HACKATHON-24H: EVALUATING LONG-HORIZON AGENTS ON PROJECT-SCALE SOFTWARE ENGINEERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Evaluating LLM-based agents on long-horizon software engineering tasks is highly challenging, as complex reasoning and execution behaviors unfold dynamically over time. To address this, we introduce Hackathon-24h, a novel evaluation protocol that shifts the paradigm from static measurement to continuous, trajectory-level process scoring. The protocol tasks agents with building project-scale systems software from scratch under a 24-hour wall-clock budget, requiring them to deliver runnable, incremental releases at regular intervals rather than withholding progress until the end. A decoupled architecture systematically snapshots and evaluates these updates against industrial conformance suites to capture continuous progress. Detailed analyses of these execution trajectories demonstrate the unique value of this continuous-delivery approach. By capturing intermediate dynamics, we uncover critical phenomena that final-state metrics completely miss such as hidden regression, delayed success, and temporal dynamics. We release our code at `https://anonymous.4open.science/r/Hackathon-24h-anonymous/`.

1 INTRODUCTION

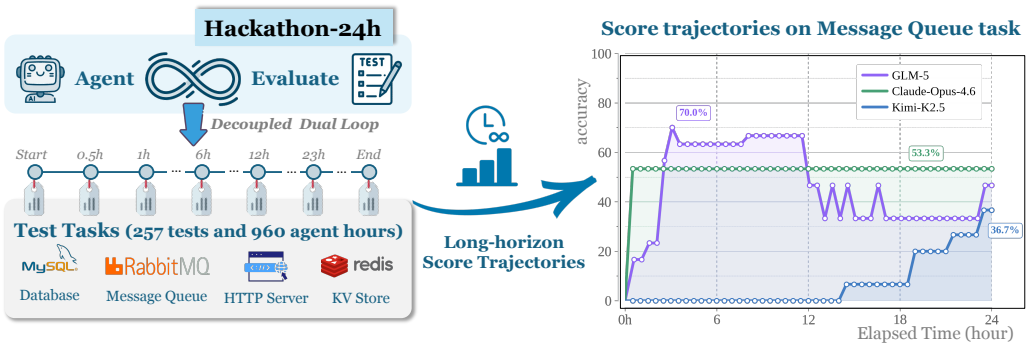


Figure 1: Overview of *Hackathon-24h*. **Left:** an agent loop builds one of four systems-software targets over 24 hours; a decoupled test loop snapshots and scores against external conformance suites every 30 minutes. **Right:** three Message Queue runs trace radically different trajectory shapes: a $70\% \rightarrow 47\%$ *hidden regression* (GLM-5), a stable 53% plateau (Claude-Opus-4.6), and a near-zero floor with a late jump to 37% *delayed success* (Kimi-K2.5), motivating our peak / final / AUC triplet.

The advent of coding agents stands as one of the most transformative breakthroughs in contemporary artificial intelligence. Moving far beyond code snippet completion (Rozière et al., 2024; Guo et al., 2024), recent large language model (LLM) agents are envisioned as independent software engineers capable of architecting, implementing, and iterating on project-scale software systems (Yang et al., 2024; Wang et al., 2025). Agents promise to fundamentally reshape the landscape of software engineering, transitioning AI from a reactive tool to a proactive, long-horizon collaborator (Kwa et al., 2026; Anthropic, 2026b).

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

Table 1: Comparison with prior coding-agent benchmarks across five dimensions: from-scratch construction, external test source, within-run sampling, trajectory scoring, and agent runtime horizon. Horizon is the budget the source paper sets, or the upper end of the agent runtime it reports when only empirical numbers are given; "–" marks papers that do not report an agent runtime budget; ◇ denotes mixed test sources.

Benchmark	From scratch	External tests	Within-run sampling	Trajectory scoring	Horizon
SWE-Bench Pro (Deng et al., 2025)	✗	✓	✗	✗	–
Terminal-Bench (Merrill et al., 2026)	✗	✗	✗	✗	2 h
NL2Repo-Bench (Ding et al., 2026)	✓	✓	✗	✗	–
WebGen-Bench (Lu et al., 2026)	✓	✗	✗	✗	–
LongCLI-Bench (Feng et al., 2026)	✗	✗	✗	✗	30 min
ProgramBench (Yang et al., 2026)	✓	◇	✗	✗	6 h
HCAST (Rein et al., 2025)	✗	✗	✗	✗	10 h
RE-Bench (Wijk et al., 2025)	✓	✗	✗	✗	8 h
Hackathon-24h	✓	✓	✓	✓	**24 h**

Progress in this direction depends on evaluation protocols that accurately reflect the capabilities and failure modes of such agents. The community has developed a rich spectrum of code and software-engineering benchmarks, evaluating function-level program synthesis (Chen et al., 2021; Austin et al., 2021), algorithmic problem solving (Jain et al., 2025), bug-fixing for real GitHub issues (Jimenez et al., 2024; Deng et al., 2025), and tool-using agents in broader environments (Merrill et al., 2026). These benchmarks have been crucial for measuring increasingly realistic coding behavior. However, during a long run, an implementation may improve, regress, plateau, or recover, and a final-state score alone cannot reveal when useful functionality emerges or whether previously achieved correctness is sustained. Following these changes requires repeated evaluation of the evolving implementation against a fixed set of requirements. To support continued development (Yao et al., 2023; Shinn et al., 2023), the evaluation must also return useful feedback while keeping hidden test cases and scoring machinery outside the agent's workspace.

To address this gap, we introduce **Hackathon-24h**, a continuous-delivery evaluation protocol for long-running coding agents (Figure 1). Agents are given a 24-hour wall-clock budget to build one of four project-scale systems from scratch: a relational database, a message queue, an HTTP server, or a Redis-compatible key-value store. Rather than scoring only the final implementation, Hackathon-24h evaluates workspace snapshots every 30 minutes. Specifically, this is a *decoupled dual-loop* architecture, which separates *development* from *evaluation*, where an external evaluator snapshots the docker workspace, executes each release in isolation, and returns anonymized feedback. Each trajectory is summarized by peak, final, and AUC scores, capturing best intermediate capability, final-state quality, and functionality sustained over time.

We instantiate this protocol with ten frontier LLMs across the four tasks under a fixed Claude Code harness, producing 40 sampled 24-hour trajectories; a separate 6-run probe varies the harness on one task (Section 3.4), and a single-case comparison examines feedback availability under a fixed progressive prompt (Section 3.5). This cohort is intended to demonstrate the evaluation setting, not to serve as a statistically powered leaderboard: the evaluation object is the deployed agent system, including the model, harness, prompts, feedback protocol, provider runtime, and wall-clock budget. The resulting trajectories expose behaviors that final-state-only scoring would obscure, including *hidden regression*, *delayed success*, and *temporal dynamics*. The strongest database run reaches only 39.6% peak performance, while the harness probe shows that harness choice can shift scores by margins comparable to model differences on the tested task. A Dolos-based analysis finds little evidence of substantial code-level copying from the selected reference implementations (Section 3.6). Our contributions are as follows:

1. **Continuous-delivery evaluation protocol.** Hackathon-24h measures 24-hour from-scratch project construction through trajectory-level checkpoints rather than final-state-only submissions.

2. **Decoupled dual-loop architecture.** The system isolates agent development from snapshot evaluation, testing runnable releases against external industrial suites while returning anonymized feedback.
3. **Empirical trajectory evidence for deployed agent systems.** A 960-agent-hour demonstration cohort and a harness-sensitivity probe reveal hidden regression, delayed success, temporal dynamics, and harness sensitivity; a complementary contamination analysis assesses token-level overlap with reference implementations.

2 HACKATHON-24H

Hackathon-24h has three components: a task suite of four systems-software targets (Section 2.1), a decoupled dual-loop architecture that runs the agent in a Docker sandbox and scores 30-minute snapshots in a separate runtime through an anonymized brief (Section 2.2), and three trajectory metrics that summarize each 24-hour run (Section 2.3).

2.1 TASK DESIGN

Selection criteria. Two properties shape our task selection. *Long-horizon depth*: each system has enough internal structure to admit incremental progress instead of a single pass-or-fail outcome, and to surface long-horizon failure modes such as *hidden regression* and *delayed success* within a 24-hour run. *Industrial conformance*: each system has a precise, publicly documented specification, with established industrial test suites serving as external judges, so correctness is not subjective. Per-task structural requirements are summarized in Table 7 (Appendix A.1).

Integration pipeline. Each software system goes through three setup steps before it becomes a scorable task. (1) *Specification staging*: the relevant protocol references (RFCs, conformance documents, design notes) are copied into the agent's workspace, so the agent grounds its implementation against the same standard the suites encode; the per-task list is given in Appendix A.2. (2) *Task instructions*: each task uses two prompts. The init prompt opens the run and carries the per-task content: protocol requirements, architecture and operational expectations, and development suggestions for that specific system, wrapped in a small shared scaffold that asks for a runnable minimum first plus a structured plan file. The iteration prompt is identical across tasks: it is reissued at each iteration with the latest available feedback brief and tells the agent to pick one item from the plan, implement it minimally, log the result, and keep the system runnable. Prompt structures and excerpts are in Appendix A.3; full templates are included in the supplementary artifact. (3) *Tier assignment*: within each task, tests are grouped into a *tier ladder*, i.e., an ordered sequence of tiers $L_0, L_1, \ldots$); L_k extends L_{k-1} with additional functional scope or workload scale, with L_0 as a smoke test for basic correctness and higher tiers adding concurrency, throughput, or compatibility coverage. The ladder gives partial credit for partial systems while extending coverage to more demanding functionality and workloads. Table 2 summarizes the four tasks and their suites; the per-tier scope is given in Table 6 (Appendix A.1). Within each task, the specification and external test suites remain fixed throughout the run, so successive checkpoints measure how the evolving implementation satisfies the same requirements.

Reference validation. Once a suite is wired into the pipeline, we run it against the canonical production reference for that system (MySQL for database, RabbitMQ for message queue, nginx for HTTP server, Redis for the KV store) and confirm it reaches full credit on every tier.

Table 2: The four tasks, their tier ladders, and the external suites that score them. Across the four tasks, 13 suites contribute 257 conformance tests in total.

Task	Tiers	Tests	Evaluation Suites	Description
Database	L0–L3	48	Sysbench, TPC-C, TPC-H	MySQL-compatible RDBMS
Message Queue	L0–L2	30	pika, OMQ, RabbitMQ PerfTest	AMQP 0.9.1 broker
HTTP Server	L0–L4	47	h1spec, CISPA, TFB	HTTP/1.1 web server
Redis KV Store	L0–L6	132	Redis TCL, redis-benchmark, YCSB, memtier	Redis-compatible store

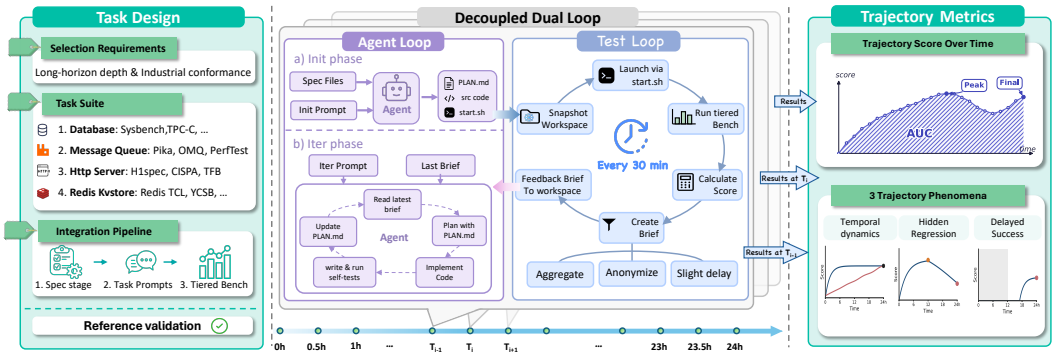


Figure 2: Hackathon-24h architecture. From left to right: task construction from staged specifications and tiered external test suites (Section 2.1); decoupled agent and evaluation loops producing scored snapshots with anonymized feedback every 30 minutes (Section 2.2); and 24-hour trajectories summarized by peak, final, and AUC (Section 2.3).

2.2 Evaluation Protocol

Long-horizon evaluation has to keep development and scoring separate for a full day of agent activity. We organize the protocol as a decoupled dual-loop architecture: two asynchronous loops run in separate runtimes and communicate only through a shared workspace and anonymized brief files (Figure 2). The arrangement is closer to a continuous-integration pipeline than to a single-shot answer check: the agent loop continuously iterates on the implementation, while the test loop independently snapshots and scores the workspace every 30 minutes over the 24-hour budget.

2.2.1 Agent Loop

The agent runs inside a Docker container with the workspace and controller as the only mounts, leaving test scripts, snapshot machinery, and the host filesystem unreachable. During initialization, the agent receives the staged protocol references and the init prompt, and from these alone produces an initial workspace with three artifacts: source code for a runnable minimum implementation, a launcher `start.sh` that the test loop will call, and a typed backlog `PLAN.md` that scopes later iterations.

At each iteration, the agent receives the iteration prompt together with the latest available anonymized brief and works against the existing workspace through a ReAct cycle (Yao et al., 2023): reading the brief, planning against `PLAN.md`, implementing code, running local acceptance tests, and updating `PLAN.md` entries are interleaved freely rather than executed in a fixed sequence.

Two invariants hold across the whole run: `start.sh` stays runnable, and iteration log entries are append-only. The session is resumed rather than restarted, so prior reasoning and tool transcripts carry into the next cycle; when context approaches the model's window the harness compacts older turns in place. Appendix A.3 summarizes the `PLAN.md` structure and provides prompt excerpts.

2.2.2 Test Loop

The test loop runs on the host. Every 30 minutes, it snapshots the workspace into an isolated test instance, launches the system via `start.sh`, runs the tier benchmarks against it, and writes a brief into the workspace at `last_brief`. Snapshot, scoring code, and benchmark assets all live outside the container, so the agent can neither read which tests are about to run nor change how they are scored.

Three design choices shape the brief feedback. *Aggregation*: overall and per-tier pass rates are accompanied by category-level error summaries. *Anonymization*: suite names are replaced with generic labels, and failed checks may be listed using opaque identifiers; test code remains outside the agent's workspace. *Slight delay*: the brief reflects the latest evaluated snapshot rather than the live implementation. This feedback supports iteration through summarized results and anonymized failure

signals. Appendix A.3 reproduces one brief verbatim. Section 3.5 examines feedback sensitivity in a single-model, single-task case study.

Snapshots are indexed by wall-clock time to compare implementations after the same elapsed budget. Task milestones instead align runs by development progress, which may take different amounts of time across agents. The 30-minute cadence is an operating point rather than a claimed optimum; Appendix D.1 quantifies the AUC information lost under coarser resampling.

2.3 METRICS

The evaluation protocol produces a sequence of checkpoint scores for each run. We summarize this trajectory through three questions: how much correctness did the implementation achieve, how much remained at the end, and how early and consistently was it available? These correspond to peak, final, and AUC, respectively.

Let T=24h be the budget and $\{s(t_i)\}_{i=0}^{n}$ the sequence of $n{+}1$ checkpoint scores with $0 < t_0 < \cdots < t_n \leq T$. Each checkpoint score $s(t_i)$ is the percentage of the task's tests passed by that snapshot, with every test weighted equally across all tiers. We compute the three metrics as follows:

$$s_{\text{peak}} = \max_i s(t_i), \qquad s_{\text{final}} = s(t_n), \qquad s_{\text{AUC}} = \frac{1}{T}\int_0^T s(t)\,dt. \tag{1}$$

AUC is computed by step-function integration: each checkpoint's score holds until the next, score 0 is assumed before the first checkpoint, and the last score extends to T. Uniform time weighting reads s_{AUC} as the score expected if the run were stopped at a uniformly random time in $[0, T]$; upweighting late checkpoints would pull AUC toward s_{final}. We read disagreements as signal: $s_{\text{peak}} - s_{\text{final}}$ flags score-level regression. This is a partial signal for the test-level *hidden regression* characterized in Section 3.3. We identify *delayed success* from the first positive checkpoint following an initial zero-score period (Section 3.3).

3 EXPERIMENTS

We first examine the 40 main-study trajectories through two research questions. **RQ1**: does the choice of summary metric (peak / final / AUC) change what we read off a single cell, and do peak, final, and AUC surface different aspects of the same trajectory? **RQ2**: which trajectory phenomena that final-state scoring suppresses (*hidden regression*, *delayed success*, *temporal dynamics* from Section 1) surface on the 40 runs, and how often? We then examine harness sensitivity (Section 3.4) and feedback sensitivity (Section 3.5), and assess code-level overlap with selected reference implementations (Section 3.6).

3.1 SETUP

Agents and models. We evaluate 10 coding-capable language models from five families: Claude (Claude Opus 4.6 (Anthropic, 2026a)), GLM (GLM-5 (GLM-5-Team et al., 2026), GLM-4.7 (Z.ai, 2025)), Kimi (Kimi-K2.5 (Kimi Team et al., 2026)), Qwen (Qwen3.5-Plus (Qwen Team, 2026), Qwen3-Max (Qwen Team, 2025b), Qwen3-Coder-Plus (Qwen Team, 2025a), Qwen3-Coder-Next (Cao et al., 2026)), and MiniMax (MiniMax-M2.7 (MiniMax, 2026b), MiniMax-M2.5 (MiniMax, 2026a)). All 10 models share the same agent harness, Claude Code, so that the harness is held constant across models; the contribution of harness choice itself is studied separately in Section 3.4. All runs in this section ask the agent to implement the system in Python.

Effort under the wall-clock budget. Across the 10 models on the same wall-clock budget, per-cell median consumption spans 26× in plan iterations, 6× in LLM steps, 5× in tool calls, and 97× in tokens, yet none of these spreads predicts cell-level outcome. Claude Opus 4.6 leads the cohort on AUC across every task on 291M median tokens per cell, while Qwen3-Max consumes 11.85B (40× more) and lands in the bottom row of Table 3. Per-model medians and the wall-clock budget rationale appear in Appendix B.3.

Table 3: Main study: 10 models × 4 tasks, one 24-hour trajectory per cell. A *cell* denotes a (model, harness, task) tuple, with the harness held constant at Claude Code in this table. Each table cell shows Peak / Final / AUC (%). All runs use the Claude Code as agent harness with 24-hour budget. Bold indicates the highest single-trajectory observation in this cohort, while underlining indicates the second-highest. Replication of any cell could shift these markers (Appendix D.2 quantifies replicated-run analysis). **These are sampled trajectories from the model × harness × task process, not stable model-level capability estimates.**

Model	Database			Msg Queue			HTTP			Redis KV		
	Peak	Final	AUC	Peak	Final	AUC	Peak	Final	AUC	Peak	Final	AUC
Claude Opus 4.6	**39.58**	**39.58**	**35.59**	53.33	**53.33**	**52.22**	**100.00**	**100.00**	**91.66**	**100.00**	**100.00**	**97.09**
GLM-5	**39.58**	**39.58**	12.88	**70.00**	46.70	45.46	91.49	91.49	72.90	96.97	96.97	75.72
Kimi-K2.5	4.17	4.17	1.43	36.67	36.67	6.33	87.23	85.11	75.22	89.39	89.39	68.51
Qwen3.5-Plus	14.58	12.50	6.83	16.67	16.67	10.54	72.34	72.34	67.73	93.18	93.18	78.28
MiniMax-M2.7	0.00	0.00	0.00	0.00	0.00	0.00	74.47	74.47	71.10	90.91	86.36	83.40
MiniMax-M2.5	0.00	0.00	0.00	0.00	0.00	0.00	70.21	55.32	58.82	84.09	84.09	64.49
GLM-4.7	0.00	0.00	0.00	0.00	0.00	0.00	76.60	74.47	67.91	29.55	29.55	20.88
Qwen3-Coder-Plus	0.00	0.00	0.00	6.67	6.67	1.17	61.70	46.81	46.23	78.03	78.03	59.64
Qwen3-Coder-Next	0.00	0.00	0.00	0.00	0.00	0.00	65.96	65.96	59.22	70.45	70.45	23.15
Qwen3-Max	0.00	0.00	0.00	0.00	0.00	0.00	42.55	42.55	36.35	75.00	75.00	53.46

3.2 Results

Table 3 presents the main study (all 40 trajectories are plotted in Appendix B.1); the two paragraphs below read RQ1 off the table, and RQ2 is taken up in Section 3.3. The 257 tests sit on a wide difficulty floor: 29 of 48 database tests and 3 of 30 message-queue tests are passed by no model in the cohort, while 12 of 47 HTTP tests and 30 of 132 Redis tests are passed by all ten. The full test-set view that unpacks each cell into the set of tests passed appears in Appendix B.2.

Peak, final, and AUC capture complementary aspects of a run (RQ1). Table 3 shows how the three metrics distinguish achieved correctness, end-of-run correctness, and correctness sustained over time. On message queue, GLM-5 reaches a peak of 70.0% but finishes at 46.7%, a 23.3 pp loss that the final score alone does not reveal. On database, its peak and final both equal 39.58%, but its AUC is only 12.88% because it reaches that level at hour 22.5. Thus, matching peak and final scores can accompany substantial differences between intermediate and end-of-run performance. Similar gaps occur elsewhere: MiniMax-M2.5 loses 14.9 pp from peak to final on HTTP, while Qwen3-Coder-Next reaches 70.45% on Redis with an AUC of 23.15%.

Higher final correctness does not imply higher correctness throughout the run. On HTTP, GLM-5 finishes at 91.49%, compared with 85.11% for Kimi-K2.5, yet their AUCs are 72.90% and 75.22%, respectively. GLM-5 makes several late gains, reaching its peak at hour 23.6, while Kimi-K2.5 attains a higher time-averaged score over the full budget. Reporting final together with AUC distinguishes improvements achieved near the end from correctness sustained during development. Section 3.3 examines the underlying trajectories at the test level.

3.3 Trajectory Phenomena Suppressed by Final-state Scoring

Section 1 identified three trajectory phenomena suppressed by final-state scoring. *Temporal dynamics* already surface in Table 3: on database, GLM-5 and Claude Opus 4.6 reach peak = final = 39.58 by passing exactly the same 19 tests. Claude Opus 4.6 first reaches the set at hour 3, while GLM-5 reaches it only at hour 22.5 after 22 hours below 17%; AUC (35.59 vs 12.88) is the only column that separates the two trajectories. Appendix B.2 unpacks this at the test-set level. Here we operationalize the remaining two on the 40 cohort trajectories (Figure 3): *hidden regression*, where a test passing at one checkpoint fails at a later one (the model-side phenomenon); and *delayed success*, where a shorter-budget benchmark would record a productive run as a zero.

Hidden regression (Figure 3a). 16 of 40 cells regress at the test-case level: 12 carry an open regression at hour 24 (color extends to the right edge), 4 regress and recover within the budget (color fades back to white). The cell where this color is most saturated in this cohort is GLM-5 on message queue, where the post-hour-12 sawtooth shows six unguarded refactors that repeatedly broke and

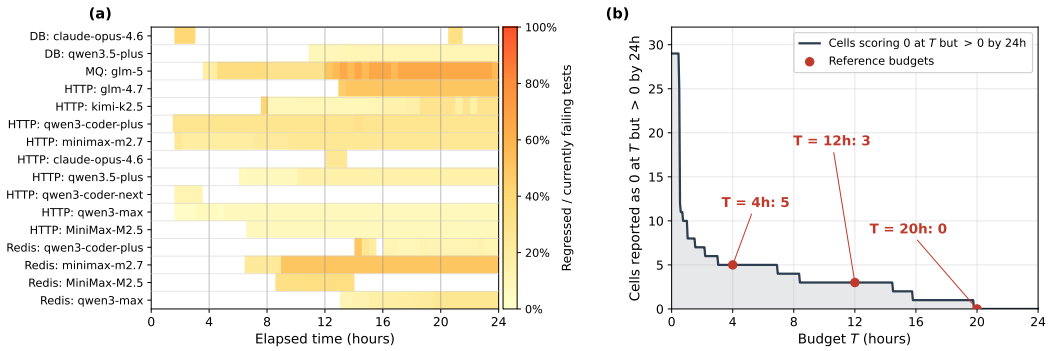


Figure 3: Trajectory phenomena on the 40 cohort cells. **(a)** Hidden regression. Each row is one of the 16 cells with at least one test that passed at some checkpoint and failed at a later one (rows grouped by task, sorted by peak regression load within task). Cell color at time T is the fraction of currently-failing tests that are regressions, with white meaning no current regression burden. **(b)** Delayed success. For each budget T, the count of cells whose score is zero across all checkpoints with elapsed time at most T but reach a positive score by hour 24.

partially refixed the same surface (peak strictly exceeds final in 4 of 6 message-queue replicates and 2 of 6 HTTP-server replicates of GLM-5 and Kimi-K2.5; see Appendix D.2). Score-level peak $-$ final ≥ 10 pp catches only 3 of these 16. Cases 2 and 3 in Appendix C illustrate oscillating and frozen post-drop regressions, respectively.

Delayed success (Figure 3b). Among runs that eventually attain a positive score, the count with no positive checkpoint yet falls steeply in the first hour, then plateaus, and reaches zero by hour 20. The latest non-zero arrivals are Kimi-K2.5 on message queue (first non-zero at hour 14.5, peak 36.67%, a mid-table productive run a 14-hour benchmark records as outright failure), Qwen3-Coder-Plus on message queue (hour 19.8, peak 6.67%), and Kimi-K2.5 on database (hour 15.8, peak 4.17%). Per-cell case studies are in Appendix C.

3.4 Harness Sensitivity

The preceding analysis characterizes trajectories produced by agents interacting with their development environments. These runs depend on both the model and the harness that organizes its prompts, tool use, and execution loop, motivating a comparison across harnesses with the model held fixed. We hold the task constant at HTTP server and vary the harness across Claude Code, OpenCode (Anomaly, 2026), and Cline (Cline Bot Inc., 2026) under two models (Kimi-K2.5 and MiniMax-M2.5); all six runs share an identical 24-hour budget, checkpoint cadence, and anonymized feedback channel, with cell-level peak, final, and AUC reported in Table 4.

Table 4: Harness sensitivity on HTTP server under two models (%). All six runs hold the task, budget, checkpoint cadence, and anonymized feedback channel constant; only the model and the agent harness vary.

Model	**Claude Code**			**OpenCode**			**Cline**		
	Peak	Final	AUC	Peak	Final	AUC	Peak	Final	AUC
Kimi-K2.5	87.2	85.1	75.2	80.9	80.9	69.7	72.3	72.3	70.2
MiniMax-M2.5	70.2	55.3	58.8	53.2	53.2	49.5	61.7	61.7	56.6

Peak-score differences across harnesses are comparable to the model gap under Claude Code. On HTTP, peak scores across the three harnesses span 14.9 pp for Kimi-K2.5 (87.2–72.3%) and 17.0 pp for MiniMax-M2.5 (70.2–53.2%). Under Claude Code, the two models differ by 17.0 pp in peak score (87.2% versus 70.2%). Thus, within this comparison, the observed peak-score differences across harnesses are similar in magnitude to the difference between the two models under a shared

harness. Final scores also vary across harnesses, spanning 12.8 pp for Kimi-K2.5 and 8.5 pp for MiniMax-M2.5 (Table 4).

Which harness reads highest under each metric depends on the model. Within Table 4, under Kimi-K2.5 the per-harness peak and final readings are Claude Code 87.2 / 85.1, OpenCode 80.9 / 80.9, Cline 72.3 / 72.3; on AUC the same six runs read Claude Code 75.2, Cline 70.2, OpenCode 69.7, because Cline reached its plateau early and held it. Under MiniMax-M2.5 the per-harness peak and AUC readings are Claude Code 70.2 / 58.8, Cline 61.7 / 56.6, OpenCode 53.2 / 49.5; on final the same runs read Cline 61.7, Claude Code 55.3, OpenCode 53.2. The flip is driven by the (MiniMax-M2.5, Claude Code) run regressing 14.9 pp in its last 8 hours (peak 70.2 collapsing to final 55.3); the same (model, task) pair under OpenCode and Cline plateaus flat and does not regress. The post-peak collapse is therefore a property of the (model, harness) pair jointly, not of the model alone.

Evaluation configuration. Together, these results show why benchmark scores should be reported with the harness configuration: both the score level and the trajectory shape vary across the evaluated model–harness combinations. Appendix D.3 provides per-tier pass rates and supporting analysis.

3.5 Feedback Sensitivity Study

Alongside harness choice, the feedback available to the agent is part of the evaluation configuration. We examine this factor on Kimi-K2.5 with Redis, comparing the main-study run with one additional run that withholds checkpoint feedback. Both use Claude Code, the same progressive prompt requesting runnable incremental releases, the same external test suite, and a 24-hour budget. External evaluation continues in both conditions; only the availability of its feedback to the agent changes.

Table 5: Feedback sensitivity on Kimi-K2.5 $\times$ Redis with the progressive prompt fixed. Each row represents one 24-hour run; scores are in %.

Checkpoint feedback	Peak	Final	AUC
Anonymized (main-study run)	89.39	89.39	68.51
None	48.48	48.48	31.54

In this pair of runs, anonymized feedback accompanies a 40.91 pp higher peak and final score and a 36.97 pp higher AUC (Table 5). This single-case comparison illustrates the relevance of feedback availability when interpreting a trajectory.

3.6 Contamination Analysis

Our tasks ask agents to implement systems with public specifications and existing open-source implementations. We therefore examine code-level overlap between the generated implementations and selected reference implementations. We use Dolos (Maertens et al., 2022), which compares token n-gram fingerprints while abstracting away identifier names. We compare agents' final submissions against `aiohttp.web` and CPython's `http.server` for HTTP, `fakeredis` for Redis, and `mysql-mimic` for the database. The analysis covers 30 (agent, task) cells with retained source code; message queue is excluded because we did not identify a suitable widely-used pure-Python AMQP broker reference.

Across the 30 cells, maximum cross-corpus similarity has a mean of 0.086 and a median of 0.071, with all values below 0.24 (Figure 4). Manual inspection of the longest matching fragments in the two highest-similarity cells identified generic HTTP request-line parsing and SQL token-table boilerplate. Together, the similarity scores and inspected matches provide little evidence of substantial code-level copying from the selected reference implementations.

4 Related Work

Software engineering benchmarks for language models. Coding benchmarks span function-level synthesis (Chen et al., 2021; Jain et al., 2025), repository-level issue resolution (Jimenez et al., 2024; Deng et al., 2025), and repository construction (Ding et al., 2026; Zhao et al., 2025). Terminal-Bench (Merrill et al., 2026) evaluates sandboxed terminal tasks, while HCAST (Rein et al., 2025)

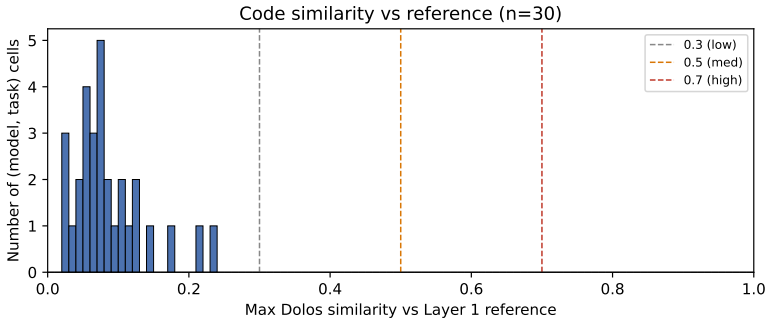


Figure 4: Maximum Dolos similarity across 30 (agent, task) cells. All values are below 0.3.

and RE-Bench (Wijk et al., 2025) extend evaluation to multi-hour tasks. ProgramBench (Yang et al., 2026) evaluates from-scratch program reconstruction under a 6-hour budget using behavioral tests. SlopCodeBench (Orlanski et al., 2026) evaluates correctness at successive specification checkpoints using author-written hidden tests and tracks trajectory-level code-quality signals. Hackathon-24h instead holds the specification fixed, samples the workspace every 30 minutes over a 24-hour run, and summarizes correctness measured by external industrial conformance suites using peak, final, and AUC. The distinction is therefore the trajectory axis and evaluation instrument, rather than the presence of intermediate evaluation alone. Table 1 compares benchmark properties.

Code-oriented LLMs and long-horizon coding agents. Code-oriented models such as Qwen3-Coder (Qwen Team, 2025a) target agentic coding, while general-purpose models including GLM-5 (GLM-5-Team et al., 2026), Kimi K2.5 (Kimi Team et al., 2026), and MiniMax M2.5 (MiniMax, 2026a) also undergo post-training targeting coding capabilities. Agent harnesses combine reasoning, tool use, and iterative self-correction (Yao et al., 2023; Shinn et al., 2023; Yang et al., 2024). We evaluate Claude Code (Anthropic, 2026b), OpenCode (Anomaly, 2026), and Cline (Cline Bot Inc., 2026). Harness design is itself an evaluation variable (OpenAI, 2026); our controlled comparison on HTTP server (Section 3.4) illustrates why scores characterize deployed agent systems rather than models alone.

5 Conclusion

Long-horizon agent evaluation has so far inherited a single-shot habit: report a number at the end of a long run and treat it as a model property. This paper argues that this habit hides what matters. Each 24-hour cell is one trajectory sampled from a deployed agent system; peak, final, and AUC give three complementary views of the same process; on one task, substituting the agent harness shifts outcomes by margins comparable to swapping the model. The harness comparison and fixed-prompt feedback case study (Section 3.5) highlight why trajectory scores should be reported with their evaluation configuration. A complementary Dolos analysis finds little evidence of substantial code-level copying from the selected reference implementations (Section 3.6). Process-sampled measurement makes questions such as when an agent peaks and what triggers a regression first-class observables, with little additional evaluation cost once periodic checkpointing is in place. We release Hackathon-24h and the harness at the website in the abstract.

The release is designed to be extended: adding a task requires a staged specification bundle, a task-specific init prompt, and an executable external suite partitioned into tiers, while snapshotting, isolated scoring, and the anonymized brief carry over unchanged, so new systems and new harnesses enter the same measurement frame. Beyond new tasks, the same process-sampled design enables several extensions. Expanding the harness sweep would generalize the current single harness-substitution study into a systematic evaluation of harness sensitivity across tasks and agent frameworks. Extending execution budgets to 48 or 72 hours would further reveal whether delayed-success trajectories eventually plateau or continue improving beyond the current 24-hour horizon.

AI USE STATEMENT

In this work, we used generative AI tools to assist with implementing experimental code and with exploratory analysis of the 24-hour trajectory data, including the identification and interpretation of candidate trajectory patterns. We additionally used generative AI tools to improve manuscript readability and identify potentially relevant literature. AI-assisted work was incorporated under author oversight, and the authors made the final decisions regarding the reported analyses, cited literature, and manuscript content.

ETHICS STATEMENT

Hackathon-24h evaluates coding agents on software implementation tasks derived from public specifications. Agent-generated code is executed in Docker-isolated environments with access restricted to designated workspaces. By exposing regressions and delayed progress during extended runs, the benchmark supports closer examination of agent reliability before deployment.

REPRODUCIBILITY STATEMENT

The anonymous repository linked in the abstract provides the benchmark framework, agent configurations, Docker setup, scoring scripts, and external asset references; prompts are documented in Appendix A.3. We plan to release the framework under an open-source license upon acceptance. Per-checkpoint scores for the 40 main-study trajectories are supplied as anonymous review supplements and will be available to researchers on request, but are not part of the public release. These scores support recomputation of peak, final, and AUC (Section 2.3). Generated system implementations, workspace snapshots, agent logs, and feedback streams are not released. A run requires Docker and model API access; API cost depends on the model and provider, and reproduction requires matching the external asset versions.

REFERENCES

Anomaly. OpenCode: The Open Source AI Coding Agent. `https://opencode.ai`, 2026.

Anthropic. Introducing claude opus 4.6. `https://www.anthropic.com/news/claude-opus-4-6`, 2026a.

Anthropic. Claude Code. `https://www.anthropic.com/product/claude-code`, 2026b.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-Coder-Next technical report. *arXiv preprint arXiv:2603.00729*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Cline Bot Inc. Cline: The Open Coding Agent. `https://cline.bot`, 2026.

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

Jingzhe Ding, Shengda Long, Changxin Pu, et al. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2026.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, Jie Sun, Yang Xiao, Sizhuo Zhou, Wenxiao Wu, Yiming Liu, Pengfei Liu, Yu Qiao, Shenglin Zhang, and Kaipeng Zhang. LongCLI-Bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *arXiv preprint arXiv:2602.14337*, 2026.

GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, et al. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring ai ability to complete long software tasks. *arXiv preprint arXiv:2503.14499*, 2026.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating LLMs on generating interactive and functional websites from scratch. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026.

Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning*, 38(4):1046–1061, 2022. doi: https://doi.org/10.1111/jcal.12662.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

MiniMax. MiniMax M2.5: Built for Real-World Productivity. `https://www.minimax.io/news/minimax-m25`, February 2026a.

MiniMax. MiniMax-M2.7: Early echoes of self-evolution. `https://www.minimax.io/news/minimax-m27-en`, 2026b.

OpenAI. Harness engineering: Leveraging Codex in an agent-first world. `https://openai.com/index/harness-engineering/`, 2026.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Frederic Sala, and Aws Albarghouthi. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Qwen Team. Qwen3-coder: Agentic coding in the world. `https://qwen.ai/blog?id=qwen3-coder`, July 2025a.

Qwen Team. Qwen3-max: Just scale it. `https://qwen.ai/blog?id=qwen3-max`, September 2025b.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents. https://qwen.ai/blog?id=qwen3.5, February 2026.

David Rein, Joel Becker, Amy Deng, Seraphina Nix, Chris Canal, Daniel O'Connel, Pip Arnott, Ryan Bloom, Thomas Broadley, Katharyn Garcia, Brian Goodrich, Max Hasin, Sami Jawhar, Megan Kinniment, Thomas Kwa, Aron Lajko, Nate Rush, Lucas Jun Koba Sato, Sydney Von Arx, Ben West, Lawrence Chan, and Elizabeth Barnes. HCAST: Human-calibrated autonomy software tasks. *arXiv preprint arXiv:2503.17354*, 2025.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2025.

Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua M Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-bench: Evaluating frontier AI r&d capabilities of language model agents against human experts. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=3rB0bVU6z6.

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

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546v1.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Z.ai. GLM-4.7: Advancing the Coding Capability. https://z.ai/blog/glm-4.7, December 2025.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025.

A Task and Protocol Details

A.1 Conformance Suites and Tier Ladder

The four tasks require Python implementations and are evaluated through external client protocols. Their thirteen external suites cover the per-tier behavior below.

Table 6: Per-tier scope of the four tasks. Suites listed are the conformance benchmarks scored at that tier.

Task	Tier	Suite	What it covers
Database	L0	Sysbench	CRUD on a single table, 8 OLTP workloads
	L1	Sysbench, TPC-C	Multi-table OLTP plus TPC-C schema and data load
	L2	Sysbench, TPC-C	Concurrent OLTP plus full TPC-C transactions
	L3	TPC-C, TPC-H	Heavier TPC-C plus 9 TPC-H analytical queries
Msg Queue	L0	pika	AMQP 0.9.1 protocol correctness
	L1	OMQ	Sustained message-delivery throughput
	L2	RabbitMQ PerfTest	Multi-publisher/consumer with large payloads
HTTP Server	L0	h1spec	HTTP/1.1 protocol conformance
	L1	CISPA	Curated HTTP/1.1 semantics subset
	L2	TFB	Plaintext and JSON endpoints
	L3	TFB	Database-backed query endpoints
	L4	TFB	Full application (Fortunes, Updates)
Redis KV	L0	TCL	RESP2 and minimal commands
	L1	TCL, redis-benchmark	Strings, expire, key management
	L2	TCL, redis-benchmark	Lists, hashes, sets, sorted sets
	L3	TCL	Persistence (RDB/AOF) and pub/sub
	L4	TCL, redis-benchmark	Transactions and pipelining
	L5	YCSB	Application workloads A–C
	L6	TCL, YCSB, memtier	Advanced commands, YCSB D–F, memtier stress

Table 6 expands the task summary in Table 2. The tiers group tests by functional scope and workload scale, yielding 3 tiers for message queue, 4 for database, 5 for HTTP, and 7 for Redis.

Table 7 summarizes the structural requirements of the four tasks alongside the per-tier coverage in Table 6.

Table 7: Structural requirements of the four implementation tasks.

Task	Structural Requirements
Database	MySQL wire-protocol compatibility, query execution, transactions, and persistent state across concurrent client sessions.
Message Queue	AMQP connection and channel state, message routing, acknowledgements, and delivery guarantees across publishers and consumers.
HTTP Server	Request parsing, response formatting, connection semantics, and application endpoints, including database-backed queries and updates.
Redis KV Store	RESP compatibility, command semantics across data types, expiration, persistence, transactions, pipelining, and pub/sub.

A.2 Per-Task Specification Materials

For each task, a curated set of protocol references is copied into the agent's workspace before the run begins. The agent reads from this directory throughout the 24-hour budget; no fetching from outside the sandbox is permitted. The materials below are the same standards that the conformance suites in Table 2 encode, so the agent and the judge are working from a shared specification.

Database. MySQL Reference Manual (markdown export), TPC-C specification, TPC-H specification.

HTTP Server. RFC 9110 (HTTP semantics), RFC 9111 (HTTP caching), RFC 9112 (HTTP/1.1 messaging), RFC 6265 (cookies), RFC 6585 (additional status codes), RFC 7617 (Basic authentication).

Message Queue. AMQP 0.9.1 specification.

Redis KV Store. RESP protocol specification, Redis 7.2 commands reference, plus per-topic notes covering `SET`, `SCAN`, `CONFIG GET`, `INFO`, `CLIENT LIST`, `OBJECT ENCODING`, persistence (RDB/AOF), pipelining, pub/sub, and transactions.

A.3 AGENT PROMPTS

Each system type uses two prompt templates: an *init prompt* (sent once at the start) and an *iteration prompt* (sent at each subsequent iteration with benchmark feedback injected). Below we show the shared structure and key excerpts.

Init prompt structure (all system types). The init prompt follows a common template:

1. **Task description:** "Build a custom [system] implementation."
2. **Execution strategy:** Deliver a runnable minimum first; leave advanced features for later iterations. Do not try to perfect everything in one shot.
3. **Protocol requirements:** Detailed specification of the target protocol (HTTP/1.1 per RFC 9110/9112, AMQP 0.9.1, MySQL wire protocol, or RESP2 for Redis).
4. **Routing/endpoints:** Required endpoints and their expected behavior.
5. **Operational requirements:** Read host/port from environment, create `start.sh`, handle graceful shutdown.
6. **PLAN.md requirement:** Maintain a plan file with state-machine format (TODO/IN_PROGRESS/DONE/BLOCKED).

Iteration prompt structure (all system types). The iteration prompt enforces a single-task discipline, interleaved with brief-driven plan reconciliation:

1. Read the latest anonymized brief at `./last_brief/last_brief_i.txt`
2. Reconcile `PLAN.md` against the brief: for every failing behavior or category named in the brief, locate the most relevant PLAN task; if it is currently marked DONE, reopen it as TODO so regressed capabilities re-enter the queue
3. Select **exactly one** highest-priority TODO item
4. Move it to IN_PROGRESS
5. Implement minimally
6. Run local acceptance tests
7. Mark DONE or BLOCKED
8. Keep the system runnable after this iteration
9. Maintain an "Iteration Log" with: iteration ID, selected task, files changed, tests executed, result summary, follow-up tasks

The brief is embedded in the iteration prompt verbatim and is presented to the agent as a lagging, directional hint rather than ground truth. Under the anonymized condition it reports overall and per-tier pass rates, category-level error summaries, generic suite labels, and opaque identifiers for failed checks; test code remains outside the agent's workspace. When the brief and a local test executed in the same iteration disagree, the prompt directs the agent to trust the local test and record the discrepancy in the iteration log. The full text of all 8 prompt templates (4 system types $\times$ 2 phases) is provided in the supplementary artifact under `prompts/`.

Sample brief. The following is one brief verbatim, drawn at the hour-4 checkpoint of the GLM-5 cohort cell on HTTP server (Table 3). Suite identifiers are folded into generic category labels (`spec`, `compliance`, `workload`); failed checks are summarized by counts and may also be listed using opaque identifiers (`rule_432b6d21`, `case_0430df3b`). One error line is wrapped for column width; the brief on disk has each entry on a single line. The first field decomposes as `{task}_{harness}_{model}_{timestamp}_{iteration}`.

```
version_id: http_server_claude_glm-5_20260406_220252__8
agent: Claude Code
snapshot_time: 2026-04-07T02:02:53
test_start:    2026-04-07T02:02:53
```

```
test_end:      2026-04-07T02:03:18
overall: Wrong Answer - spec: 8 workload(s) failed
score: 72.3% (34/47)

--- tiers ---
L0: Wrong Answer (25/33) -- spec: 8 workload(s) failed
  spec: 25/33 passed
L1: Wrong Answer (7/8) -- compliance: 1 rule(s) failed
  compliance: 7/8 passed [FAIL: rule_432b6d21]
L2: Accepted (2/2)
  workload: 2/2 passed
L3: Wrong Answer (0/2) -- workload: 2 test(s) failed
    (first error: Connection error: HTTP Error 404: Not Found)
  workload: 0/2 passed [FAIL: case_0430df3b, case_0254d7ea]
L4: Wrong Answer (0/2) -- workload: 2 test(s) failed
    (first error: Connection error: HTTP Error 404: Not Found)
  workload: 0/2 passed [FAIL: case_cb3498b7, case_85ef1b70]
```

B Extended Main-Table Results

B.1 Trajectory Atlas

Figure 5 shows all 40 main-study trajectories, with models in rows and tasks in columns. GLM-5 reaches its message-queue peak early and subsequently regresses, while its database score rises late in the run. Kimi-K2.5 makes early progress on HTTP and Redis but remains at zero for more than 14 hours on message queue and database. Six of the ten database runs remain at zero throughout, and none exceeds 40%; nine Redis runs finish above 70%. These trajectories provide the time-resolved view underlying the summaries in Table 3.

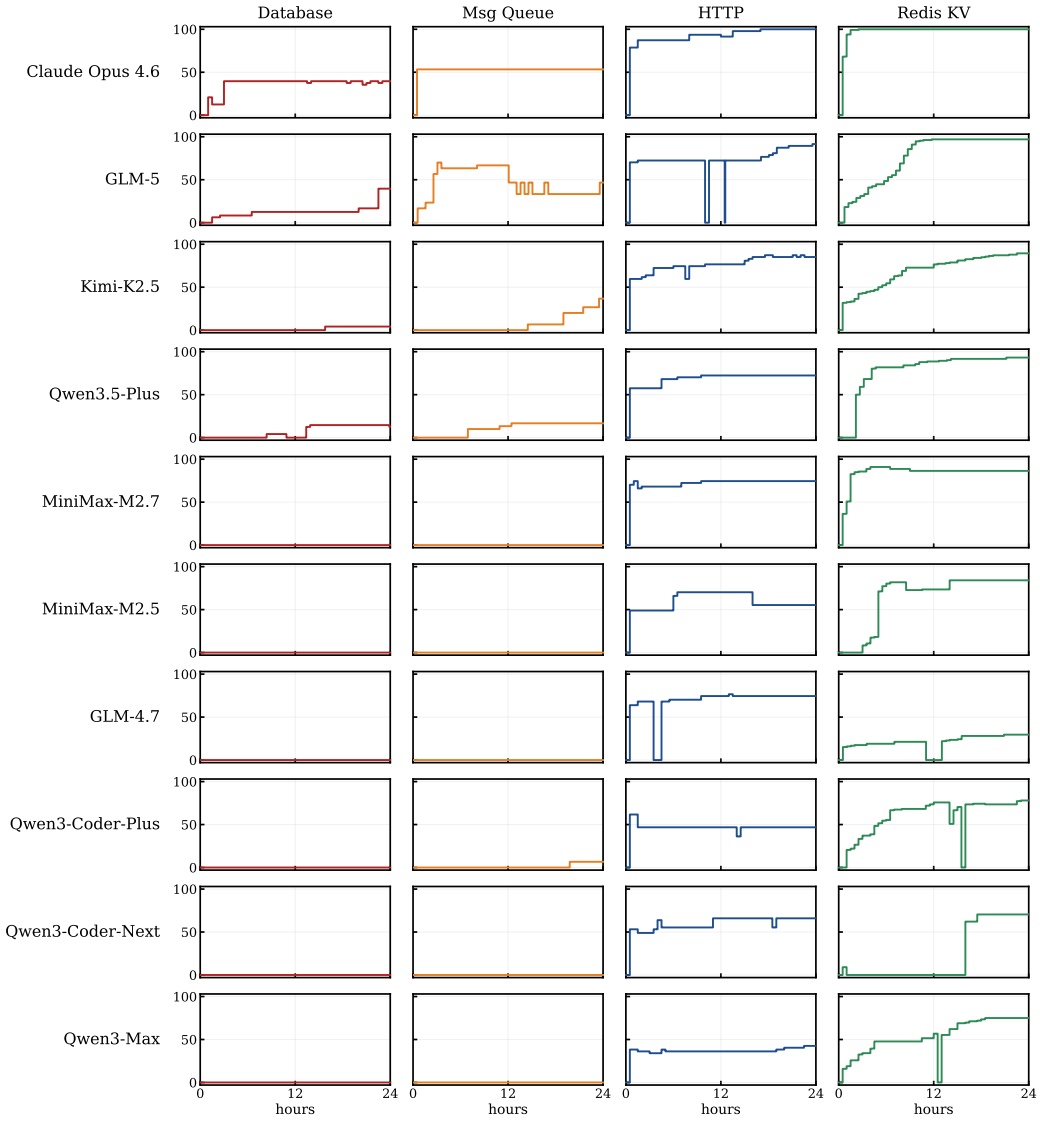


Figure 5: Score trajectory of all 40 main-table cells over the 24-hour budget. Rows: 10 models in the order of Table 3. Columns: 4 tasks. Each subplot shows checkpoint scores as a step function. Colors identify the four tasks (Database, Message Queue, HTTP, Redis KV).

B.2 Test-Set View of the Demonstration Cohort

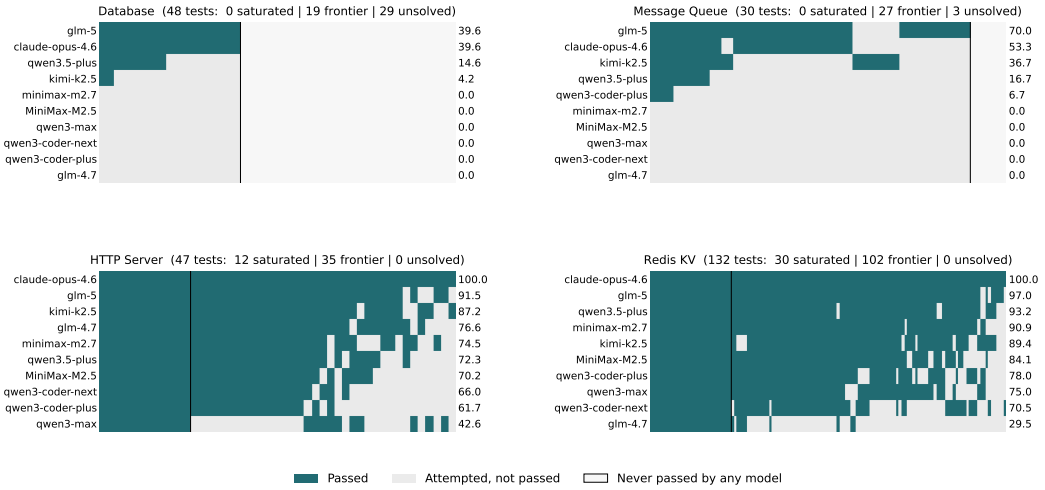


Figure 6: Test-set view of the demonstration cohort. For each task, rows are the 10 evaluated models (sorted by peak), columns are the task's test cases (sorted by descending count of models passing). Columns split into three regions: *saturated* (passed by all 10), *frontier* (passed by 1–9), *unsolved* (passed by 0). The number on the right of each row is the model's peak score (%). Database and Message Queue have no saturated tests and a long unsolved tail; HTTP and Redis have saturated heads and per-model variation in the frontier middle.

Figure 6 unpacks each cell of Table 3 into the set of tests passed at any checkpoint during the run. Database and message queue have no tests passed by all ten models, while 29 of 48 database tests and 3 of 30 message-queue tests are passed by none. HTTP and Redis have 12 of 47 and 30 of 132 tests, respectively, passed by all ten models; every remaining test is passed by at least one model.

On HTTP, GLM-5 and Kimi-K2.5 pass partially distinct sets of tests over their runs: GLM-5 passes 43 distinct tests and Kimi-K2.5 passes 42, with 39 shared, four unique to GLM-5, and three unique to Kimi-K2.5 (Jaccard similarity 0.85). Kimi-K2.5's 42-test union exceeds its single-checkpoint peak count of 41, illustrating how the set of passing tests changes during a run.

B.3 Per-Run Effort Profile

Each cell runs unattended for 24 hours on a single Docker-capable host with no GPU required. The agent loop accumulates iterations (one PLAN.md TODO cycle each), LLM steps (one assistant turn each, typically issuing one or more tool calls), tool invocations, and tokens (input, output, cache-read, and cache-creation tokens reported by the harness, summed across all turns). Per-cell volumes vary widely across models on the same wall-clock budget: a single cell can consume between 13 and 884 plan iterations, 369 to 5,805 LLM steps, 374 to 6,946 tool calls, and 53M to 33B tokens (Table 8). The median cell does 96 iterations, 2,313 steps, 3,008 tool calls, and 384M tokens.

Effort under a fixed budget. Median resource use varies across models despite the shared 24-hour budget (Table 8). These counts describe the iterations, LLM steps, tool calls, and tokens consumed by each deployed agent system; they do not measure useful progress per action.

Wall-clock budget. We use elapsed time because it gives every run the same development window while allowing iteration and tool-use rates to vary. Table 8 summarizes resource consumption under this shared budget.

Table 8: Per-model effort profile on the 10-model × 4-task demonstration cohort. **Med. Iters / Steps / Tool Calls / Tokens** are medians across the model's four (model, task) cells. The token column is the sum of input, output, cache-read, and cache-creation tokens reported by the harness; these four fields are disjoint, so the sum equals total tokens billed by the provider for the run. Providers without prompt caching report zero in the cache-read and cache-creation columns, with all repeated context counted as fresh input. Rows in the order of Table 3.

Model	Med. Iters	Med. Steps	Med. Tool Calls	Med. Tokens (M)
Claude Opus 4.6	48	1,964	3,532	291
GLM-5	206	3,799	3,804	441
Kimi-K2.5	114	1,428	1,731	177
Qwen3.5-Plus	37	2,032	2,671	310
MiniMax-M2.7	568	4,898	4,994	544
MiniMax-M2.5	129	1,454	1,566	160
GLM-4.7	22	822	1,026	122
Qwen3-Coder-Plus	182	4,132	4,300	485
Qwen3-Coder-Next	82	2,228	2,789	510
Qwen3-Max	158	2,580	5,264	11,850

C Trajectory Phenomena: Case Studies

This section examines four individual runs: one peak-zero case, two hidden regressions, and one late breakthrough. Trajectory data come from checkpoint scores, and agent actions from iteration logs.

Case 1 (Confident hallucination): MiniMax-M2.5 on Message Queue. **Setting:** MiniMax-M2.5 building an AMQP 0.9.1 broker from scratch over 24 hours. **Trajectory:** 0% across all 47 test cycles (hours 0.5–23.5). Not a single test ever passed. **What happened:** The agent completed 85 improvement iterations and marked 28 plan tasks "DONE" in its `PLAN.md`. It built an elaborate broker with connection handling, channel management, and queue declaration. However, a frame-size calculation error (expecting 89 MB when the AMQP handshake sends 22 bytes) caused the initial connection to fail before any protocol exchange could occur. The agent never diagnosed this root cause, instead attributing repeated failures to "the test framework using an incompatible protocol version." **Why this matters:** The gap between the agent's self-assessment (28 tasks completed, extensive code written) and external reality (0% conformance) is maximal. No outcome-only benchmark can distinguish "confidently wrong for 24 hours" from "never started."

Additional log checks. We inspected four of the eleven peak-zero runs: this case, MiniMax-M2.5 and Qwen3-Coder-Plus on database, and GLM-4.7 on message queue. All four combined positive self-assessments with zero-pass feedback and an identifiable implementation bug; observed responses included marking tasks complete, reframing failures as stability issues, and attributing errors to the evaluation environment.

Case 2 (Catastrophic regression): GLM-5 on Message Queue. **Setting:** GLM-5 building an AMQP 0.9.1 broker. Peak 70.0%, final 46.7% (23.3 pp gap). **Trajectory excerpt:** 0.6h: 16.7% → 2.6h: 56.7% → 3.1h: **70.0%** (peak) → 3.6h–7.6h: 63.3% → 8.1h–11.6h: 66.7% → 12.1h: 46.7% (20 pp drop) → [oscillates 33–47% from hour 13] → 23.6h: 46.7%. **What happened:** GLM-5 built a functional broker reaching 70% in just 3 hours, then spent the remaining 21 hours making unguarded modifications that repeatedly broke existing functionality. Six distinct regression events occurred, the largest being a 20 pp single-step drop. The agent attempted to add advanced features (consumer priorities, message TTL) without protecting the working baseline. **Why this matters:** A peak-only benchmark would report 70.0% and would record GLM-5 as strong on this task. The 23.3 pp gap records a loss of previously achieved correctness that the peak score alone does not reveal.

Case 3 (Unnoticed self-destruction): MiniMax-M2.5 on HTTP Server. **Setting:** MiniMax-M2.5 building an HTTP/1.1 server. Peak 70.2%, final 55.3% (14.9 pp gap). **Trajectory excerpt:** 0.5h: 48.9% → 6.5h: **70.2%** (peak) → [holds 70.2% through hour 15.5] → 16.0h: 55.3% → [holds 55.3% for 16 consecutive cycles through hour 23.5]. **What happened:** After peaking at 70.2%, the agent entered a verification loop, reporting "all endpoints verified working" while the score had

already dropped to 55.3%. The agent's verification only checked HTTP 200 status codes, never testing the compliance behaviors (chunked encoding, connection persistence, content-length accuracy) that the benchmark actually evaluated. The score remained at 55.3% for the final 16 checkpoints, spanning hours 16.0–23.5. **Why this matters:** The agent's self-assessment was consistently positive during a period of degraded performance. Unlike Catastrophic regression (where the score actively oscillates), here the damage happened once and then froze; the agent did not even attempt recovery because it did not know it had regressed.

Case 4 (Late breakthrough): Kimi-K2.5 on Message Queue. **Setting:** Kimi-K2.5 building an AMQP 0.9.1 broker. Peak = final = 36.7%. **Trajectory excerpt:** 0% from run start through hour 14.0 $\rightarrow$ 14.5h: 6.7% (first non-zero) $\rightarrow$ [held 6.7% for 9 consecutive checkpoints through hour 18.5] $\rightarrow$ 19.0h: 20.0% $\rightarrow$ 21.5h: 26.7% $\rightarrow$ 23.5h: **36.7%**. **What happened:** The agent spent 14 hours at 0%, methodically building AMQP frame parsing, connection negotiation, and channel management. The first non-zero score, 6.7%, appeared at hour 14.5. Progress then accumulated in three discrete steps over the remaining 9.5 hours (6.7% $\rightarrow$ 20.0% $\rightarrow$ 26.7% $\rightarrow$ 36.7%), demonstrating that the foundation work had been productive even though it produced no benchmark signal until very late. **Why this matters:** A 12-hour benchmark would report total failure (0%); even an 18-hour benchmark would have caught only the 6.7% plateau and missed the climb to 36.7%. The full 24-hour budget reveals that the agent was making genuine architectural progress during the zero-score period, challenging the assumption that early checkpoint performance predicts final capability.

D Sensitivity and Robustness

D.1 Checkpoint Cadence Sensitivity

No checkpoint interval follows from first principles. We use 30 minutes as a runtime hyperparameter chosen to match the timescale of agent progress. Finer sampling increases evaluation compute and can cause conformance-suite executions to overlap, whereas coarser sampling increasingly obscures trajectory changes. To quantify the latter effect, we post-hoc resample all 40 main-study trajectories at 1-, 2-, 3-, 4-, and 6-hour cadences and recompute step-function AUC using the definition in Section 2.3. Table 9 reports the mean AUC decrease relative to the 30-minute baseline.

Table 9: Checkpoint cadence sensitivity across the 40 main-study trajectories. Values are mean AUC loss in percentage points relative to the 30-minute baseline.

Cadence	**1 h**	**2 h**	**3 h**	**4 h**	**6 h**
Mean AUC loss (pp)	1.06	2.46	3.56	5.40	7.78

The loss rises monotonically from 1.06 pp at 1 hour to 7.78 pp at 6 hours. This analysis characterizes the resolution–evaluation-cost trade-off of the chosen operating point; it does not establish 30 minutes as theoretically optimal.

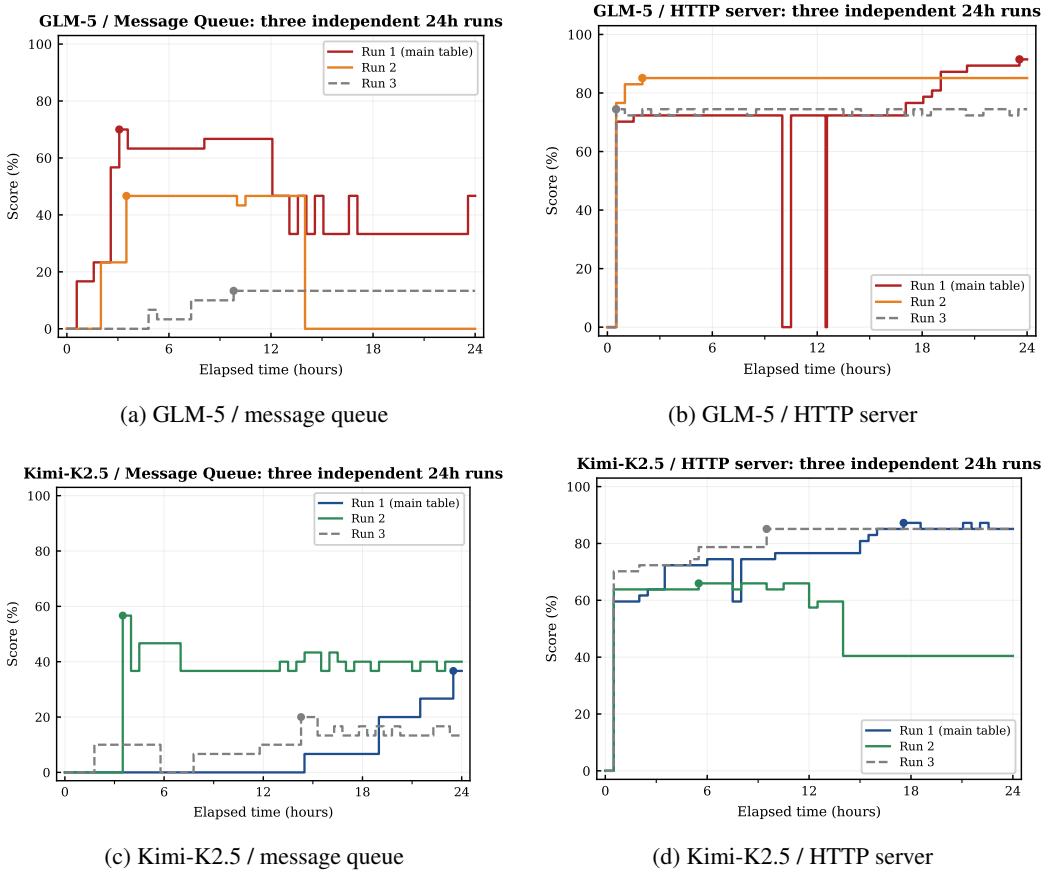
D.2 Replicated-Run Analysis

The main results table reports a single 24-hour run per (model, task) cell. To check whether peak, final, and AUC at n=1 reflect a model's typical behavior on the cell or one draw from a wider distribution, we added two independent runs for each of four main-table cells, giving three runs per cell: GLM-5 and Kimi-K2.5 on message queue, and the same two models on HTTP server (Table 10, Figure 7). Each replicate reuses the configuration of its main-table cell exactly (Claude Code harness, anonymized brief, the task's full external suite, 24-hour budget, identical prompt).

Replicate variation. Peak exceeds final in four of six message-queue runs and two of six HTTP runs. Within GLM-5/message queue, peaks range from 13.3% to 70.0%, and AUC ranges from 9.34% to 45.46%. Across the four cells, sample SDs span 6.1–28.5 pp (Table 11). The trajectories in Figure 7 show how these differences emerge over time. Characterizing run-to-run variation more fully remains future work.

Table 10: Replicated-Run Analysis on message queue and HTTP server: each cell shows **peak / final / AUC** (%) for one 24-hour run. Run 1 in each row is the value reported in Table 3; Runs 2 and 3 are independent re-launches of the identical configuration. 30 tests for message queue, 47 tests for HTTP server.

Model	Task	Run 1 (main table)	Run 2	Run 3
GLM-5	message queue	70.00 / 46.70 / 45.46	46.67 / 0.00 / 21.81	13.33 / 13.33 / 9.34
GLM-5	HTTP server	91.49 / 91.49 / 72.90	85.11 / 85.11 / 83.07	74.47 / 74.47 / 72.25
Kimi-K2.5	message queue	36.67 / 36.67 / 6.33	56.67 / 40.00 / 34.10	20.00 / 13.33 / 9.91
Kimi-K2.5	HTTP server	87.23 / 85.11 / 75.22	65.96 / 40.43 / 52.79	85.11 / 85.11 / 79.52



(a) GLM-5 / message queue

(b) GLM-5 / HTTP server

(c) Kimi-K2.5 / message queue

(d) Kimi-K2.5 / HTTP server

Figure 7: Three independent 24-hour runs each of GLM-5 (top row) and Kimi-K2.5 (bottom row) on message queue (left column) and HTTP server (right column). Run 1 in each panel is the cell reported in Table 3. Peak strictly exceeds final in 6 of 12 replicates across the four cells, showing score-level regression on both tasks.

Table 11: Within-cell dispersion over the three replicates in Table 10. Each entry reports **mean / range / sample SD**: means are in %, and ranges and sample SDs are in percentage points. With n=3 these describe the observed spread of this case study, not a confidence interval.

Model	Task	Peak	Final	AUC
GLM-5	message queue	43.3 / 56.7 / 28.5	20.0 / 46.7 / 24.1	25.5 / 36.1 / 18.3
GLM-5	HTTP server	83.7 / 17.0 / 8.6	83.7 / 17.0 / 8.6	76.1 / 10.8 / 6.1
Kimi-K2.5	message queue	37.8 / 36.7 / 18.4	30.0 / 26.7 / 14.5	16.8 / 27.8 / 15.1
Kimi-K2.5	HTTP server	79.4 / 21.3 / 11.7	70.2 / 44.7 / 25.8	69.2 / 26.7 / 14.4

D.3 HARNESS SENSITIVITY: PER-TIER PROFILES

Table 12 expands the Kimi-K2.5 HTTP results in Section 3.4. OpenCode is the only harness to pass any L4 tests (50%) and reaches 100% on L1–L3; Claude Code has the highest L0 pass rate (90.9%). Cline completes 40 plan iterations, compared with 188 for Claude Code and 180 for OpenCode.

Table 12: Harness sensitivity: per-tier final pass rates, AUC, and plan iterations for Kimi-K2.5 × HTTP server under a 24-hour budget. **Iters** counts plan iterations (one `PLAN.md` TODO item each).

	Final Per-Tier Pass Rate (%)						
Harness	**L0**	**L1**	**L2**	**L3**	**L4**	**AUC**	**Iters**
Claude Code	90.9	87.5	100.0	50.0	0.0	75.2	188
OpenCode	75.8	100.0	100.0	100.0	50.0	69.7	180
Cline	72.7	87.5	100.0	50.0	0.0	70.2	40