ASYNC-RBENCH: BENCHMARKING ASYNCHRONOUS TASK EXECUTION IN LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents increasingly execute tasks asynchronously: a main agent continues working while its subagents run, and the task state may change before their results arrive. The main agent must decide which work to update or preserve and which checks to repeat, while avoiding prohibited changes. Final task success alone does not reveal whether these results were handled correctly, and natural asynchronous arrivals are hard to reproduce. We introduce Async-RBench, a benchmark for asynchronous task execution in LLM agents, with 200 executable tasks covering eight scenarios. A scenario controller regulates result delivery using predefined task-state conditions, while an Event Response Contract (ERC) specifies what each response must change, preserve, avoid, and re-verify. The Dynamic Replanning Score (DRS) weights the response process and outcome equally and is reported separately from task success. We evaluate nine agents and find DRS scores of 19.2–45.1, with the lowest scores on duplicate reports and subagent failures. Under our execution protocol, all nine agents obtain lower task success with asynchronous than with batched delivery (26.0% vs. 29.7% on average). For three agents, prompting the main agent to rebuild its plan after every result improves DRS by only 2.6–3.5 points. Together, these findings highlight the challenges of reliable asynchronous task execution and the need to evaluate how agents handle incoming results alongside end-to-end task success.[1]

1 INTRODUCTION

Main–subagent architectures are increasingly common in large language model (LLM) agents: a main agent decomposes tasks, delegates subtasks to subagents, and integrates their results (Fourney et al., 2024; Masters et al., 2025). Subagents can run with separate contexts, specialized roles (Hong et al., 2024; Qian et al., 2024; Li et al., 2023), tools, and instructions (Wu et al., 2024; Gao et al., 2024), and production coding agents such as Claude Code and Codex provide built-in support for subagents (Anthropic, n.d.; OpenAI, n.d.). Rather than idling until every subagent finishes, the main agent can keep working and consume results *asynchronously* as they arrive. We call such an arrival during ongoing execution an *asynchronous interruption*.

The task state may change between a subagent's launch and the arrival of its result (Ginart et al., 2024; Yang et al., 2026). By the time a result arrives, the main agent may already have committed to a plan, written artifacts, or run validation, and the right response depends on how the result relates to that state. A late authoritative result must be acted on and may invalidate earlier work. A stale or duplicate report may require leaving valid work unchanged and avoiding repeated side effects. If a subagent fails, the main agent must recover the affected subtask while preserving valid work on other subtasks (Geng & Chang, 2025). In Figure 1, subagent B's cleanup changes the code base after subagent A has reported passing tests, so A's report no longer verifies the repository, although no message flags it as outdated. Responding correctly means adapting the patch to the new base, keeping C's server setup, and rerunning the affected tests; reusing A's report leaves the new release unverified. A correct response makes the changes required by a subagent result, preserves valid progress, avoids prohibited side effects, and performs the required validation. Correctness requires both making necessary changes and avoiding unnecessary ones. These errors are not symmetric in cost: ignoring a result that invalidates earlier work leaves artifacts the agent's own evidence no longer

[1]We release our code at https://anonymous.4open.science/r/Async-RBench-723A.

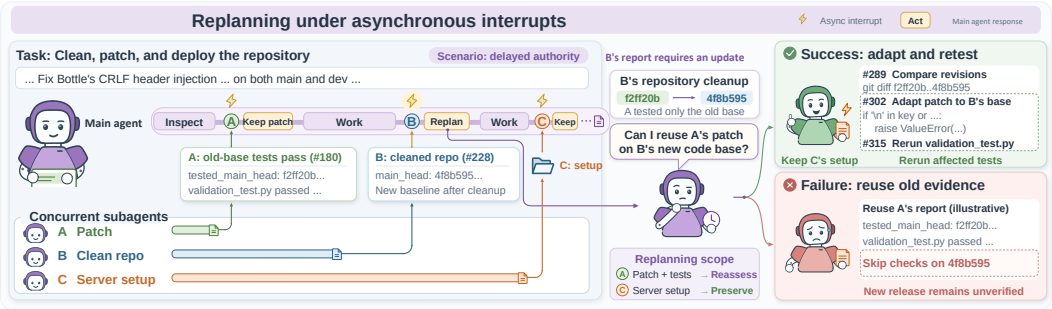


Figure 1: A delayed-authority interruption in a repository-repair task. Lightning bolts mark subagent results reaching the main agent, and yellow boxes mark its responses. The right panels contrast the correct response with an illustrative failure.

supports, whereas rebuilding work the result does not affect repeats side effects, discards verified artifacts, and spends calls the remaining task still needs.

Existing benchmarks do not isolate this capability. Evaluations of interruptible, dynamic, and multi-agent systems inject disruptions through user instructions (Zou et al., 2026; Rana et al., 2025; Wu et al., 2026), environment events (Froger et al., 2026; Ji et al., 2026), interrupted execution (Gulati, 2026; KC & Budathoki, 2026), or tool and delegation failures (Zhu et al., 2026; Chen et al., 2026c), most of which make explicit that something must be handled. A subagent result may not explicitly indicate that earlier work has become invalid. The main agent must assess the result's validity, recency, and scope to decide whether existing work needs to change. The closest benchmark includes subagent reports but neither specifies per-event response requirements nor separates event response from task success (Xiong et al., 2026). Evaluation poses two further obstacles. First, final task success alone cannot distinguish correct event handling from success or failure elsewhere in the task. An agent may handle an event correctly yet fail another requirement, or pass the final checks despite repeating a side effect. Second, asynchronous execution is nondeterministic. Subagent results may reach the main agent unpredictably, making responses difficult to compare across runs and agents.

We introduce Async-RBench, a benchmark for asynchronous task execution in LLM agents, with 200 executable tasks adapted from Terminal-Bench, Gaia2, OSWorld, SWE-bench, and MultiAgentBench (Merrill et al., 2026; Froger et al., 2026; Xie et al., 2024; Jimenez et al., 2024; Zhu et al., 2025a). The tasks evenly cover eight interruption scenarios, ranging from delayed authoritative results and stale updates to duplicate reports and subagent failures (Table 2). Subagents execute concurrently in isolated workspaces. A scenario controller buffers results and applies case-specific release conditions, with a maximum-hold fallback to avoid deadlock. Therefore, each task's target event reaches the main agent under designed preconditions rather than natural timing (Figure 2). Each target event has an Event Response Contract (ERC) available only to the evaluator. It specifies required changes, valid work to preserve, prohibited side effects, validation evidence, and the expected outcome, without prescribing a unique action sequence. The *Dynamic Replanning Score* (DRS) weights the response process and the event outcome equally and is reported separately from end-to-end task success S.

We evaluate nine agents and report three findings: **(1) Handling asynchronous results remains challenging.** Overall DRS ranges from 19.2 to 45.1, with the lowest scenario means on Duplication and replay (8.6) and Subagent failure (12.2). **(2) Task success is lower under asynchronous delivery.** Under our execution protocol, all nine agents have lower task success rate estimates with asynchronous than with batched delivery (26.0% vs. 29.7% on average), and eight consume more tokens. Selected paired traces illustrate two failure patterns under Async: the main agent terminates before integrating required artifacts, or acts on assumptions before inspecting the underlying task evidence. **(3) Mandatory plan rebuilding yields limited gains.** Full-rebuild prompts the main agent to rebuild its remaining plan after every subagent result. For the three agents evaluated, it improves DRS by only 2.6–3.5 points while increasing model calls by 7.5–10.2%. Together, these findings highlight the difficulty of reliable asynchronous task execution. By evaluating event responses separately from end-to-end task success, Async-RBench provides a basis for comparing agents and assessing improvements in this setting.

Table 1: Comparison with related benchmarks. *Async delivery*: subagent results reach the main agent while it continues executing. *Post-event checks*: behaviors evaluated after an event. *Per-event scoring*: each event response is scored separately from end-to-end task success. ✓ explicit; ◐ partial or related; ✗ no explicit design.

Benchmark	Interruption events	Setting	Post-event checks					Eval.
		Async delivery	Update plan	Fix affected	Keep valid	No side eff.	Re-verify	Per-event scoring
Gaia2 (2026)	Environment/tools	✗	✓	✓	◐	✓	✗	◐
Robotouille (2025)	Delays/state changes	✗	✗	◐	✗	◐	✗	✓
MultiAgentBench (2025a)	Cooperation/competition	✗	✗	✗	✗	✗	✗	◐
InterruptBench (2026)	Requirement changes	✗	◐	◐	◐	✗	◐	✗
ToolMaze (2026)	Tool failures	✗	◐	◐	✗	✗	✗	✓
OrchestraBench (2026c)	Delegation failures	✗	◐	◐	◐	◐	✗	✓
ClawArena-Team (2026)	Workspace updates	✓	◐	◐	◐	◐	◐	✗
Async-RBench (ours)	**Subagent results**	✓	✓	✓	✓	✓	✓	✓

2 Related Work

Agent benchmarks. Many agent benchmarks pose a task and emphasize task-level outcomes, covering general assistance (Liu et al., 2024; Mialon et al., 2024), web and mobile interfaces (Zhou et al., 2024; Koh et al., 2024; Drouin et al., 2024; Rawles et al., 2025), tool use with users (Yao et al., 2024; Barres et al., 2025; Chen et al., 2026a), and long-context or MCP-based workflows (Li et al., 2026; Luo et al., 2025). Multi-agent benchmarks extend this to collaboration and competition among agents (Zhu et al., 2025a) and to orchestration plans evaluated in deterministic simulation (Ren et al., 2026). Other benchmarks provide complementary checks. AgentBoard measures intermediate progress (Ma et al., 2024), ToolSandbox checks milestones along the trajectory (Lu et al., 2024), and AppWorld checks for unintended state changes (Trivedi et al., 2024), while recent work examines corrupt success, silent failures, and process-level defects (Cao et al., 2026; Advani, 2026; He et al., 2026). These checks provide progress, state-integrity, or process-quality measures.A related line of work diagnoses failures after the fact, labelling trajectories by cause and attributing each failure to the step where the run first departs from a valid solution (Cemri et al., 2025; Zhu et al., 2025b; Barke et al., 2026; Chen et al., 2026b). Async-RBench instead scores the response to each target event against an ERC written for that event and reports DRS separately from task success.

Asynchronous execution and interruptions. Multi-agent frameworks give separate agents distinct roles and conversational contexts (Hong et al., 2024; Qian et al., 2024; Li et al., 2023; Wu et al., 2024; Gao et al., 2024), and Magentic-One uses an orchestrator that plans, tracks progress, and replans to recover from errors (Fourney et al., 2024). Masters et al. (2025) frame such orchestration by a manager agent as a unifying research challenge. Single-agent loops such as ReAct interleave reasoning and actions with observations from the environment (Yao et al., 2023). LLMCompiler dispatches tool calls in parallel once their dependencies are resolved, passes their outputs to dependent calls, and replans by sending intermediate results back to the planner (Kim et al., 2024). On the benchmark side, Gaia2 evaluates agents in independently evolving environments under time constraints (Froger et al., 2026), Robotouille studies asynchronous planning under execution delays (Gonzalez-Pumariega et al., 2025), and ClawArena and SentinelBench cover evolving information and long-running monitoring (Ji et al., 2026; Maldaner et al., 2026). Other work studies user requirement changes (Zou et al., 2026; Rana et al., 2025), interruptions during reasoning (Wu et al., 2026), paused or handed-over execution (Gulati, 2026; KC & Budathoki, 2026), or injected tool and delegation failures (Zhu et al., 2026; Chen et al., 2026c), and HANDRAISER learns when an agent should interrupt a peer (Wang et al., 2026). AgentChangeBench also reports shift-level recovery metrics (Rana et al., 2025). A subagent result need not itself require a plan change. It may be authoritative, stale, duplicated, or partial, and the correct response may be to keep existing work unchanged. The closest benchmark, ClawArena-Team, evaluates how a main agent delegates to a fixed subagent pool, assigns permissions, and integrates results (Xiong et al., 2026), but it does not define a separate response contract for each subagent-result event. Table 1 compares their event sources, delivery settings, post-event checks, and use of per-event scoring.

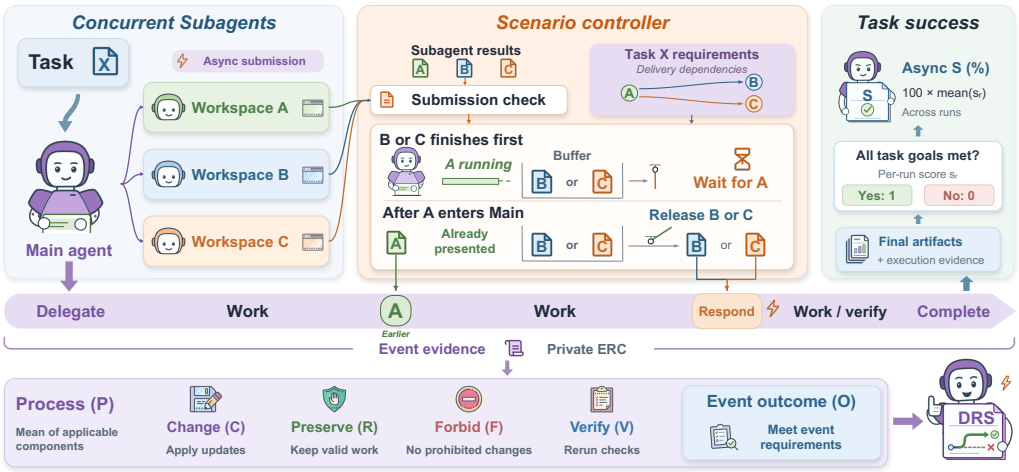


Figure 2: Overview of Async-RBench. Subagents A–C run concurrently in separate workspaces while the main agent continues working. The scenario controller checks submitted results and applies case-specific release conditions, with a maximum-hold fallback. B's and C's reports are held until A's is presented, even if they finish first. Task-level checks determine success S. The response to the target event and its artifacts are evaluated against the ERC to compute DRS (Section 3.2).

3 Async-RBench

Async-RBench addresses both evaluation obstacles identified in Section 1: controlled delivery applies predefined release conditions (Section 3.1), and per-event contracts score the response separately from task success (Section 3.2).

3.1 Controlled Asynchronous Delivery

For each task, the main agent delegates subtasks to subagents, which run concurrently in isolated workspaces while the main agent continues working (Figure 2). Each submitted result is checked against public requirements on content, files, and format and then held by the scenario controller. The main agent sees it only once it is *presented*, that is, included in the input of a main-agent call.

Release conditions. Each case specifies the subagent workloads, the relations among their results (e.g., one result supersedes, duplicates, or conflicts with another), and the conditions for presenting each result to Main. These conditions depend on observable task progress, such as which results Main has already received and which artifacts exist, and are designed so that the prior work referenced by an ERC is generally available when its event is presented. In Figure 2, if B or C finishes before A, its result is buffered until A has been presented in an earlier Main request. Async adds at most one new result to each Main request. After a result is presented,the main agent receives a handling window of at least one and, by default, at most four responses; subsequent results remain queued until this window closes. To prevent indefinite blocking, the controller releases a result once its maximum hold time is reached and logs the fallback release (Appendix C.1).

The controller therefore enforces each event's release prerequisites and inter-result relations without scripting the exact payload, task state, or wall-clock timing. It does not tell Main whether to adopt a result, what existing work it affects, or how to respond. Main must determine the result's validity and scope, then decide what to revise, what to preserve, and which checks to rerun.

Event evidence. For a presented event e_i, we record its result r_i, the task state s_i^- immediately before presentation, the main agent's trace τ_i from presentation to run termination, and the ERC-relevant state s_i^+ at termination:

$$Z_i = \left(r_i, s_i^-, \tau_i, s_i^+\right). \tag{1}$$

Presentation logs identify whether the target result reached the main agent. For presented events, Z_i provides the evidence used to score the response.

3.2 Event Response Contracts and Scoring

Event Response Contract. Each target event has an evaluator-only ERC,

$$\Gamma_i = (C_i, R_i, F_i, V_i, O_i)\,, \tag{2}$$

The ERC defines four process components: required changes C_i, valid work to preserve R_i, prohibited changes F_i, and required validation V_i. The outcome component O_i specifies the conditions that must hold at run termination. Each component contains scoring points tied to concrete artifacts, operations, or validation records; inapplicable components are excluded from scoring. ERCs specify required state changes, preserved invariants, prohibited side effects, and validation evidence without prescribing a unique action sequence.

Dynamic Replanning Score (DRS). Each component X_i receives a score $\sigma(X_i) \in [0, 1]$, the weighted mean of its scoring points. Process points (C, R, F, V) are evaluated on the actions, artifact changes, and validation records in τ_i, and outcome points (O) on the final state s_i^+. Scoring points, weights, and applicability are fixed before evaluation. The process score P_i averages σ over the applicable components among C_i, R_i, F_i, and V_i, with unmet requirements scoring 0, and the outcome score is $Q_i = \sigma(O_i)$. For a run with n target events,

$$\text{DRS} = \frac{100}{n} \sum_{i=1}^{n} \frac{P_i + Q_i}{2}. \tag{3}$$

Scoring both process and outcome distinguishes cases that task success alone conflates. A run that reaches the required outcome through a prohibited action loses process credit, while a run that fails to reach that outcome can still earn credit for satisfied process requirements. Later repairs can improve the outcome score but do not erase a prohibited side effect.

If the main agent ends the run before the target result is presented, the event scores 0. For example, in one Scope and dependencies run, GPT-5.6 Terra adds late throughput and memory requirements to a research proposal ($\sigma(C) = 1$) and keeps the original research goals ($\sigma(R) = 1$) but never substantively checks the revision ($\sigma(V) = 0$). No prohibited change is declared, so F is inapplicable, and the final proposal meets the event requirements ($Q = 1$), giving DRS $= 100 \cdot (2/3 + 1)/2 \approx 83.3$ (Appendix C.3).

Task success and aggregation. End-to-end task success S is the percentage of runs whose final artifacts and execution evidence satisfy all task goals. Each task has one target event ($n = 1$). Other subagent messages provide context and are not scored separately, although later changes to artifacts within the target event's scope are still checked against its ERC. Scenario DRS averages runs within a scenario, and overall DRS is the unweighted mean of the eight scenario scores.

3.3 Tasks and Scenarios

The evaluation set contains 200 tasks adapted from the five source benchmarks (40 each) and divided evenly among eight scenarios (25 each). The tasks comprise 621 initial subtasks, each assigned to one subagent (Appendix A.2). A disjoint set of 47 tasks is used only during construction. Table 2 groups the scenarios by how a result relates to the current task state: its validity, its completeness, or its effect on dependencies and scheduling. Each task carries one primary scenario label, but its ERC, derived from task evidence and dependencies, determines what must be revised or retained.

Construction and acceptance. An LLM agent screens source tasks and traces, with fixed versions and environments, for decision points at which a delegated result could change an earlier decision. Researchers then check the basis of that decision, the validity of the new result, and its effect on existing work, design the subagent workflows, result contents, and release conditions (introducing an event where the source task has none), and derive the ERC from the artifacts expected before the event, the scope the result affects, and the evidence validation must produce. Evaluated agents never see the source traces. Before admission, trial runs check that events trigger in the intended state, reference and equivalent solutions check solvability and that verifiers accept equivalent valid solutions, and invalid variants (e.g., omitted revisions, incorrect rollbacks, duplicated side effects) check that verifiers reject them (Appendix A).

Table 2: Example task and evaluation focus for each scenario, grouped by result validity (first four), information completeness (next two), and dependencies and scheduling (last two). Examples are condensed from actual cases.

Scenario	Example task	Evaluation focus
Delayed authority	*Archive an AWS invoice; its fields arrive after the folder and summary table are checked.*	**Adopt authoritative evidence**; save the attachment and verify the record.
Staleness and ordering	*Negotiate a cart purchase; an old offer arrives after the new $42 agreement is saved.*	**Reject stale updates**; keep the current agreement over old terms.
Conflicting results	*Diagnose slow DB writes; duplicate-index evidence arrives after causes are ranked.*	**Arbitrate by evidence**, not arrival order; revise the root-cause ranking.
Duplication and replay	*Build a scheduling app; a completion message is replayed after its tests are integrated.*	**Stay idempotent**; avoid repeated side effects and needless rebuilding.
Partial to complete	*Merge two spreadsheets into a CSV; column extractions arrive before the complete merge.*	**Separate provisional from complete evidence**; confirm row alignment.
Subagent failure	*Coordinate restaurant deliveries; a delivery subagent declines its suborder.*	**Recover selectively**; reassign the failed suborder and keep valid routes.
Scope and dependencies	*Add slideshow audio; the WAV arrives after a single-slide insertion is planned.*	**Extend playback scope**; keep the track and verify playback across slides.
Resource stragglers	*Fix Matplotlib figure pickling; a key diagnosis is delayed as worker slots shrink.*	**Prioritize critical work**; weigh waiting against cancelling; keep finished work.

4 Experiments

We first validate DRS (Section 4.2), then evaluate agents' responses to asynchronous results (Section 4.3), compare task success across delivery modes (Section 4.4), and examine the effect of mandatory plan rebuilding (Section 4.5).

4.1 Experimental Setup

Agents. We evaluate Claude Sonnet 5 (Anthropic, 2026), GPT-5.6 Terra and GPT-5.6 Luna (OpenAI, 2026), Qwen3.7 Plus (Yang et al., 2025), Gemini 3 Flash High (Google DeepMind, 2025), Qwen3.5 27B (Yang et al., 2025; Qwen Team, 2026), GLM5.3 Flash (GLM-5-Team et al., 2026; Z.ai, 2026), Kimi K2.5 (Kimi Team, 2026), and DeepSeek V4 Pro (DeepSeek-AI et al., 2024). For each evaluated agent, the main agent and all subagents use the same underlying model. All agents use a common interface, tool set, and evaluation pipeline. We use short names (e.g., Terra, Luna, Gemini) where unambiguous.

Delivery modes. Our main setting, asynchronous delivery (Async), presents results individually while the main agent continues working, following the release rules in Section 3.1. The baseline, batched delivery (Batched), presents all results together after every required subagent workflow terminates. Both modes run subagents concurrently with the same tasks, workflows, and resource limits, but use different result-delivery protocols. We report DRS for Async and task success S for both modes. Because subagents are executed independently in the two modes, their outputs and the main agent's subsequent actions may differ across runs.

Protocol. Each agent is evaluated on all 200 tasks three times per delivery mode, yielding 600 runs per agent per mode. Runs across the two modes are paired by task and repetition index, with matched environments, task and verifier versions, and model configurations. Each run allows at most 100 main-agent responses and 40 responses per subagent execution. Each response is capped at 16,384 output tokens, and at most three subagents run concurrently. A run ends when the main agent calls `finish`, responds without a tool call, exhausts its budget, or reaches 2,400 seconds. Runs affected by infrastructure failures are rerun (Appendix C). For each scenario, we report the mean and sample SD of its three repetition-level mean scores.

Table 3: DRS (0–100) under asynchronous execution. Scenario columns follow Table 2 and report the mean and sample SD over three repetitions. The Calls column counts main-agent and subagent responses. The last row averages across agents. Bold marks column maxima.

Agent	Delay	Stale	Conflict	Replay	Partial	Failure	Scope	Strag.	Calls	DRS
Claude Sonnet 5	**38.7**±1.8	**53.4**±1.2	39.9±2.1	15.5±1.2	29.8±1.7	**35.8**±1.3	66.9±1.9	80.5±1.5	77.1	**45.1**
GPT-5.6 Terra	34.2±1.5	43.9±2.9	**41.5**±1.3	**18.5**±1.6	21.4±1.3	16.2±1.9	**69.4**±2.0	72.2±2.4	48.1	39.7
Qwen3.7 Plus	33.0±1.3	45.5±2.9	38.8±1.5	12.6±1.6	21.2±2.4	6.9±1.5	61.8±2.4	**85.9**±2.6	75.9	38.2
GLM5.3 Flash	29.0±2.7	47.8±2.2	27.0±2.3	11.6±2.2	**31.2**±2.1	5.0±0.6	53.5±2.4	62.3±1.4	76.5	33.4
GPT-5.6 Luna	34.0±1.2	47.8±1.2	31.7±1.4	4.2±0.3	22.2±1.2	14.9±1.2	43.8±1.3	50.5±2.2	45.5	31.1
DeepSeek V4 Pro	16.2±1.6	47.6±2.2	20.8±3.2	2.7±0.7	24.2±2.4	16.8±2.8	62.0±2.1	54.7±3.2	108.0	30.6
Qwen3.5 27B	28.7±1.4	32.0±2.9	31.5±3.1	6.1±0.2	20.0±1.0	4.2±0.2	61.1±1.4	54.6±2.2	119.6	29.8
Kimi K2.5	27.5±2.3	30.0±1.9	20.4±1.3	3.1±0.4	12.9±1.3	8.5±1.3	51.4±1.2	55.3±2.3	125.2	26.1
Gemini 3 Flash-H	20.7±1.1	13.8±0.4	29.3±1.3	2.7±0.4	8.5±1.1	1.7±0.4	47.2±2.4	29.9±2.3	95.0	19.2
Mean	29.1	40.2	31.2	8.6	21.3	12.2	57.5	60.7	85.7	32.6

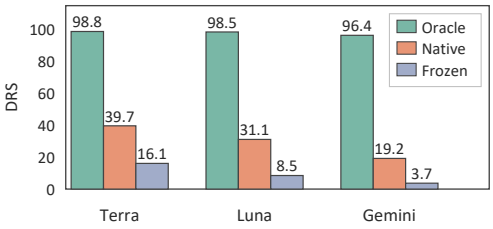


Figure 3: DRS of Oracle, Native, and Frozen responses to the same target events.

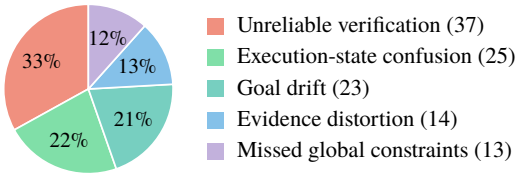


Figure 4: Distribution of 112 issue labels across 106 Async runs; a run may receive multiple labels.

4.2 Validation of DRS

Verifiers agree with human judgments. In an initial human audit of 405 randomly sampled runs, automated decisions agree with human judgments on 96.6% of scored items. All disagreements trace to infrastructure failures, which we fixed before rerunning every affected run (Appendix A.3).

DRS separates responses of different quality. For Terra, Luna, and Gemini, which rank high, middle, and low in Table 3, we score three conditions from the same pre-event state (artifacts, main-agent context, subagent states, and event content) under the same limits, ERCs, and scoring rules. *Native* lets the main agent respond freely, as in the main experiment. After the target result is presented, *Frozen* replays a continuation generated by the same model without access to that result. *Oracle* executes a reference recovery program written for that event. As shown in Figure 3, for all three agents, DRS is lowest for *Frozen* (3.7–16.1), higher for *Native* (19.2–39.7), and highest for *Oracle* (96.4–98.8). This ordering shows that DRS distinguishes responses informed by the target result from continuations generated without it. The near-ceiling Oracle scores show that the evaluated ERCs can be satisfied. Oracle knows each event's requirements, so it is a reference rather than a capability-matched baseline. Frozen earns some credit because unadapted actions can still preserve valid work or avoid prohibited side effects.

4.3 Agent Performance under Asynchronous Delivery

Overall DRS ranges from 19.2 for Gemini 3 Flash High to 45.1 for Claude Sonnet 5 (Table 3). Even the highest DRS is below 50 on the 0–100 scale, whereas Oracle range from 96.4 to 98.8 (Figure 3).

Scores are lowest on duplicate reports and subagent failures. Averaged across agents, DRS is 8.6 on Duplication and replay and 12.2 on Subagent failure, compared with 57.5 on Scope and dependencies and 60.7 on Resource stragglers. The ERCs for the two lowest-scoring scenarios emphasize preserving valid work and avoiding unnecessary side effects. A duplicate report adds no new evidence, while a subagent failure requires recovering the affected subtask without disturbing valid work elsewhere. These averages also include zero scores from valid runs in which the main agent ter-

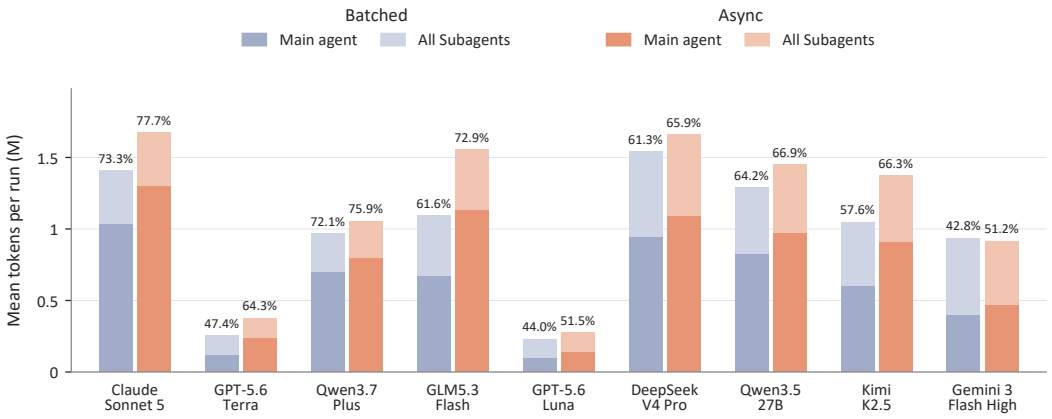


Figure 5: Mean token consumption per run by the main agent and all subagents. Each pair shows batched and asynchronous delivery; percentages give the main agent's share.

minates before receiving the target result, and the scenarios use different source tasks. The score gaps therefore do not, by themselves, establish that preserving valid work is the main source of difficulty.

Rankings shift across scenarios and differ from task success. Claude Sonnet 5 leads overall but trails Terra on Duplication and replay (15.5 vs. 18.5), and Qwen3.7 Plus leads on Resource stragglers (85.9) but scores only 6.9 on Subagent failure. DRS is also complementary to task success: Qwen3.7 Plus ranks third on DRS but seventh on asynchronous S (Table 4), whereas DeepSeek V4 Pro ranks sixth and third. Across agents, more model calls do not necessarily correspond to higher DRS: Terra scores 39.7 with 48.1 calls per run, whereas Kimi K2.5 scores 26.1 with 125.2 calls (Appendix C.5).

Where responses fail. Following prior failure taxonomies (Cemri et al., 2025) and trace-based diagnosis (Zhu et al., 2025b; Barke et al., 2026; Chen et al., 2026b), we label issues in a sample of 106 Async runs (Figure 4; excerpts in Appendix D). Unreliable verification is the most common label; one example is a test that prints a failure message but does not signal failure. Other examples include using a deferred result (execution-state confusion), fabricating a missing receipt (goal drift), presenting static code analysis as runtime coverage (evidence distortion), and accepting plans that pass local checks but jointly exceed a task-wide budget (missed global constraints).

4.4 Task Success Is Lower under Asynchronous Delivery

Under our execution protocol, all nine systems obtain lower task-success point estimates with asynchronous than with batched delivery, by 1.0–6.0 percentage points (Table 4), and the mean falls from 29.7% to 26.0%. Eight of the nine also consume more tokens, and main-agent consumption rises in all nine (Figure 5); for Terra it doubles while subagent consumption barely changes (Appendix C.5). Paired traces of tasks solved only under batched delivery illustrate two observed failure paths, without establishing that asynchronous delivery caused them.

Table 4: Task success rate S (%) under batched and asynchronous delivery.

Agent	Task success		Change
	Batched	Async	ΔS
Claude Sonnet 5	**42.3**	38.2	−4.2
GPT-5.6 Terra	41.3	**38.8**	−2.5
Qwen3.7 Plus	21.3	18.7	−2.7
GLM5.3 Flash	25.5	20.8	−4.7
GPT-5.6 Luna	40.2	34.3	−5.8
DeepSeek V4 Pro	35.7	34.7	**−1.0**
Qwen3.5 27B	19.7	17.0	−2.7
Kimi K2.5	23.2	19.2	−4.0
Gemini 3 Flash-H	18.3	12.3	−6.0
Mean	29.7	26.0	−3.7

Ending before required artifacts are integrated. In a Conflicting results scheduling task, Qwen3.5 27B receives a validation report under asynchronous delivery while the second plan set still sits in another subagent's workspace. The main agent emits its final answer, and shutdown cancels that subagent before it delivers the plans, so the terminal checks find the plan set and the decision manifest missing; the batched run receives both plan sets and passes.

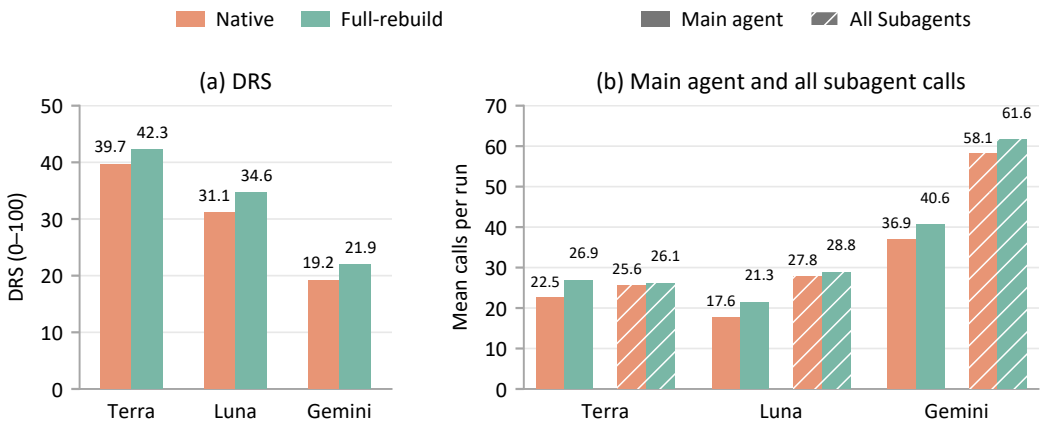


Figure 6: DRS and mean model calls per run under Native execution and Full-rebuild. In (b), solid bars count main-agent calls and hatched bars count calls summed over all subagents.

Acting before examining the original evidence. In a WAL recovery task, two agents that succeed under batched delivery invoke SQLite under asynchronous delivery before any report arrives and then construct records from assumptions instead of decoding the encrypted log (Appendix D).

4.5 Mandatory Plan Rebuilding Yields Limited Gains

Under Native execution, the main agent decides for itself whether and how to revise its plan when a result arrives. Full-rebuild instead instructs the main agent to rebuild the entire remaining plan from the current task state whenever a subagent result is presented, keeping completed work and existing artifacts. We run it on Terra, Luna, and Gemini, separately from the main experiment.

Full-rebuild raises DRS on all three agents (Figure 6a), from 39.7 to 42.3 for Terra, 31.1 to 34.6 for Luna, and 19.2 to 21.9 for Gemini, while mean model calls per run increase by 7.5–10.2% (Figure 6b). The additional calls occur mainly in the main agent for Terra and Luna and are distributed roughly evenly between the main agent and subagents for Gemini. Thus, the Full-rebuild instruction yields modest DRS gains at additional computational cost. Because we do not report whether agents actually rebuild their plans as instructed, the results cannot distinguish failures to follow the instruction from errors that persist after rebuilding.

5 Conclusion

We introduced Async-RBench to evaluate how LLM agents handle subagent results asynchronously. Controlled delivery and event-specific contracts allow response quality to be scored separately from final task success. Across nine agents, DRS reaches at most 45.1 and is lowest on duplicate reports and subagent failures. Under our execution protocol, all nine agents have lower task-success point estimates with asynchronous than with batched delivery. For three agents, prompting the main agent to rebuild its plan after every result improves DRS by only 2.6–3.5 points while increasing model calls by 7.5–10.2%. These findings motivate strategies that track which work a result affects and which remains valid, evaluated through both event responses and end-to-end task success.

Limitations. Each task isolates one target event, excluding overlaps. DRS conservatively scores valid runs terminating before target delivery as zero, conflating them with response errors. Controller-based delivery with timeout fallback is reproducible but may not reflect real arrivals. Main and subagents share one model, favoring comparability over heterogeneity. Independent subagent runs introduce variation into delivery-timing comparisons. Human-written ERCs and verifiers may omit valid responses despite sampled human review.

AI Use Statement

In this work, LLM agents are the object of study: the analyzed trajectories and trained checkpoints are produced by the agents themselves and are reported as experimental results. As authors, we used generative AI tools for tasks requiring disclosure: producing initial annotation labels for trajectory analysis, distilling open-source documentation into the skill library, and assisting with implementing the framework and analysis code. We also used generative AI tools for drafting and polishing text, creating figures, and formatting references. We have reviewed all AI-assisted work: all labels were manually audited, and all code was verified and tested by the authors. We take responsibility for the final content of this work.

Reproducibility Statement

Sections 3 and 4 elaborate on the full benchmark framework, evaluation metrics, execution modes, model configurations, run budgets, and aggregation procedures. Appendix A documents task sources, case construction workflows, scenario specifications, and verifier acceptance criteria; Appendix B presents representative cases for all eight scenarios; Appendix C specifies result delivery boundaries, termination and rerun rules, cost accounting conventions, and execution and scoring versioning; and Appendix D provides the trajectory evidence underlying the failure analysis. The supplementary material includes the benchmark implementation code, case specification files, verifiers, configuration files, and run manifests, with each analyzed run uniquely tied to its model identifier, configuration parameters, test case, verifier, and protocol version, while private verifier data, access credentials, and license-restricted assets are excluded from the public release package.

Ethics Statement

This work introduces Async-RBench, a benchmark for evaluating asynchronous task execution in LLM agents. The study does not involve human subjects, sensitive personal data, or potentially harmful applications. All source tasks are adapted from publicly available, previously published benchmarks (Terminal-Bench, Gaia2, OSWorld, SWE-bench, and MultiAgentBench) in compliance with their respective usage licenses.

This research received no commercial sponsorship, and the authors declare no conflicts of interest. All authors have read and adhere to the ICLR Code of Ethics.

References

L. Advani. From confident closing to silent failure: Characterizing false success in LLM agents. *arXiv preprint arXiv:2606.09863*, 2026.

Anthropic. Claude Sonnet 5 system card. System card, 2026. URL https://www.anthropic.com/claude-sonnet-5-system-card. Published June 2026.

Anthropic. Create custom subagents. Claude Code documentation, n.d. URL https://code.claude.com/docs/en/sub-agents. Accessed September 17, 2026.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing AI Agent Failures from Execution Trajectories. *arXiv preprint arXiv:2602.02475*, 2026. URL https://arxiv.org/abs/2602.02475.

Victor Barres, Honghua Dong, Soumya Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Hongliu Cao, Ilias Driouich, and Eoin Thomas. Beyond task completion: Revealing corrupt success in LLM agents through procedure-aware evaluation, 2026.

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

Jing Chen et al. AppWorld-UL: Benchmarking diverse agent-user interactions for tool-use. *arXiv preprint arXiv:2607.20536*, 2026a.

Mengzhuo Chen, Junjie Wang, Fangwen Mu, Yawen Wang, Zhe Liu, Huanxiang Feng, and Qing Wang. Seeing the Whole Elephant: A Benchmark for Failure Attribution in LLM-based Multi-Agent Systems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19888–19905, 2026b. doi: 10.18653/v1/2026.acl-long.912. URL https://aclanthology.org/2026.acl-long.912/.

Yidian Chen, Yingzi Gu, Natan Vidra, Spurthi Setty, and Sharon Zheng. OrchestraBench: Evaluating multi-agent orchestration failure modes, recovery, and decomposition quality, 2026c.

DeepSeek-AI et al. DeepSeek-V3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How Capable are Web Agents at Solving Common Knowledge Work Tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 11642–11662, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. *arXiv preprint arXiv:2411.04468*, 2024. URL https://arxiv.org/abs/2411.04468.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, et al. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=9gw03JpKK4.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. AgentScope: A Flexible yet Robust Multi-Agent Platform. *arXiv preprint arXiv:2402.14034*, 2024. URL https://arxiv.org/abs/2402.14034.

Longling Geng and Edward Y. Chang. ALAS: transactional and dynamic multi-agent LLM planning. *CoRR*, abs/2511.03094, 2025. doi: 10.48550/ARXIV.2511.03094. URL https://doi.org/10.48550/arXiv.2511.03094.

Antonio A. Ginart, Naveen Kodali, Jason Lee, Caiming Xiong, Silvio Savarese, and John Emmons. Asynchronous tool usage for real-time agents. *CoRR*, abs/2410.21620, 2024. doi: 10.48550/ARXIV.2410.21620. URL https://doi.org/10.48550/arXiv.2410.21620.

GLM-5-Team et al. GLM-5: From vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Gonzalo Gonzalez-Pumariega, Leong Su Yean, Neha Sunkara, and Sanjiban Choudhury. Robotouille: An asynchronous planning benchmark for LLM agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OhUoTMxFIH.

Google DeepMind. Gemini 3 Flash model card. Model card, 2025. URL https://deepmind.google/models/model-cards/gemini-3-flash/. Published December 17, 2025.

Aryan Gulati. ContinuityBench: A framework and taxonomy for evaluating agent recovery from interrupted state. In *ICML 2026 Workshop on Combining Theory and Benchmarks*, 2026. URL https://openreview.net/forum?id=3N3BzvoLbG.

Jiawei He, Jie Jia, Chenbo Liu, Chaoyi Xue, Yapeng Song, Xikai Yang, and Dong Sun. ProcCtrl-Bench: Evaluating process-level defects and control preservation in LLM coding agents, 2026.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

Haonian Ji, Kaiwen Xiong, Siwei Han, Peng Xia, Shi Qiu, Yiyang Zhou, et al. ClawArena: Benchmarking AI agents in evolving information environments, 2026.

Carlos E. Jimenez et al. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Dipesh KC and Anjila Budathoki. Handoff debt: The rediscovery cost when coding agents take over interrupted tasks, 2026. URL https://arxiv.org/abs/2606.02875.

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*, volume 235, pp. 24370–24391, 2024. URL https://proceedings.mlr.press/v235/kim24y.html.

Kimi Team. Kimi K2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *Advances in Neural Information Processing Systems*, volume 36, pp. 51991–52008, 2023. doi: 10.52202/075280-2264. URL https://proceedings.neurips.cc/paper/2023/hash/a3621ee907def47c1b952ade25c67698-Abstract-Conference.html.

Kai Li et al. AgencyBench: Benchmarking the frontiers of autonomous agents in 1m-token real-world contexts. In *Annual Meeting of the Association for Computational Linguistics*, pp. 7422–7440, 2026.

Xiao Liu et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, 2024.

Jiarui Lu et al. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. *arXiv preprint arXiv:2408.04682*, 2024.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. AgentRewardBench: Evaluating Automatic Evaluations of Web Agent Trajectories. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=fQcUZMPIvu.

Zhen Luo et al. MCP-Universe: Benchmarking large language models with real-world model context protocol servers. *arXiv preprint arXiv:2508.14704*, 2025.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Advances in Neural Information Processing Systems*, 2024.

Matheus Kunzler Maldaner, Adam Fourney, Amanda Swearngin, Hussein Mozzanar, Gagan Bansal, Maya Murad, Rafah Hosn, and Saleema Amershi. SentinelBench: A benchmark for long-running monitoring agents, 2026. URL https://arxiv.org/abs/2606.05342.

Charlie Masters, Advaith Vellanki, Jiangbo Shangguan, Bart Kultys, Jonathan Gilmore, Alastair Moore, and Stefano V. Albrecht. Orchestrating human-AI teams: The manager agent as a unifying research challenge. In *Proceedings of the 2025 7th International Conference on Distributed Artificial Intelligence*, pp. 91–107. Association for Computing Machinery, 2025.

Michael A. Merrill et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, 2026.

Grégoire Mialon et al. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations*, 2024.

OpenAI. GPT-5.6 system card. System card, 2026. URL https://deploymentsafety.openai.com/gpt-5-6. Published July 9, 2026.

OpenAI. Subagents. Codex documentation, n.d. URL https://developers.openai.com/codex/subagents. Accessed September 17, 2026.

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)*, pp. 15174–15186, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Qwen Team. Qwen3.5-27B model card, 2026. URL https://huggingface.co/Qwen/Qwen3.5-27B. Accessed September 12, 2026.

Manik Rana, Calissa Man, Anotida Expected Msiiwa, Jeffrey Paine, Kevin Zhu, Sunishchal Dev, Vasu Sharma, and Ahan M. R. AgentChangeBench: A multi-dimensional evaluation framework for goal-shift robustness in conversational AI, 2025. URL https://arxiv.org/abs/2510.18170.

Christopher Rawles et al. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, 2025.

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, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Danqing Wang, Da Yin, Ruta Desai, Lei Li, Asli Celikyilmaz, and Ansong Ni. Learning to interrupt in language-based multi-agent communication. In *Conference on Language Modeling*, 2026. URL https://arxiv.org/abs/2604.06452.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed 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*, 2024. URL https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/.

Tsung-Han Wu, Mihran Miroyan, David M. Chan, Trevor Darrell, Narges Norouzi, and Joseph E. Gonzalez. Are large reasoning models interruptible? In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2510.11713.

Tianbao Xie et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, 2024.

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, 2026.

An Yang et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Xin Yang, Letian Li, Zimo Ji, Terry Jingchen Zhang, and Wenyuan Jiang. Position: Multi-agent systems should prioritize concurrency control. *CoRR*, abs/2608.18092, 2026. doi: 10.48550/ARXIV.2608.18092. URL https://doi.org/10.48550/arXiv.2608.18092.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Z.ai. GLM-5.3-Flash model card, 2026. URL https://huggingface.co/zai-org/GLM-5.3-Flash. Accessed September 12, 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/4410c0711e9154a7a2d26f9b3816d1ef-Abstract-Conference.html.

Dongsheng Zhu, Xuchen Ma, Yucheng Shen, Xiang Li, Yukun Zhao, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. When tools fail: Benchmarking dynamic replanning and anomaly recovery in LLM agents, 2026.

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*, 2025a.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM Agents Fail and How They can Learn From Failures. *arXiv preprint arXiv:2509.25370*, 2025b. URL https://arxiv.org/abs/2509.25370.

Henry Peng Zou, Chunyu Miao, Wei-Chieh Huang, Yankai Chen, Yue Zhou, Hanrong Zhang, Yaozu Wu, Liancheng Fang, Zhengyao Gu, Zhen Zhang, Kening Zheng, Fangxin Wang, Yi Nian, Shanghao Li, Wenzhe Fan, Langzhou He, Weizhi Zhang, Xue Liu, and Philip S. Yu. When users change their mind: Evaluating interruptible agents in long-horizon web navigation, 2026. URL https://arxiv.org/abs/2604.00892.

A Task-Set Construction and Scenario Specification

A.1 Task Sources and Construction Process

Candidate tasks are drawn from Terminal-Bench, Gaia2, OSWorld, SWE-bench, and MultiAgent-Bench (Merrill et al., 2026; Froger et al., 2026; Xie et al., 2024; Jimenez et al., 2024; Zhu et al., 2025a), covering terminal interaction, desktop applications, software engineering, and multi-agent collaboration. Source tasks provide the original objectives, initial environments, and acceptance criteria, from which we design subagent workflows, result-version relations, and release conditions.

Construction begins by fixing the source version, task identifier, and initial environment, and checking that the task has an explicit objective, an executable workflow, and a verifiable final state. An LLM agent then flags decision points that subagent results may affect; its suggestions only support screening. Researchers examine information dependencies to identify work that can be delegated and the plans, artifacts, or checks that new results may affect, and they determine the scenario labels and scoring requirements. They then design the subagent workflows, result contents, version relations, release conditions, and event timeline, and write the Event Response Contract (ERC). Human review checks that the source evidence, the validity of the new result, its task consequences, and the scoring requirements agree; a candidate becomes an executable instance only when it is clear which work must be revised, preserved, or revalidated after the event. Figure 7 summarizes the process, and Appendix A.5 describes execution acceptance.

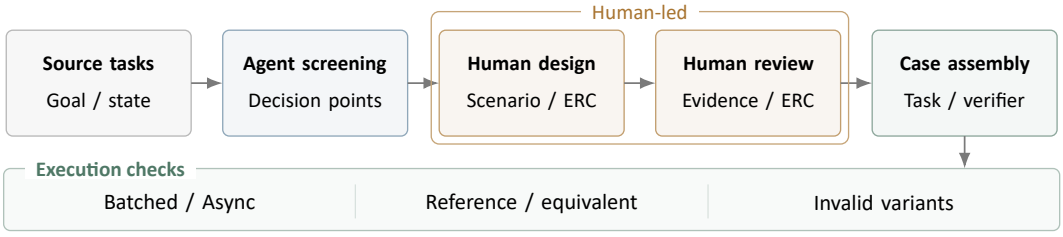


Figure 7: Task construction and acceptance. LLM-agent screening feeds human-led scenario and ERC design and review; assembled instances then undergo execution checks in both delivery modes, with reference or equivalent solutions and invalid variants.

A.2 Task Composition and Aggregation

The construction set contains 47 tasks, and the evaluation set contains 200 tasks, with no overlap between them. The evaluation set covers eight scenarios with 25 tasks each and five original sources with 40 tasks each. Its 200 tasks comprise 621 initial subtasks: 36.5% of tasks start with two subagents, 20.0% with three, and 43.5% with four or more, and 52.0% are rated medium and 48.0% hard.

Evaluation weights tasks equally, without additional scenario weights. Because all eight scenarios contain the same number of tasks, averaging scenario means is equivalent to averaging all tasks under complete coverage; each scenario contributes one eighth of the overall result. The construction set is excluded from all reported results.

A.3 Sampled Human Review and Verifier Acceptance

Human review during construction checks that task requirements are clear, required inputs are available, and scoring conditions match the task objectives. Following work that assesses automatic trajectory evaluators against expert judgments (Lù et al., 2025), we also audit the verifiers. We randomly sampled 405 run records from the evaluation set. Auditors examined task requirements, initial states, presented results, trajectories, tool outputs, and final artifacts; a model's completion claim alone was never taken as evidence of success. A verifier decision counts as a false rejection if the auditor judges an artifact valid but the verifier rejects it, and as a missed violation in the opposite case. In the initial audit, item-level agreement between human judgments and automated decisions was 96.6%. Every disagreement traced to external infrastructure failures; no task-design issue was found in the sam-

ple. We then screened all run records across models, modes, and repetitions for the same anomaly types, fixed the environment, and reran all affected runs. The reruns passed run-validity checks and replaced the affected records, while task failures attributable to the models were retained. These conclusions apply to the audited sample and versions; they do not establish that the verifiers have no false rejections or missed violations.

A.4 Asynchronous Scenario Taxonomy

The eight scenarios are organized by result validity, information completeness, task dependencies, and execution scheduling. Each instance has one primary scenario label and may require revision, preservation, avoidance of forbidden changes, and revalidation. The label describes the principal event relation; task evidence and dependencies determine the specific response requirements.

Result validity. In *Delayed authority*, the main agent progresses using provisional information before a result arrives that supersedes or corrects it, requiring revision of the affected work. In *Staleness and ordering*, an older result is presented after a newer state has taken effect; the main agent must recognize the version relation and avoid rolling back valid state. In *Conflicting results*, outputs from valid workflows support different conclusions; the main agent must resolve them using provenance, scope, and task constraints. *Duplication and replay* requires idempotent handling of repeated results, avoiding duplicate writes, repeated submissions, or unnecessary rebuilding.

Information completeness. *Partial to complete* presents local evidence or an intermediate artifact before the full result. The main agent must distinguish provisional information from completion evidence and complete integration and validation once the full result arrives. In *Subagent failure*, a subagent fails to complete its assigned work, reports failure, or declines further execution. The main agent must recover the affected subtask while preserving valid work completed by other subagents.

Task dependencies and execution scheduling. *Scope and dependencies* requires the main agent to propagate changes in scope, dependency relations, or acceptance conditions to the relevant artifacts and checks while preserving unrelated work. *Resource stragglers* combines delayed critical results with changing resource constraints. The main agent must use task dependencies to choose among waiting, cancellation, reassignment, and continued execution while preserving valid completed state.

One event may impose several response requirements. For example, a late authoritative result may require revising one dependent file, preserving unaffected test results, and rerunning the relevant checks on the updated state. The ERC therefore translates scenario relations into specific revision scope, preservation requirements, and validation evidence.

A.5 Case Specification and Execution Acceptance

Each instance includes public task information and private evaluation information, as summarized in Table 5. The main agent can use task requirements, the initial environment, available tools, and presented subagent results. Scenario labels, expected responses, and internal verifier state are withheld from systems under evaluation.

The ERC binds required changes, preservation of valid work, forbidden changes, revalidation, and response-outcome conditions to concrete artifacts, states, operations, or validation evidence. It specifies the required task state without prescribing a unique action sequence. A reference execution provides one feasible path; other implementations that satisfy the same requirements should also pass.

Before admission, each instance undergoes execution acceptance. We check that the environment starts, subagents can submit qualified results, events trigger in the intended state, and the verifier can assess both the event response and the final state. Reference executions and equivalent implementations test feasible completion paths and expected acceptance. Invalid variants test whether the verifier detects omitted revisions, incorrect rollback, duplicate side effects, and missing revalidation.

The task, scenario specification, and verifier are stored under the same version; Appendix C describes result delivery and execution rules.

Table 5: Case components and information boundaries. Public version and dependency evidence comes from the task or presented results; the evaluator retains the full affected scope, response requirements, and scoring evidence.

Information	Access boundary	Content and use
Information available to the main agent		
Task and initial environment	Always visible	Goal, initial state, tools, and permitted operations.
Subagent result	After presentation	New evidence or artifacts, received when included in a main-agent call.
Version and dependency relations	Partly visible	Supports judgments of result validity and affected work.
Evaluator configuration and records		
Result-release conditions	Hidden from the main agent	Determine when qualified results may be presented.
Event Response Contract (ERC)	Hidden from the main agent	Defines revision, preservation, forbidden changes, revalidation, and outcome requirements.
Hidden verifier	Hidden from the main agent	Checks event responses and final task state.
Logs and state snapshots	Retained by evaluator	Record result presentation, main-agent actions, and state changes.

B Representative Cases across Eight Scenarios

This appendix presents one recorded run for each of the eight scenarios. Each figure retains relevant excerpts from the task, subagent results, and the main agent's subsequent actions. E identifies trace events, `[...]` marks omissions, and dashed boxes highlight the evidence discussed below. In the figures, "Main agent" denotes the main agent and "Subagent" denotes a subagent. Fields such as `replayed` and `replay_of_completion_id` come from the evaluator's delivery records, which the framework logs separately from the result presented to the main agent. The analysis is limited to the records shown: subagent reports and the main agent's final responses describe their execution but do not independently establish task success.

B.1 Delayed authority: Select a deployment backend

The task requires choosing a parallel backend, profile version, and world size from the profile applicable to the final deployment. A subagent first returns authoritative profile v2, recommending pipeline parallelism with a world size of two; a later candidate implementation report still cites v1. The main agent then writes v2 into both the deployment configuration and the batch lineage record. Arrival time and profile applicability must be distinguished: a later report does not necessarily provide a more current basis for deployment.

B.2 Staleness and ordering: Fix Bottle header injection

After the repository is sanitized and rewritten, a subagent reports new branch revision identifiers. A patch report arrives later but still cites pre-rewrite revisions as its test basis. The main agent subsequently runs a local header check covering carriage returns, line feeds, and null characters; the shown command exits successfully. The ordering can be misleading: a newly received report may rely on an outdated code state. Test results from old revisions cannot directly substitute for checks of the current repository.

B.3 Conflicting results: Recover and remove a Git secret

Repository scanners return different findings about a secret. Scans of the working tree and ref-reachable history find no match, whereas the object-database report locates the target in an unreachable object. These apparently conflicting results cover different repository scopes. The subsequent recovery uses the object-database finding, and the cleanup report records reflog expiration and garbage

Delayed authority | Case: Select a deployment backend

Task (E2)

Select backend, profile_version and world_size from the profile evidence you determine applies to the final deployment.

Child (E60)

Inspected /app/profiles/authoritative.json. Profile version v2, world_size 2, ethernet interconnect. Tensor parallel not supported; pipeline parallel supported. Recommended backend: pipeline.

Child (E323)

ColumnParallelLinear and RowParallelLinear implemented in /app/parallel_linear.py with bias zero-init, all-gather/all-reduce semantics per original Megatron-LM API.
[...]
Candidate profile v1 declares world_size 4; validator ran with world_size 1.

Main agent (E551)

```
cat > /app/deployment.json << 'EOF'
{
  "backend": "pipeline",
  "profile_version": "v2",
  "world_size": 2,
[...]
cat > /app/batch-lineage.json << 'EOF'
{
  "backend": "pipeline",
  "profile_version": "v2",
```

(a)

Staleness and ordering | Case: Fix Bottle header injection

Task (E2)

Fix Bottle's CRLF header injection vulnerability (CWE-93) on both main and dev branches and write the original report format to /app/report.jsonl.

Child (E232)

replaced repository history with the authoritative post-sanitization main/dev histories, pruned the object database, and produced a verified complete bundle.
main_head: "4f8b595fa29a1798808c478bd17e8e67f2d544be"
dev_head: "c9757b12c48941e2a48ed25246947aaab2273856"

Child (E297)

Produced a single Bottle CWE-93 patch and standalone regression test; applied and tested both staged pre-rewrite refs.
tested_main_head: "f2ff20b7b19c7e24d8050e2589db973206727857"
tested_dev_head: "d7659db8a15f09e933cae547ff74ad19789a751c"

[...]
validation_test.py passed against patched archives of main and dev; all exercised CR, LF, and NUL header-name/value cases raised ValueError.

Main agent (E340)

```
import bottle
for k,v in
  [('X\nY','ok'),('X\rY','ok'),('X\0Y','ok'),('X','v\n'),('X',
  'v\r'),('X','v\0')]:
 try: bottle.BaseResponse(headers=[(k,v)])
 except ValueError: pass
 else: raise SystemExit('failed '+repr((k,v)))
print('header controls rejected')
```

Main agent (E340, E341)

```
git add bottle.py && git commit -m 'security: reject control
  characters in headers'
MAIN=$(git rev-parse HEAD)
git checkout dev
exit_code: 0
```

(b)

Figure 8: **(a)** Delayed authority: deployment backend selection. **(b)** Staleness and ordering: Bottle header injection.

collection. The main agent needs to check which repository states each scanner covers rather than choosing between reports solely because they say "found" or "not found."

B.4 Duplication and replay: Build OfficeTaskScheduler

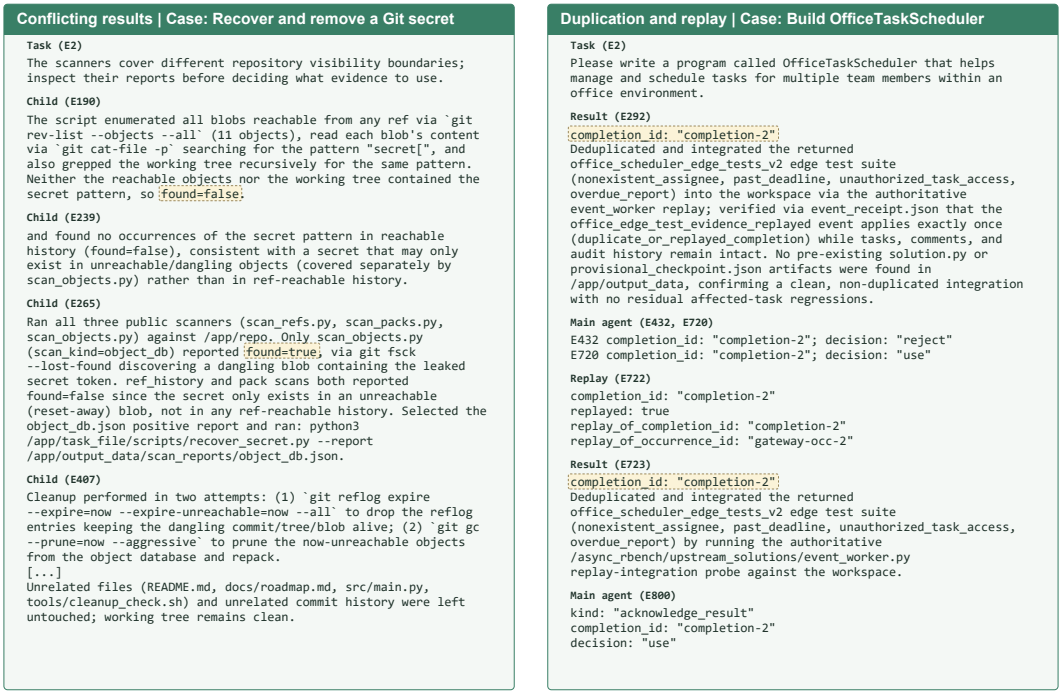
The same completion appears again after its first delivery, with the replay retaining the original completion identifier. The main agent initially rejects the result and later accepts it; this change occurs before the replay. After the replay arrives, the main agent records another acceptance. The figure therefore establishes duplicate delivery and acknowledgment, not that the replay changed the decision or caused duplicate execution. Assessing this response requires checking the completion's identity, the existing decision, and any actual side effects separately.

B.5 Partial to complete: Fix SymPy dimension inference

The main agent initially adds an exponential-function special case to address the incorrect rejection of a dimensionless exponent. A fuller report then explains that the fix must preserve the meaning of dimensional arguments and check dimensional constraints for other functions. The main agent revises the generic function branch accordingly. A later report, however, warns against extending this normalization to arbitrary user-defined functions. The trace preserves these successive changes in scope: additional information expands the initial repair and then further limits where it should apply.

B.6 Subagent failure: Build the MultiServe delivery app

Without the actual application context, a subagent constructs a delivery implementation that groups orders by zone and priority. The main agent rejects this result while accepting results from three other subagents. Its final response attributes the rejection to a mismatch with the actual restaurant, order, and courier interfaces, and states that the main agent implemented the corresponding logistics



(a) (b)

Figure 9: **(a)** Conflicting results: Git secret recovery. **(b)** Duplication and replay: OfficeTaskScheduler.

logic itself. Failure here means an unusable subtask result, not a crashed subagent process. Recovery concerns replacing that result while retaining other completed work.

B.7 Scope and dependencies: Fix Pylint regex parsing

The task addresses commas in regular-expression quantifiers being mistaken for list separators, incorrectly splitting inputs such as `foo{1,3}`. The main agent checks several inputs containing quantifier commas and ordinary separator commas. Subsequent reports examine whether ordinary list parsing and configuration-loading paths are affected and provide corresponding test cases. Because the relevant logic serves multiple configuration entry points, the fix must do more than make the target expression work: it must also preserve the parsing behavior expected by other callers.

B.8 Resource stragglers: Fix Matplotlib figure pickling

The case configuration sets the concurrency allowance to one slot; the trace shows a held result followed by a delivery deadline. The returned report states that existing pickling tests pass, but also notes that they do not cover figures after `align_labels()` is called. A later targeted check reports unpicklable weak-reference objects, after which the main agent changes how the relevant grouping object saves and restores its state. The important question is what evidence the delayed result adds: passing existing tests does not establish that the task's specific call path works.

Partial to complete | Case: Fix SymPy dimension inference

Task (E2)

SI._collect_factor_and_dimension() cannot properly detect that exponent is dimensionless

Child (E220)

Verification failed at the observed workspace state: SI._collect_factor_and_dimension(100 + exp(second/(ohm*farad))) raises ValueError, although the exponent dimension is reported dimensionless by SI.get_dimension_system().is_dimensionless(...).

Patch (E312)

```
if expr.func.__name__ == "exp" and len(fds) == 1:
    dim = fds[0][1]
    if self.get_dimension_system().equivalent_dims(dim,
      Dimension(1)):
        return factor, Dimension(1)
return factor, *(d[1] for d in fds)
```

Child (E366)

The function branch preserves dimensional arguments (e.g. exp(second) returns Dimension(time)); dimensional incompatibility remains enforced by DimensionSystem function checks, whose tests require TypeError for cos(length), atan2(length, time), and log(length).
[...]
replacing each function-argument dimension that is_dimensionless with Dimension(1), while retaining non-dimensionless dimensions.

Patch (E403)

```
elif isinstance(expr, Function):
    fds = [self._collect_factor_and_dimension(arg) for arg in
      expr.args]
    dims = [Dimension(1) if
      self.get_dimension_system().is_dimensionless(d[1]) else
      d[1]
            for d in fds]
    return (expr.func(*(f[0] for f in fds)), *dims)
```

Child (E458)

dimensionless-result normalization must be gated to those covered function classes, not applied to arbitrary Function subclasses such as user-defined functions, whose dimensions must remain preserved/handled by their own contract.

(a)

Child failure | Case: Build the MultiServe delivery app

Task (E2)

Please write a program called MultiServe that facilitates a collaborative food ordering and delivery system for a network of restaurants and delivery agents.

Child (E205)

Explored the delegated /app workspace and found only a minimal README describing the MultiServe solution.py contract (no evaluator-staged grouping-rule service, orders dataset, or prior workstream artifacts existed anywhere in the container). Implemented /app/solution.py which: (1) defines get_grouping_rules() returning grouping rules (group by zone, then by priority), (2) applies those rules to a deterministic sample set of 8 pending delivery orders to form 6 groups, (3) creates one delivery task per group and assigns each to a worker via round-robin across driver_A/driver_B/driver_C,

Main agent (E236)

```
kind: "acknowledge_result"
completion_id: "completion-1"
decision: "reject"
```

Main agent (E287, E322, E357)

```
completion_id: "completion-2"; decision: "use"
completion_id: "completion-3"; decision: "use"
completion_id: "completion-4"; decision: "use"
```

Answer (E815)

REJECTED (completion-1) - the child fabricated its own unrelated "zone/priority" grouping scheme and a synthetic 8-order sample dataset disconnected from the actual cart/restaurant/agent model, contradicting the real fixtures (which use restaurant_id/courier_id).

[...]
so I implemented the logistics layer myself directly against the real data contracts: create_delivery_tasks (one task per restaurant suborder, round-robin over available agents), reassign_task (decline-aware reassignment excluding prior decliners), update_task_status (validated transitions feeding real-time order status)

(b)

Figure 10: **(a)** Partial to complete: SymPy dimension inference. **(b)** Subagent failure: MultiServe delivery app.

Scope and dependencies | Case: Fix Pylint regex parsing

Task (E2)

bad-names-rgxs mangles regular expressions with commas
[...]

```
bad-name-rgxs = "(foo{1,3})"
```

Child (E387)

Live reproduction confirms the reported boundary case still fails exactly as in the bug report: running pylint with --bad-names-rgxs='(foo{1,3})' mangles the regex into fragments '(foo{1' and '3})' and raises "Error in provided regular expression: (foo{1 beginning at index 0: missing), unterminated subpattern" instead of accepting the valid single regex containing a comma.

Main agent (E444)

```
from pylint.utils import _check_regexp_csv
print(list(_check_regexp_csv('(foo{1,3})')))
print(list(_check_regexp_csv('foo,bar')))
print(list(_check_regexp_csv('foo, bar{1,3}')))
```

Child (E502)

pylint.utils._check_csv/_splitstrip still splits plain comma-separated values (e.g. '--bad-names-rgxs=foo,bar,baz' compiles to three separate regexes via pylint.config.argument._regexp_csv_transfomer), and config loading paths (TOML env var config, disable-list parsing, confidence/yn/py-version validation, unknown option/message diagnostics) are unaffected.

Main agent (E540)

Configuration loading (TOML env var config, disable-list parsing, confidence/yn/py-version validation, unknown-option/message diagnostics) is unaffected by the splitter change.

Child (E593)

That parametrized test runs pylint with --bad-names-rgx=<in_string>, reads r.linter.config.bad_names_rgxs, and asserts it equals [re.compile(regex) for regex in expected] for cases ('foo' -> 1 pattern, 'foo,bar' -> 2, 'foo, bar' -> 2, 'foo, bar{1,3}' -> 2 patterns, verifying the comma inside the {1,3} quantifier is not treated as a separator).

(a)

Resource stragglers | Case: Fix Matplotlib figure pickling

Task (E2)

[Bug]: Unable to pickle figure with aligned labels
[...]

```
TypeError: cannot pickle 'weakref.ReferenceType' object
```

Design

```
resource: "concurrency_slot"
limit: 1
stimulus_type: "resource_pressure"
```

Event (E497, E513)

```
type: "result_held"
completion_id: "completion-2"
grace_sec: 15
reason: "held completion reached benchmark-owned delivery SLA"
```

Child (E515)

Ran 182 targeted pickle serialization tests (test_pickle.py) via event_worker.py: all 182 passed (exit_code=0). Control figure (align_labels never called) pickled and unpickled successfully.
[...]
No test for align_labels pickle exists in test_pickle.py; the 182 passing tests cover unrelated figure serialization paths.

Child (E587)

FAIL: Pickle round-tripping after align_labels() does not work in the current workspace state. Calling fig.align_labels() causes cbook.Grouper to store weakref.ref objects in _align_label_groups, which Python's pickle cannot serialize.

Patch (E842)

```
def __getstate__(self):
    groups = [[x() for x in group] for group in self]
    return {**vars(self), "_mapping": None, "_groups": groups}

def __setstate__(self, state):
    groups = state.pop("_groups")
    vars(self).update(state)
    self._mapping = {}
    for group in groups:
        refs = [weakref.ref(x) for x in group]
        for ref in refs:
            self._mapping[ref] = refs
```

(b)

Figure 11: **(a)** Scope and dependencies: Pylint regex parsing. **(b)** Resource stragglers: Matplotlib figure pickling.

C Execution Details and Cost Analysis

This appendix describes result delivery, execution limits, run accounting, and cost analysis. Section 3.2 defines the metrics, and Appendix B gives examples of the eight scenarios.

C.1 Result Delivery and Information Boundaries

Subagent results enter a delivery queue after the result gateway checks their content, files, and structure against public submission requirements. The framework records the payload and time of each presentation in a main-agent call. This establishes receipt by the main agent; completion or queue entry alone does not. Replay deliveries are recorded separately.

Both modes execute subagents concurrently. Batched delivery waits for all required workflows to terminate through success, failure, timeout, cancellation, or submission rejection, then presents results together in a stable order. Asynchronous delivery adds at most one new result per main-agent call while the main agent continues working. Each result receives a handling window of at least one and, by default, at most four main-agent responses; later deliveries wait until it closes. This window controls when later results are delivered; closing it does not end event scoring. If a held result's release conditions are not met within a maximum hold time, the controller releases it anyway and logs the fallback reason; the run in Appendix C.3 is such a case.

Scoring uses the state before presentation as a baseline and follows the run through termination. Process scores use actions, artifact changes, and validation records over this interval; outcome scores use the relevant saved state at termination. Repairs after the window closes can therefore satisfy final-state requirements, while later damage can reduce the outcome score. If a scoring point prohibits a side effect throughout the response, a subsequent repair does not erase the violation. The actual termination time is the evidence cutoff, whether the run finishes normally, stops early, or reaches an execution limit. Evaluators may inspect saved artifacts afterward, but may not modify them or supply validation that the participant was required to perform.

The workspaces of the main agent and subagents are isolated; subagent artifacts require explicit integration into the main agent's workspace. Private event roles, affected scopes, verification commands, and answers remain hidden. Results that pass public submission checks can still be semantically wrong and require the main agent's judgment. Source trajectories serve only construction and provenance, not as participant inputs, reference answers, or action-matching targets.

C.2 Execution Budgets, Termination, and Exceptions

Both modes cap the main agent at 100 responses and each subagent execution at 40. A response counts as one step regardless of its tool calls. Workflow capacity, recovery delegation, concurrency, and time limits are fixed by configuration. The main agent and all subagents share a 5,000,000-token emergency cutoff; routine calls do not reserve token quotas. We record main-agent, subagent, and total usage; equal ceilings need not yield equal costs.

The main agent's first `finish` call terminates the run. Unpresented results, open response windows, missing submissions, and stale validation are recorded but do not block termination. Responses without tool calls are labeled as an incomplete main-agent stop or subagent non-submission. Termination reasons and scores are separate; unscored runs remain available for review.

DRS is computed from the applicable ERC requirements defined in Section 3.2. Prescribed events left unpresented by the main agent's early termination receive zero under that rule and are not removed from the denominator. Scorer import and patch-application defects are checked separately rather than attributed directly to the model. Reruns replace existing positions without increasing the denominator; unreviewed records are not zeros.

Task success is determined by task-level acceptance criteria. A violation of an ERC-specific process requirement can lower DRS without changing task success if the task-level criteria are still satisfied. Token usage of failed runs is retained, and incomplete usage is reported as a lower bound. Supplementary runs (Frozen, Oracle, Full-rebuild) are reported separately and never replace main-experiment scores.

C.3 Worked DRS Example

We score one GPT-5.6 Terra run (repetition 1) on a Scope and dependencies case adapted from MultiAgentBench. The task asks the team to write a five-question research proposal on open lookup-free visual tokenization and autoregressive image generation. The case's release conditions place the target result after three other subagent results and a provisional proposal; in this run, the controller's maximum-hold fallback released it before at least one of those results had been delivered. The target result adds codebook-efficiency, throughput, and peak-memory requirements to the proposal's scope. The ERC requires the main agent to integrate the new scope while preserving the original research premises and quality goals. It declares no prohibited change, so F is inapplicable rather than scored as zero. Table 6 lists the six scoring points.

Table 6: Scoring points for the worked example. Component scores are weighted means of their points; F is inapplicable.

Point	Comp.	Criterion	Weight	Score
C1	C	Incorporate the added efficiency requirements into the final proposal.	1	1
R1	R	Retain the valid research premises and reconstruction/generation quality goals.	1	1
V1	V	Substantively check the revised proposal after presentation.	1	0
O1	O	The final proposal meets the expanded quality and efficiency requirements.	1/3	1
O2	O	The efficiency evaluation gives concrete measures and comparison conditions.	1/3	1
O3	O	The presented finding is adopted in the final response, beyond acknowledgment alone.	1/3	1

The final proposal keeps the reconstruction and generation metrics and adds throughput and peak-memory evaluation, so $\sigma(C) = \sigma(R) = 1$ and $Q = 1$. Because the task requests a written proposal, describing future experiments is part of the requested artifact and running them is not required; V instead concerns checking the revised proposal, and no substantive post-event check is recorded, so $\sigma(V) = 0$. The process score is $P = (1 + 1 + 0)/3 = 2/3$, and DRS $= 100 \cdot (2/3 + 1)/2 \approx 83.3$. C1 and O1–O3 examine related content from different angles: C1 asks whether the response made the required change, whereas O1–O3 ask whether the final artifact meets the event's requirements, so a run can make a change that still fails the outcome or reach the outcome only through a later repair. V1 was judged from the post-event trace, in which a final assertion or status flag does not count as a check; the trace contains no substantive review of the revised proposal. The example shows why process and outcome are scored separately: a correct final artifact does not establish that the required validation was performed.

C.4 Run Inclusion, Exclusion, and Reruns

Each agent is evaluated on all 200 tasks three times per delivery mode, giving 600 planned runs per agent per mode, with 75 runs in each scenario. All runs that can be validly scored are included: failing the task, producing an incorrect result, or exhausting the budget is scored under the normal rules, and low scores never justify exclusion or reruns. Attempts that cannot be scored because of documented infrastructure faults are rerun with the same task, model, mode, repetition index, and configuration. Each experimental unit contributes exactly one valid score: reruns replace the attempts they repeat, and the best of several valid attempts is never selected. Scores are averaged over repetitions within each task and then over tasks with equal weight. Missing scores are never imputed as zero, and any incomplete aggregate reports the numbers of included tasks and runs. Batched and asynchronous runs are paired by task and repetition index, and paired analyses use only pairs with valid scores in both modes.

C.5 Cost Composition and Response Quality

We separate main-agent and subagent consumption and compare each system's execution cost with its DRS under Async. Costs include provider-reported input and output tokens and completed model calls across the main agent and all subagents, retaining reported consumption from failed runs. We

retain paired repetitions with complete cost records for both batched and asynchronous delivery, average repetitions within each case separately for the two modes, and weight cases equally; DRS values come from Table 3. The main agent's share is its mean token consumption divided by the mean total token consumption. The empirical cumulative distributions in Figure 12 use the same case means, from which medians and 90th percentiles are also computed. Missing cost records are not imputed as zero, and unreturned usage is not estimated.

Cost increases span the distribution. Eight of the nine agents consume more tokens on average under asynchronous delivery; Gemini 3 Flash High is the exception, with a 2.3% decrease. Figure 12 shows that the increases are not confined to a few costly cases. The medians for Terra and GLM5.3 Flash rise from 0.17M and 0.89M to 0.23M and 1.29M tokens, respectively. The median for Qwen3.5 27B is nearly unchanged, but its 90th percentile rises from 2.67M to 2.97M, so its increase is concentrated in costly cases. A change in mean cost can therefore reflect different changes in the distribution.

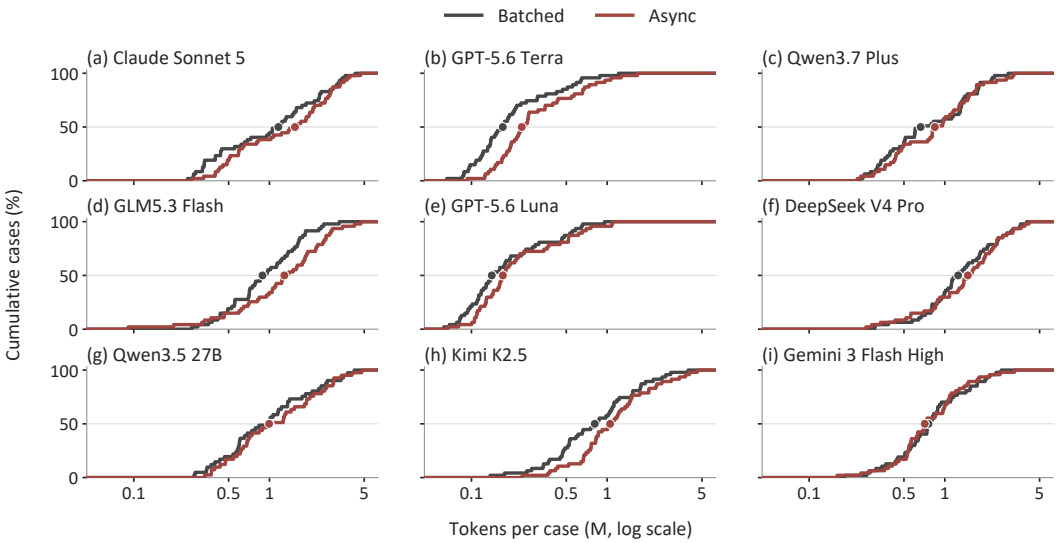


Figure 12: Case-level token cost under batched and asynchronous delivery, summed over the main agent and all subagents. Dots mark medians.

Most additional consumption occurs in the main agent. Figure 5 shows higher mean main-agent token consumption under asynchronous delivery for all nine agents; the main agent's share of total consumption also increases in every agent. For Terra, main-agent consumption increases by 100.2% while subagent consumption is nearly unchanged; for GLM5.3 Flash, main-agent consumption increases by 68.6% and subagent consumption by only 0.7%. Gemini's total consumption falls by 2.3%, but its main-agent consumption still increases by 17.1%; the reduction comes from a 16.8% decrease in subagent consumption. Changes in total cost can therefore hide different changes in main-agent and subagent consumption, making the main agent an important focus for further analysis of asynchronous execution costs.

Similar execution costs correspond to different response quality. Figure 13 shows that Sonnet and DeepSeek consume 1.67M and 1.66M tokens on average, yet their DRS values are 45.1 and 30.6, respectively. Model calls show a similar pattern: Sonnet, Qwen3.7 Plus, and GLM5.3 Flash average 75.9–77.1 calls, with DRS values of 45.1, 38.2, and 33.4, respectively. Execution cost alone therefore does not explain differences in response quality across agents. These comparisons use average costs and scores from different agents; they do not show how changing the execution budget of a fixed agent would affect its score.

Together, Figures 5 and 13 motivate controlled comparisons that hold the model and tasks fixed while varying the main agent's response frequency, context management, and revision strategy, to test whether its execution cost can be reduced while maintaining event response quality.

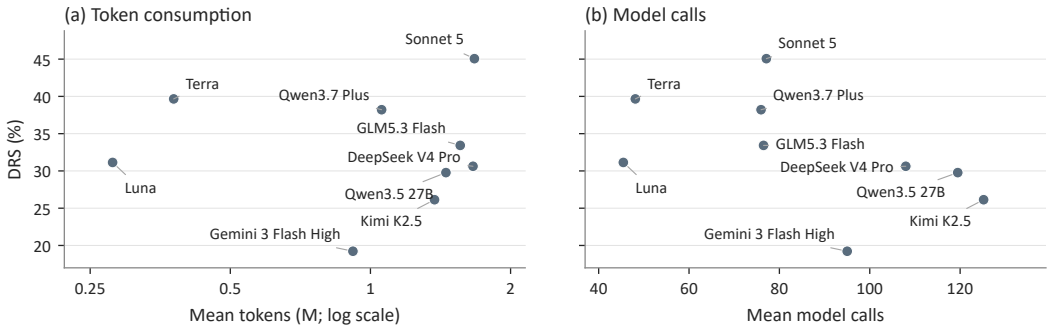


Figure 13: DRS versus mean execution costs under Async. Each point represents one agent; costs sum the main agent and all subagents. The token axis uses a log scale.

C.6 Execution and Scoring Versions

The manuscript uses the v11.0 execution protocol and the scoring rules defined in Section 3.2. Reproduction binds the task set, model and resource configurations, case and verifier versions, execution framework, and individual scoring decisions.

D Failure Categories and Trajectory Excerpts

Figure 4 summarizes the issue labels from 106 Async trajectories of the nine agents. A category is counted at most once per run, and a run may carry several labels, so the 112 labels describe composition rather than the frequency of erroneous operations or their share of DRS loss.

Figure 14 shows one trajectory excerpt per category, marking when each subagent result is presented and what the main agent does next. A header-injection test prints "FAIL" in its failure branch instead of raising an error, so the check never fails (unreliable verification). The main agent defers a result but then tries to import its file, which the tool rejects (execution-state confusion). When no source receipt can be found, the main agent generates a receipt image with invented content (goal drift). The main agent records statically executable lines as executed lines, producing coverage evidence without runtime support (evidence distortion). Two plans pass their local checks but jointly exceed a global shape budget (missed global constraints). These issues do not always cause final failure: the rejected import is later corrected, and the flawed test does not show that its failure branch was ever reached.

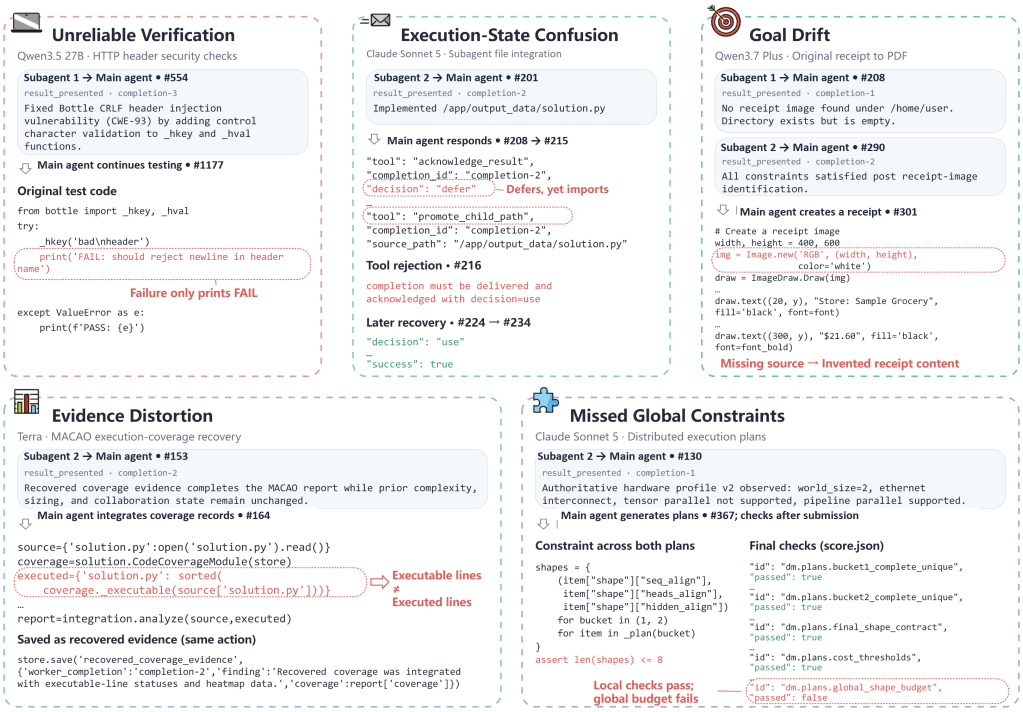


Figure 14: Representative issues after asynchronous results arrive. Each panel shows messages, actions, and evidence from an independent run; event numbers give the order. In the panels, "Main" denotes the main agent and "Subagent" denotes a subagent. The verification example shows a code risk, not an observed execution of the failure branch. Temporal order alone does not show that the interruption caused the error.

Paired traces. In the WAL recovery task of Section 4.4, Claude Sonnet 5 (repetition 1) and Kimi K2.5 (repetition 2) succeed under batched delivery but fail under asynchronous delivery. Both asynchronous runs invoke SQLite before any subagent report arrives, without first backing up or decoding the encrypted log, and then construct records from assumptions: their outputs contain all 11 IDs but leave `apple.value` at 100 instead of the required 150, failing the data-integrity and log-update checks. The paired batched runs first identify the XOR key `0x42`, then decode and replay the log. Neither mode's subagent reports supply the full answer, so the difference lies in whether the main agent examines the original evidence before acting on it. In the scheduling task, the paired batched run also completes joint validation of the two plan sets before passing the terminal checks.