCE_STATEMENT	symptom	A worker/unit states that another agent's in-progress or broken edits affect its build/tests.	206/791
DUPLICATE_SUPPORT_WORK	contributing cause	Several threads install the same package / set up the same tool.	6/899
TRUE_SEMANTIC_HIDDEN_DEPENDENCY	root cause	A unit genuinely needs another unit's product (an interface, format or semantics the plan contract leaves open or gets wrong) to implement its own part correctly; the only false-independence split that threatens DAG validity.	5/107
VERIFICATION_ORDER_DEPENDENCY	contributing cause	A unit implements against the plan's contract, but its own unit check / tests need a sibling's product, so it waits, polls or defers its checks (a latency cost; the DAG is valid for implementation).	68/107
WORKER_WAITS_ON_PARENT	contributing cause	In a model-initiated arm a worker waits for code the parent is writing (the parent kept a unit its workers depend on).	4/59

bucket	type	definition	flagged/elig.
PARENT_WIP_INTERFERENCE_AUDITED	contributing cause	Audited version of BUILD_TEST_INTERFERENCE_PARENT_WIP: the parent's in-progress edits break or hang a worker's checks (or the reverse).	3/59
worker			
COLD_WORKER	contributing cause	Workers begin from the instruction (not from the parent's exploration) and re-explore before their first edit.	141/791
REPEATED_REPO_EXPLORATION	contributing cause	Workers re-read most of the files the parent already read before delegating.	131/491
DUPLICATE_VERIFICATION	contributing cause	The same test command is run by two or more threads (typically each worker and then the parent, or every DAG-Sched unit).	109/899
FULL_SUITE_RERUN_NO_EDIT	contributing cause	A thread reruns the same full suite with no edit anywhere in the cell in between (e.g. to filter the output).	40/1199
TOOL_HANG	harness property	A single command runs into the Claude Code 2-minute Bash cap or runs for minutes (recursive greps, hung tests); present in Single too.	147/1199
UPSTREAM_ANSWER_SEARCH	contributing cause	An agent looks for the upstream answer (git log –all / fsck / reflog, pip download, curl to GitHub/PyPI) instead of implementing.	278/1199
THRASHING	symptom	One thread reruns the same test command many times (fix-test loop).	5/1199
ENV_REPAIR_IN_TAIL	contributing cause	After the workers finish, the parent installs or repairs dependencies (environment work beyond the task).	8/491
RERUN_TO_FILTER_OUTPUT	contributing cause	An agent reruns the same suite only to see or filter its output (11-31 s per audited case).	7/18
harness			
MEMORY_KILL	infrastructure artifact	A process is killed (exit 137 / 'Killed' / OOM) while other agents run in the same task container (requested resources 2 CPUs / 8 GiB per task.json; host co-scheduling is a competing explanation).	14/1199
CPU_SLOWDOWN	infrastructure artifact	The same command runs much slower while another heavy tool is in flight.	20/899
CONTENTION_STATEMENT	symptom	An agent says the machine is loaded / memory pressure / killed by contention.	22/1199
D_PROCESS_STARTUP_LAG	harness property	A DAG-Sched unit's first model input comes long after the scheduler started its process.	2/300
MODEL_STALL	infrastructure artifact	A single model call takes >= 300 s (provider latency outlier).	8/1199
POST_MODEL_CLI_TAIL	infrastructure artifact	The CLI keeps running >= 120 s after the last model output; inflates agent/trial time, not total time. Two kinds: after a final answer (hung background command / shutdown; Codex testem MAS and DAG-Sched) or with the transcript ending inside a tool call whose process is killed later (the protonmail cells, which also carry exit-137 kills).	8/1199
HARBOR_EXCEPTION	infrastructure artifact	Harbor recorded an exception (ApiRateLimitError label on killed CLIs, safety refusal, timeout).	15/1200
PARENT_CONTEXT_COST	harness property	Most of a multi-agent cell's tokens are the parent's growing context (cost, not latency).	79/199
UNIT_PROCESS_DEATH	infrastructure artifact	Every / several DAG-Sched unit processes are killed (exit 137) during concurrent test runs (DAG-Sched failure cause).	3/26
task-DAG			
LOW_WIDTH	task property	The DAG has at most 2 concurrently runnable nodes.	144/1200
LOW_WORK_SPAN	task property	The reference DAG's work/span is below 1.5.	300/1200
TINY_TASK	task property	The single agent (same system) finishes in under 3 minutes, so fixed overheads dominate.	96/1200
INHERENT_BRANCH_IMBALANCE	task property	One DAG node holds at least half of the reference work.	648/1200
EXPENSIVE_GLOBAL_VERIFICATION	task property	The repository's full suite is slow: the single agent spends >= 120 s or >= 25% of its time in full-suite runs.	76/1200
SHARED_STATE_HEAVY_TASK	task property	Plan units share files (two or more units list the same path).	336/1200
PLAN_TOO_FINE	task property	Many very small units: >= 6 plan units and the median DAG-Sched unit lasts < 60 s.	14/300
PLAN_COMPILE_ONLY_CHECKS	harness property	Most unit checks in the plan are temporary/local or compile-only checks, not the repository suite.	468/1200
positive			
PLAN_INFO_BENEFIT	positive mechanism	With the plan, the first edit of any thread comes at most half as late as the single agent's first edit.	444/600
PLAN_INFO_BENEFIT_NO_DELEGATION	positive mechanism	Sub-case: MAS+DAG did not delegate (a single agent with the plan) and still edited at most half as late as Single.	12/57
EARLY_FANOUT	positive mechanism	The parent fans out early to several workers.	126/491
BALANCED_FANOUT	positive mechanism	Worker durations are balanced.	253/542
IMMEDIATE_READY_DISPATCH	positive mechanism	DAG-Sched starts every ready unit within 2 s of it becoming ready.	300/300
DEPENDENCY_TRIGGERED_SPAWN	positive mechanism	A model-initiated parent launches a later worker within 30 s after another worker finished (dependency-aware staging).	7/491
REDUCED_INTEGRATION	positive mechanism	A delegating parent finishes shortly after its workers.	58/491
VERIFICATION_SKIP	positive mechanism	The cell runs no full repository suite while the same system's single agent ran at least one (saves time; relies on the verifier).	159/899

bucket	type	definition	flagged/elig.
VERIFICATION_SKIP_REGEX_ONLY	symptom	Sensitivity version of VERIFICATION_SKIP with the original regex on both sides (overcounts: the audit found 17 of 36 sampled flags were detector misses).	205/899
PROMPTED_NO_TEST_WORKERS	harness property	Sub-case of the verification mechanism: the Claude Code parent tells workers not to run builds/tests.	59/292
ADAPTIVE_PRUNING	positive mechanism	The parent stops a worker it no longer needs (interrupt_agent / TaskStop on an agent).	6/491
MIDFLIGHT_STEERING	positive mechanism	The parent sends a running worker a follow-up task (Codex followup_task; payload encrypted, so whether it is a contract update or new work is not observable).	18/491
WARM_PARENT_CRITICAL_PATH_WIN	positive mechanism	The parent keeps the critical implementation and the cell still beats the same task's DAG-Sched cell.	15/491
INTEGRATION_CATCHES_WHAT_D_MISSES	positive mechanism	MAS+DAG passes where the same task's DAG-Sched cell fails, and the MAS+DAG parent edited in its tail (integration repair).	4/243
TAIL_CAUGHT_DEFECT	positive mechanism	The integration tail found and fixed a real defect (the benefit side of the tail).	7/36
VERIFICATION_SKIP_GENUINE_PROMPT_SETUP	harness property	Verification skip explained by the arm's setup (DAG-Sched template 'unit check passes', plan unit checks that are temporary/local/compile-only).	10/36
VERIFICATION_SKIP_EMERGENT	positive mechanism	Verification skip by the free choice of a MAS or MAS+DAG parent (targeted checks only).	8/36

F.4 SECONDARY MECHANISMS AND NEGATIVE CONTROLS

The main text develops four core mechanisms. This appendix gives five secondary ones in the same format: the observation and aggregate evidence, a real trace and the deepest explanation the evidence supports, then a summary of the owner, a counterexample or limitation, and a possible mitigation. No mitigation was tested; no new runs were made for this analysis. The secondary mechanisms are less important than the core four: each is rarer, harness-specific, or a weaker explanation of latency. The mechanism forensics cover the three original systems (Codex × GPT-5.6 Sol, Claude Code × Fable 5, Claude Code × Opus 5.5) on the PARALLEL tasks; Pi × Kimi K3 is in the quantitative results but was not audited at trace level. Every trace below is one run ($N = 1$) whose excerpts were checked against the raw transcript; times are seconds from the cell's first model input.

F.4.1 S1. VERIFICATION-ORDER COUPLING

A unit's own existing tests can require a downstream unit's change even when the implementation dependency points the right way. Flagged in 5 of 300 DAG-Sched cells; every flagged cell passed, so the cost is time, not correctness.

Claude Code × Fable 5 DAG-Sched on `kombu` (pass): the upstream unit `virtual-sac-core` finds that `test_memory.py` hangs, because consumer registrations now live in shared broker state and leak across connections until the downstream unit's reset exists. The run takes 960 s against 776 s for Single and 492 s for MAS+DAG. The same shared-state hang also slows MAS's worker on this task, so it is not specific to DAG-Sched. The edge is correct for implementation; the coupling is in verification order, which the plan format cannot express. This is not a wrong work edge.

Owner	Task or decomposition property.
Counterexample or limitation	Rare, one deep read, and the MAS+DAG comparison is a different single run.
Possible mitigation (untested)	Mark test-order couplings in the plan, or scope a unit's check to tests its own change can make pass.

F.4.2 S2. STRAGGLERS AND GRANULARITY

Most stragglers are not bad workers; the straggler is usually the task's largest reference unit running at about its replay time. Over the audited straggler set, straggler duration divided by the duration of the task's largest reference branch has median 0.96; 71% are within 1.2 times and 17% at or above 1.5 times. The straggler bucket flags 123 of 721 cells with at least two worker or unit threads.

Claude Code × Fable 5 MAS on `kombu` (pass, 1216 s against 776 s for Single): the core worker runs 890 s, against a replay time of 882 s for the same reference unit (ratio 1.01). The reference records this unit as large but cohesive: its parts share one broker state, so splitting it would create private

hand-offs. Granularity is a property of the decomposition; the straggler here is the certified-cohesive unit.

Owner	Mostly task or decomposition property; a minority involves the harness (a fixed tool time-out) or policy (repeated exploration).
Counterexample or limitation	Some stragglers run well beyond their replay time and remain unexplained, and several audited Claude Code × Fable 5 stragglers were environment or harness artifacts. Codex workers that received a parent follow-up are not counted as stragglers, because their measured span includes time the parent spent idle before the follow-up.
Possible mitigation (untested)	Repartition a unit only where replay evidence shows it was bundled without need; do not split certified-cohesive units.

F.4.3 S3. Cold workers and context reconstruction

Workers re-read repository context that the parent has already explored. The median share of the parent's read files that a worker reads again is 0.67, and 73% of delegated MAS and MAS+DAG cells are at or above one half (504 delegated cells, including a few control-task cells). A worker's first model call carries a median of 10.6k input tokens. Codex spawns request the full history (`fork_turns=all`), but the worker rollouts contain the task and the plan and not the parent's exploration; what a forked worker receives is inferred from token counts, not confirmed from the CLI source.

Claude Code × Fable 5 MAS on `ofetch` (pass, 632 s against 654 s for Single): the parent reads `fetch.ts`, `types.ts` and `index.ts` in its first seconds and spawns at 322 s; the test worker then reads the same three files again. Workers receive the task and, in MAS+DAG, the plan, but not the parent's in-turn findings.

Owner	Mixed: an inherent coordination cost plus what the harness passes at spawn (inferred).
Counterexample or limitation	The signal does not separate good from bad runs, and some re-reading is needed before editing. It is a real overhead but a weak explanation of latency.
Possible mitigation (untested)	Pass compact findings (a file map and the relevant interfaces) in the spawn brief.

F.4.4 S4. Shared-tree git interference (Claude Code)

In a shared checkout, a parent's `git stash` can remove a sibling's in-flight edits. Flagged in 71 of 791 delegated MAS, MAS+DAG and DAG-Sched cells, all Claude Code; of 18 audited cases, 3 were harmful. Parents used separate worktrees in 13 of 292 cells, with no conflicts.

Claude Code × Fable 5 MAS on `flipt` (pass, 422 s against 399 s for Single): the parent runs a build command that includes `git stash` on three files a worker is editing, notices that it "could clobber the worker's in-flight edits", and pops the stash; the worker meanwhile reports that two of its files "vanished from the worktree". A baseline-comparison habit (stash, test, pop) is safe in a single-agent tree and destructive in a tree shared with live workers.

Owner	Harness or runtime artifact (shared checkout), triggered by model policy.
Counterexample or limitation	Rare, and this cell still passed.
Possible mitigation (untested)	Per-worker worktrees, or baseline checks in a separate worktree.

F.4.5 S5. Concurrency and memory pressure in DAG-Sched

DAG-Sched dispatches every ready unit at once with no concurrency cap, and heavy concurrent builds can be killed (exit 137). Memory kills occur in 14 of 1199 cells over all arms, 5 of them among the 300 DAG-Sched cells.

Claude Code × Fable 5 DAG-Sched on a second `flipt` task (pass, 819 s against 899 s for Single): two units' Go compilations of the same large parser are killed, and each unit states that its build was likely killed by memory pressure from parallel builds and retries with reduced parallelism. All

agents of a cell share one container (2 CPUs, 8 GiB); the out-of-memory cause rests on the exit code and the units' own statements, with no kernel log.

Owner	Harness or runtime artifact (an uncapped scheduler in one container).
Counterexample or limitation	Rare, the task still passed, and the comparison with MAS+DAG is between two single runs, not a measured cost of contention.
Possible mitigation (untested)	Resource-aware dispatch, for example a cap tied to container CPUs or memory.

F.4.6 GOOD RUNS AS NEGATIVE CONTROLS

Mechanism claims are more persuasive when a real run without the pattern sits next to one with it. All runs below pass.

- *Early useful fan-out and a short scoped tail* (Codex × GPT-5.6 Sol MAS+DAG, `koota`): the parent spawns its first worker at 4.6 s, spawns five product units within seconds, makes no source edits itself, and ends with a 18 s tail; total 268 s against 541 s for Codex Single (2.02×). On the same task, Claude Code × Opus 5.5 MAS ends with a 273 s tail that includes a fixed `sleep 240` started after the worker's completion notification was queued, and takes 590 s against 391 s for Opus Single. This pair crosses systems, so it contrasts patterns, not systems.
- *Scoped versus broad verification* (Claude Code × Fable 5, `dulwich`, same model): MAS checks the touched modules and ends with a 48 s tail (total 368 s); MAS+DAG's workers finish earlier, but its parent runs the full suite twice plus a stash check and ends with a 276 s tail (total 481 s); Single takes 645 s.
- *A warm parent on the hard unit* (Claude Code × Fable 5, `dynaconf`): in MAS+DAG the parent keeps the coupled merge core and edits it while four workers run (total 503 s); in DAG-Sched a cold unit owns the same core and spends minutes re-reading and debugging it (total 937 s); Single takes 1146 s. Retained work by a warm parent is therefore not always a loss.

F.4.7 MINOR BUCKETS

Support-surface ownership gaps. An interface change can affect test doubles, helpers or callers outside every unit's product-file scope. This is flagged in 2 DAG-Sched cells and explains 4 of the 26 audited DAG-Sched failures; in Claude Code × Fable 5 DAG-Sched on `wazero` (fail, 342 s) two units wrote test files that call a helper defined in another unit's test file, and in the verifier's build that helper is undefined, so the package does not compile. *Parent exits while workers run.* In a small number of Claude Code cells the parent ended its turn while workers were still running, a harness lifecycle issue. These cells were not re-verified at trace level for this draft, and we draw no conclusion from them.

F.5 TRACE EXCERPTS FOR THE CORE MECHANISMS

Each excerpt below was re-opened in the raw transcript and matched (prep/W5/MECHANISMS.md, section 7). Lines starting with = are verbatim substrings of the raw JSONL field; lines starting with ~ are the catalog's rendering of a tool call whose key argument was matched. `t` is seconds from the cell's first model input and `L` the line in the transcript file. Paths are relative to the artifact root; `...` abbreviates a directory prefix given in the surrounding text. Em dashes inside transcripts are rendered as en dashes.

F.5.1 CORE MECHANISM 1: SERIAL PARENT TIME BEFORE THE FIRST SPAWN

Run `f5-go-git-worktree-merge-conflicts-mas-r01`. Source:
`traces/paper-eval-20260923/fable5-4arm/jbom02/cells/f5-go-git-worktree-merge-conflicts-mas-r01/harbor/f5-go-git-worktree-merge-conflicts-mas-r01/go-git-worktree-merge-conflicts__ugiUZXS/agent/sessions/projects/-app/b5ebbb80-2fb6-4a2b-bc7e-b8e8a13d02a2.jsonl`.

```
= [t=423.4s] L72 say:  I have a good picture now.  Let me verify a few
API details (‘plumbing.Hash‘ parsing, billy ‘util‘ helpers,
‘checkoutFile‘) before decomposing the work.
```

```
~ [t=707.9s] L96 edit:  Write /app/internal/merge/merge.go
~ [t=767.0s] L102 edit:  Write /app/worktree_merge.go
= [t=770.5s] L107 Bash:  go build ./...  && go vet ./internal/merge/ .
2>&1 | head -20
= [t=797.0s] L111 say:  Everything compiles.  Now I'll launch the two
background workers in parallel on disjoint files.
~ [t=820.2s] L112 spawn:  Agent description='Implement diff3 merge
engine' ...  "you must ONLY create/modify these two files ...
/app/internal/merge/merge.go (replace the existing placeholder)"
```

F.5.2 CORE MECHANISM 2: THE PARENT-OWNED VERIFICATION TAIL

Run `f5-jelmer__dulwich-1586-mas-dag-r01`. Source: `traces/paper-eval-20260923/fable5-4arm/jbom02/cells/f5-jelmer__dulwich-1586-mas-dag-r01/harbor/f5-jelmer__dulwich-1586-mas-dag-r01/jelmer__dulwich-1586__PP4JJKr/agent/sessions/projects/-dulwich/953dce9e-5881-4a0b-a900-6ff84450bdbf.jsonl`.

```
= [t=256.3s] L149 exec:  PYTHONPATH=$(pwd) python3 -m unittest
tests.test_suite 2>&1 | tail -6
= [t=341.7s] L150 out:  Ran 1943 tests in 84.951s ...  FAILED
(failures=1, errors=3, skipped=41, expected failures=1)
= [t=345.7s] L153 say:  Neither ruff nor mypy is installed here.  The
full suite has 1 failure + 3 errors -- I need to identify them and check
whether they're pre-existing:
= [t=346.5s] L154 exec:  PYTHONPATH=$(pwd) python3 -m unittest
tests.test_suite 2>&1 | grep -E "^(FAIL|ERROR):"
= [t=435.4s] L159 exec:  git -C /dulwich stash -q && PYTHONPATH=$(pwd)
python3 -m unittest tests.contrib.test_swift.SwiftObjectStoreTests 2>&1 |
tail -3; git -C /dulwich stash pop -q ...
= [t=440.5s] L163 say:  All four failures are pre-existing on the
pristine HEAD (3 swift contrib errors + the pack delta one), so the suite
is clean relative to baseline.
```

Run `f5-jelmer__dulwich-1586-mas-r01`. Source: `.../f5-jelmer__dulwich-1586-mas-r01/harbor/f5-jelmer__dulwich-1586-mas-r01/jelmer__dulwich-1586__ojCvi5z/agent/sessions/projects/-dulwich/48e3792b-1238-43cc-9a02-5c7bd20f94d0.jsonl`.

```
= [t=324.1s] L80 say:  All five workers are done.  Now I'll verify the
integrated result:  run the original reproducer, the combined test
modules, and check the diff.
= [t=332.6s] L88 exec:  PYTHONPATH=/dulwich python3 -m unittest
tests.test_refs tests.test_repository tests.test_config tests.test_ignore
tests.test_file tests.test_index tests.test_object_store tests.test_pack
tests.test_porcelain 2>&1 | tail ...
= [t=344.6s] L98 say:  One failure out of 788 -- let me confirm it's the
pre-existing Rust-extension one the worker identified.
= [t=368.4s] L106 say:  Confirmed:  that failure exists on the clean tree
too ...  Work is complete.
```

Run `o55-koota-entity-snapshot-rollback-mas-r01`. Source: `traces/paper-eval-20260923/opus55-ab/cells/o55-koota-entity-snapshot-rollback-mas-r01/harbor/o55-koota-entity-snapshot-rollback-mas-r01/koota-entity-snapshot-rollback__ZEG9JQT/agent/sessions/projects/-app/ea76b82a-bc62-46cc-bf1a-a65144f1ffb5.jsonl`.

```
= [t=316.6s] L171 say:  Lint is clean.  I'm waiting for the test worker
to finish its run.
~ [t=317.6s] L172 ScheduleWakeup delaySeconds=1200 "Fallback in case the
test-writing worker's completion notification doesn't arrive."
= [t=317.8s] L175 queue-operation enqueue:  <task-notification> ...
<status>completed</status> <summary>Agent "Write snapshot test suite"
finished</summary>
```

```
= [t=319.9s] L176 Bash:  sleep 240; cd /app/packages/core && npx vitest
run tests/snapshot.test.ts 2>&1 | tail -6
= [t=560.7s] L177 out:  Test Files 1 passed (1) ...  Tests 87 passed (87)
```

F.5.3 CORE MECHANISM 3: THE PARENT RETAINS CRITICAL WORK

Run `o55-go-genai-streamed-function-args-mas-r01`. Source: `traces/paper-eval-20260923/opus55-ab/cells/o55-go-genai-streamed-function-args-mas-r01/harbor/o55-go-genai-streamed-function-args-mas-r01/go-genai-streamed-function-args__XbsdMCK/agent/sessions/projects/-app/25555bbd-1b06-4080-be23-27efce68586d.jsonl`.

```
~ [t=120.5s] L52 edit:  Write /app/function_call_accumulator.go
= [t=207.4s] L82 say:  Existing tests first, then I'll hand out the test
work.
= [t=238.3s] L90 say:  Tests pass except `TestTable`, which also fails
without my change (it needs a replay directory that isn't set up here).
I'm starting three parallel test workers, each writing its own new test
file.
~ [t=248.5s] L91 spawn:  Agent description='Unit tests for accumulator'
~ [t=257.4s] L93 spawn:  Agent description='Stream + chat integration
tests'
~ [t=262.9s] L95 spawn:  Agent description='Live tool call tests'
= [t=284.7s] L112 exec:  sleep 240; ls /app/*_test.go | grep -E
"accumulator|streamed|live_function"
```

F.5.4 CORE MECHANISM 4: WHAT THE RUNTIME SCHEDULER GIVES UP

Run `codex56i-e-textual-kitty-key-phases-dag-sched-r01`. Source: `traces/codex-gpt56sol-integrity-20260924/pod-e/matrix-dag-sched/cells/codex56i-e-textual-kitty-key-phases-dag-sched-r01/harbor/codex56i-e-textual-kitty-key-phases-dag-sched-r01/textual-kitty-key-phases__7CnJDwt/`.

```
= [t=179.5s] agent/workers/01-kitty-extended-parser ...  L244 say:  ...
Key-code `0` text events ...  Expanded parser tests ...  Validation:  81
tests pass; mypy passes.
= verifier/test-stdout.txt L129:  > event = _key_event("\x1b[0;;229u")
= verifier/test-stdout.txt L139:  > assert len(events) == 1
= verifier/test-stdout.txt L140:  E AssertionError:  assert 8 == 1
```

Run `codex56i-e-textual-kitty-key-phases-mas-dag-r01`. Source: `.../codex56i-e-textual-kitty-key-phases-mas-dag-r01/.../rollout-2026-09-25T02-46-55-01a0d675-....jsonl`.

```
= [t=203.9s] L308 say:  The specified suites now pass cleanly:  89 tests.
...  I'm doing one final end-to-end contract check through the real
parser and app--including alterna[...]
= [t=223.9s] L325 patch:  *** Update File:
/app/src/textual/_xterm_parser.py
```

Run `f5r-rochacbruno__dynaconf-243-dag-sched-r01`. Source: `.../rochacbruno__dynaconf-243__GBvkBh2/agent/sessions/01-settings-merge-core/-dynaconf/be2d3b18-5979-4c54-97fb-8529df1aaa96.jsonl`.

```
= [t=89.8s] L48 exec:  cd /tmp && pip download dynaconf==2.2.0 --no-deps
-q 2>&1 | tail -1; ls /tmp/dynaconf-2.2.0* 2>/dev/null
= [t=346.3s] L75 say:  All callers are compatible.  Now implementing.
First the helper in `dynaconf/utils/__init__.py`:
= [t=550.1s] L137 say:  Something's off with the dunder path.  Let me
compare against the original code to see if this is a pre-existing quirk
or a regression from my change:
= [t=936.7s] L277 say:  Unit `settings-merge-core` is complete.
```

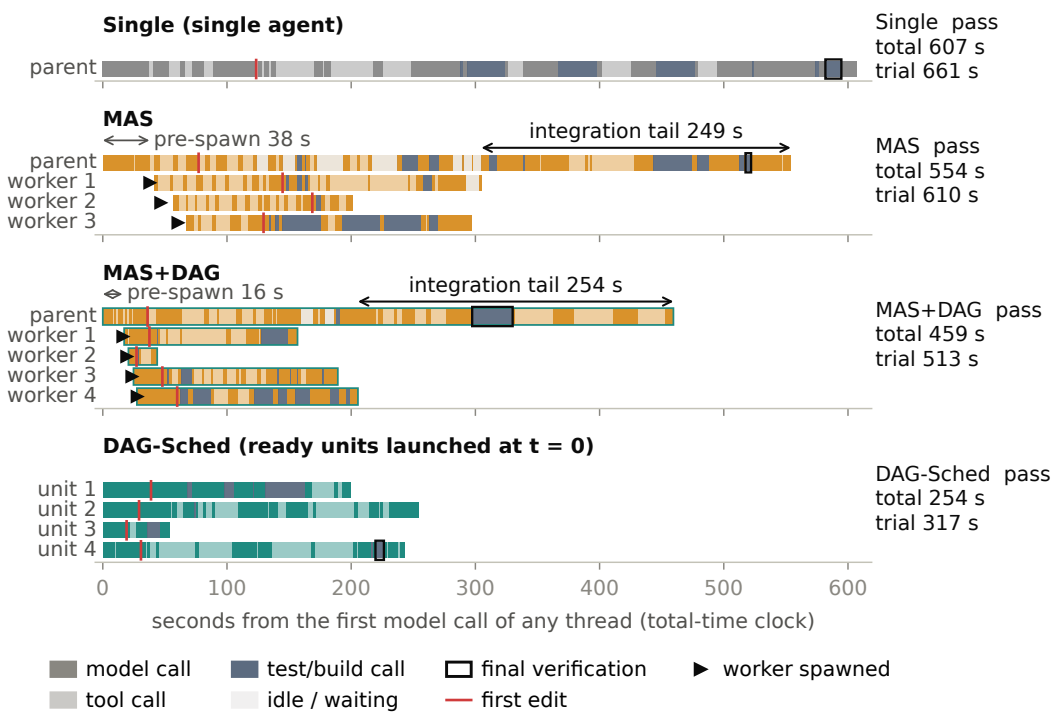


Figure 19: One task, all four arms (Codex × GPT-5.6 Sol, `navidrome`, every arm passes; total-time clock). MAS issues its first spawn call at 38 s and ends with a 249 s parent tail; MAS+DAG issues it at 16 s and still has a 254 s tail; DAG-Sched starts its 4 units at once and ends at 254 s against 607 s for Single. Disclosures: part of MAS+DAG's tail adds a test-mock support file (`tests/mock_user_props_repo.go`) that the verifier replaces with the gold version before grading, so that work does not affect the verdict; no DAG-Sched unit edits any test file. The DAG-Sched/Single total-time ratio here (2.39×) is above the Codex mean (2.02×). $N = 1$, illustrative, not typical.

Run `f5-rochacbruno__dynaconf-243-mas-dag-r01`. Source: `.../f5-rochacbruno__dynaconf-243-mas-dag-r01/.../rochacbruno__dynaconf-243__2smhFpk/agent/sessions/projects/-dynaconf/22be927c-c293-404c-829c-5a7398a2217a.jsonl`.

```
= [t=7.5s] L18 say:  I'll start by reading the key files ..., then
delegate the independent units to parallel workers while I take the core
`Settings.set`/`update` changes myself.
~ [t=283.2s] L76 edit:  Edit /dynaconf/dynaconf/utils/__init__.py
~ [t=332.4s] L110 edit:  Edit /dynaconf/dynaconf/base.py
= [t=338.5s] L118 say:  All workers have finished.  Let me review the
final state of `_merge_before_set` and then run behavioral checks.
```

F.6 One task in all four arms: navidrome (illustrative)

Figure 19 shows one Codex × GPT-5.6 Sol task in all four arms. It was the main-text timeline of an earlier draft. We moved it here because it needs several disclosures to be read correctly, and because its DAG-Sched gain is larger than the system average.

G Details moved from the main text

This appendix holds detail moved out of the nine-page main text; each part names the section it extends.

G.1 INTRODUCTION: ADDITIONAL CONTEXT

With a tested reference, a task-level outcome separates into three quantities: the *available parallelism* of one tested decomposition, the *planning* that finds useful units and dependencies, and the *orchestration* that turns them into concurrent execution (launching ready work, allocating the critical path, transferring context, integrating and verifying). In the four-arm protocol (§3), the single agent, Single, is the baseline. The autonomous multi-agent system, MAS, receives only the task and decides for itself whether, when and what to delegate. The same MAS given the reference plan as text, MAS+DAG, still makes every spawn decision. A deterministic DAG scheduler, DAG-Sched, receives the same plan and launches a fresh worker for every ready unit, with no parent model. The protocol partially factors planning from orchestration; it does not separate them causally, because MAS+DAG's prompt differs from MAS's beyond the plan and DAG-Sched differs from MAS+DAG in more than the scheduler. In the traced runs, the parent often keeps completion-critical product work in its own lane while workers write tests or support code.

G.2 BENCHMARK: CONSTRUCTION AND COVERAGE DETAILS

This part extends §2.

Screens before replay. Two cheap screens decide whether a candidate is worth replay budget: a six-question semantic rubric (PASS/FAIL/UNCERTAIN) and a gold-check probe that runs every local check against the base tree and the gold solution, rejecting vacuous, invented or invalid checks. Neither screen is structural evidence; the rubric's measured calibration is weak, which is why we do not use it as a verifier (Appendix B.4). Every intervention test uses a fresh worker with no memory of the trajectory, the frozen task environment, a declared information boundary and the original verifier, unchanged. Refuted claims enter at most two repair rounds; the revised graph is replayed, and remaining disagreements are recorded. Appendix B.5 traces one candidate (`tengo` callable isolation) from trajectory to refuted claim to released structure.

Coverage on the released tasks. Over the 104 released graphs (442 work units), unit-level isolation claims are 411 validated, 26 refuted and 8 inconclusive; 745 pairwise sibling-independence claims are validated (72 inconclusive, none refuted); and 103 of the 104 whole-graph recompositions are validated against the verifier (one untested). Edges are rare: 80 of the 100 PARALLEL plans have no dependency between units. A dependency test was attempted on all 57 PARALLEL edges, but only 17 were validated; most of the rest are inconclusive under the contract-first control. Necessity by omission validated 284 unit claims, refuted 93 and left 45 inconclusive and 23 untested. By the release's own certification level, 99 of the 100 PARALLEL graphs carry a verifier-passing recomposed replay (Appendix B.3).

Construction yield. The pipeline ran as a repeatable process rather than hand annotation (Table 19). Across 1016 task-by-wave construction attempts it produced 345 per-task construction deliverables, of which 278 contained a DAG, 235 recomposed to a verifier-passing tree and 138 reached full certification. Curation produced a historical 122-task candidate corpus and then the 104-task release. Human decisions entered through curation: 5 manual-curation and 3 salvage admissions, and one released graph admitted by manual review on the gold solution (Appendix B.3). Replay cost $13955 over all attempts, of which $2067 belongs to the released tasks. That is about $134 of replay per released task, and about 10% of construction attempts end in the release. The construction manifests have no structured repair count, so we do not report repair frequency.

Selection. A task enters only if a seed trajectory succeeded, the constructor produced a graph, and replay admitted it. From the historical corpus, the final cut removed 7 graphs whose compiled plan run exposed a hidden dependency (a rule fixed before the final runs; it can favour a plan executor and so DAG-Sched), 6 FrontierSWE tasks (a curation decision with no recorded task-level reason), and archived controls (Appendix B.2).

Release composition. The 104 tasks come from 4 upstream benchmarks: DeepSWE (Huang et al., 2026) (41+1), SWE-rebench V2 (Badertdinov et al., 2025; 2026) (36+1), SWE-Bench Pro (Deng et al., 2025) (22+1) and Terminal-Bench 2 (Merrill et al., 2026) (1+1); counts are PARALLEL+control

Table 19: Construction funnel (counts from the construction and release manifests; definitions in Appendix B.2). Rows above the release are candidate-pool counts, not release statistics.

stage	tasks
task-by-wave construction attempts	1016
per-task construction deliverables	345
with a candidate DAG	278
recomposition passes the verifier	235
fully certified	138
historical curated corpus	122
public release (100 PARALLEL + 4 control)	**104**

(Appendix B.1). Each task ships the reference DAG with every claim's replay label, W, S and width, the rendered plan text, and the scheduler used for the DAG-Sched arm. Full distributions are in Appendix A.

Table 20 maps each requirement of the measurement to the mechanism that meets it.

Table 20: What measuring task structure apart from system behaviour requires, and how MACE-Bench provides it.

	requirement	**how MACE-Bench meets it**
1	parallel opportunity measured apart from any system	reference work DAG with measured W, S and width
2	decomposition executable, not only plausible	each unit replayed by a fresh worker; outputs recomposed under the unchanged verifier
3	dependencies causal, not trajectory order	producer output withheld, with a contract-first control
4	latency and cost, not only success	success over all tasks; time and cost ratios on tasks both arms solve
5	an explanation of why a system is slow	phase partition of transcripts; trace catalog with verified examples (§5)

G.3 PROTOCOL DETAILS

Resource limits and worker caps. Each cell has 2 CPUs and 8 GiB, shared by every agent in the cell. The final configuration allows 64 concurrent workers, more than the widest plan (11 units). All Opus 5.5 MAS cells and some Codex MAS and MAS+DAG cells ran with an eight-agent prompt cap (Appendix C.1.2).

Contrast confounds in detail. MAS+DAG's prompt adds concurrent-launch, critical-path and verification instructions. MAS+DAG $\rightarrow$ DAG-Sched also changes the process model and integration (DAG-Sched's units check only their own scope).

Integrity and hosts. Leak channels and a few harness and verifier defects led to reruns chosen by exposure, never by outcome (267 Codex and 43 Claude Code cells, each replacing the original; Appendix C). Arms were not interleaved on hosts (same-host comparisons in Appendix L), and plans come from a Fable 5 baseline, which may favour Claude Code $\times$ Fable 5.

G.4 RESULTS: FULL STATISTICS AND EXPLORATORY ANALYSES

This part extends §4. Values are in the order Codex, Fable 5, Opus 5.5, Kimi K3 unless stated otherwise.

Table 21: Four systems, four arms, on the 100 PARALLEL tasks, one run per cell. Pass = passes / tasks, with no-verdict cells counted as fails; Δ = paired pass difference vs Single (points). Speedup and cost vs Single are geometric means over task pairs in which *both* arms passed (n pairs; cost pairs in parentheses where fewer), so they are conditional on success. Trial time covers the whole Harbor trial; total time runs from the first to the last model output over all agents. Brackets: 95% task-bootstrap intervals (task sampling only, $N = 1$ per cell). Codex DAG-Sched uses the 9 quota-rerun cells. Agent-time speedups: Table 26; Pi × Kimi K3 details: Table 11.

arm vs Single	pass	Δ pass (pp)	speedup, trial	speedup, total	cost vs Single	n
Codex × *GPT-5.6 Sol*; Single passes 83/100						
MAS	84/100	+1.0 [−7.0, +9.0]	0.98× [0.92, 1.05]	1.00× [0.93, 1.07]	2.06× [1.92, 2.21]	76
MAS+DAG	89/100	+6.0 [−1.0, +13.0]	1.26× [1.18, 1.35]	1.30× [1.21, 1.41]	1.80× [1.66, 1.95]	79
DAG-Sched	92/100	+9.0 [+2.0, +17.0]	1.73× [1.60, 1.88]	2.02× [1.84, 2.23]	1.14× [1.05, 1.24]	80
Claude Code × *Fable 5*; Single passes 81/100						
MAS	81/100	+0.0 [−7.0, +7.0]	0.99× [0.93, 1.06]	0.97× [0.91, 1.04]	1.14× [1.06, 1.23]	74
MAS+DAG	89/100	+8.0 [−1.0, +17.0]	1.18× [1.10, 1.26]	1.17× [1.09, 1.25]	1.02× [0.95, 1.11]	75
DAG-Sched	93/100	+12.0 [+3.0, +21.0]	1.78× [1.63, 1.96]	1.94× [1.75, 2.16]	1.10× [1.01, 1.21]	76
Claude Code × *Opus 5.5*; Single passes 83/100						
MAS	85/100	+2.0 [−5.0, +9.0]	0.95× [0.88, 1.01]	0.95× [0.88, 1.03]	1.23× [1.14, 1.33]	78
MAS+DAG	92/100	+9.0 [+1.0, +17.0]	1.20× [1.12, 1.30]	1.26× [1.16, 1.38]	0.96× [0.89, 1.04]	79
DAG-Sched	90/100	+7.0 [+0.0, +14.0]	1.58× [1.42, 1.76]	1.77× [1.56, 2.00]	1.32× [1.21, 1.46]	80 (79)
Pi × *Kimi K3*; Single passes 82/100; 3 agent timeouts scored as fails (Table 11)						
MAS	82/100	+0.0 [−6.0, +6.0]	0.96× [0.89, 1.03]	0.95× [0.88, 1.02]	1.06× [0.98, 1.14]	78
MAS+DAG	87/100	+5.0 [−4.0, +14.0]	1.02× [0.93, 1.11]	1.01× [0.91, 1.11]	0.97× [0.87, 1.07]	74
DAG-Sched	92/100	+10.0 [+2.0, +18.0]	1.78× [1.58, 2.00]	1.84× [1.62, 2.09]	1.00× [0.89, 1.13]	77

Single to MAS: delegation detail. When MAS delegates, it spawns a median of 3, 2, 2 and 2 workers. Its trial-time speedup over Single is 0.98×, 0.99×, 0.95× and 0.96×, and every interval contains one.

Plan and first spawn. On the 227 tasks where MAS and MAS+DAG both delegated in the three original systems, the median first spawn moved from 0.18 to 0.09 of total time.

Harness robustness. With Kimi K3 fixed, under Pi and Claude Code MAS stays near Single (trial 0.96× and 0.99×), MAS+DAG is little faster (1.02× and 1.09×, both intervals include one), and DAG-Sched is fastest (1.78× and 1.54× against Single; Appendix D.1).

Codex DAG-Sched before the quota reruns. Before the 9 reruns, Codex DAG-Sched passed 87/100 with a 2.01× total-time speedup over Single.

Verification split. Splitting DAG-Sched cells by whether they ran a full suite gives DAG-Sched/Single total-time speedups of 2.01× (skipped) against 1.72× (ran) on Fable 5 and 1.67× against 2.09× on Opus 5.5, where the split reverses (Codex in §4).

Deeper plans. The DAG-Sched/Single trial-time speedups on the 20 deeper plans in §4 use both-pass $n = 17$, 12 and 15 (three original systems).

Verification split under a broader detector. Under a broader suite detector, the Codex gap between DAG-Sched cells that skipped and ran a full suite shrinks to 2.05× against 1.95× (DAG-Sched/Single total-time speedup; observational; the split is confounded with task; Appendix F.1.5).

Success conditioning. Speedups and cost ratios are conditional: they are computed only on tasks that both compared arms solved, while pass rates cover all tasks. The result is therefore not "twice as fast at the same accuracy". DAG-Sched passes more tasks than Single in point estimate on every system, with a paired pass difference of +9.0, +12.0, +7.0 and +10.0 points (intervals in Table 21). Because DAG-Sched verifies less than Single, the solution-derived plan is a likely contributor to this gain, which again marks DAG-Sched as a runtime executing an oracle-derived plan. The per-task contingency of both-pass, Single-only, other-only and neither is in Appendix E.2; for example, on Codex Single and DAG-Sched both pass 80 tasks, only Single passes 3 and only DAG-Sched passes

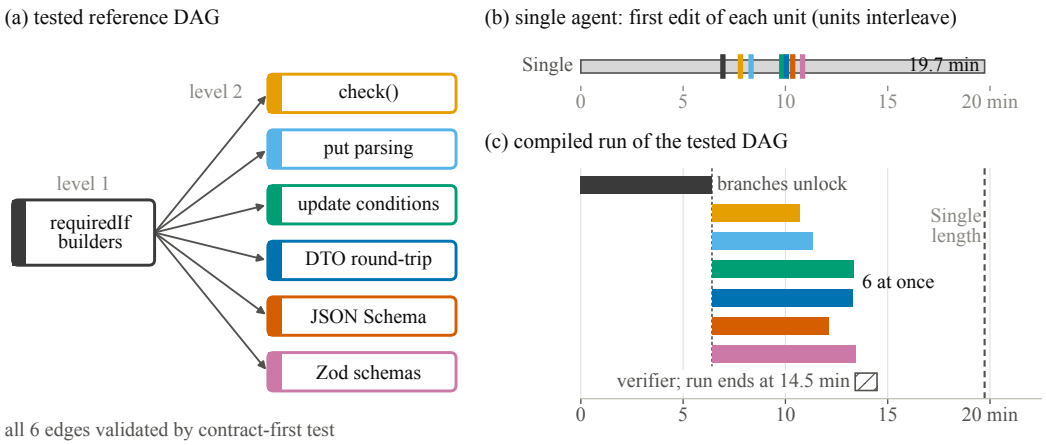


Figure 20: Trajectory order is not work dependency (DeepSWE `dynamodb-toolbox` conditional attributes). (a) The tested reference DAG: one producer (requiredIf builders) and 6 branches; all 6 edges were validated by a contract-first test; $W = 31.7$ min, $S = 11.2$ min, $W/S = 2.83$, width 6. (b) One successful single-agent trajectory of 19.7 min; each mark is the first edit of one of the 7 product units (6.9 to 10.8 min), and the agent interleaves units, so marks are not unit intervals. (c) The compiled certification replay of that DAG (not the DAG-Sched arm): the producer runs from 0 to 6.41 min, the 6 branches then run at once, and the verifier ends at 14.48 min. Panels (b) and (c) are separate single runs shown on one minute axis for scale; the gap is not a measured speedup.

12. Because MAS and DAG-Sched are each compared with Single on their own both-pass pairs, we do not divide the two geometric means. The direct comparison exists on matched MAS–DAG-Sched pairs in which both passed: for the three original systems, DAG-Sched is 2.02×, 2.00× and 1.83× faster than MAS on total time (trial 1.74×, 1.80×, 1.70×; $n = 81$, 77, 80). No matched value was computed for Pi × Kimi K3; there, MAS stays near Single while DAG-Sched reaches 1.84× Single's total-time speed.

Against the reference (exploratory). We divide each task's total-time speedup by its reference W/S. This ratio is not a realized fraction of a bound: W sums fresh-worker replay times, which can exceed Single's own run (Appendix Figure 20), so the ratio compares two differently measured quantities. For the three original systems its median is 0.49, 0.51 and 0.49 for MAS, 0.70, 0.62 and 0.65 for MAS+DAG, and 1.02, 0.94 and 0.92 for DAG-Sched. DAG-Sched's ratio is close to one. Parallel execution alone on a shared 2-CPU container would not obviously achieve that, so part of DAG-Sched's gain likely comes from plan information, less verification and less per-worker exploration.

Width versus work scale (exploratory). Figure 21 bins the same per-task DAG-Sched/Single total-time speedup by plan width and by measured reference work W, pooling the four systems. Within the evaluated range, the speedup rises with width, from 1.54× at width two to 2.32× at width 6+ (a pooled trend of 1.12× per unit of width), and does not rise with W (0.98× per doubling). Per-system rank correlations agree (Spearman $\rho = 0.19$ to 0.38 with width, -0.18 to 0.00 with W). Some of this is mechanical: width counts the units and DAG-Sched launches one agent per ready unit. The analysis is exploratory, with no multiplicity correction, and it does not show that task horizon is irrelevant.

G.5 Mechanisms: owners, mitigations and secondary mechanisms

This part extends §5. For each core mechanism it gives the owner (inherent coordination cost, model or orchestration policy, prompt or protocol design, harness or runtime artifact, task or decomposition property, or mixed), a counterexample or limitation, and a possible mitigation. No mitigation was tested. It then gives the good-run controls and the secondary mechanisms listed in Table 22.

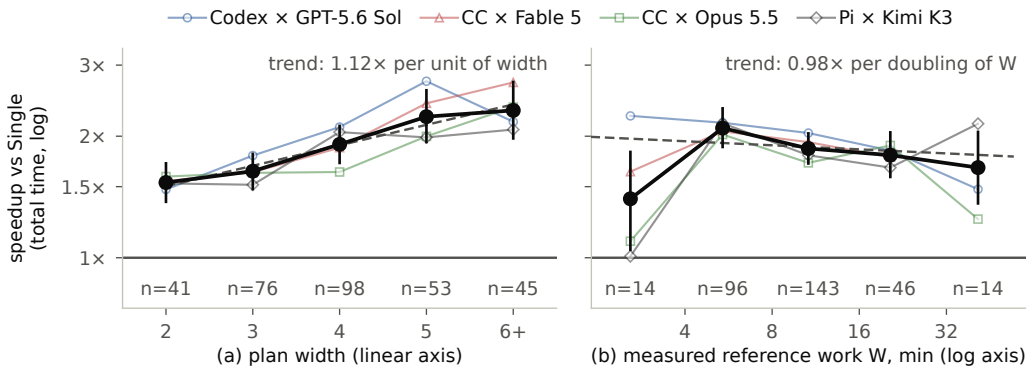


Figure 21: Exploratory: DAG-Sched/Single total-time speedup on tasks both arms solved, binned by (a) plan width and (b) measured reference work W (log2 bins). Black: pooled geometric mean over the four systems with 95% bootstrap intervals and bin size n; faint lines: per system; dashed: pooled trend. The speedup tracks width, which also counts the agents DAG-Sched launches, more than work scale. No multiplicity correction.

Core mechanism 1: serial parent time before the first spawn (§5.1). After dispatch, the worker's first model call follows about 2.0 to 2.9 s later; only 6 first calls took 40 s or more. The `go-git` parent launches its workers at 820 s (excerpt in Appendix F.5.1). In the `go-git` trace the MAS prompt asks for an inspect-and-decompose phase, and the parent fills it with its own model turns, going past localization to implement the integration surface.

Owner	Mixed: prompt design (the prefix is requested) and orchestration policy (the parent finishes its own piece first); not a harness artifact.
Counterexample or limitation	Among runs that already fan out early, first-spawn share does not separate good from bad outcomes; writing integration code before spawning is shown directly only in this trace; MAS+DAG's earlier spawning is confounded with plan information and prompt wording.
Possible mitigation (untested)	Expose ready units for delegation once their interfaces are known, rather than after the parent's own implementation compiles.

Core mechanism 2: the parent-owned verification tail (§5.2). The MAS and MAS+DAG prompts assign the tail to the parent ("integrate … into one verified solution"; "run the task's checks yourself"). In the `dulwich` pair (excerpts in Appendix F.5.2), MAS+DAG's 9 workers finish at 205 s, before MAS's 5 finish at 320 s. Across the population, tail share is only weakly related to speedup, so we do not claim that the tail predicts latency. Global verification is serial because only the parent composes and checks the result, and its length is a choice of scope and of waiting.

Owner	Mixed: one composed check is an inherent coordination cost; scope, reruns and sleep length are model policy; the timer-based wait idiom is a harness artifact.
Counterexample or limitation	Good runs also verify; they verify promptly rather than less, and real catches exist, if rarely.
Possible mitigation (untested)	Scope the final check to touched modules plus one global run, avoid repeated full suites without new evidence, and wait on completion events rather than timers.

Core mechanism 3: the parent retains critical work (§5.3). In the `go-genai` run (excerpt in Appendix F.5.3) the parent also runs the existing tests before spawning; the run takes 617 s against Single's 398 s.

Owner	Model or orchestration policy in MAS; prompt design in MAS+DAG, whose prompt tells the parent to implement critical-path work itself.
Counterexample or limitation	Retention sometimes wins: a warm parent that keeps one hard, coupled unit can beat a cold worker on it (15 flagged cells; the `dynaconf` pair in §5.4); among hand-read good and bad runs, retention does not separate the groups.
Possible mitigation (untested)	When product units are independently executable, give them to workers and treat tests and documentation as secondary work.

Core mechanism 4: what the runtime scheduler gives up (§5.4). Excerpts for `textual-kitty` are in Appendix F.5.4. The dominant DAG-Sched failure is a within-scope coverage miss, which better ownership contracts would not catch; a composed check might, but here it recovers at most about a third.

Owner	Mixed: scheduler design (no composed step, no reassignment) and task properties (one hard unit, difficulty).
Counterexample or limitation	On Fable 5 `dynaconf`, MAS+DAG's warm parent keeps the settings-merge core and ends at 503 s, while DAG-Sched's cold unit for the same core debugs for minutes and DAG-Sched ends at 937 s (Single: 1146 s); the warm-parent pairs are hand-picked, and which effect wins is a task property.
Possible mitigation (untested)	A bounded final compose-and-check unit, or selective repair; whether it would keep DAG-Sched's latency advantage is unknown, and it would change what DAG-Sched measures.

G.5.1 Good runs as controls

Real good runs show the opposite pattern. In Codex MAS+DAG on `koota`, the parent spawns its first worker at 4.6 s, keeps no source edits for itself, and ends with a 17.5 s tail, reaching 2.02× Single's speed. The Opus 5.5 MAS run on the same task, with the `sleep 240` tail, reaches 0.66×. Because this pair is cross-system, it contrasts the two patterns rather than the systems. The same-system `dulwich` pair (Figure 22b) shows the same contrast in the tail.

G.5.2 Secondary mechanisms

Five further mechanisms have real but smaller or narrower evidence. Each is summarized below and in Table 22; full write-ups with traces are in Appendix F.4.

Cold workers. Workers re-read a median 0.67 of the files the parent already read (at least half in 73% of 504 delegated cells, including 13 control-task cells). Codex parents request the full history at spawn (`fork_turns=all`), but worker first-call token counts suggest that workers receive the task and plan without the parent's exploration; this is inferred, not confirmed from the CLI source. The overhead is real but does not distinguish good from bad outcomes.

Stragglers and granularity. The slowest worker usually is the task's largest reference unit running near its replay time (median ratio 0.96; 71% within 1.2×), so a long worker is often a decomposition property rather than a bad worker.

Verification-order coupling. In 5 DAG-Sched cells a unit's existing tests need a downstream unit's fix (Fable 5 `kombu`); the implementation edge is correct, and the plan format cannot express the coupling.

Shared-tree git interference. In Claude Code only, a parent's baseline `git stash` in a checkout shared with live workers removed a sibling's in-flight edits in 3 of 18 audited cases.

Memory pressure in DAG-Sched. The uncapped scheduler runs concurrent builds in one container, and 5 of 300 DAG-Sched cells show killed compilers (exit 137).

G.6 Related work and limitations: extended

Software-engineering benchmarks. SWE-bench (Jimenez et al., 2024) and its successors, including SWE-Bench Pro (Deng et al., 2025), SWE-rebench (Badertdinov et al., 2025; 2026), DeepSWE (Huang et al., 2026) and Terminal-Bench (Merrill et al., 2026), pair a repository environment with an instruction and tests that grade the final state. They measure whether a task was solved, not what work structure it contains. MACE-Bench keeps their tasks and verifiers unchanged and adds a tested reference for the work each task contains.

Multi-agent benchmarks and systems. Multi-agent benchmarks grade collaboration through milestones or designer-specified dependencies (Zhu et al., 2025; Geng & Chang, 2025). Concurrent 2026 work evaluates orchestration plans in simulation (Ren et al., 2026), subagent orchestration (Xiong et al., 2026) and manager-built task graphs for computer use (Koh et al., 2026); DeLM (Mao & Mirhoseini, 2026) runs parallel agents over a dependency-tagged task queue on repository tasks, Co-Coder (Yang et al., 2026) partitions coding work to shorten the critical path, and TIPEX (Xu et al., 2026) studies structural parallelism as a latency–token trade-off (Appendix J). These systems

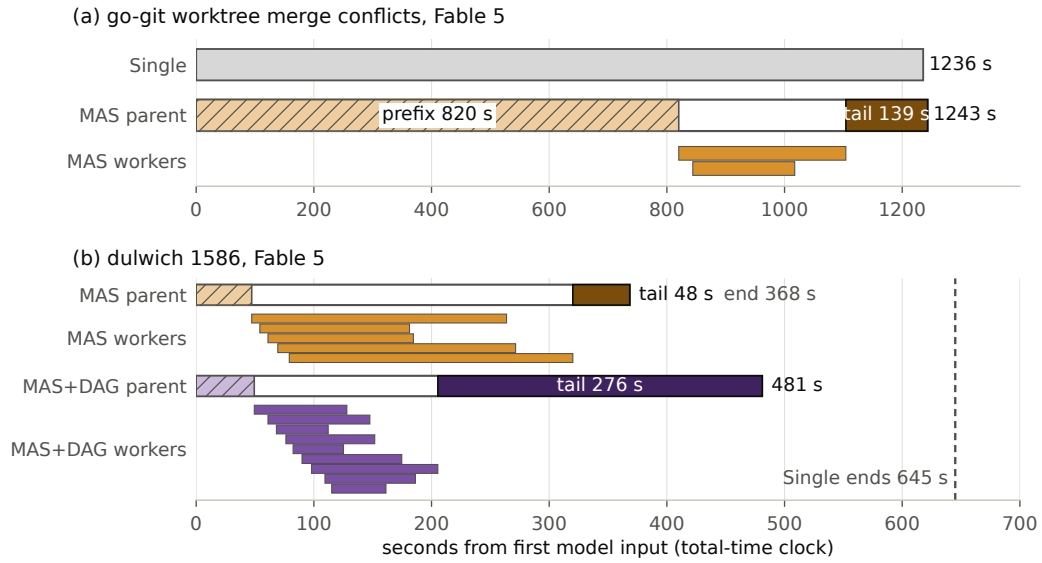


Figure 22: The parent's serial phases, not worker time, set the end-to-end time (Claude Code × Fable 5, total-time clock; every run shown passes). (a) `go-git`: MAS's parent works alone for 820 s of 1243 s before its first worker; Single takes 1236 s. (b) `dulwich`: MAS+DAG's workers finish before MAS's, but MAS+DAG's parent spends 276 s verifying (two full suites and a baseline check) against MAS's 48 s (touched modules), so MAS+DAG ends later. $N = 1$ runs, illustrative.

report their own gains; none supplies a per-task reference, tested against the task's verifier, against which any system's realized parallelism can be read. Frameworks and DAG runtimes decompose work among agents (Hong et al., 2024; Qian et al., 2024; Wu et al., 2024; Kim et al., 2024; Yu et al., 2025); controlled comparisons find regime-dependent gains (Smit et al., 2024; Wang et al., 2024; Kim et al., 2025) and communication overhead (Zhang et al., 2025a), and failure studies find coordination failures hard to locate (Cemri et al., 2025; Zhang et al., 2025b). In these works the task structure comes from the system under test, a planner or the benchmark designer, and it is not tested by intervention against the original verifier. Our forensics differ from failure taxonomies in that each loss is measured against a per-task reference for what the task permitted. Table 23 positions MACE-Bench against these benchmarks and systems.

Work and span. W, S and W/S follow the work–span model of parallel computation (Brent, 1974; Blumofe & Leiserson, 1999). What is new is measuring them for worker-sized software-engineering units whose independence and dependencies were tested by replay.

Limitations in more detail. The benchmark intentionally selects long-horizon tasks with replay-supported parallel opportunity. The reference is derived from one successful solution, and the plan given to MAS+DAG is therefore a partial oracle; no Single-plus-plan control was run. DAG-Sched also loses the parent's accumulated context, not only its warm process. Integrity reruns, not only ordinary reruns, differ by cell (Appendix C). The mechanism analysis is observational, and the audit covers the three original systems, not Pi × Kimi K3.

Mechanism summary. Letting the model decide when and what to delegate spawns workers and spends more, while the parent's serial time before delegation, the product work it keeps, and the verification it runs afterwards take up much of each run. A runtime that launches ready work removes those serial phases, at the price of the warm, composing agent that sometimes catches what isolated units miss. The four systems show a consistent latency ordering (MAS close to Single, DAG-Sched fastest), and the traces of three systems suggest why.

Beyond software engineering. Although MACE-Bench instantiates the method on software engineering, the construction protocol is not inherently code-specific. It could apply to domains where

Table 22: Mechanisms behind the gap between spawned workers and realized parallelism (three original systems, PARALLEL tasks; post-hoc catalog, counts are direct, latency associations observational). Each trace is one verified run ($N = 1$). Mitigations are proposals; none was tested.

mechanism	measured signal	best trace	owner	possible mitigation
Core				
serial pre-spawn parent time	dispatch median $\leq$0.02 s over 1412 spawns (1437 with controls); prefix 77–94% model time	Fable 5 MAS go-git: first worker at 820 s of 1243 s	prompt + policy	delegate once interfaces are known
parent-owned verification tail	tail in 281/491 delegated cells; real catch 7/36	Fable 5 dulwich: tail 48 s (MAS) vs 276 s (MAS+DAG), both pass	inherent + policy	scoped final check
timer-based waiting (Claude Code)	sleep overhang in 54 cells, none on Codex	Opus 5.5 MAS koota: `sleep 240` after a queued completion	harness + policy	event-driven waiting
retained critical work	131/491 delegated cells	Opus 5.5 MAS go-genai: parent writes the feature, workers write tests	policy (MAS), prompt (MAS+DAG)	give product units to workers
DAG-Sched loses warm, composed checking	23/300 all-units-pass failures; 15/26 within-unit misses	Codex DAG-Sched textual-kitty: "81 tests pass", verifier fails	scheduler + task	bounded compose/check
Secondary				
cold workers	re-read median 0.67; 73% $\geq$0.5	Fable 5 MAS ofetch: worker re-reads the parent's files	inherent + harness	pass compact findings
straggler / granularity	straggler / reference ratio median 0.96	Fable 5 MAS kombu: core worker runs about as long as its replay time	task	repartition only with replay evidence
verification-order coupling	5/300 DAG-Sched cells	Fable 5 DAG-Sched kombu: tests hang until the downstream reset exists	task	mark test-order couplings
shared-tree git interference	3/18 audited cases harmful	Fable 5 MAS flipt: parent's `git stash` removes a worker's edits	harness + policy	per-worker worktrees
memory pressure (DAG-Sched)	5/300 DAG-Sched cells	Fable 5 DAG-Sched flipt: two compiler kills	harness	resource-aware dispatch

Table 23: Positioning. ✓: provided; ○: partial or designer-given; –: not provided. "Reference work graph" means a per-task graph of work units and dependencies; "executable validation" means structural claims tested by running them against the task's success criterion.

	real-repo SWE	real-repo MAS	reference work graph	executable structural validation	available vs realized parallelism
SWE-bench family, Terminal-Bench	✓	–	–	–	–
MultiAgentBench, REALM-Bench	–	✓	○	–	–
OrchBench, ClawArena-Team, MACU	–	✓	○	–	–
LLMCompiler-style DAG runtimes	–	✓	○	–	–
DeLM, Co-Coder, TIPEX (concurrent)	○	✓	○	–	○
MACE-Bench (ours)	✓	✓	✓	✓	✓

candidate work units can be executed independently, outputs can be composed, and structural claims can be tested against an executable task-level success criterion.

H EXTENDED LIMITATIONS

H.1 LIMITATIONS AND DISCUSSION

Limits of the evaluation. *One run per cell.* Every interval is a task bootstrap and describes which tasks were sampled, not how a rerun of the same task would vary; no statement in this paper is about run-to-run variance, and single cells do vary (§C.1.3). *The scheduler's gain has two unresolved confounds.* DAG-Sched runs the composed suite less often than Single, and the plan carries information a single agent can use; the split by verification reverses on Opus 5.5 and the control that separates information from parallelism was not run (§F.1.5). We therefore say that the confounds *may* inflate DAG-Sched's speedup, not that they do, and not that its speedup is parallelism. *DAG-Sched differs from MAS+DAG in more than scheduling:* worker context, briefs, tools, failure handling and process model all change with the parent (§C.1.2). *Timing definitions differ in what they include.* Trial time contains environment setup and the verifier, total time excludes both; the Codex DAG-Sched/Single ratio is 2.02× on total time and 1.73× on trial time, and neither may be quoted without its name. *Reruns after an infrastructure failure.* The 9 Codex DAG-Sched cells whose unit workers hit the API quota were rerun on 2026-09-25, a day after the cells they replace, and one of them ran through a Codex CLI login on a ChatGPT subscription rather than the API, so its cost is a list-price estimate and its latency path differs; the rerun cells are single runs like every other cell (§C.1.3). *Scoring rules differ by harness for API safety refusals*, affecting one Codex MAS cell (Table 21). *Integrity conditions differ by cell* (§C.1.4): outside the rerun sets, Claude Code cells ran with web tools on and on full-history images, Codex kept cells ran web-on on full-history images (audited clean), and 2 Opus 5.5 cells that read future history keep their verdicts; no system ran every cell under web-off, history-cut conditions. One earlier-version Claude Code DAG-Sched leaf attempted to fetch the upstream source file its task changes; the fetch failed (§C.1.2). The rerun notes add an untested caution: cells that did not look in the history may be slightly easier than an average clean run. *Selection toward DAG-Sched.* The evaluation set excludes the 7 graphs whose compiled run exposed a hidden dependency, the failure mode DAG-Sched shares with the compiled executor; this favours DAG-Sched on pass rates and possibly on speed (§C.1.1). *Same-model provenance.* The source trajectories, constructions, reviews and most compiled workers used Fable 5, so Claude Code × Fable 5 is evaluated on tasks that model solved once and on plans distilled from its solutions (§C.1.1). *Days and attempts.* The arms of one system ran on different days, and some cells had more than one attempt, one of which the documented rule selects (§C.1.1, §C.1.4); a same-day sensitivity leaves the ordering unchanged (Appendix L). *Mostly flat plans.* 80 of the 100 plans have no dependency between units, so DAG-Sched mostly launches every unit at once, and the width trend uses plan width, which counts support units. *Mixed substrate.* Claude Code DAG-Sched is mixed-harness (8 Fable 5 and 23 Opus 5.5 cells on the fixed version) and its port has not had an independent review; some kept MAS and MAS+DAG cells ran under the old eight-agent prompt cap, and the Claude Code tool-batch limit may cap wide plans (§C.1.2); Opus 5.5 ran on 2 hosts with MAS+DAG on a loaded host, which the same-host rows check but do not remove (Table 27). *Heuristic measures.* Full-suite detection, command classes and most taxonomy labels are rule-based; shares depend on the detector, recall is limited for interference, false independence and contention, and the out-of-memory cause of exit-137 kills is inferred without kernel logs. Killed cells may carry partial costs. *Exploratory subgroups.* The width, horizon and delegation splits are exploratory with no multiplicity correction; the width trend is partly built into DAG-Sched, and the non-delegated groups are chosen by the model. *The taxonomy is post hoc* (§F.1.6): it was derived from traces already seen, its categories overlap, and it supports no additive account of lost time. *Generality.* Two harnesses and three models, each with its default multi-agent mode; other harnesses, prompts or orchestration policies may behave differently; other harness and model systems are future work. The evaluation set was frozen during the runs (§C.1.1). Measured reference work is moderate (median W 10.0 minutes); "long-horizon" refers to the repository-level task and the extended agent interaction (Appendix C.3), not to graph depth or to W.

Limits of the reference. *Solution-conditioned, non-unique reference.* Every graph is recovered from one successful trajectory and its gold solution, so it is one valid decomposition among several; its granularity moves width and work/span, and it is neither an oracle nor an upper bound on the task. Because W sums single isolated replays, a run of the same decomposition can exceed W/S: DAG-Sched's total-time speedup over Single exceeds the task's W/S on 51%, 47% and 44% of both-pass tasks (40%, 42% and 36% on trial time), and DAG-Sched executes this decomposition

only. *Fan-out partly by design.* The graphs encode work dependencies only; the contract-first control removes interface dependencies, so the shape of the released graphs, and the trace-order statistics of §A.4.1, describe independently authorable work under that definition, not every dependency an engineer would draw. *Coverage of the intervention tests.* In the public release, all 57 PARALLEL edges had a dependency test attempted, but only 17 were validated; the rest are inconclusive or refuted (Appendix B.3). In the historical pre-release pool, 61 of 135 PARALLEL edges were tested and 20 validated, and 72 edges were supported under the operative layer largely by model review. "Validated" in this paper names the outcome of an experiment; review-supported claims are labelled as such. *W is a reference quantity.* Node durations are single isolated replays by fresh workers. Over the 53 historical-pool tasks with a recorded single-agent wall the ratio of W to that wall has median 0.99 but IQR 0.78–1.31 (W exceeds the wall on 26 of them), so W/S is a ratio internal to the reference and a realized speedup on a given task can exceed it. *The plan text supplies specification.* MAS+DAG and DAG-Sched receive units, dependencies and file scopes recovered from a verified solution, so their gains over Single measure structure plus solution-derived information, not structure alone; the gains of §F.1.5 are the visible part of this. *Two repairs are not equally validated.* The spice ownership repair composed without conflict in a rerun whose verifier still failed (§A.4.3); the parallel quantum-espresso repair is not validated by any run. *Certification strength varies.* In the public release, 99 of 100 PARALLEL graphs carry a verifier-passing recomposed replay and one a manual-review record only. In the historical pool, 38 of 112 PARALLEL graphs were fully certified, 68 partially and 6 carried a manual-review record with no recomposed replay; partial graphs are often held by the shared-file independence rule, which an audit found conservative (9 of 12 rejected graphs would be included by a human); certification level is a published field, not a quality tier. *Sources.* The public release draws on 4 benchmarks, and three of them (DeepSWE, SWE-rebench V2, SWE-Bench Pro) supply all but 2 of the 104 tasks; the FrontierSWE tasks of the historical pool are excluded. *Shared files.* Siblings that edit a common file are independent as work but may not compose mechanically; the hidden dependencies of §A.4.3 concentrate there, and DAG-Sched's shared-tree interference and false-independence labels (§F.1.4) are the same phenomenon seen from execution. *Constructor study.* One seed per cell on 18 tasks and no forward arm.

I EXTENDED RELATED WORK

Agentic coding benchmarks. SWE-bench (Jimenez et al., 2024) fixed the unit that later repository-level benchmarks inherit: an environment at a base commit, an issue text, and tests that grade the final state (OpenAI, 2024; Deng et al., 2025; Badertdinov et al., 2025; Merrill et al., 2026). Where subtask structure exists it is a sequential grading rubric (LHTB's graded subtasks (Li et al., 2026)) or an optional step list, not a set of independently authorable units whose independence was tested; the scaffold around the model moves results substantially (Yang et al., 2024; Wang et al., 2025; Xia et al., 2025). We use these items unchanged, including their verifiers, and add an executable, partly replay-tested reference for the work each task contains.

Multi-agent systems. Role-based frameworks decompose work among specialized agents (Hong et al., 2024; Qian et al., 2024; Chen et al., 2024b; Wu et al., 2024; Tao et al., 2024), and DAG-driven runtimes execute planner-produced task graphs in parallel (Kim et al., 2024; Yu et al., 2025). Controlled comparisons find the gains regime-dependent: debate- and discussion-style systems often fail to beat a well-prompted single agent (Smit et al., 2024; Wang et al., 2024), adding calls is not monotone in accuracy (Chen et al., 2024a), the sign of the multi-agent effect depends on task structure and on the strength of the single-agent baseline (Kim et al., 2025), coordination carries token overhead (Zhang et al., 2025a), and accuracy-only evaluation rewards needlessly complex and costly agents, so cost should be reported beside accuracy (Kapoor et al., 2025). Failure studies attribute a substantial share of multi-agent failures to inter-agent misalignment and find that even locating the responsible agent and step in a failed trace is hard (Cemri et al., 2025; Zhang et al., 2025b). In this work the task structure is produced by the system under test or by an annotator and is not tested by intervention against the task's verifier; failures are attributed by reading traces, not against a per-task reference for what the task permitted.

Agent and multi-agent benchmarks. Benchmarks of agent behaviour score outcome and sometimes tokens (Liu et al., 2024; Mialon et al., 2023; Yao et al., 2025; Xu et al., 2024), and multi-agent benchmarks grade collaboration through milestones or designer-specified dependency structure (Zhu

et al., 2025; Geng & Chang, 2025). Our focus is narrower: real coding tasks, a reference for the work structure recovered from a verified solution rather than given by construction, and the conditional comparison of planning and orchestration on the original task.

Work, span and scheduling. W, S and W/S are the work, span and parallelism of the multithreaded-computation model (Brent, 1974; Blumofe & Leiserson, 1999; Cormen et al., 2009), also called average parallelism (Eager et al., 1989); greedy scheduling of a dependency graph runs within a factor of two of optimal (Graham, 1969), and a fixed serial fraction bounds the achievable speedup (Amdahl, 1967). What is new here is measuring these quantities for engineering tasks whose units are worker-sized and whose dependencies, where the replay reached them, were tested against the task's verifier rather than declared. Task-model induction (Jiang et al., 2026), adapted as one of our constructors, recovers control-flow models from traces; we find that it recovers order rather than dependency (Appendix A.2.4).

J Concurrent work

Several 2026 preprints, concurrent with this work, overlap it. DeLM (Mao & Mirhoseini, 2026) runs parallel agents over a shared task queue with dependency tags on repository tasks, and MACU (Koh et al., 2026) has a manager build a DAG and dispatch subagents on its ready frontier, reporting wall-clock gains; both are systems that our runtime arm resembles, and both report their own system's gains. OrchBench (Ren et al., 2026) evaluates orchestration plans in a deterministic simulator, Co-Coder (Yang et al., 2026) partitions coding work to shorten the critical path, ClawArena-Team (Xiong et al., 2026) isolates a leader's delegation skill, and TIPEX (Xu et al., 2026) studies structural parallelism as a trade between latency and tokens. None of them provides a per-task reference work structure, tested against the task's own verifier, against which any system's realized parallelism can be read, nor a controlled ladder that holds the harness and model fixed while changing who plans and who spawns.

K Trace analysis: taxonomy, splits and annotated traces

How the labels were made. One workstream extracted events from every final transcript (model calls, tool calls with their command text, spawns, edits and test or build commands with their intervals), derived categories bottom-up from traces it had read, wrote a rule for each, and read 101 cells by hand; a second workstream recomputed the counts. The taxonomy is therefore post hoc and exploratory: it names mechanisms and their frequency, it does not test them. Each cell gets one primary label and any number of secondary labels; categories co-occur and overlap in time, so the counts in Table 24 are not additive and no share of lost time is assigned to a category. Flagged-against-unflagged speed and cost contrasts (in the trace-analysis report and in Figure 23) are observational and confounded with task difficulty.

Worst MAS runs. Among MAS cells whose total time exceeds 1.25× Single's, the primary labels are, on Codex (23 cells), integration tail 16, straggler 3 and cold worker 2; on Fable 5 (32), serial prefix 15, integration tail 6 and no delegation 6; on Opus 5.5 (25), serial prefix 8, integration tail 8 and cold worker 3. Secondary causes co-occur in most of these cells.

Verification split. Where Single ran a full suite, DAG-Sched did not in 67%, 60% and 51% of cases under the regular expression (n = 61, 57, 55) and in 57%, 51% and 46% under the broader rule. On trial time the split is 1.75× against 1.66× (Codex), 1.83× against 1.64× (Fable 5) and 1.48× against 1.89× (Opus 5.5, reversed), skipped against ran. MAS+DAG also skips a full suite more often than Single (50%, 56% and 62% of MAS+DAG cells under the regular expression). Figure 24 shows the DAG-Sched/Single speedup split by whether the DAG-Sched cell ran a full test suite.

Figure 25 splits MAS+DAG cells by whether the model delegated, each against matched Single on the same tasks; the split is chosen by the model, so the comparison is observational.

Starting hypotheses the traces did not support. *Duplicated context*: every Codex spawn requests the full history (`fork_turns=all`), but its workers' first calls are small (so the forked

Table 24: Post-hoc taxonomy of orchestration inefficiencies (exploratory). The categories were derived bottom-up from the final traces after they had been read; they are not the pilot taxonomy of the 2026-09-22 plan (Appendix T). Counts: flagged cells out of 100 PARALLEL cells per arm, as Codex / Fable 5 / Opus 5.5. Primary: cells (of the 900 MAS, MAS+DAG, DAG-Sched cells) whose primary label is the category. Typical s: median of the category's own seconds measure in flagged cells (an estimate). Labels come from rules plus 101 cells read by hand; recall is limited for interference, false independence and contention. Categories co-occur and overlap in time, so the counts are *not additive* and no share of any arm's lost time is attributed to a category. The two gain rows are mechanisms that may inflate a speedup, not losses.

category	primary cause type	MAS	MAS+DAG	DAG-Sched	primary	typical s
integration tail: the parent integrates and verifies after the last worker	orchestration	62/40/32	69/47/33	2/1/1	218	184
serial prefix: late first spawn	orchestration	5/75/39	0/58/12	0/0/0	111	206
no delegation: a model-initiated arm spawns no worker	orchestration decision (often rational)	0/20/31	0/9/48	n/a	103	n/a
straggler or oversized unit	task DAG (+ orchestration when units are bundled)	12/10/2	18/11/5	22/32/19	80	171
cold worker: re-exploration before the first edit	harness–model	21/18/34	3/18/9	3/24/15	59	125
shared-tree interference	orchestration / harness	7/13/5	7/16/1	28/20/26	55	n/a
parent keeps the largest unit	orchestration (sometimes beneficial)	15/42/28	3/40/24	n/a	34	103
false independence: coupled units run in parallel	task DAG	5/6/11	2/10/13	15/27/33	31	n/a
no integrator: every unit succeeds, the composed tree fails	orchestration policy (outcome)	n/a	n/a	8/6/10	20	n/a
harness artifact (exit-137 kill, refusal, timeout)	implementation	1/3/2	0/2/2	0/0/0	10	n/a
resource contention	implementation	5/3/5	3/3/2	6/10/5	14	n/a
parent re-edits worker files	orchestration / interface	55/29/6	53/20/6	n/a	12	n/a
duplicate verification	orchestration	22/31/20	15/30/20	2/3/1	8	92
fixed-sleep polling (Claude Code)	harness–model	0/26/27	0/27/23	0/1/4	6	60
over-delegation	orchestration	1/3/1	5/2/1	n/a	5	93
tool hang	harness–model (noise)	2/7/11	1/6/2	0/11/4	2	n/a
gain: plan information (first edit at most half of Single's)	gain mechanism	n/a	92/44/38	100/93/80	n/a	n/a
gain/risk: no full suite where Single ran one	gain mechanism (risk)	9/13/19	22/19/21	42/34/28	n/a	n/a

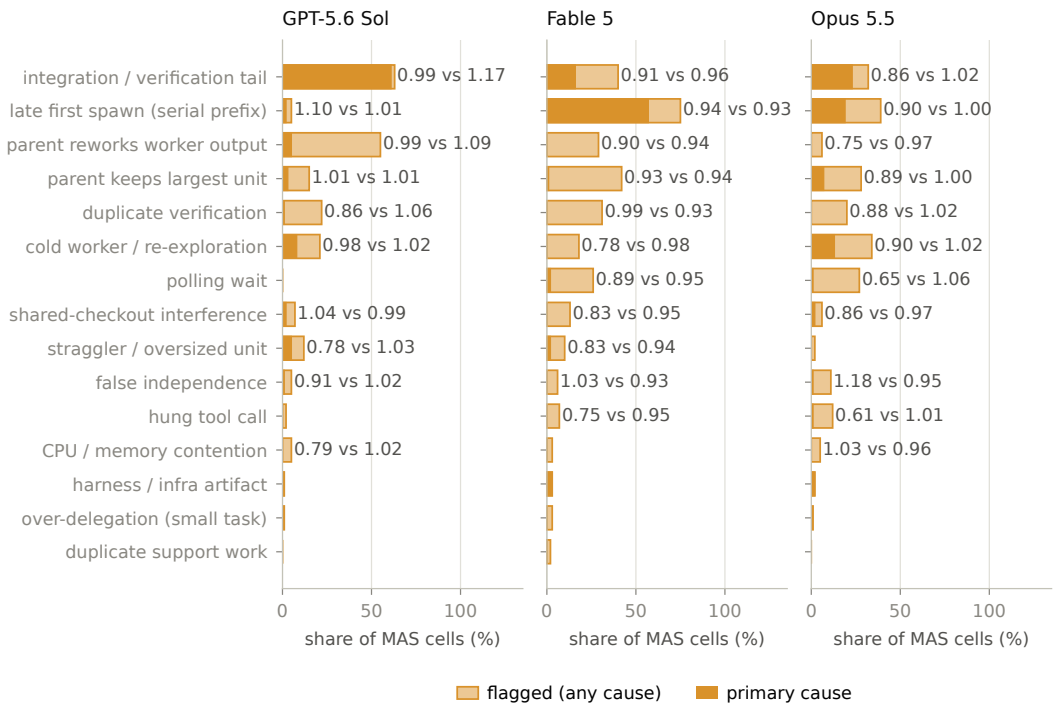


Figure 23: Arm MAS: share of PARALLEL cells flagged with each post-hoc category (light) and share where it is the primary label (solid), with the median MAS/Single total-time speedup of flagged against unflagged cells where at least five are flagged. Observational and non-additive: categories co-occur and overlap in time, flagged cells differ in task, and the contrast is not an effect size (all-verdict contrasts, labels as recorded). $N = 1$ run per cell.

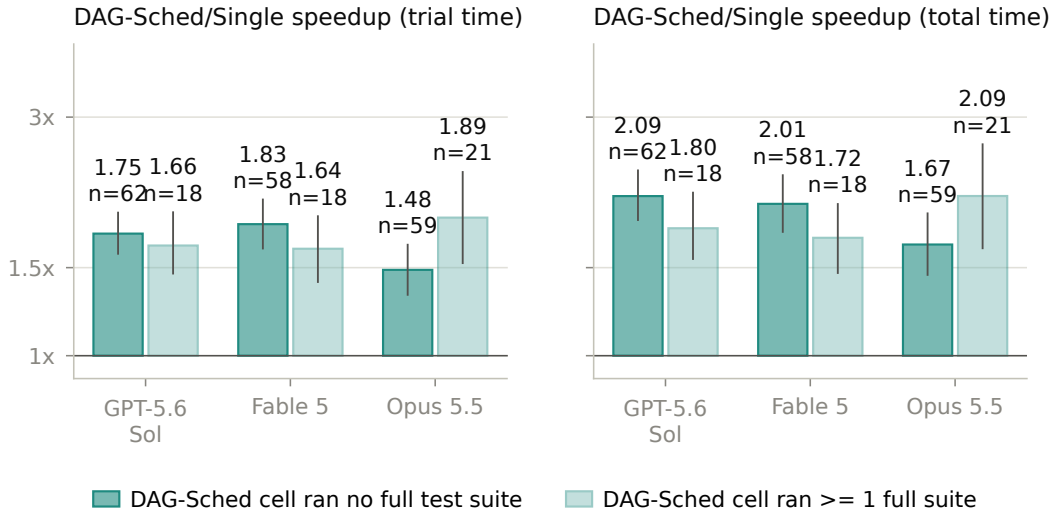


Figure 24: DAG-Sched/Single speedup on trial and on total time (both-pass pairs, one panel per timing definition) split by whether the DAG-Sched cell ran a full test suite, detected by a regular expression over commands. Observational, not an effect estimate: the gap goes the expected way on Codex and Fable 5 and reverses on Opus 5.5, and under a broader detector the Codex gap nearly vanishes (§F.1.5). Skipped verification may inflate DAG-Sched's speedup. $N = 1$ run per cell.

content is inferred, not confirmed from the CLI source) (§C.1.2); for the 117 spawns whose parent held at least 40 thousand tokens, the largest worker first call was 15952 tokens. What looks like duplicated context is re-reading. *Adaptive pruning* of the plan by the model-initiated parent was not

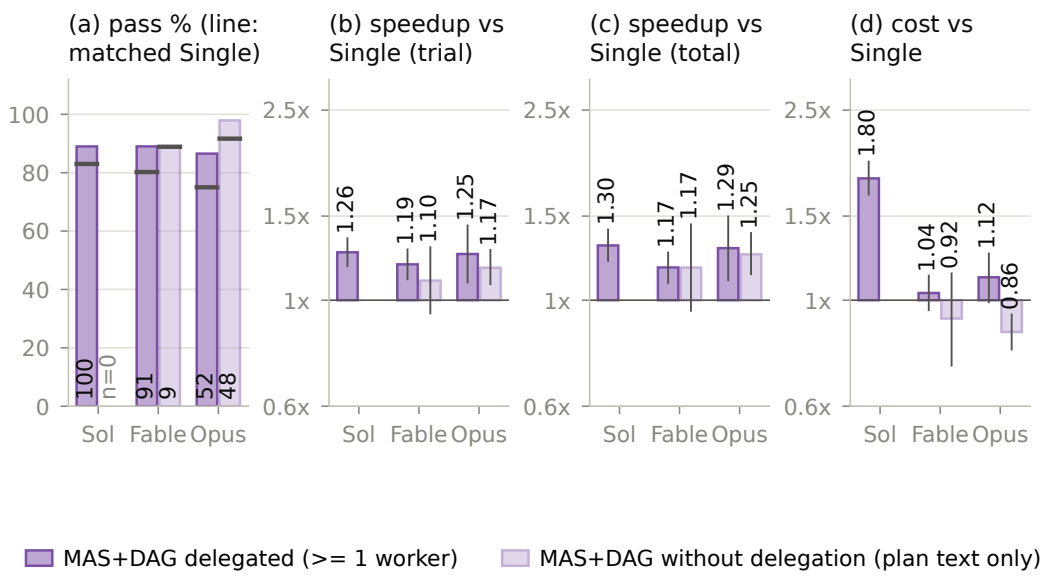


Figure 25: Arm MAS+DAG split by whether the model delegated, each against matched Single on the same tasks (both-pass geometric means; paired pass verdicts). The split is chosen by the model: Opus 5.5 did not delegate on tasks that Single also found easier, so the comparison is observational. Codex delegated in every cell; Fable 5 declined in 9 cells. $N = 1$ run per cell.

observed. *Load imbalance* and *oversized branch* could not be told apart in the traces and are merged into the straggler category. *Contention as a general tax*: rare overlap of heavy calls (§F.1.6), with a matched slowdown when it happens.

Harness observations. In Claude Code headless mode a parent can end its turn while background subagents still run; the run then ends without integration (2 Opus 5.5 MAS cells; 1 further cell with a negative tail, 3 in all, was an exit-137 kill during a parent tool call; all 3 tails are clamped to zero). Harbor labels exit-137 kills as rate-limit errors (13 agent-level cells). Both are harness fixes in Table 17.

Attempt selection. 157 cells have more than one archived Harbor trial (4 Codex, 84 Fable 5, 69 Opus 5.5); each run's documented rule picks one (for Claude Code, the first attempt with a verdict). The rule decides the verdict in a few documented cells: a Fable 5 MAS+DAG cell whose first attempt timed out in the verifier keeps its fail although a second, unarchived attempt passed, and an Opus 5.5 MAS+DAG cell keeps its failing first attempt over a later pass from a false host-suspend retry; across the Opus 5.5 retry audit the verdict flips in the two directions cancel (`analysis/final/issues.md` section 3).

Claude Code DAG-Sched leaf tools. A read-only scan of every Claude Code DAG-Sched leaf log (`paper/figures/trace-20260925/scan_cc_d_leaf_tools.py`, per-cell detail in `paper/figures/trace-20260925/cc_d_leaf_tools.json`) covers 104 Fable 5 and 104 Opus 5.5 cells (96 and 81 on the earlier harness version). No leaf was offered the sub-agent tool (0 and 0 earlier-version cells) and no leaf called it (0 and 0 cells). Earlier-version leaves were offered the messaging tools in 96 and 81 cells; 1 Fable 5 and 3 Opus 5.5 cells contain a call to one of them (listing agents or stopping a task). One earlier-version Fable 5 leaf (1 cell) attempted to fetch, from the upstream repository, the `ansible` source file that its task changes, a listed leak channel; the fetch failed ("Socket is closed"), so the cell keeps its verdict.

Ten annotated traces. Each was re-verified against raw timestamps by the trace analysis and is a single run; values are from the example files (`analysis/trace_examples/`), rounded to whole seconds on the total-time clock unless marked trial, and first-spawn and tail times can differ by a few seconds from the report's text, which times the parent's spawn call rather than the worker's

first model call. The list stays on the total-time clock because the timelines are drawn on it; each arm's trial time is in the same example file (§F.1.4 quotes the trial times of the cases used in the main text).

- *Four-arm figure* (Codex × GPT-5.6 Sol, `navidrome`, Figure 19; part of MAS+DAG's tail adds a test-mock file that the verifier replaces, and no DAG-Sched unit edits a test file): Single 607 s; MAS 554 s (spawn 38 s, tail 249 s); MAS+DAG 459 s (spawn 16 s, tail 254 s); DAG-Sched 254 s with no tail; all pass.
- *A good active run* (Codex × GPT-5.6 Sol MAS+DAG, `textual-kitty`): 5 workers from 12 s, a 51 s tail, 249 s against Single's 540 s.
- *A good runtime-initiated run* (Claude Code × Opus 5.5 DAG-Sched, `dulwich`): 11 units at once, mostly without a full suite; DAG-Sched/Single 7.17× on total time but 4.45× on trial time.
- *A bad autonomous run* (Codex × GPT-5.6 Sol MAS, `kcp-go`): 1415 s against Single's 590 s, with a 281 s tail of stress loops, repeated full suites and rework.
- *MAS to MAS+DAG* (Codex × GPT-5.6 Sol, `hydra-zen`): 469 s to 192 s; spawn 80 s to 10 s; tail 219 s to 86 s.
- *MAS+DAG to DAG-Sched, a win* (Claude Code × Opus 5.5, `testem`): MAS+DAG 588 s with a 341 s tail; DAG-Sched 71 s; DAG-Sched/Single 6.64× total, 4.63× trial.
- *MAS+DAG beats DAG-Sched* (Claude Code × Fable 5, `kombu`): MAS+DAG 492 s against DAG-Sched 960 s; the upstream unit's own checks could not pass until its dependent unit landed, and the scheduler starts that unit only after the upstream one exits (a verification-order coupling, not a wrong work edge; the same hang appears in MAS).
- *Contention* (Claude Code × Opus 5.5 DAG-Sched, `flipt`): DAG-Sched 1824 s against MAS+DAG 715 s; 4 cold Go builds killed with exit code 137 (likely out of memory).
- *Over-delegation and contention* (Claude Code × Opus 5.5 DAG-Sched, `remeda`): DAG-Sched 565 s against MAS+DAG 183 s; 4 small units, each running the full test suite.
- *A long critical worker* (Codex × GPT-5.6 Sol MAS, `filecoin`; Codex worker spans can include parent follow-ups, so we do not count this as a straggler case): MAS 438 s against Single's 154 s; the critical worker runs to 381 s while the parent waits.

L SENSITIVITY, ROBUSTNESS AND CONTROLS

Everything here concerns the final matrix of §4 (PARALLEL unless stated, $N = 1$ per cell, Codex DAG-Sched with its quota-fix rerun cells) and is recomputed from the committed cells (§L).

Pairing and the rerun cells. Table 25 puts the primary both-pass values beside all-verdict pairs. No ordering between arms changes. Without the one Codex DAG-Sched rerun cell served through a Codex CLI login (§C.1.3), Codex DAG-Sched/Single is 1.74× on trial and 2.03× on total time and DAG-Sched/MAS+DAG 1.39× and 1.58×, against 1.73×, 2.02×, 1.38× and 1.57× with it. The all-verdict Opus 5.5 cost ratios are inflated by killed Single cells whose recorded costs are partial; with raw-only costs the Opus 5.5 DAG-Sched/Single all-verdict cost ratio matches the both-pass 1.32×. Dropping every task with a killed cell moves each Claude Code ratio to Single by at most 0.02.

Timing definitions. Table 26 and Figure 26 give every comparison under trial, agent and total time. The Codex DAG-Sched/Single value is 1.73× on trial time, 1.96× on agent time and 2.02× on total time.

Hosts. On same-host Opus 5.5 pairs DAG-Sched/Single is 1.49× and MAS+DAG/Single 1.28× on trial time, against 1.58× and 1.20× on all hosts; DAG-Sched/MAS+DAG is 1.27× [1.03, 1.60] on 19 pairs. Fable 5 also spans two hosts in every arm; its same-host DAG-Sched/Single trial-time value is 1.77× ($n = 70$). Figure 27 compares all both-pass pairs with the same-host pairs.

Days. The arms of a system ran on different days (§C.1.4). Restricting each comparison to task pairs whose two trials started on the same UTC day gives, on trial time (geometric means, 95% task-bootstrap intervals): MAS/Single 0.97× ($n = 62$), 0.99× ($n = 70$) and 0.95× ($n = 78$); MAS+DAG/Single 1.23× [1.14, 1.33] ($n = 34$), 1.17× ($n = 69$) and 1.20× ($n = 78$); DAG-Sched/Single 1.68× [1.45, 1.97] ($n = 32$), 1.76× ($n = 67$) and 1.51× [1.35, 1.69] ($n = 61$);

Table 25: Pairing sensitivity (PARALLEL). Primary: both-pass pairs. All-verdict: every task where both cells have a verdict. Codex DAG-Sched uses the 9 quota-fix rerun cells (§C.1.3); without the one rerun cell served through a Codex CLI login, Codex DAG-Sched/Single is 1.74× on trial and 2.03× on total time (n=79). Geometric means, 95% task-bootstrap intervals, $N = 1$ per cell. The all-verdict Opus 5.5 cost ratios are inflated by Opus Single cells killed with exit code 137 (likely out of memory), whose recorded costs are partial (§C.1.3).

comparison	metric	both-pass (primary)	all-verdict
Codex × GPT-5.6 Sol			
MAS/Single	trial	0.98× [0.92, 1.05] (n=76)	0.98× [0.92, 1.03] (n=99)
MAS/Single	total	1.00× [0.93, 1.07] (n=76)	0.98× [0.92, 1.04] (n=99)
MAS/Single	cost	2.06× [1.92, 2.21] (n=76)	2.14× [2.01, 2.28] (n=99)
MAS+DAG/Single	trial	1.26× [1.18, 1.35] (n=79)	1.26× [1.19, 1.34] (n=100)
MAS+DAG/Single	total	1.30× [1.21, 1.41] (n=79)	1.30× [1.21, 1.39] (n=100)
MAS+DAG/Single	cost	1.80× [1.66, 1.95] (n=79)	1.86× [1.73, 1.99] (n=100)
DAG-Sched/Single	trial	1.73× [1.60, 1.88] (n=80)	1.74× [1.62, 1.87] (n=100)
DAG-Sched/Single	total	2.02× [1.84, 2.23] (n=80)	2.02× [1.86, 2.21] (n=100)
DAG-Sched/Single	cost	1.14× [1.05, 1.24] (n=80)	1.14× [1.07, 1.23] (n=100)
DAG-Sched/MAS+DAG	trial	1.38× [1.29, 1.49] (n=88)	1.38× [1.30, 1.47] (n=100)
DAG-Sched/MAS+DAG	total	1.57× [1.44, 1.73] (n=88)	1.56× [1.44, 1.70] (n=100)
DAG-Sched/MAS+DAG	cost	0.62× [0.59, 0.65] (n=88)	0.62× [0.59, 0.65] (n=100)
Claude Code × Fable 5			
MAS/Single	trial	0.99× [0.93, 1.06] (n=74)	0.96× [0.91, 1.01] (n=99)
MAS/Single	total	0.97× [0.91, 1.04] (n=74)	0.94× [0.89, 1.00] (n=99)
MAS/Single	cost	1.14× [1.06, 1.23] (n=74)	1.17× [1.10, 1.25] (n=99)
MAS+DAG/Single	trial	1.18× [1.10, 1.26] (n=75)	1.18× [1.11, 1.26] (n=99)
MAS+DAG/Single	total	1.17× [1.09, 1.25] (n=75)	1.18× [1.10, 1.25] (n=99)
MAS+DAG/Single	cost	1.02× [0.95, 1.11] (n=75)	1.04× [0.97, 1.12] (n=99)
DAG-Sched/Single	trial	1.78× [1.63, 1.96] (n=76)	1.72× [1.59, 1.87] (n=99)
DAG-Sched/Single	total	1.94× [1.75, 2.16] (n=76)	1.86× [1.70, 2.04] (n=99)
DAG-Sched/Single	cost	1.10× [1.01, 1.21] (n=76)	1.11× [1.02, 1.21] (n=98)
DAG-Sched/MAS+DAG	trial	1.47× [1.35, 1.60] (n=85)	1.47× [1.36, 1.58] (n=100)
DAG-Sched/MAS+DAG	total	1.61× [1.45, 1.78] (n=85)	1.57× [1.44, 1.72] (n=100)
DAG-Sched/MAS+DAG	cost	1.06× [1.00, 1.13] (n=85)	1.06× [1.01, 1.12] (n=99)
Claude Code × Opus 5.5			
MAS/Single	trial	0.95× [0.88, 1.01] (n=78)	0.94× [0.88, 1.01] (n=100)
MAS/Single	total	0.95× [0.88, 1.03] (n=78)	0.93× [0.85, 1.01] (n=100)
MAS/Single	cost	1.23× [1.14, 1.33] (n=78)	1.28× [1.19, 1.38] (n=100)
MAS+DAG/Single	trial	1.20× [1.12, 1.30] (n=79)	1.19× [1.11, 1.29] (n=100)
MAS+DAG/Single	total	1.26× [1.16, 1.38] (n=79)	1.20× [1.08, 1.33] (n=100)
MAS+DAG/Single	cost	0.96× [0.89, 1.04] (n=79)	1.01× [0.93, 1.10] (n=100)
DAG-Sched/Single	trial	1.58× [1.42, 1.76] (n=80)	1.58× [1.43, 1.75] (n=100)
DAG-Sched/Single	total	1.77× [1.56, 2.00] (n=80)	1.67× [1.47, 1.89] (n=100)
DAG-Sched/Single	cost	1.32× [1.21, 1.46] (n=79)	1.35× [1.24, 1.48] (n=99)
DAG-Sched/MAS+DAG	trial	1.30× [1.18, 1.44] (n=85)	1.32× [1.21, 1.44] (n=100)
DAG-Sched/MAS+DAG	total	1.37× [1.22, 1.56] (n=85)	1.39× [1.24, 1.55] (n=100)
DAG-Sched/MAS+DAG	cost	1.37× [1.26, 1.50] (n=84)	1.33× [1.23, 1.44] (n=99)
Paired pass difference, percentage points (all tasks; no-verdict cells counted as fails)			
Codex DAG-Sched−Single	pass	+9.0 [+2.0, +17.0] (n=100)	n/a
Codex DAG-Sched−MAS+DAG	pass	+3.0 [−1.0, +8.0] (n=100)	n/a
Fable 5 DAG-Sched−Single	pass	+12.0 [+3.0, +21.0] (n=100)	n/a
Fable 5 DAG-Sched−MAS+DAG	pass	+4.0 [−3.0, +11.0] (n=100)	n/a
Opus 5.5 DAG-Sched−Single	pass	+7.0 [+0.0, +14.0] (n=100)	n/a
Opus 5.5 DAG-Sched−MAS+DAG	pass	−2.0 [−9.0, +5.0] (n=100)	n/a

Table 26: The same comparisons under the three timing definitions of §C.1.3, each in its own column (never combined). Both-pass pairs, geometric means, 95% task-bootstrap intervals, $N = 1$ per cell. Trial time also contains environment setup and the verifier run; our reading (not tested) is that this shared time is one reason why trial-time ratios lie closer to one.

comparison	trial	agent	total	n
Codex × GPT-5.6 Sol				
MAS/Single	0.98× [0.92, 1.05]	0.99× [0.92, 1.07]	1.00× [0.93, 1.07]	76
MAS+DAG/Single	1.26× [1.18, 1.35]	1.30× [1.21, 1.40]	1.30× [1.21, 1.41]	79
DAG-Sched/Single	1.73× [1.60, 1.88]	1.96× [1.79, 2.17]	2.02× [1.84, 2.23]	80
DAG-Sched/MAS+DAG	1.38× [1.29, 1.49]	1.52× [1.40, 1.67]	1.57× [1.44, 1.73]	88
Claude Code × Fable 5				
MAS/Single	0.99× [0.93, 1.06]	0.98× [0.92, 1.05]	0.97× [0.91, 1.04]	74
MAS+DAG/Single	1.18× [1.10, 1.26]	1.18× [1.10, 1.26]	1.17× [1.09, 1.25]	75
DAG-Sched/Single	1.78× [1.63, 1.96]	1.94× [1.76, 2.16]	1.94× [1.75, 2.16]	76
DAG-Sched/MAS+DAG	1.47× [1.35, 1.60]	1.59× [1.44, 1.76]	1.61× [1.45, 1.78]	85
Claude Code × Opus 5.5				
MAS/Single	0.95× [0.88, 1.01]	0.95× [0.88, 1.03]	0.95× [0.88, 1.03]	78
MAS+DAG/Single	1.20× [1.12, 1.30]	1.26× [1.16, 1.37]	1.26× [1.16, 1.38]	79
DAG-Sched/Single	1.58× [1.42, 1.76]	1.73× [1.53, 1.95]	1.77× [1.56, 2.00]	80
DAG-Sched/MAS+DAG	1.30× [1.18, 1.44]	1.35× [1.20, 1.52]	1.37× [1.22, 1.56]	85

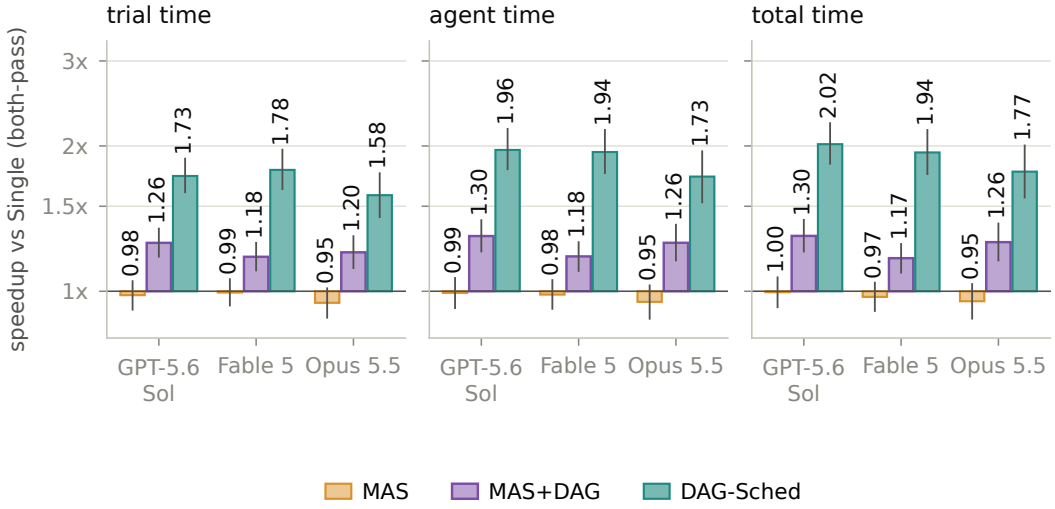


Figure 26: MAS, MAS+DAG, DAG-Sched speedup against Single under each timing definition, one panel per definition (never combined): trial, agent and total time (§C.1.3). Both-pass pairs, geometric means, 95% task-bootstrap intervals. $N = 1$ run per cell. The ordering is the same under all three definitions; the gap between trial and total time is largest for DAG-Sched.

and DAG-Sched/MAS+DAG 1.38×, 1.47× and 1.31× ($n = 81, 81, 65$). The ordering is unchanged; the Codex MAS+DAG/Single and DAG-Sched/Single same-day sets are small because Codex Single ran mostly on other days. Same-day pairing does not remove time-of-day effects.

Tokens and spend. Recorded spend per pass, over all PARALLEL cells, is $2.01, $4.13, $3.26 and $1.97 for Codex Single–DAG-Sched, $9.06, $10.24, $7.84 and $7.84 for Fable 5, and $1.65, $2.23, $1.53 and $1.78 for Opus 5.5 (Figure 28). Spend per pass mixes success and cost and is shown for budgeting only; the paper's cost comparisons are the paired ratios of Table 21.

Available against realized parallelism. Figure 29 plots each task's trial-time speedup against its reference W/S; the total-time converted fractions are in §E.1.3. The rank correlation of DAG-Sched/Single (trial) with W/S (also in §E.1.4) is 0.22 [-0.00, 0.42], 0.48 [0.27, 0.65] and 0.15 [-0.08, 0.36] (exploratory, no multiplicity correction; the interval excludes zero on Fable 5 only).

Table 27: Claude Code host sensitivity (PARALLEL): all both-pass pairs against the both-pass pairs whose two cells ran on the same host. Opus 5.5 MAS+DAG ran on one host under heavy load and most of Opus 5.5 DAG-Sched on another. Geometric means, 95% task-bootstrap intervals, $N = 1$ per cell. Codex is not involved.

comparison	metric	all hosts	same host
Claude Code × Fable 5			
MAS+DAG/Single	trial	1.18× [1.10, 1.26] (n=75)	1.17× [1.09, 1.25] (n=73)
MAS+DAG/Single	total	1.17× [1.09, 1.25] (n=75)	1.16× [1.08, 1.24] (n=73)
DAG-Sched/Single	trial	1.78× [1.63, 1.96] (n=76)	1.77× [1.60, 1.95] (n=70)
DAG-Sched/Single	total	1.94× [1.75, 2.16] (n=76)	1.90× [1.71, 2.12] (n=70)
DAG-Sched/MAS+DAG	trial	1.47× [1.35, 1.60] (n=85)	1.48× [1.36, 1.62] (n=81)
DAG-Sched/MAS+DAG	total	1.61× [1.45, 1.78] (n=85)	1.61× [1.46, 1.78] (n=81)
Claude Code × Opus 5.5			
MAS+DAG/Single	trial	1.20× [1.12, 1.30] (n=79)	1.28× [1.15, 1.43] (n=45)
MAS+DAG/Single	total	1.26× [1.16, 1.38] (n=79)	1.36× [1.20, 1.54] (n=45)
DAG-Sched/Single	trial	1.58× [1.42, 1.76] (n=80)	1.49× [1.29, 1.73] (n=38)
DAG-Sched/Single	total	1.77× [1.56, 2.00] (n=80)	1.68× [1.40, 2.01] (n=38)
DAG-Sched/MAS+DAG	trial	1.30× [1.18, 1.44] (n=85)	1.27× [1.03, 1.60] (n=19)
DAG-Sched/MAS+DAG	total	1.37× [1.22, 1.56] (n=85)	1.36× [1.04, 1.81] (n=19)

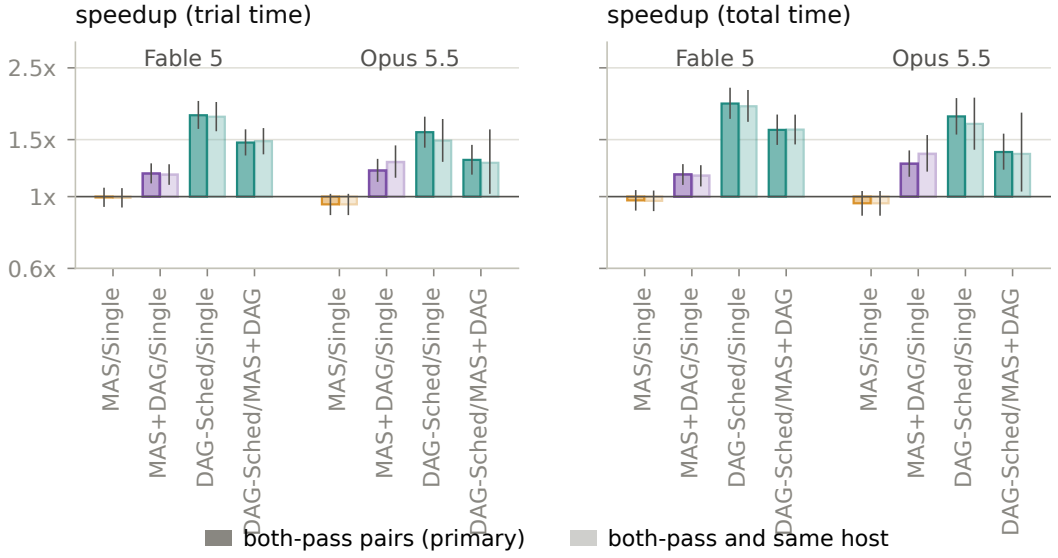


Figure 27: Claude Code timing: all both-pass pairs against the pairs whose two cells ran on the same host. Opus 5.5 MAS+DAG ran on one host under heavy load and most Opus 5.5 DAG-Sched cells on another. $N = 1$ run per cell. The direction and ordering hold on same-host pairs; the Opus 5.5 DAG-Sched/MAS+DAG same-host interval is wide ($n = 19$).

Controls (descriptive). On the 4 width-one controls every cell has a verdict. The arms passed 3, 3, 4 and 4 of them on Codex (Single–DAG-Sched), 4, 4, 4 and 4 on Fable 5, and 4, 4, 4 and 4 on Opus 5.5. DAG-Sched/Single on trial time was 1.20× ($n = 3$), 1.48× ($n = 4$) and 0.99× ($n = 4$). 4 tasks support no inference.

Recomputation checks. An independent recomputation from the raw artifacts, sharing no code with the workstreams that produced the numbers, reproduces every pass count exactly, 1246 of 1246 trial and agent times within a hundredth of a second, 1244 of 1246 total times within five seconds, and every speedup point estimate within 0.020. All 312 plotted values and table cells of the analysis figures, and all 3733 generated macros, equal the analysis's number file; an independent review of the numbers before the quota-fix rerun reproduced 208 of 216 headline ratios within two hundredths and explained the rest (`analysis/final/review/REVIEW.md`). In this paper every final-evaluation statistic is a macro: `paper/check_trs_macros.py` re-derives each `trs` and `trx` value from the number file, regenerates the `tcs` values from the trace examples, and fails on any macro used but not defined; the `M` values are report facts with a source comment and are not re-derived.

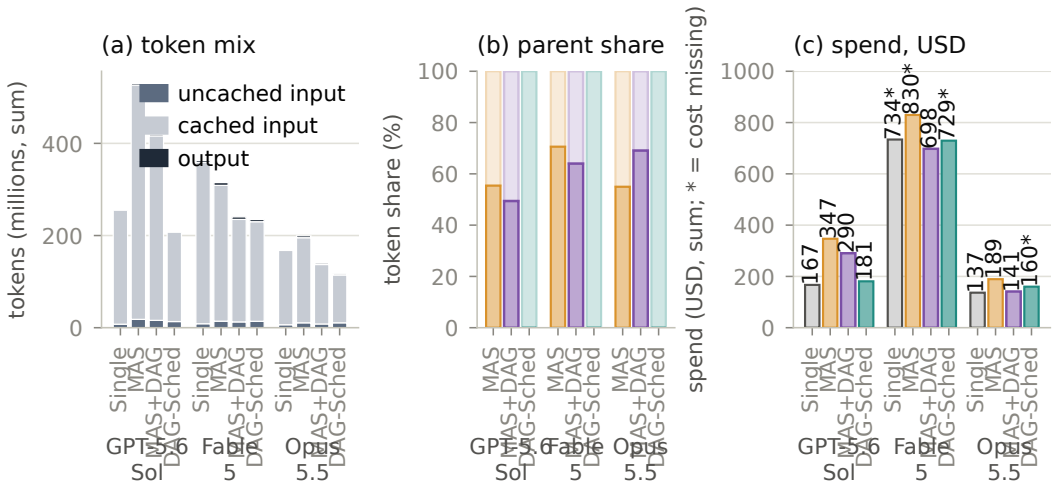


Figure 28: Tokens and spend summed over all PARALLEL cells of each arm, all verdicts (Codex DAG-Sched counts its final rerun cells, not the superseded quota cells, and one rerun cell's cost is a list-price estimate). Claude Code records no per-thread cost, so the parent-versus-worker panel is in tokens; some cells have no recorded cost. $N = 1$ run per cell.

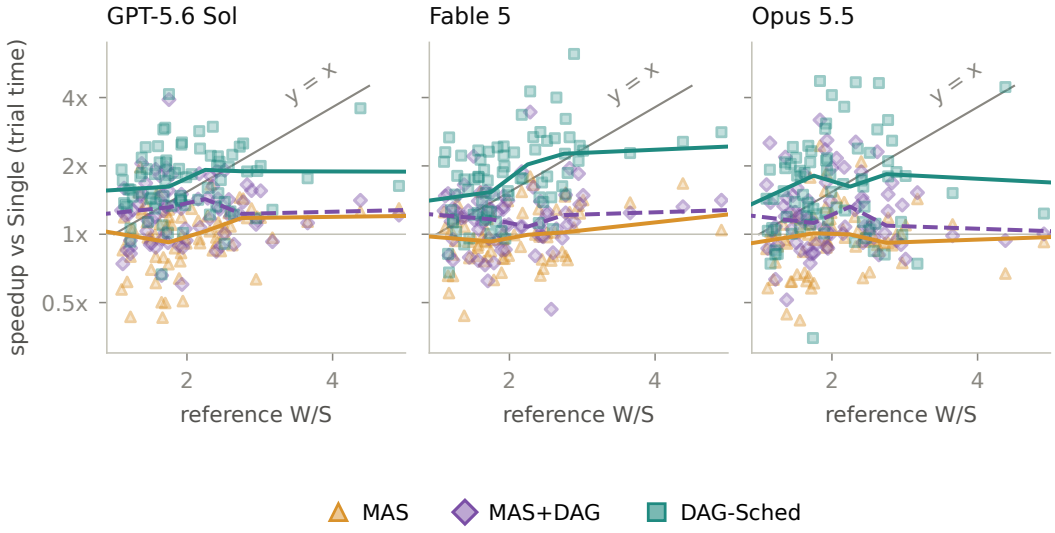


Figure 29: Available parallelism (reference work over span, W/S) against realized trial-time speedup over Single, both-pass pairs; lines are medians per W/S bin. W/S describes one reference decomposition under single-sample durations and is not an upper bound. $N = 1$ run per cell.

Not analyzed in this paper. Results restricted to the legacy CORE subset, to the long-and-parallel regime of Table 5 or to fully certified graphs; replicate variance (one run per cell); and a structure-only plan text without objectives or paths.

Harness and model robustness. Other harness and model systems are future work; this paper reports three systems.

M FULL REFERENCE WORK DAG METHODOLOGY

The construction and validation pipeline is Figure 7 in §A.2.

This appendix gives the complete definition of the artifact and of every control that produces a certification verdict. The main text (§A.1, §A.2) states the principles; this is the specification a reader needs to re-derive a verdict from a released record.

M.1 Artifact schema

Each released graph is a `reference-dag-v2` document with `work-dependency-v2` semantics. Its top level carries the task identifier, the source benchmark, the task instruction reference, the success criterion (the unchanged original verifier on the final workspace), the shared *contract facts*, the nodes, the edges, the independence records, the recomposition record, the minimality judgment, the reviewer verdict, the construction provenance, a list of unrepresented work, structure metrics, and the mechanically derived status.

Node fields. A node has an identifier and title; an *objective* (a coherent worker-level assignment); declared *inputs* (contract facts and predecessor outputs) and *outputs* (named artifacts); the *paths* it expects to touch; a *necessity* label (`necessary` or `useful`); a *role* (`work` or `integration`); a natural-language *verification* statement and a runnable *verification command* (the local check); *evidence spans* citing trajectory steps and gold hunks; a *granularity* label; per-claim *claims* (necessity, executable, nontrivial, local-check), each with status, method, evidence and notes; a measured *duration*; and a *necessity basis* (`product` or `support`). One node may change many files; two independent nodes may change the same file.

Edge fields. An edge has a source, a target, a *kind* (`artifact` for produced state, `state` for environment state), a *rationale*, a *claim* record for the work-dependency outcome, and a *work-dependency* record naming the type the evidence supports: `work`, `interface`, or `verification-only`. An edge asserts that the target cannot be *performed* without concrete state or output produced by the source. The following establish no edge on their own: a published interface the target codes against, chronological order in the trajectory, an import or build dependency, a check of the target that needs the assembled program, shared ownership of a file.

Contract facts. Facts describe interfaces and task requirements (type signatures, module paths, the names and shapes of functions siblings call). A fact must come from the task, the base state, or an explicit interface agreement; a fact that could only be learned by solving a predecessor's task cannot be a free starting fact, and review challenges facts of that kind. Facts must not carry gold-specific private algorithms; a leakage scan over worker-facing text enforces the interface-versus-solution rule.

Product and support work; necessity. A node is *product* work when the verifier depends on its output and *support* work otherwise (tests, documentation, harnesses, scratch tooling). Headline structure (product width, product work, product span) is computed over product nodes. Necessity is tested by omission (§M.4); a node whose omission leaves the verifier passing has refuted product necessity and may remain as `useful` support only when its output is consumed by a necessary node and the trajectory shows it was used.

Structure metrics. Node count, product width (the largest set of product nodes with no directed path between any two), depth (dependency levels), edge count, product work (sum of isolated node durations), product span (longest path over product nodes by duration), work over span, and branch balance (second-largest product branch over the largest). Support nodes are reported separately; an "executable width" that includes them is a distinct field and is never called parallelism. Durations are single samples from the certification replay.

M.2 Claims, outcomes and status

Every structural claim carries an outcome in {`validated`, `refuted`, `inconclusive`, `untested`}, with the method that produced it and the evidence path. The status is derived mechanically by one function and by nothing else: `certified` when recomposition and every required claim validate; `executable-inconclusive` when recomposition passes the verifier but evidence is incomplete, including infrastructure failures; `failed-review` when a required claim is

refuted; `failed-verifier` when the recomposed union fails the verifier; `unreplayed` when no execution evidence exists. Pre-replay screens and model reviews can send a candidate back to construction; they cannot certify, widen, narrow, or skip a control.

M.3 Pre-replay screens

Screens decide what is worth funding; they establish nothing about the graph. A *static screen* rejects file-shard nodes, thin branches, and evaluator wording in worker-facing text. A *gold check probe* runs every node's local check against the base tree and against the gold solution and gates out checks that pass on base (vacuous), fail on gold (invented internals), or need a file the node itself must write (unprobeable). A model *review* under a fixed protocol asks separately whether each producer is needed to author its consumer and whether it is needed to verify it, and returns accept, revise or reject with blocking issues. A frozen six-question *screening rubric* answers only pass, fail, or uncertain on grounding, semantic work units, plausible independence, plausible dependencies, granularity, and omitted parallelism; calibrated blind on 29 stored candidates it produced no false rejects and five action-level false accepts, and its dependency question was wrong on seven of twelve edges it passed with a known outcome. It is a sanity check, not a verifier.

M.4 Replay controls

Every control is a real agent execution in the task's frozen container by a fresh worker with no memory of the trajectory, judged by the unchanged verifier or by the node's own local check.

Isolation. Each node is executed by a fresh worker that starts from the base commit with its ancestors' outputs pre-applied and committed, and nothing else. The worker receives the objective, the contract facts, and its declared inputs; it must produce its declared outputs and pass the local check. The captured output is the worker's own delta relative to the pre-applied state. A failed pre-application cannot establish an own-node patch.

Sibling independence. For each pair of nodes with no directed path, both isolation runs are examined with the other's output withheld, and the union of all node deltas is composed mechanically. The claim validates when both nodes delivered their outputs and the union applied and passed the verifier. Under the frozen rule a pair that touched a common file stays `inconclusive` even when the union composed cleanly and passed; an audit found this the one repeated conservative rejection in the corpus (§Q), and the relaxation was specified but not applied for this release.

Withholding, stage one. For each edge $u \to v$, v is executed with u's output withheld. If v fails its check without any stand-in for u and review names the produced state v needed, the edge validates as a work dependency. If v passes with an independently verified alternative, the edge is refuted. If v *reconstructs* u's work, the control is inconclusive at this stage and proceeds to stage two. Reconstruction is detected by content: v's control patch defines top-level symbols that u's positive patch defines, repeats a substantial fraction of u's added lines on a shared file, or writes real content into files reserved to u, including data files.

Contract-first withholding, stage two. After a reconstruction-inconclusive stage one, or after any stage-one check failure with output, v runs again under a stricter instruction: u's output is withheld; v is told the published interface of u; it must not re-implement u; it may scaffold u's interface only with stubs that fail loudly with a fixed marker string; and it must write a `BLOCKED` report if its work is impossible without u's produced state. The outcome is read from what the worker did. (i) v authored its implementation in full against the interface and its check failed only by executing a marked scaffold: the edge is *refuted as interface*; the work was authorable and only verification needed u. (ii) v wrote `BLOCKED`, or its check failed with no stand-in present and review names the produced state it needed: the edge is *validated* as a work dependency. (iii) v reconstructed u again, used a silent stand-in, or failed for an unattributed reason: *inconclusive*, with the reason recorded. Type declarations count as interface; a stub module whose header carries the marker before its first definition is a whole-file scaffold; a check failure that names a scaffold-only file is attributed to scaffolding even when no marker appears in the output, which matters for compile-time failures in typed languages. Every verdict is re-scorable offline from the stored control patch and check output.

Necessity by omission. A node's contribution is omitted from the union and the verifier is rerun. A node whose omission leaves the verifier passing has refuted product necessity. A failed patch application is not evidence of necessity; a revision's ablation requires a successful-application sentinel before a verifier failure is read.

Recomposition. The union of every node's own delta is applied to a clean checkout in topological-level order. Byte-identical sections are deduplicated; a sibling-path overlap with conflicting hunks, or two distinct creations of one path, is a composition conflict that refutes composition before the verifier runs. The unchanged verifier judges the composed state once. No adjudication step repairs a failed recomposition.

Timing. Isolation runs yield one duration per node. Work, span and work over span are computed from these single samples and exclude scheduling, composition and final verification.

M.5 Bounded revision and evidence reuse

After replay, at most two model-proposed revision rounds may repair a candidate: merge redundant work, fix a check the gold probe shows to be wrong, add a dependency the controls support, or re-isolate a consumer whose edge was refuted as interface into a sibling with a fake-based local check and a composition-time real check. Each round records its diagnosis, the parent graph, the proposal, a fresh review, evidence reuse, and the replay. Pre-revision artifacts remain intact; a rejected revision never replaces the last replay-bound deliverable. A refuted edge parks the graph in `failed-review`; the revision removes exactly the refuted edges, re-isolates the consumer, and re-replays only the changed context. Stand-in files a sibling writes for an interface are recognized mechanically and stripped from the union, so they never become width. The reviewer sees the control evidence beside a proposed removal and rejects one whose interface is published in no contract fact. Unchanged nodes reuse evidence only when task bytes, image, model and scaffold, objective, check, predecessor contracts and patch hashes all match. A refuted claim is repaired and re-replayed; it is never deleted from a certified artifact.

M.6 Provenance and version boundary

A graph carries its construction method, prompt and contract hashes, the trajectory and gold it was built from, every prior boundary a revision changed and why, and the pipeline code version that certified it. Artifacts certified under the earlier `verification-coupled-v1` semantics, in which a check that needed the assembled program could establish an edge, retain that label and are not promoted; a schema change does not recertify evidence. Replay reuse, resumed queues, admission and reporting enforce the boundary.

M.7 Admission logic

Pool from structure. A task enters the release when its verifier passes on the gold solution in the release container, a candidate graph's recomposed union passes the unchanged verifier (or, for manual-review admissions, the gold passes and the static review below was completed), and the graph carries a per-claim evidence record. The pool follows from intrinsic structure alone: product width at least two is PARALLEL; product width one is CONTROL. No language-model arm's success or wall time was used in selecting tasks.

Certification level as metadata. *Fully certified*: recomposition and every required claim validated. *Verifier-passing, partial*: recomposition passes the verifier but at least one independence, necessity or local-check claim is inconclusive or refuted; the per-claim record says which, and a refuted local check is flagged in the level string. *Serial/control*: a width-one graph whose replay passed. *Manual review*: see below. Certification level is published per task and is not a pool.

Legacy analytic tiers. The release keeps the 2026-09-20 curation's tiers as metadata because the evaluation uses them as covariates. CORE: product work of at least eight minutes, second-largest product branch of at least two minutes, no refuted claim, no reviewer grade below acceptable, compiled execution not failed. EXTENDED: a PARALLEL task that misses one of those thresholds, most

often a shorter horizon (33 tasks) or a shorter second branch (35), or a refuted claim (43). All 77 EXTENDED tasks pass the verifier under recomposition. Neither tier is a correctness or quality tier of the public benchmark.

Manual-review admission rule. A candidate whose pipeline run was cut off before replay, or whose replay failed for an infrastructure reason, may be admitted after a zero-spend static review of its stored evidence if (i) the gold solution passes the verifier, (ii) the reviewed graph has at least two substantive product branches with checks that passed the gold check probe or were repaired for an identified probe-plumbing cause, (iii) every manual edit is recorded with its evidence, and (iv) a static consistency check passes (inputs and outputs wired, every path in the gold solution, acyclic, no unordered nodes sharing a path unless flagged). Such tasks carry `certification_level: manual-review`, `verifier_passed: gold-only`, and the exact automated failure that prevented certification. Manual review is not certification.

N CONSTRUCTOR STUDY: METHODS, FULL MATRIX AND HISTORY

The paired comparison is §A.2.4; this appendix gives the constructor prompts' framing, the preregistered design and censoring rule, the full matrix with its unequal denominators, and the two earlier comparisons that did not survive audit.

N.1 CONSTRUCTORS

The five strategies and the trace-cleaning step are described in §A.2.1; every constructor receives the same four inputs (task instruction, cleaned trajectory, artifact inventory, gold solution) and targets the same definition (§M). The construction model refuses several natural framings of "reason backward from the solution"; the accepted backward prompt states its direction in one sentence and supplies the solution as evidence. We record this as a limit on how faithfully backward deduction can be instructed on this model.

N.2 PREREGISTERED DESIGN AND CENSORING RULE

The comparison was preregistered before any arm ran (`docs/constructor-comparison-20260920.md`). Task set: 21 tasks drawn by a fixed rule from the released corpus, so that every task has a successful trajectory and a verifier-passing graph from the frozen pipeline and construction success is not confounded by broken tasks. Each arm ran one method with no fall-through, then the pre-replay screens, review, replay with every control, and at most two bounded repairs, under the production budgets (per-task cap scaled by planned leaves). Compiled execution ran on every arm's verifier-passing graph with at least two nodes. Relaunches retried only infrastructure statuses, never a screened-out or failed construction, so no arm received a second construction draw.

Censoring. A cell is *censored* when its failure side is infrastructure (provider or harness failure) or the per-task cost cap before a verdict existed; it is *scored* otherwise. Screened-out, rejected and composition-conflict outcomes are verdicts and are never censored. The *paired common evaluable subset* is the set of tasks on which every run arm has a scored cell: 18 of 21 tasks. The 3 excluded tasks and the arm that censored each are visible in Figure 30.

Forward arm. The provider's safety layer refused the forward arm's stage-one draft on 21 of 21 tasks in both of 2 launches. The arm therefore has no scored cell and is excluded from every comparison. This is a limitation on how the inductive framing can be instructed on this provider, not a finding about the method.

N.3 FULL OUTCOME MATRIX

The paired table is Table 4 in §A.2.4.

On the full matrix, with unequal denominators (direct 21, hierarchical 20, backward 19, TMI 19 scored tasks), verifier-passing graphs are 13 / 14 / 7 / 10, fully certified 6 / 6 / 2 / 3, and product-

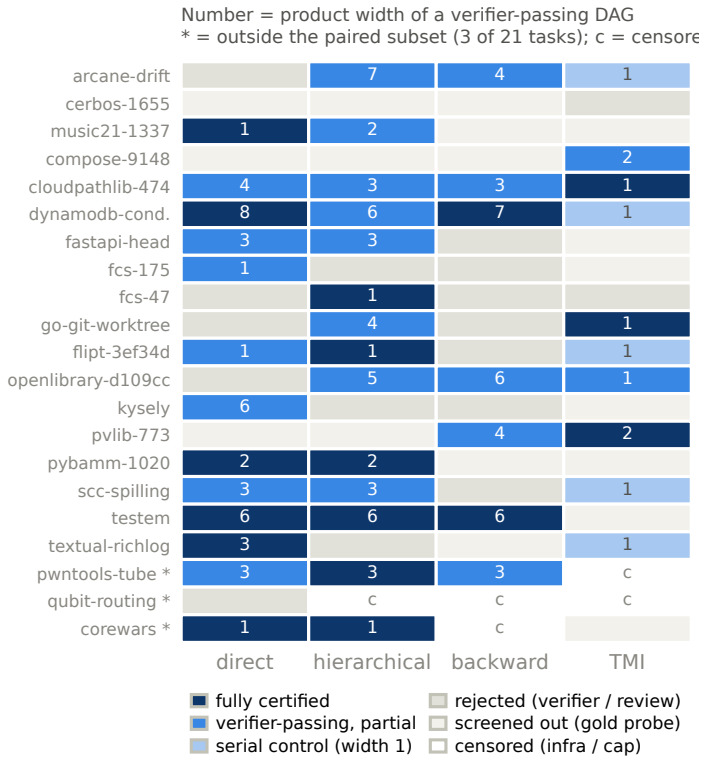


Figure 30: Full 21-task by four-arm outcome matrix of the preregistered constructor comparison. Each cell is the final pipeline outcome of one arm on one task; hatched cells are censored by infrastructure or the cost cap and define the 3 tasks excluded from the paired subset of §A.2.4. Censoring is sparse and arm-specific, so the paired subset loses few tasks.

parallel 9 / 11 / 7 / 2. Censored cells per arm: direct 0, hierarchical 1, backward 2, TMI 2. The main text uses the paired subset because these denominators differ by censoring, not by any property of the graphs.

Cost and latency. Median cost per task: direct $17, hierarchical $27, backward $31, TMI $37 (totals $400, $528, $651, $655). Median construction latency in minutes: direct 8.8, hierarchical 9.5, backward 13.9, TMI 23.6. Screened-out candidates on the paired subset: direct 3, hierarchical 3, backward 5, TMI 6. Median work over span of verifier-passing graphs: direct 1.59, hierarchical 1.50, backward 2.67, TMI 1.00. Cross-arm node-path alignment (mean Jaccard of node path sets): direct versus hierarchical 0.59, direct versus backward 0.48, direct versus TMI 0.21.

N.4 Two earlier comparisons that did not survive audit

The preregistered comparison is the third constructor comparison in the project; the first two are reported because several earlier conclusions rested on them.

Phase A (four methods, eight tasks). Direct, forward, backward and hierarchical each produced one candidate for the same 8 repository tasks from identical inputs (32 candidates), each reviewed once and replayed once with controls. As first scored, 3 certified and 16 failed review, with 20 refuted edges; the natural reading was that constructors declare false dependencies. Inspection of the stored control patches showed that 16 of the 20 refutations were the control misreading reconstruction: with the producer withheld, the consumer rebuilt the producer's symbols and passed its own check. Rescoring the same records under a content-based reconstruction detector gave 7 certified and 8 failed review (Figure 31). After an evidence-only necessity pass and at most two automated repairs, 13 of 32 certified; hierarchical led with 5 of 8 at about $32 per certified graph. Two facts invalidate

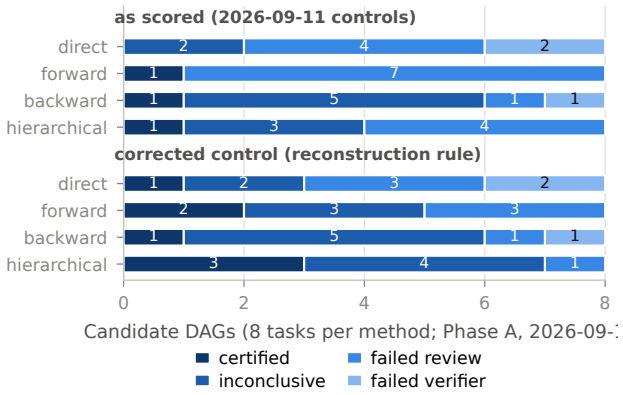


Figure 31: Historical Phase A comparison (four methods, 8 tasks, single seeds): first-pass outcomes as scored and after correcting the reconstruction control on the same stored records. Invalidated as a ranking (shared prompt opener, empty trajectory); kept for the research record.

Phase A as a ranking: the four prompts shared one opener and differed by one sentence, so the arms were four emphases of one recipe; and every construction before the trace-rendering fix of 2026-09-15 ran with an empty trajectory in the prompt, so Phase A compared gold-only construction. Its conclusions about the controls, the check probe and the granularity rule stand, because those instruments do not read the trajectory.

Corrected comparison (five methods, fifteen tasks). After three independent reviewers audited the arms, the prompts were rewritten to differ procedurally, the shared decomposition policy was moved into the hierarchical arm alone, TMI's contracting step was corrected, and every arm ran with a fixed construction budget on a 15-task development set (75 task-runs). Certification collapsed in every arm relative to the earlier hierarchical run on the same rules (7 of 15): direct 2, forward 0, backward 1, hierarchical 1, TMI 0. A forensic audit found that a large share of its evidence was self-censored: the gold check probe errored on 35 of 75 runs because of a path error, a revoked API key produced empty patches on every node of the forward arm's runs, pre-application conflicts were recorded as refutations, and the fallback integrator emitted empty patches. It is not a ranking. Over the induced task models of that run, TMI observed 55 sequential adjacencies between phases, could name a consumed artifact for 37, and the withholding controls validated 7: order is not dependency.

Production choice. The production constructor is the hierarchical work-unit policy with the corrective screen retry and the gold check probe, with direct as the fall-through when the probe gate rejects. It was the only configuration with a measured yield at scale when production started (37 of 78 certified in a fleet queue under the earlier semantics, before the methodology freeze). The preregistered comparison is consistent with the choice without establishing a winner.

O Compiled-execution audit: definitions, raw outcomes and fidelity

§A.3 defines compiled execution, its plan schema, executor and refusal policy; §A.4.3 reports the audit by failure attribution with its category table. This appendix gives the raw strict-policy view (Figure 32), the denominators, the run-level fidelity audit and the per-task list, so that a reader can recompute either view.

Denominators used in the paper. 111 compiled runs exist across the release and the archived controls (one task, `spice-circuit-simulator-in-rust`, was run twice on 2026-09-22 and its repaired-plan rerun is the run counted); 109 of them are attributable to the graph (categories 1–3); 102 are released tasks; 20 released tasks have no compiled run; and the run-level fidelity audit below

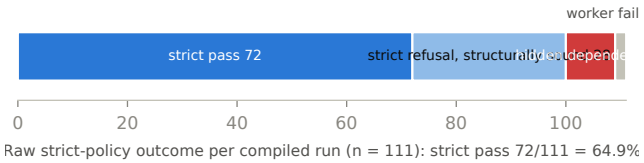


Figure 32: Raw strict-policy view of the same 111 compiled runs, by category. This is the executor's fail-closed outcome, not a statement about graph validity; Figure 13 in §A.4.3 gives the attributed view.

covers 42 runs (41 distinct graphs) that were executed through the plan executor with full schedule capture.

O.1 Run-level fidelity audit

The six fidelity properties and the aggregate result (0 violations; realized maximum concurrency equal to declared level width in 42 of 42 runs) are stated in §A.3; this subsection gives the population and the raw outcomes. The 42 runs cover 38 released tasks (one run twice), 1 archived control and 2 tasks outside the release; the other compiled runs of Table 6 have no fidelity audit. At this level 35 passed the verifier, 4 refused composition on a sibling file overlap (under the release categories two of these are hidden dependencies, `multi-gitter` is a strict-policy refusal whose diagnostic composition passed, and `drizzle`'s strict re-run passed and is its release run), 1 failed after a two-node chain's nodes both succeeded (composition semantics or a hidden interface dependency), and 2 were environment failures, one of them a safety refusal on a worker's build command. Mean concurrency ranged from 1.6 to 5.9 (median 2.65); orchestration overhead (wall minus critical path) from 0.7 to 6.5 minutes (median 1.7); the median ratio of compiled wall to the graph's critical path was 1.31. Figure 14 in §A.4.3 plots realized against declared concurrency.

O.2 Per-task audited list

Strict passes (category 1). `abs-module-cache-flags`, `aiomonitor-task-snapshots-diff`, `arcane-drift-detection-baselines`, `bandit-interprocedural-taint-checks`, `cerbos__cerbos-1655`, `clack-async-autocomplete-options`, `crossplane__provider-aws-1240`, `cuthbertlab__music21-1337`, `dai-lab__copulas-100`, `docker__compose-9148`, `drizzle-orm-window-function-builders`, `dynamodb-toolbox-conditional-attribute-requirements`, `dynamodb-toolbox-lazy-recursive-schemas`, `fastapi-implicit-head-options`, `filecoin-project__specs-actors-378`, `frontier-cs-netdeny-175`, `frontier-cs-netdeny-47`, `frontier-cs-netdeny-87`, `geo-shapeindex-serialization`, `go-genai-streamed-function-args`, `go-git-worktree-merge-conflicts`, `goccy__go-yaml-103`, `googlefonts__ufo2ft-480`, `happy-dom-abort-pending-body-reads`, `httpx-deterministic-cookie-store`, `httpx-multipart-response-parsing`, `instance_ansible__ansible-1b70260d5aa2f6c9782fd2b848e8d16566e50d85-vba6da65a0f3baefda7a058ebbd0a8dcafb8512f5`, `instance_ansible__ansible-b2a289dcbb702003377221e25f62c8a3608f0e89-v173091e2e36d38c978002990795f66cfc0af30ad`, `instance_ansible__ansible-deb54e4c5b32a346f1f0b0a14f1c713d2cc2e961-vba6da65a0f3baefda7a058ebbd0a8dcafb8512f5`, `instance_ansible__ansible-f8ef34672b961a95ec7282643679492862c688ec-vba6da65a0f3baefda7a058ebbd0a8dcafb8512f5`, `instance_flipt-io__flipt-3ef34d1fff012140ba86ab3cafec8f9934b492be`, `instance_flipt-io__flipt-b2393f07d893024ab1e47ea2081e0289e1f9d56f`, `instance_flipt-io__flipt-cf06f4ebfab7fa21eed3e5838592e8e44566957f`,

instance_flipt-io__flipt-dbe263961b187e1c5d7fe34c65b000985a2da5a0, instance_future-architect__vuls-54e73c2f5466ef5daec3fb30922b9ac654e4ed25, instance_navidrome__navidrome-5e549255201e622c911621a7b770477b1f5a89be, instance_navidrome__navidrome-66b74c81f115c78cb69910b0472eeb376750efc4, instance_navidrome__navidrome-f7d4fcdcc1a59d1b4f835519efb402897757e371, instance_protonmail__webclients-0d0267c4438cf378bda90bc85eed3a3615871ac4, instance_protonmail__webclients-3f22e2172cbdfd7b9abb2b1d8fd80c16d38b4bbe, instance_protonmail__webclients-5e815cfa518b223a088fa9bb232a5fc90ab15691, jelmer__dulwich-1586, jumptrading__influx-spout-94, kcp-go-multiplexed-kcp-streams, knative__client-682, koota-entity-snapshot-rollback, kysely-window-grouping-helpers, llm-inference-batching-scheduler, mit-ll-responsible-ai__hydra-zen-104, narwhals-dev__narwhals-2747, obsidian-linter-link-format-conversion, ofetch-per-origin-circuit-breaker, onedump-dump-encryption-pipeline, openmdao__openmdao-2931, ossf__package-analysis-839, psd-tools-blend-range-api, pwntools-tube-multiplexing, qubit-routing, quill-shared-toolbar-focus, reshard-c4-data, shepmaster__snafu-47, tengo-destructuring-bindings, testem-per-launcher-reports, textual-richlog-follow-state, theacodes__nox-187, ts-pattern-match-each, twpayne__chezmoi-1034, valibot-recursive-schema-composition, vegaprotocol__vegawallet-567, wazero-multi-module-snapshots, winning-avg-corewars, yaegi-go-embed-directives.

Full task identifiers and per-task fields are in `paper/figures/data/compiled_audit_tasks.json`.

P Dataset provenance

P.1 Source benchmarks

The public release draws on 4 sources (Table 7, Appendix B.1): DeepSWE, SWE-rebench V2, SWE-Bench Pro and Terminal-Bench 2. Table 3 in §A.2.1 gives the composition of the historical curated pre-release pool, which drew on 7 sources, all software tasks with hermetic verifiers and GPU-free containers: DeepSWE (Huang et al., 2026), original repository feature tasks with hand-written verifiers; SWE-rebench (Badertdinov et al., 2025; 2026), monthly batches of repository tasks run under network-denied egress because the only observed leakage channel was live upstream fetching; the public slice of SWE-Bench Pro (Deng et al., 2025); Terminal-Bench 2.0 (Merrill et al., 2026); FrontierCS (Mang et al., 2026) competitive-programming tasks with graded verifiers; FrontierSWE (Proximal Labs, 2026) ultra-long-horizon engineering tasks with graded verifiers; and Long-Horizon-Terminal-Bench (Li et al., 2026). Several other sources were scouted and excluded on evidence: BrowseComp-Plus search tasks, because their parallel branches were substitutes rather than complementary work (removing one did not change the outcome); tau-bench, because policy gates force admissible width to one; GAIA and MathArena Apex, because omission controls passed from model memory or reduced to width one after verification work was removed; and several very-long-horizon sources whose verifiers could not be run reproducibly. The ledger (`docs/research-history-and-findings.md`, §15) lists every source decision with its evidence.

P.2 Certification metadata

Every task carries three independent fields: the *pool* (public admission decision from intrinsic structure), the *certification level* (the replay controls' verdict on the graph's claims), and the *compiled-execution status* (the outcome of one strict deterministic run where one exists). The counts in the rest of this paragraph are over the historical curated pre-release pool (122 tasks); the public-release

counts are in Appendix B.3. Over that pool the levels are: fully certified 45, verifier-passing partial 68 (of which 22 carry a refuted local-check claim), serial/control 3, manual review 6. Over the PARALLEL pool: fully certified 38, partial 68 (refuted local check 22), manual review 6; 106 (95%) are certified by replay at least partially. Mechanically derived graph statuses over the set: certified 45, executable-inconclusive 28, failed-review 43 (a refuted local-check or necessity claim with a verifier-passing union), unreplayed 6 (the manual-review admissions). 116 tasks have a recomposed union that passed the verifier; 6 have gold-only verifier evidence. Replay cost over the released graphs totals $2,603 (median $15.7 per task).

Partial certification. A partial graph's recomposed union passes the unchanged verifier while at least one claim is inconclusive or refuted. The two rules that most often hold graphs at partial are conservative: sibling independence stays inconclusive on any shared file even when the union composed cleanly and passed, and necessity is ablated by patch hunk rather than by node output on shared files (§Q). A refuted local check means a node's own check failed in isolation while the union passed; the check quality, not the work, is in question. Neither rule was relaxed for this release.

Manual-review admissions. 6 PARALLEL tasks carry `certification_level: manual-review`: five FrontierSWE tasks (`dart-style-in-haskell`, `quantum-espresso-pwx-in-rust`, `notebook-compression`, `verilog-simulator-in-swift`, `spice-circuit-simulator-in-rust`) whose pipeline runs were cut off by the acquisition freeze or failed on probe-environment and capture-hygiene causes, and one SWE-rebench task (`rochacbruno__dynaconf-243`) whose replays died on a since-fixed workspace-root bug. Each was rebuilt deterministically from its stored pipeline artifacts by `experiments/manual-curation-20260921/build_curated.py`, which records every edit with its evidence, copies the provenance (candidate and reviewed graphs, reviews, check-probe results, replay records) and runs the static consistency check of §M.7; the spice graph additionally passed an adversarial static review with edits (`docs/spice-adversarial-review-20260921.md`). After a compiled probe refused composition on a shared module-registration file, the spice and quantum-espresso graphs were repaired so that one root unit owns every registration or manifest file (contract facts F10 and F18; `docs/compiled-release-20260922-opus5-probe.md`); the spice repair composed without conflict in a rerun whose graded verifier still fell short (§A.4.3), the quantum-espresso repair is not validated by a run, and the new root unit of the latter carries no trace steps, so that graph leaves the trace-order population. Their metadata records `verifier_passed: gold-only` and the exact automated failure that prevented certification. Two further FrontierSWE candidates (`lua-native-compiler`, `revideo-perf-opt`) were not admitted and are preserved as case studies (§R).

Salvage admissions and the held task. A zero-spend audit of every task excluded from the release (`docs/salvage-audit-20260921.md`) recovered tasks whose exclusion was bookkeeping rather than evidence: `kombu-single-active-consumer-priority` (verifier-passing deliverable dropped because its graph file had not been synced) was admitted with its replay evidence, and `rochacbruno__dynaconf-243` as above. A third, `pebble-durability-wait-apis`, was admitted and then *held* the same day: its released deliverable is unreplayed with an invalid round-two replay, and the verifier-passing replay belongs to the pre-revision graph whose derived status is failed-review. It sits under `archive/held/` with the reason in its metadata and is in neither pool. A fourth candidate (`sqlfmt-create-table-ddl-formatting`) was not admitted because its verifier-passing graphs use the earlier artifact semantics.

P.3 LEGACY ANALYTIC TIERS

The 2026-09-20 curation assigned CORE (29) and EXTENDED (77) to PARALLEL tasks by the thresholds of §M.7; the finalization kept them as metadata because the evaluation uses CORE as a covariate. CORE tasks have product work median 12.0 minutes (IQR 10.9 to 15.7), second-branch median 4.0 minutes, work over span median 1.89, and product widths two / three / four / five or more of 11 / 9 / 6 / 3; 17 are fully certified and 22 have a strict compiled pass. An EXTENDED task is a valid PARALLEL task that misses one CORE threshold.

P.4 CONTROLS: SELECTION RULE AND ARCHIVE

The 10 controls were chosen from 37 certified width-one candidates by a deterministic rule (`docs/control-selection-20260921.md`): cover every source with an eligible control (7 sources); cover both structures (4 serial with no support nodes, 6 support-parallel with up to 5 support nodes, so the over-parallelization test sees narrow and wide support structure); spread single-agent horizon from 7 to 125 minutes where recorded (5 known); spread task type (feature, bugfix, debugging, competitive programming, optimization, model training, data audit); prefer fully certified graphs and compiled-execution passes; binary verifiers over graded ones; at most two controls per source and no two with the same source, structure and type. 7 of the retained controls are fully certified. The other 27 certified width-one tasks are archived under `archive/controls/` with their task directories, reference runs and metadata intact, listed in the manifest, and excluded from the release.

P.5 FUNNEL BY STAGE

Under the frozen final methodology and the regenerated curation input (one production deliverable per task, experiment and holdout records excluded): 345 tasks attempted, 278 with a construction deliverable, 235 with a verifier-passing recomposed union, 225 at least partially certified, 138 fully certified, 146 in the usable pool after curation (30 CORE, 77 EXTENDED, 39 control candidates), 199 rejected by curation (89 of them verifier-passing, 84 for product work under the EXTENDED floor), and 122 in the historical curated pool, from which the 104-task public release was cut (Appendix B.2). Across all 12 production waves the curation input holds 1016 task-wave rows, 525 verifier-passing. Replay spend over the funnel totals $5,875.

Q PIPELINE HISTORY THAT A REPRODUCTION NEEDS

The full ledger, with commits and evidence paths for every entry, is `docs/research-history-and-findings.md`; its period files under `docs/research-history/` hold the day-by-day record. This appendix keeps only what a reader needs to interpret the released artifacts and to avoid the bugs that changed earlier conclusions.

Rule. A run that did not happen as specified is inconclusive or censored, never refuted. Infrastructure failures censor; measurement failures produce wrong verdicts from correct runs and are re-scored offline; methodology failures are wrong definitions and are dated.

Representation. The artifact went from a topology tree, to a typed graph with explicit independence claims and contract facts, to a minimal series-parallel tree, to the graph-native reference DAG under verification-coupled semantics (v1), to the reference DAG under work-dependency semantics with product and support roles (v2) and the contract-first edge type. A series-parallel tree cannot express a diamond and its conversion invented orderings: release v1 carried 72 artifact edges from that conversion. Under v1 a check that needed the assembled program could establish an edge; an audit of release v1 (88 graphs, 848 structural relations) found independence claims validated at scale (343 of 400) while dependency edges almost never were (1 of 448 by intervention; 199 graded validated on definitional grounds about integration nodes). A parallelism audit of 118 certified v1 graphs found that most validated edges recorded verification or integration order: in a stratified sample of sixteen graphs, at least six were implementable in parallel against their published interfaces and 4 were genuinely serial. The contract-first control (§M.4) resolved this by asking whether the consumer could be *authored* without the producer; in the pilot that introduced it, 8 of 9 reconstruction-inconclusive edges were refuted as interface and the one genuine produced-state dependency validated. v1 artifacts retain their label and are not promoted.

Calibrating the contract-first reading. The stage-two rule was wrong four ways in its first complete run and four more in its third, every one visible in stored evidence: scaffold names read as reconstruction; silent placeholders making check failures uninterpretable; reconstruction in data invisible to a code detector; reserved files forbidding legitimate stub modules; a marker two lines from a definition; a marker outside the inspected log tail; a type declaration and a one-line signature tripping symbol rules; compile-time failures with no marker in the output. Worker behaviour was

consistent across all 26 stage-two runs; every fix was to how evidence was read. The calibration set was 9 edges in three languages.

Where certification is conservative. Of 15 verifier-passing but uncertified graphs in the first production waves, 12 were inspected by three independent readers; 9 would be included by a human and 3 were correctly rejected (single-file refinement chains whose integration nodes re-derived their inputs). The file-overlap independence rule was the sole blocker on 4 of the 12. Certification was kept frozen and the relaxation specified with its adversarial cases; the release publishes certification level per task instead.

Where the new semantics first landed. The work-dependency semantics were introduced through an eight-case diagnostic pilot; no case received an accepted work-mode certificate, because interface edges were still declared as work, controls reconstructed in scratch state, and local checks covered too little. The requalification pilot that followed certified 2 of 6 tasks under work semantics against 5 under legacy semantics, because reconstruction had no repair path; the contract-first control supplied it.

Production and freeze. Production ran in 12 waves under a frozen code snapshot. On 2026-09-21 acquisition was frozen (no new trajectories, replays or model calls); the historical release candidate was finalized into two pools (later cut to the 104-task public release, Appendix B.2), the control pool was reduced from 37 to 10 with the rest archived, 5 FrontierSWE graphs were admitted after manual static review and 3 tasks were recovered by a salvage audit (one later held, §P), and the curation precedence bug in Table 29 was fixed. Spend on the project's evaluation-side replays and compiled runs is recorded per task in the release metadata (`replay_usd_real`); the pilots cost $289 and $214, and the compiled reference runs in the 42-run fidelity audit $329 (the 102 released reference runs spent $877, of which $186 is the Opus 5 spice rerun).

R QUALITATIVE CASE STUDIES

Six released or preserved graphs illustrate the structural situations MACE-Bench distinguishes (one item card is Figure 8 in §A.2.3; the pilot trace readings are §5). Every fact below is read from the task's released files (`reference_dag.json`, `metadata.json`, `reference_runs/`) or from the named audit document.

Genuinely parallel: `dynamodb-toolbox-conditional-attribute-requirements`. The worked example of §A.1 (Figure 6), a DeepSWE repository feature. Its 7 product nodes form a fan-out of width 6 with 6 edges and depth 2: product work 31.7 minutes, span 11.2 minutes, work over span 2.83, second branch 5.5 minutes, branch balance 0.97. Under compiled execution every branch ran concurrently (realized maximum concurrency 6, mean 3.1); the run took 14.5 minutes against a critical path of 11.7 minutes (orchestration overhead 2.8 minutes), a factor of 2.41 below the sum of worker time. This is the regime the benchmark is for: several substantive branches and a short span.

Genuinely serial: `httpx-multipart-response-parsing`. A DeepSWE feature retained as a CONTROL. Its graph is a chain with a single product node per level and no support node; gold ablation found no independent group and the recomposed union passed the verifier, and the compiled reference run passed under the strict policy. A system that spawns workers on this task can only pay coordination cost.

Product-serial with support parallelism: `frontier-cs-netdeny-87`. A FrontierCS competitive-programming task retained as a CONTROL. One dominant solver node carries the product work; the other nodes are support (generators, checkers, a local judge) that run in parallel with it and were consumed during the trajectory but do not change the verifier outcome when omitted. The compiled run realized the graph's full executable width while its product width is one; the fidelity record shows realized concurrency equal to declared width. The case separates "a wide graph" from "available product parallelism".

Under-parallelized decomposition. The parallelism audit of the certified v1 fleet (`docs/dag-parallelism-audit-2026-09-16.md`) found width-one graphs whose only edges were verification order: a consumer whose check needed the assembled program had been serialized behind a producer it could have been authored alongside (for example a query engine feeding a scripting module, or a core module feeding its tests). In the audit's stratified sample of sixteen certified graphs, at least six were implementable in parallel against their published interfaces and only 4 were genuinely serial. These graphs motivated the contract-first control and the v2 semantics; they are not in the release.

Hidden dependency versus policy refusal: `participle-grammar-conflict-analysis` and `lindell_multi-gitter-215`. The first is one of the 9 category-3 compiled runs (Table 28): every node passed its own check in isolation and the mechanically composed state failed the original verifier, so one sibling's correctness depended on state another sibling produced, which no pairwise control had exercised because each node was checked alone. The second is a category-2 run: two siblings edited the same product file, the strict policy refused composition, and the diagnostic composition passed the verifier; the release records it as a policy refusal, not a hidden dependency, and the pilot's MAS+DAG run on the same task passed. Both graphs keep their certification records; only the first carries a hidden-dependency compiled-execution status.

Shared-file composition: `lua-native-compiler` (not released). A FrontierSWE task with a solved trajectory whose reviewed graph made the back-ends, the interpreter, the dump format and the parser port siblings of one core node. In the replay every sibling also edited the same hub files (code generation, runtime generation, glue and entry point), so the mechanical union conflicted although each sibling's own delta passed its check. A single-file-ownership repair with a pre-declared registration surface was proposed but not applied, because no successful replay supports it and the trajectory shows the runtime generator edited across phases. The graph is preserved under `experiments/manual-curation-20260921/case-studies/` as the canonical shared-file composition failure; the 28 strict-refusal runs of Table 6 are its benign relatives, where siblings touched a shared file in disjoint regions and the diagnostic composition passed.

S REPRODUCING EVERY NUMBER

No statistic in this paper is typed by hand, except the pilot-trace minutes of Appendix U, which are read from the cited pilot records under a stated exemption. Values prefixed `R` are regenerated from the released files (`release/benchmark-20260920/`), the regenerated curation manifest (`release/dags-20260919/curation-20260921/`), the constructor comparison results (`traces/constructor-compare-20260920/`), the compiled-run audits and the pilot records by one deterministic script, `paper/figures/make_figures.py`, which also renders every figure and writes the numbers behind each figure to `paper/figures/data/*.json`. Values prefixed `M` exist only in a committed report and are listed in `paper/numbers-manual.tex` with their source path. The appendix table of compiled runs is rendered by `paper/sections/appendix/gen_tables.py` from the same data. The research ledger (`docs/research-history-and-findings.md`) links every experiment named in this paper to its report, evidence directory and commits.

Final four-arm evaluation. Values prefixed `trs` come from `analysis/final/numbers.json`, which `analysis/final/build_numbers.py` computes from the 1248 cells listed in `analysis/final-cells-manifest.json` (and only those; older runs, superseded cells and every compiled-executor arm are excluded by run identifier); `analysis/final/make_numbers_tex.py` writes them to `paper/figures/candidates/numbers-20260925.tex`. Values prefixed `trx` (dictionary-valued entries of the same file) and the generated tables in `paper/sections/tables-20260925/` are written by `paper/figures/trace-20260925/make_trs_tables.py`; values prefixed `tcs` (the annotated traces) by `paper/figures/trace-20260925/make_case_macros.py` from `analysis/trace_examples/`. The final-evaluation figures in `paper/figures/trace-20260925/` are rendered by `analysis/final/figs/` and carry a JSON sidecar with every plotted value. `paper/check_trs_macros.py` re-derives every `trs` and `trx` value from the number file, re-

generates the `tcs` file, and fails on any macro used in the paper but not defined. The provenance of each number, from raw trial records to macro, is in `analysis/final/provenance.md`.

Release layout. `manifest.json` lists the 122 tasks of the historical curated pool (the 104 public-release tasks are marked by `in_evaluation_set` and listed in `evaluation_set.json`) with pool (PARALLEL or CONTROL), source, certification level, legacy analytic tier, admission provenance and structure metrics, plus the 27 archived controls and the held task. Each `tasks/<task>/` folder holds `task.json` (the original task: instruction, base commit, image, timeouts), `reference_dag.json` (the graph with per-claim evidence and provenance), `execution_plan.json` and `execution_plan.py` (the deterministic orchestration interpretation and its rendering), `metadata.json` (pool, legacy tier and reasons, source, horizon, width, work and span, certification level, verifier status, compiled-execution outcome and category, reviewer grade, caveats, admission note), and `verifier/` (the original tests; solutions are never included). `reference_runs/<task>/` holds one compiled execution per task that has one (schedule, record, metrics). `archive/controls/` holds the controls removed from the release and `archive/held/` the task withdrawn after admission, both with metadata intact.

Validation. `scripts/validate_release.py` checks the manifest against the task folders (pool and source counts, duplicate identifiers, directory completeness, execution plans, DAG validity, archive consistency, README, leak patterns) and exits non-zero on any error; `scripts/run_all_tests.sh` runs every unit-test module. Both are deterministic and offline.

Certification code. Construction, review, replay controls, the gold check probe, contract-first stage two, bounded revision, status derivation, the execution-plan compiler, the plan executor and the fidelity audit are in the repository at the commit hash recorded in every artifact's provenance. Every verdict in a replay record is re-scorable offline from the stored control patches and check outputs. The manual-review admissions are rebuilt deterministically from their stored pipeline artifacts by `experiments/manual-curation-20260921/build_curated.py`, which records every edit with its evidence and runs the static consistency check.

Build. `cd paper && latexmk -pdf main.tex`. The figure script needs `matplotlib` and `numpy`; it is byte-deterministic on re-run.

T ANALYSIS PLAN OF THE 2026-09-22 DRAFT AND THE PILOT TAXONOMY

The plan below was written in the 2026-09-22 draft, after the pilots had been read and before any final-matrix run existed; it is reproduced as written (bracketed notes excepted), including its references to the compiled arm. Of it, only the outcome readings are used on the final matrix (§C.1.5); the final analysis departs from the rest in the ways listed there, and the final traces were labelled with the post-hoc taxonomy of §F.1.6, not with Table 30, which was applied to the pilots only. Because the plan and its taxonomy were shaped by the pilots, we do not call them pre-registered. The full protocol, the failure-analysis coding scheme with its per-category evidence fields, and the optional intervention-study protocol are maintained with the paper source (`paper/analysis-protocol.md`, `paper/failure-analysis-plan.md`, `paper/intervention-plan.md`).

The planned metrics, outcome definitions and primary analyses, the reading of every outcome pattern and the failure taxonomy (Table 30) follow; the intervention protocol and the candidate interventions read off the pilots are in the released analysis plan and are not reported here. The conditions they refer to are those of the pilot design, with the compiled executor as fourth arm.

T.1 METRICS, OUTCOMES AND PRIMARY ANALYSES AS PLANNED (2026-09-22)

Metrics. Per task and arm: success under the unchanged verifier (reward against the task's acceptance threshold for graded verifiers); wall minutes, with timeouts recorded as censoring; workers spawned; realized concurrency (peak and mean active workers, overlap fraction, and worker-on-worker overlap as distinct from any-worker-active time); parent wait and merge tail; per-worker start and end times, file events and tokens; model time; and dollars only where runs are billed through

an API key, otherwise marked unavailable rather than estimated. Compiled walls are executor walls and exclude pod scheduling; single and multi-agent walls are end-to-end trial walls, so a compiled-versus-agent comparison favours the compiled arm slightly, and the paper says so wherever it makes one.

Outcomes. A run *succeeded* when the verifier reward is at or above the task's acceptance threshold (1.0 for binary verifiers, the recorded threshold for graded ones); it *failed* when the verifier ran and the reward is below threshold; it is *censored* when it cannot be scored (infrastructure exception, rate limit, host suspend, missing wall time, agent timeout). Censored runs leave success denominators and every latency statistic; their counts are reported per cell, timeouts separately.

Primary analyses, fixed before any final run. (1) Success rate per arm with a Wilson interval, per regime and on controls. (2) Matched speedup per arm against Single: for each task where both have a successful replicate, median single wall over successes divided by median arm wall over successes, aggregated as a geometric mean with a bootstrap interval over tasks; the matched count is printed and matched sets are never merged across arms. (3) An equal-success view over tasks every arm solved, the headline cross-arm comparison. (4) Converted fraction: matched speedup over work/span per task, reported against both W/S and single-wall over span (§H.1). (5) Realized concurrency: maximum and mean active workers, overlap fraction. (6) Cost on all runs and on successes, dollars only where the whole population was billed through an API key, otherwise tokens and model time with dollars marked unavailable; each arm placed on the time–cost plane. (7) Overhead decomposition: parent wait, merge tail, worker startup, verification, as fractions of wall. (8) The regime interaction, pre-declared as a hypothesis and not fitted. There is no post-hoc task selection on evaluation results, and pilots are never pooled with the matrix. [Note added 2026-09-25: the slate rule below removes the graphs whose compiled run exposed a hidden dependency; it is pre-specified but not neutral, because the DAG-Sched arm executes the same units (§C.1.1).]

Task set, controls, replicates. The release is the frozen artifact; the matrix runs on an *evaluation slate* derived from it by a rule fixed before any run: every released task minus those whose compiled run exposed a hidden dependency (category 3, 8 tasks at the time of writing) and minus the one Long-Horizon-Terminal-Bench control, which the evaluation plan excludes, giving 113 tasks (104 PARALLEL, 9 CONTROL); the 19 slate tasks with no compiled run yet are compiled and validated first, and a hidden dependency found there removes the task before the matrix starts. This is an evaluation-slate rule, not a benchmark admission rule: the release keeps every task with its three per-task fields (§A.2.3). Speedup claims are made on the slate's PARALLEL tasks stratified by the regimes of Table 5, computed from the reference graph's replay durations; tasks without durations are reported but excluded from strata. The width-one controls run under the same arms and are reported separately and never enter speedup aggregates: a system that spawns workers on a serial task pays coordination for nothing, and that cost is the control's measurement. The graph is never shown to Single or MAS. Three replicates per cell where budget allows, one otherwise, with the replicate count printed in every table. [Resolved in the final matrix: the evaluation set of §C.1.1, frozen during the runs, replaces the slate rule above; one run per cell; systems, versions and substrate in §C.1.1–§C.1.2; the Codex harness was run as one of the three systems; other harness and model systems are future work.]

T.2 READING OF EVERY OUTCOME PATTERN AS PLANNED

No pattern is the intended one. Similar gains across the three parallel arms mean systems already exploit the opportunity; MAS alone trailing means planning; MAS and MAS+DAG trailing together means orchestration; an ordered triple means both, to be split by regime; no arm beating Single means overhead exceeds opportunity on this substrate; gains only in the long-and-parallel cell mean horizon and branch substance gate usefulness; speed rising with cost means latency was bought with inference; and the compiled arm losing to the orchestrator on a task means the reference graph was wrong there. Each shortfall between available and realized parallelism is then coded against the failure taxonomy of Table 30, fixed before results are read, with one primary category per run under a stated precedence; Appendix U applies it to the pilots and records the rules that had to be corrected when applied literally.

T.3 Precedence and logging

Two further details belong here.

Precedence for the primary category. For a failed run the primary category is the first that fires of A2, I1, L1, then the worker and integration proxies; for a successful run it is D1 by active agents, then D3, then the largest in minutes of S1, C2, O2, I1 and A2. The precedence was fixed after one pass over the pilots of Appendix U and before any final run; the corrections to the literal rules that pass produced are listed in the plan's §1a.

Logging fields added for the final matrix. The evaluation harness and the plan executor now record, with no change to what any agent does: authentication and billing mode, cost availability, censoring reason and the timeout cap; per worker, tokens, distinct model calls, workspace (shared or worktree), time from spawn to first event and first-edit fraction; path-normalized shared-file lists split between parent and worker; the ratio of worker-on-worker overlap to any-worker overlap; distinct model calls per model-time bucket; and per node in compiled runs, dollar cost, token usage and setup seconds. The instrumentation audit and its verification (`docs/instrumentation-audit-20260921.md`, `docs/instrumentation-verify-20260922.md`) list every field with the code location that writes it. Still missing, and named for the matrix: per-worker patch hashes and hunks, reference-node mapping of worker output, verifier failed-test lists, retry seconds, worker outcome text, and per-node environment-setup versus agent time in compiled runs.

U Pilot runs (not reported)

Before the final evaluation, small pilots ran an earlier arm set whose fourth arm was the compiled plan executor of Appendix A rather than the runtime scheduler DAG-Sched. Those pilots shaped the protocol but are superseded by the final four-arm evaluation; none of their numbers is reported in this paper, and no pilot run enters any comparison.

Table 28: Every compiled run outside the strict-pass category ($n = 111$ compiled runs in all; the 72 strict passes are listed below the table). Category: 2 strict shared-path refusal whose diagnostic composition passes (structurally executable); 3 hidden dependency (structural failure), with its kind; 4 worker stochastic failure (not charged to the graph). Width is the graph's product width; set marks released tasks against archived controls. The hidden dependencies are few and named; the strict refusals are the conservative policy at work.

Task	Source	Cat.	Outcome	Width	Set
abs-stepped-slices	deep-swe	2	strict refusal, diagnostic pass	2	evaluation
awilix-async-container-initialization	deep-swe	2	strict refusal, diagnostic pass	2	evaluation
axelrod-python__axelrod-699	swerebench-v2	2	strict refusal, diagnostic pass	3	evaluation
gembaadvantage__uplift-374	swerebench-v2	2	strict refusal, diagnostic pass	4	evaluation
go-critic-doc-link-checker	deep-swe	2	strict refusal, diagnostic pass	2	evaluation
googleforgames__open-match-1268	swerebench-v2	2	strict refusal, diagnostic pass	1	archived-control
grafana__grafana-plugin-sdk-go-330	swerebench-v2	2	strict refusal, diagnostic pass	2	evaluation
helm-unified-manifest-stream	deep-swe	2	strict refusal, diagnostic pass	4	evaluation
instance_ansible__ansible-942424e10b2095a17...	swe-pro	2	strict refusal, diagnostic pass	4	evaluation
instance_ansible__ansible-d62496fe416623e88...	swe-pro	2	strict refusal, diagnostic pass	2	evaluation
instance_flipt-io__flipt-e42da21a07a5ae3583...	swe-pro	2	strict refusal, diagnostic pass	3	evaluation
instance_internetarchive__openlibrary-d109c...	swe-pro	2	strict refusal, diagnostic pass	4	evaluation
instance_navidrome__navidrome-5001518260732...	swe-pro	2	strict refusal, diagnostic pass	4	evaluation
lindell__multi-gitter-215	swerebench-v2	2	strict refusal, diagnostic pass	5	evaluation
moov-io__customers-179	swerebench-v2	2	strict refusal, diagnostic pass	2	evaluation
narwhals-dev__narwhals-247	swerebench-v2	2	strict refusal, diagnostic pass	3	evaluation
olivere__elastic-1415	swerebench-v2	2	strict refusal, diagnostic pass	3	evaluation
pvlib__pvlib-python-278	swerebench-v2	2	strict refusal, diagnostic pass	3	evaluation
pvlib__pvlib-python-773	swerebench-v2	2	strict refusal, diagnostic pass	2	evaluation
pybamm-team__pybamm-1020	swerebench-v2	2	strict refusal, diagnostic pass	2	evaluation
query-persist-restored-query-state	deep-swe	2	strict refusal, diagnostic pass	2	evaluation
remeda__remeda-443	swerebench-v2	2	strict refusal, diagnostic pass	4	evaluation
returns-validated-error-accumulation	deep-swe	2	strict refusal, diagnostic pass	4	evaluation
scc-bounded-memory-spilling	deep-swe	2	strict refusal, diagnostic pass	3	evaluation
sql-formatter-bigquery-pipe-formatting	deep-swe	2	strict refusal, diagnostic pass	2	evaluation
tengo-callable-instance-isolation	deep-swe	2	strict refusal, diagnostic pass	2	evaluation
textual-kitty-key-phases	deep-swe	2	strict refusal, diagnostic pass	4	evaluation
tikv__client-go-843	swerebench-v2	2	strict refusal, diagnostic pass	2	evaluation
dasel-html-document-format	deep-swe	3	hidden dependency: product-file composition conflict	3	evaluation
drivendataorg__cloudpathlib-474	swerebench-v2	3	hidden dependency: product-file composition conflict	4	evaluation
ink-grid-box-layout	deep-swe	3	hidden dependency: product-file composition conflict	3	evaluation
instance_ansible__ansible-d30fc6c0b359f6311...	swe-pro	3	hidden dependency: product-file composition conflict	4	evaluation
instance_protonmail__webclients-8be4f6cb938...	swe-pro	3	hidden dependency: product-file composition conflict	2	evaluation
opa-template-string-reconstruction	deep-swe	3	hidden dependency: verifier fails on the composed state	1	archived-control
participle-grammar-conflict-analysis	deep-swe	3	hidden dependency: verifier fails on the composed state	4	evaluation
python-statemachine-state-data-scoping	deep-swe	3	hidden dependency: product-file composition conflict	4	evaluation
spice-circuit-simulator-in-rust	frontierswe	3	hidden dependency: verifier fails on the composed state	6	evaluation
instance_ansible__ansible-502270c804c33d3bc...	swe-pro	4	worker failure	3	evaluation
true-myth-iterable-collection-combinators	deep-swe	4	worker failure	5	evaluation

Table 29: Failures that changed a stated conclusion or a released verdict. *Infra* censored runs; *measurement* produced wrong verdicts from correct runs; *methodology* was a wrong definition. Each row names what it invalidated; every re-scoring was done offline on stored evidence.

Failure	Class	Mechanism	Invalidated / consequence
Reconstruction read as refutation	measurement	consumer rebuilt withheld producer and passed its own check; detector only saw "touched producer's file"	16 of 20 Phase A refutations; "constructors declare false dependencies" withdrawn
Reconstruction read as validation	methodology	symbol overlap on a shared file certified the edge	interface dependencies certified as work under v1; motivated v2 semantics
Double-applied predecessor patches	infra	node captures diffed against base while inputs were pre-applied	17 of 28 verifier failures in the first graph-native wave were "patch does not apply"; its yield of 27 certified of 74 withdrawn
Definitional dependency edges	methodology	integration node "consumes predecessors by definition" graded validated	all 199 such edges in release v1
Tree representation	methodology	series-parallel tree cannot express a diamond; conversion invented orderings	72 artifact edges in release v1; tree retired
Empty trajectory in constructor prompt	infra	wrapper re-rendered to an empty step list	every construction before 2026-09-15 was gold-only; Phase A ranking withdrawn
Node checks needing node-written test files	measurement	check failed when the worker named the file differently, passed when the worker graded itself	31 of 137 checks; gold check probe made a hard gate
Fail-open review	measurement	a review that errored was recorded as accept	candidates funded without a review verdict; fixed before the production waves
Stale replay cache	infra	evidence reuse matched on a stale key after a code change	reused verdicts recomputed; reuse key now includes code version and patch hashes
Frozen-image capture mismatch	infra	"base commit missing" in images built the same night	27 tasks censored in one wave; re-replayed
Workspace root / empty leaf patches	infra	repositories mounted at a non-default root	18 of 25 replays in one wave had empty patches; re-queued (workspace root now read from the image)
Silent scaffold stubs	measurement	consumer's check failed on behaviour with no attributable cause	would have certified five interface edges as work in the first contract-first run
Reconstruction in data	measurement	consumer repaired the database itself and passed; code detector blind	the one genuine serial control refuted in that run; reserved-file rule added
File-overlap independence rule	methodology (conservative)	any shared file keeps a sibling pair inconclusive despite clean composition	sole blocker on 4 of 12 audited rejections; relaxation specified, not applied
Hidden sibling dependency	methodology gap	pairwise controls pass; compiled merge fails the verifier or product files conflict	9 of 109 attributable compiled runs; compiled execution adopted as a validation instrument
Curation input precedence	infra (bookkeeping)	experiment records shadowed production deliverables; verifier-passing rows dropped for a missing local graph file	two tasks lost from the 2026-09-20 release; fixed 2026-09-21 (curation regeneration report, §P)

Table 30: Failure taxonomy for the gap between available and realized parallelism: eight families, each category with an observable signature in the recorded artifacts and a decision rule fixed in advance (`paper/failure-analysis-plan.md`). L2 and W2 are charged to the task or to variance, not to the system; O1 needs per-worker startup recorded before it is attributed in agent arms. Every loss the paper names has a signature a reader can check in the released records.

Code	Category	Signature and rule
D1	Planning: missed independent work	peak active agents (parent included when it edits product files) at most half the reference width while at least two independent product units of two minutes or more existed
D2	Planning: poor decomposition	one worker holds at least 80% of summed worker busy time, or the median worker is busy under two minutes
D3	Planning: support mistaken for product	at least half of worker busy time on support paths of the reference graph
S1	Scheduling: late launch	first spawn after 30% of the wall
S2	Scheduling: serialized independent work	worker-on-worker overlap under a quarter of any-worker-active time with two or more workers
S3	Scheduling: critical path starts late	the longest reference branch's worker starts after a quarter of the wall
A1	Assignment: duplicate work	a redispatch of the same prompt, or at least 30% of one worker's diff duplicated by another
A2	Assignment: ownership overlap	independent workers edited the same product file, weighted by whether a conflict followed
C1	Context: missing information	a worker re-derives upstream state, or its first product edit comes after 40% of its busy time
C2	Context: avoidable wait	parent wait of a quarter or more of the wall with idle workers
W1	Worker: branch failure	a node or worker did not deliver its unit (error, safety refusal, timeout)
W2	Worker: variance	wall coefficient of variation at least 0.3, or a success flip, across replicates of one cell (needs three replicates)
L1	Load balance: straggler	longest worker at least twice the second, with mean concurrency at most half the peak
L2	Load balance: bad units	the reference's second branch under two minutes; charged to the task
I1	Integration: merge conflict	strict refusal or product-file conflict at composition; conflict markers in agent merges
I2	Integration: locally passing, globally failing	every node passed its check and the union fails the verifier
O1	Overhead: startup dominates	fixed per-worker cost at least 30% of wall
O2	Overhead: coordination dominates	parent planning, waiting and merging at least the parallel saving
O3	Overhead: economics	cost ratio above speedup, or model-time ratio above 1.5 with speedup below 1.2