RETHINKING VERIFICATION IN AGENT HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent harnesses enable large language models to act through tools, observe the results, and iteratively work toward a task goal. Verification checks and revises agents' work to improve output quality. Yet there is limited guidance on how harnesses should organize this effort. We study verification as an axis of test-time scaling by examining when to intervene, who checks the work, and how checking leads to revision. Our framework spans four layers of the agent lifecycle: initial instructions, mid-turn reminders, completion gates, and independent review. We compare 17 designs across seven frontier coding models on Terminal-Bench (TB) and SWE-bench Verified (SWE), and introduce VisualBench for expert evaluation of visual and interactive artifacts. Initial instructions and execution-time reminders yield modest, model-dependent average gains on the coding benchmarks, without substantially increasing average output token use. Completion gates can yield larger gains but may substantially increase cost. Independent review by a critic offers a more favorable accuracy–cost tradeoff, improving average accuracy by 5.9–9.5 percentage points on TB and 3.9–6.6 on SWE at 1.13–1.25× and 1.00–1.04× baseline cost, respectively. On VisualBench, independent review achieves net preferences of 13.2–23.8 points over the baseline across its four designs, exceeding the evaluated L0–L2 designs. Additional review rounds or independent critics further improve average TB accuracy by up to 3.5 percentage points. Together, these results provide empirical guidance for verification design in agent harnesses and support independent review as a practical axis of test-time scaling.

1 INTRODUCTION

Coding agents built on large language models (LLMs) tackle long-horizon software engineering tasks, including resolving repository-level issues across large codebases (Yang et al., 2024; Anthropic, 2026a; OpenAI, 2026; Google, 2026). These agents run inside harnesses that let the model act through tools, observe the results, and iterate toward a task goal (Yao et al., 2022; Yang et al., 2024). Beyond exposing tools, the harness specifies the agent's instructions, controls intervention timing, and coordinates verification before accepting its work (Bolin, 2026; Lopopolo, 2026). These choices allocate computation to checking and revising the agent's work, making verification an important part of harness design (Lopopolo, 2026).

Prior work develops learned verifiers for selecting solutions (Cobbe et al., 2021; Lightman et al., 2023), self-correction through critiques and reflections (Madaan et al., 2023; Shinn et al., 2023), and tool-assisted checks that return execution evidence for repair (Gou et al., 2023; Chen et al., 2023; Huang et al., 2023a). In practice, verification in agent harnesses is often shaped by engineering heuristics and iterative experience: Claude Code provides hooks for completion checks (Anthropic, 2026b), while Codex uses self-review and iteration (Lopopolo, 2026). Yet systematic evidence remains limited on how to design and scale verification, when a design improves performance, and whether its gains justify the additional computation. Without this guidance, current designs may be suboptimal for a given model or task.

In this paper, we study verification as an axis of test-time scaling in agent harnesses. We organize the design space around three questions: when to intervene, who should check the work, and how findings should lead to revision. Following the agent lifecycle, **L0: Initial instructions** and **L1: Execution-time reminders** shape what the agent is told before and during execution to guide its own checking; **L2: Completion gates** reopen its work for self-checking; and **L3: Independent review** assigns a separate critic to check the work and return findings for repair before it is accepted.

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.

This framework lets us compare prompting the agent to verify, extending its self-checking, and introducing an independent checking perspective. Within these layers, we vary message role, intervention timing, critic context, and review strategy, then scale independent review through additional rounds or critics.

We compare 17 verification designs on Terminal-Bench (TB) (Merrill et al., 2026) and SWE-bench Verified (SWE) (Jimenez et al., 2023; OpenAI, 2024) tasks using seven frontier coding models: Qwen3.7-Max (Qwen Team, 2026a), Qwen3.8-Max (Qwen Team, 2026b), GLM-5.2 (Z.ai, 2026), DeepSeek V4-Pro (DeepSeek-AI et al., 2026), DeepSeek V4-Flash (DeepSeek-AI et al., 2026), Kimi-K2.7-Code (Moonshot AI, 2026), and MiniMax-M2.5 (MiniMax, 2026). We also introduce an internal VisualBench of 18 visual and interactive tasks, evaluated through blinded expert comparisons of functionality and visual quality. L0 initial instructions and L1 execution-time reminders yield modest, model-dependent average gains on the coding benchmarks, using 0.90–$1.00\times$ and 0.75–$0.93\times$ baseline output tokens, respectively. L2 completion gates can yield larger gains but may substantially increase cost: rejection-based designs cost 1.60–$1.76\times$ baseline on TB and 1.48–$1.56\times$ on SWE. L3 independent review by a separate critic offers a more favorable accuracy–cost tradeoff, improving average accuracy by 5.9–9.5 percentage points on TB and 3.9–6.6 on SWE at 1.13–$1.25\times$ and 1.00–$1.04\times$ baseline cost, respectively, including critic review and agent revision. On VisualBench, all four independent-review designs outperform the evaluated L0–L2 designs, with net preferences of $+13.2$–$+23.8$ points. Changes-based review outperforms trajectory-based review both with and without required visual inspection. Additional review rounds or independent critics further improve average TB accuracy by up to 3.5 percentage points. Together, these results provide empirical guidance for verification design in agent harnesses and support independent review as a practical axis of test-time scaling, enabling users to configure verification effort for different tasks to improve agent performance.

2 Related work

Verification. There is a line of works exploring verification for assessing candidates, guiding self-correction, and checking outputs against external evidence. Learned verifiers score complete solutions or intermediate reasoning steps to guide candidate selection (Cobbe et al., 2021; Lightman et al., 2023; Wang et al., 2024a). Code-focused assessment ranges from trained critics that highlight bugs (McAleese et al., 2024) to review protocols that combine agent judgments or introduce structured disagreement (Tang et al., 2024; Qiu & Gill, 2026). Self-correction methods use natural-language critiques and reflections to revise outputs or improve subsequent attempts (Madaan et al., 2023; Shinn et al., 2023). Training-based approaches further teach models to improve across successive attempts through fine-tuning or multi-turn reinforcement learning (Qu et al., 2024; Kumar et al., 2024). Studies of reasoning correction and code repair examine the limits of self-generated feedback and show that reported gains depend on feedback quality and the cost of additional attempts (Huang et al., 2023b; Olausson et al., 2023; Kamoi et al., 2024). Tool-assisted verification grounds error detection in evidence from code execution and external tools (Chen et al., 2023; Gou et al., 2023), while dedicated test-generation and execution agents return test feedback to guide code revision (Huang et al., 2023a). Together, these strands provide the assessment and feedback mechanisms from which harness-level verification designs can be built.

Agent harnesses. Agent harnesses determine how models interact with tools and environments by organizing instructions, tool interfaces, execution flow, and context. Prior work studies interleaved reasoning and action (Yao et al., 2022), agent–computer interfaces (Yang et al., 2024), execution platforms (Wang et al., 2024b), and structured repair workflows (Xia et al., 2024). These studies examine how the surrounding execution design shapes agent behavior. Harness design also governs how computation and information are used across attempts, including agent sampling and selection (Zhu et al., 2025) and reuse of coding rollouts through structured summaries (Kim et al., 2026). Agent harnesses have become an essential part of modern coding agents, including Claude Code (Anthropic, 2026a), Codex CLI (OpenAI, 2026), and Gemini CLI (Google, 2026).

These works establish useful assessment mechanisms and agent workflows, but offer limited comparative guidance on how a harness should organize checking and revision. We systematically study verification design in agent harnesses across multiple models and benchmarks, examining when to

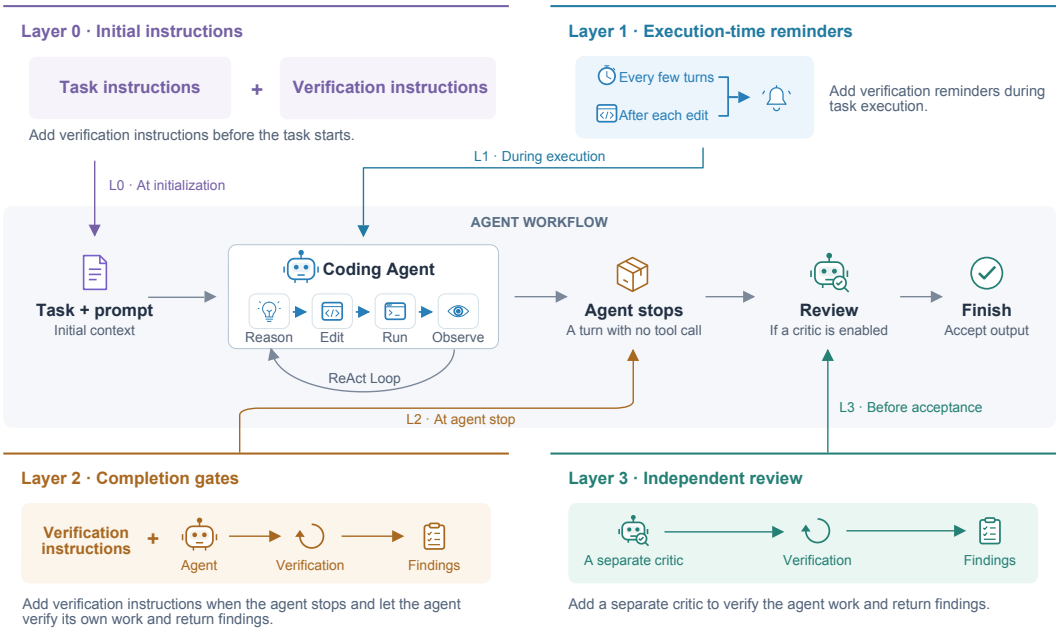


Figure 1: Framework overview.

Table 1: Verification layers and their roles in the execution and revision process.

Layer	When	Who checks	How execution continues
L0	At initialization	Agent	Use initial verification instructions to guide checking and revision.
L1	During execution	Agent	Check recent changes and revise ongoing work after a reminder.
L2	At completion	Agent	Resume work for self-checking and revision after a check request or rejected completion.
L3	Before acceptance	Separate critic	Return critic findings to guide agent revision.

intervene, who performs the checks, how feedback leads to revision, and how additional verification effort affects performance and cost.

3 Verification Designs

In this section, we introduce verification designs by following the agent lifecycle, from receiving a task to delivering a result as shown in Figure 1. Table 1 summarizes when each layer intervenes, who checks the work, and how execution continues.

Before execution begins, the agent receives the task and the instructions that guide its work. At this stage, **L0: Initial instructions** introduces verification guidance into the initial prompt. We vary the message role of the verification instructions: *System* or *User*. In the *System* variant, we append it to the system prompt as standing guidance. Because users do not always include verification instructions in their requests, we introduce a *User* variant that appends the same prompt to the initial user request. Both variants ask the agent to run the deliverable through its intended interface, inspect the output, and fix observed defects. The L0 verification instructions are in Appendix E.1. For visual or interactive artifacts, the instruction also asks the agent to render the result and exercise its core interactions. The L0 verification instructions for visual tasks in our VisualBench are in Appendix E.1.

As execution proceeds, changes to the codebase accumulate across turns. In our experiment, we find that only 6.2–13.6% of detected editing turns are followed by a recognized checking command

before the next editing turn or the end of the run, as reported in Appendix Table 7. We therefore introduce **L1: Execution-time reminders** to encourage the agent to check recent changes as it works, creating opportunities to identify and correct errors before further changes accumulate. These reminders arrive after tool results and before the next model call, so the agent can check its recent changes before continuing. We evaluate three designs. *Periodic reminder (System)* sends a reminder a few turns after the edit. *After-edit reminder (System)* sends the same reminder after a turn that changes a file. These two designs use a marker identifying the message as an automated self-verification checkpoint. *After-edit reminder (User)* uses the same edit-based trigger but delivers the reminder as a user message without that marker. Thus, the first comparison varies the trigger, while the second varies the message role and the presence of the automated marker. The L1 verification instructions are in Appendix E.2.

When the agent attempts to finish, **L2: Completion gates** reopen the agent's work for checking and revision. To examine how the requested action guides subsequent checking and revision, we vary whether the gate asks the agent to inspect changes or run checks, or rejects completion and asks it to continue working. We vary the message role (*System* or *User*) and whether user messages carry an automated marker to study how these choices affect the agent's response. We also vary intervention timing to test whether reserving time for checking and repair improves outcomes. The L2 prompts and triggering rules are in Appendix E.3.

Before final acceptance, the harness can also assign verification to a separate model, which we call the *critic*. This introduces **L3: Independent review**: the critic starts in a fresh context and assesses the agent's deliverable. The critic receives the task specification and access to the live sandbox, where it can run its own checks. The critic returns its findings to the agent for revision. We vary the critic's context and whether its checks are guided by task requirements alone (*Task-first*) or also by the agent's assumptions and claims (*Claim-guided*). The *Changes* condition provides the critic with the task and the agent's changes. The *Trajectory* condition provides the task, changes, and the agent's execution trajectory. The critic instructions are in Appendix E.4.

4 Experimental setup

Models. We evaluate seven coding models through Alibaba Cloud Model Studio (DashScope): Qwen3.7-Max (Qwen Team, 2026a), Qwen3.8-Max (Qwen Team, 2026b), GLM-5.2 (Z.ai, 2026), DeepSeek V4-Pro (DeepSeek-AI et al., 2026), DeepSeek V4-Flash (DeepSeek-AI et al., 2026), Kimi-K2.7-Code (Moonshot AI, 2026), and MiniMax-M2.5 (MiniMax, 2026). Qwen3.7-Plus serves as the independent critic for every coding model.

Benchmarks. We evaluate on task subsets from Terminal-Bench 2.1 (TB) (Merrill et al., 2026) and SWE-bench Verified (SWE) (Jimenez et al., 2023; OpenAI, 2024), both with automatic graders. Task selection aims to cover a range of difficulties, using baseline success rates for TB and human time-to-fix tiers followed by baseline evaluation for SWE. Appendix A.1 provides task counts and the selection procedure. Each coding condition is configured for three trials per model and task.

VisualBench contains 18 visual and interactive tasks. Each provides a task description and any supporting files, and asks the agent to produce an artifact, such as a playable game, an interactive 3D configurator, or a sales data explorer. For models without native visual capabilities, a `look_at` tool renders the artifact and returns a description from Qwen3.7-Plus. Raters from an internal company data team compare artifacts from a verification condition and the baseline for the same task and model, with conditions hidden and sides randomized. They receive the task brief and short interaction instructions, and judge functionality and visual quality against the task requirements. Outcomes are recorded as wins, losses, or ties; cases where both artifacts fail are recorded separately. We evaluate one representative design from each of L0–L2 and four L3 designs, with two trials configured per task, model, and condition. Appendix C provides the full descriptions of all 18 tasks, expert instructions, and screenshots of the evaluation interface.

Environment. All experiments run in E2B cloud sandboxes (E2B, 2026), orchestrated with Harbor (Harbor Framework Team, 2026). Each trial starts in a fresh sandbox containing the task files, dependencies, and execution tools. The agent interacts with this environment by reading and editing files, running commands, and inspecting outputs. For independent review, the critic accesses the same workspace to examine the deliverable and execute its own checks. The critic does not have

Table 2: Accuracy changes (pp) on Terminal-Bench across verification methods and models.

Method	Variant	Qwen 3.7-Max	Qwen 3.8-Max	GLM 5.2	DeepSeek V4-Pro	DeepSeek V4-Flash	Kimi K2.7 Code	MiniMax M2.5	Average
L0 Initial instructions									
1 Verification instruction	System	+7.1	+8.0	+6.0	0.0	+4.8	−2.4	0.0	+3.3
2 Verification instruction	User	+3.6	+4.0	+6.0	+1.2	−1.2	−8.3	+3.6	+1.2
L1 Execution-time reminders									
3 Periodic reminder	System	+10.7	+12.0	+1.2	+6.0	−1.2	−1.2	0.0	+3.8
4 After-edit reminder	System	+7.1	+12.0	+2.4	0.0	+1.2	−2.4	+4.8	+3.5
5 After-edit reminder	User	+14.3	**+13.3**	+4.8	+3.6	+4.8	+1.2	−3.6	+5.4
L2 Completion gates									
6 Inspect changes	Automated	+3.6	+10.7	−3.6	+4.8	−8.3	−4.8	+3.6	+0.7
7 Run and check	Automated	+10.7	+12.0	+1.2	+8.3	−10.7	−3.6	+1.2	+2.6
8 Run and check	System	+10.7	+8.0	0.0	+9.5	+1.2	−2.4	+4.8	+4.5
9 Reject completion	Automated	+4.8	+10.7	−2.4	+8.3	+2.4	+1.2	+3.6	+4.0
10 Reject completion	User	+9.5	+4.0	−1.2	+3.6	+4.8	−4.8	−1.2	+2.1
11 Recheck before finishing	User	+15.5	+12.0	+2.4	+6.0	+1.2	−4.8	−2.4	+4.1
12 Recheck before finishing	User / early	+9.5	**+13.3**	+3.6	+9.5	+6.0	−7.1	0.0	+4.8
13 Reject completion	User / early	+8.3	+8.0	0.0	**+17.9**	+1.2	0.0	+7.1	+6.0
L3 Independent review									
14 Claim-guided review	Changes	+10.7	+6.7	+3.6	0.0	0.0	**+8.3**	**+11.9**	+5.9
15 Task-first review	Changes	**+19.0**	+10.7	0.0	+9.5	+3.6	−1.2	+7.1	+6.9
16 Claim-guided review	Trajectory	+14.3	**+13.3**	**+9.5**	+13.1	+1.2	+2.4	+5.4	+8.4
17 Task-first review	Trajectory	+15.5	+10.7	+4.8	+14.3	**+7.1**	+7.1	+7.1	**+9.5**

permission to modify the deliverable. The agent can then revise the artifact using the returned feedback.

Metrics. For TB and SWE, we report accuracy (the percentage of tasks solved) and cost. We average accuracy and cost across trials for each model–task pair. Cost includes the coding agent, the critic, and subsequent revisions, priced separately by model. Figure 2 reports each design's mean cost relative to its baseline, with per-model plots in Appendix B.2. Appendix A.2 lists the API cost prices, and Appendix B.3 separates agent and critic output tokens.

5 Results

In this section, we analyze L0 initial instructions in Section 5.1, L1 execution-time reminders in Section 5.2, L2 completion gates in Section 5.3, and L3 independent review in Section 5.4.

5.1 L0: Initial instructions

L0 verification instructions yield modest average coding gains, with substantial differences across models and benchmarks. Instructions with the system role improve accuracy by $+3.3$ points on TB and $+3.9$ points on SWE, compared with $+1.2$ and -1.1 points for instructions with the user role, as shown in Tables 2 and 3. On SWE, MiniMax-M2.5 benefits from both variants ($+12.6/+8.0$ points for system/user), whereas Qwen3.8-Max declines under both ($-2.3/-2.3$). Qwen3.8-Max instead gains $+8.0/+4.0$ points on TB with the system/user roles. On VisualBench, L0 yields an average net preference of -3.2 points over the baseline, as shown in Figure 3. DeepSeek V4-Flash receives a net preference of $+5.6$ points, while GLM-5.2 receives -8.3 points despite its positive accuracy gains on both coding benchmarks.

Cost changes are modest: system/user instructions incur $1.08/1.03\times$ baseline cost on TB and $0.94/0.93\times$ on SWE, as shown in Figure 2. Average output tokens and agent turns remain close to or below baseline levels across both benchmarks and message roles, at 0.90–$1.00\times$ and 0.94–$0.99\times$, respectively, as reported in Appendix Table 6.

These average gains are not accompanied by consistently increased generation or longer interactions. Overall, we recommend concise verification instructions in the system prompt, as system

Table 3: Accuracy changes (pp) on SWE-bench Verified across verification methods and models.

Method	Variant	Qwen 3.7-Max	Qwen 3.8-Max	GLM 5.2	DeepSeek V4-Pro	DeepSeek V4-Flash	Kimi K2.7 Code	MiniMax M2.5	Average
L0 Initial instructions									
1 Verification instruction	System	+1.1	−2.3	+6.9	+2.3	+4.6	+2.3	+12.6	+3.9
2 Verification instruction	User	0.0	−2.3	−2.3	−3.4	−5.7	−2.3	+8.0	−1.1
L1 Execution-time reminders									
3 Periodic reminder	System	+1.1	0.0	+3.4	−1.1	0.0	+5.7	+12.6	+3.1
4 After-edit reminder	System	**+4.6**	0.0	+5.7	+3.4	−2.3	−1.1	**+14.9**	+3.6
5 After-edit reminder	User	−4.6	0.0	+1.1	−2.3	−1.1	−1.1	+5.7	−0.3
L2 Completion gates									
6 Inspect changes	Automated	−2.3	+1.1	0.0	−9.2	+6.9	−2.3	+9.2	+0.5
7 Run and check	Automated	−5.7	+4.6	0.0	−1.1	0.0	−3.4	+2.3	−0.5
8 Run and check	System	+3.4	−2.3	0.0	+1.1	+8.0	−6.9	+6.9	+1.5
9 Reject completion	Automated	0.0	+2.3	+9.2	+4.6	**+10.3**	−2.3	+12.6	+5.3
10 Reject completion	User	0.0	0.0	**+13.8**	0.0	+3.4	0.0	+11.5	+4.1
11 Recheck before finishing	User	**+4.6**	−6.9	−2.3	−3.4	+1.1	−4.6	+9.2	−0.3
12 Recheck before finishing	User / early	−4.6	+2.3	+1.1	+5.7	**+10.3**	+5.7	+8.0	+4.1
13 Reject completion	User / early	+2.3	+1.1	+11.5	+4.6	+9.2	+3.4	+11.5	+6.2
L3 Independent review									
14 Claim-guided review	Changes	−1.1	**+5.7**	+5.7	**+6.9**	−3.4	+13.8	+8.0	+5.1
15 Task-first review	Changes	0.0	+4.6	0.0	**+6.9**	+3.4	+17.2	+13.8	**+6.6**
16 Claim-guided review	Trajectory	**+4.6**	+2.3	+5.7	0.0	+2.3	**+20.7**	+6.9	+6.1
17 Task-first review	Trajectory	+1.1	+3.4	−3.4	+4.6	−2.3	+16.1	+8.0	+3.9

instructions outperform user instructions on average in the coding benchmarks, while their uneven benefits favor leaving room for other harness instructions.

The case studies illustrate how L0 instructions can alter verification behavior that is already present. On `bn-fit-modify`, MiniMax's baseline run inspects the generated files and their row counts but misses the required edge orientation. With the verification instruction, the agent revisits this requirement, corrects the CSV convention, and passes, as shown in Appendix Figure 25. On `git-multibranch`, Qwen3.7-Max checks deployment endpoints in both conditions. However, the instructed run deletes the repository to prepare for another test, removing the deployment hook, and fails despite subsequent reconstruction; the baseline passes, as shown in Appendix Figure 26.

L0 Takeaways

Keep verification guidance in system prompt lightweight. Use concise instructions, leaving room for other harness guidance. Use user message for task requirements and context.

5.2 L1: Execution-time reminders

Periodic and after-edit L1 reminders have similar average effects with the system message role: +3.8 versus +3.5 points on TB and +3.1 versus +3.6 on SWE. Switching the message role of after-edit reminders to user raises the average gain to +5.4 points on TB but lowers it to −0.3 on SWE, as shown in Tables 2 and 3. On VisualBench, MiniMax-M2.5 receives +12.5 points, while GLM-5.2 and Qwen3.7-Max each receive −12.5 points. Across the three L1 designs, average output tokens decrease by 14–25% on TB and 7–10% on SWE, while cost ranges from 0.90–1.12× and 0.85–0.89× baseline, respectively, as shown in Figure 2 and Appendix Figure 15. Higher accuracy can also coexist with less generated output. On TB, Qwen3.7-Max gains +7.1/ + 14.3 points with after-edit system/user reminders while using 33%/28% fewer output tokens, respectively. On VisualBench, L1 yields an average net preference of −3.2 points over the baseline, as shown in Figure 3. Overall, we recommend verification reminders after code edits to focus the agent on task requirements and the resulting artifact alongside the code itself. The average accuracy gains with fewer output tokens on TB and SWE support using L1, while the VisualBench results motivate checks that also cover rendered output and interactive behavior.

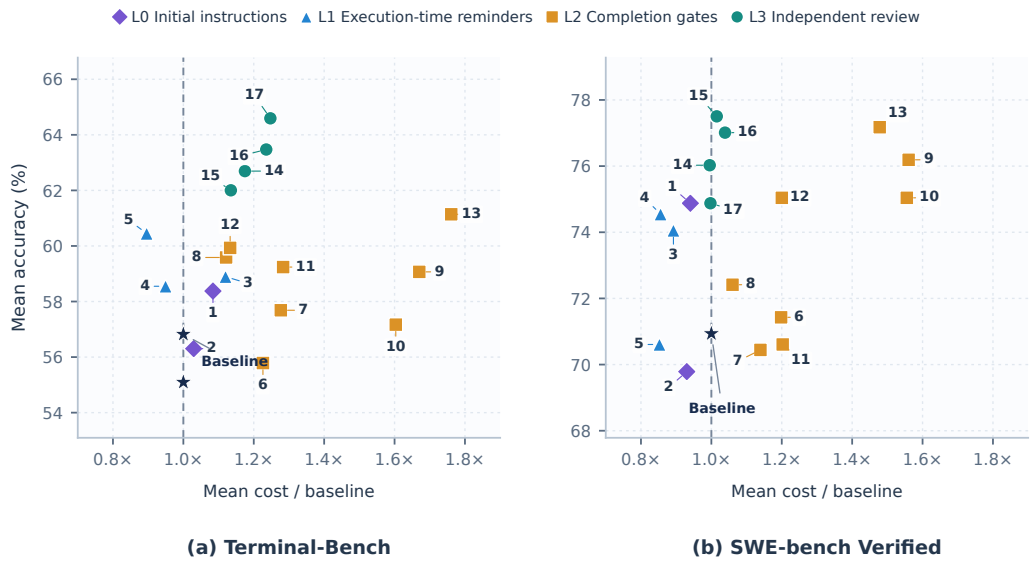


Figure 2: Mean accuracy versus mean cost relative to baseline (method IDs are in Tables 2 and 3).

Trajectory analysis distinguishes continued execution from task-relevant verification. On TB, 90–97% of after-edit user reminders are followed by execution commands within two agent turns, as reported in Appendix Table 8. The checks themselves also differ in coverage. On `nginx-request-logging`, Qwen3.7-Max checks the Nginx configuration, HTTP responses, and access log after a reminder and passes, whereas the baseline stops with an unfinished plan, as shown in Appendix Figure 27. However, Kimi-K2.7-Code verifies compilation while overlooking invalid word substitutions on `overfull-hbox`, and DeepSeek V4-Flash verifies output equality yet misses the runtime criterion on `query-optimize`, as shown in Appendix Figure 28 and Appendix Figure 29.

L1 Takeaways

Use verification reminders after code edits to guide the agent in checking both the code and the resulting artifact against task requirements. Include checks of rendered output and interactive behavior when relevant.

5.3 L2: Completion Gates

L2 yields limited gains relative to its cost, even when rejection and earlier intervention improve average accuracy. Automated *Reject completion* yields $+4.0/+5.3$ points on TB/SWE, compared with $+0.7/+0.5$ for *Inspect changes* and $+2.6/-0.5$ for automated *Run and check*. Using the system role for *Run and check* raises the gains to $+4.5/+1.5$ points. Earlier triggering also improves both user rechecking and user rejection: rechecking rises from $+4.1/-0.3$ to $+4.8/+4.1$ points, while rejection rises from $+2.1/+4.1$ to $+6.0/+6.2$ points, the largest average gains among the tested L2 designs, as shown in Tables 2 and 3. Across the rejection designs, cost reaches 1.60–$1.76\times$ baseline on TB and 1.48–$1.56\times$ on SWE. Early user rechecking achieves its gains at a lower cost of $1.13/1.20\times$ baseline, respectively, as shown in Figure 2. On VisualBench, early rechecking yields an average net preference of $+2.7$ points, as shown in Figure 3. GLM-5.2 receives $+9.4$ points, while DeepSeek V4-Flash receives -8.0 points. Overall, we recommend avoiding completion-time self-checking, as its gains are limited relative to the additional cost and reopening work does not consistently improve the final artifact.

The L2 cases illustrate both useful repair and invalid final states. On TB, model-level medians range from 12–19 additional steps after user rejection, compared with 4–6 after *Inspect changes*. File-changing actions follow user rejection in 86–96% of L2 runs in which user rejection is triggered, as

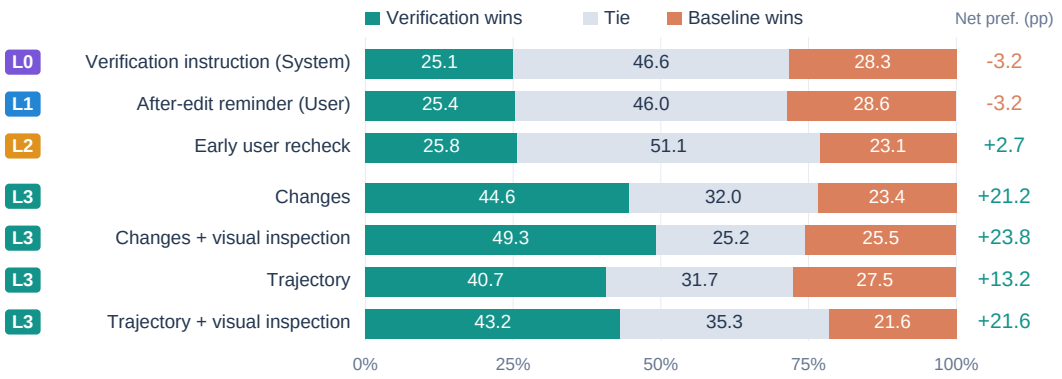


Figure 3: VisualBench win, tie, and loss results.

reported in Appendix Table 9. On `regex-log`, MiniMax-M2.5 resumes after early user rejection and finds that its regex misses dates placed before the IP address. It removes this ordering assumption and passes grading, whereas the baseline retains it and fails, as shown in Appendix Figure 30. On `git-multibranch`, Kimi-K2.7-Code responds to an early recheck with deployment tests that commit and push test content to the served branches. It leaves that content in place and fails, whereas the passing baseline restores the required branch contents, as shown in Appendix Figure 31. These cases show why further execution alone is insufficient: a completion gate creates an opportunity for repair, but the agent must still identify the relevant defect and preserve a valid final artifact.

L2 Takeaways

Avoid completion-time self-checking. Asking the main agent to recheck its own work can improve accuracy, but the gains are limited relative to the additional cost.

5.4 L3: Independent Review

L3 delivers consistent improvement across all four designs on TB and SWE at modest additional cost. The gains range from $+5.9$–9.5 points on TB and $+3.9$–6.6 on SWE, as shown in Tables 2 and 3. Cost, including critic review and agent revision, is 1.13–$1.25\times$ baseline on TB and 1.00–$1.04\times$ on SWE, as shown in Figure 2. Compared with the strongest L2 design, the strongest L3 design achieves a larger gain on TB ($+9.5$ versus $+6.0$ points) and a comparable gain on SWE ($+6.6$ versus $+6.2$), at lower cost on both benchmarks. On VisualBench, all four L3 designs receive positive net preference, ranging from $+13.2$ to $+23.8$ points and exceeding the evaluated L0–L2 designs, as shown in Appendix Table 12. Changes-based review outperforms trajectory-based review both without required visual inspection ($+21.2$ versus $+13.2$ points) and with it ($+23.8$ versus $+21.6$ points). Required visual inspection improves net preference for both contexts, with a larger increase for trajectory-based review. Both changes-based designs receive positive net preference for all seven models, while trajectory-based designs retain some negative model-level results. Across the three benchmarks, the full trajectory does not consistently improve outcomes, and trajectory-based designs incur 29–43% higher critic cost on TB and SWE. Overall, we recommend task requirements and changes as the default context for a separate, low-cost critic.

The trajectories illustrate how independent review turns additional checks into concrete repairs. On `regex-log`, MiniMax-M2.5 tests its regex and declares completion, but the critic tests a case with the date appearing before the IP address and exposes a missed match. The agent then revises the regex and passes, whereas the baseline performs its own tests but still fails, as shown in Appendix Figure 32. On `cobol-modernization`, Qwen3.7-Max's implementation passes its own checks for valid and invalid transactions. The critic tests an arithmetic underflow and finds that Python writes a negative sign that COBOL's unsigned field omits, breaking the required record format. The agent reproduces the discrepancy, repairs the numeric formatting, and passes the external grader; the baseline declares success after its own comparisons but fails grading, as shown in Appendix Figure 33. In both examples, the critic tests a case that the agent overlooked and supplies an executable

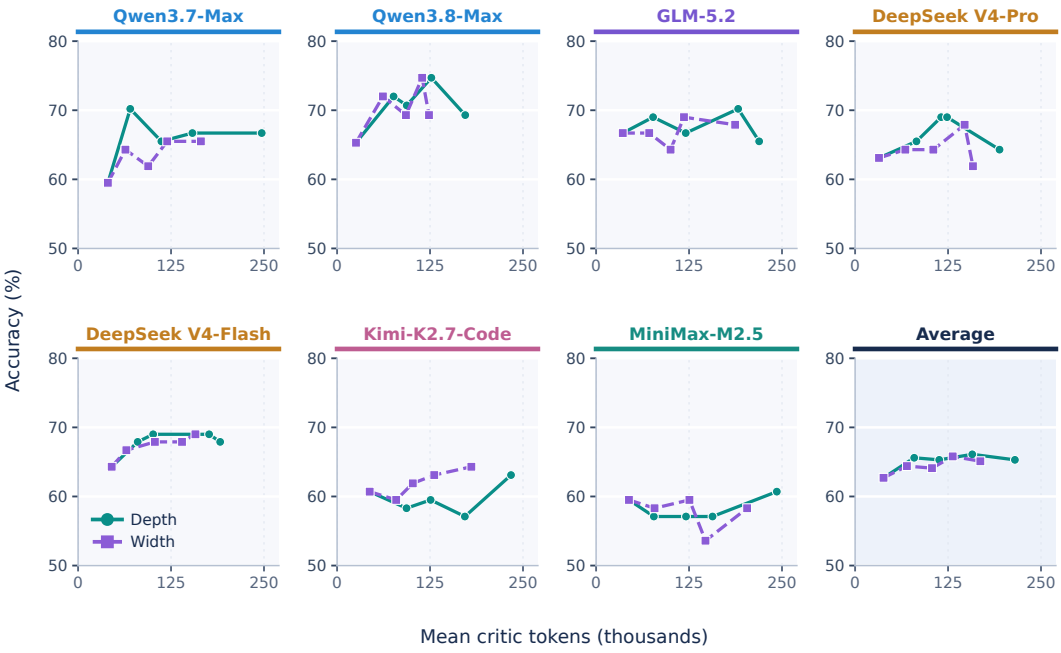


Figure 4: Scaling independent review on TB through additional review rounds (depth) and critics (width), showing accuracy versus mean critic token use.

failure that guides revision. This illustrates how an independent checking perspective can address gaps in the agent's self-verification. On `cancel-async-tasks`, GLM-5.2's critic confirms that asynchronous cleanup remains incomplete after cancellation, but the run reaches its 45-minute limit without a completed repair after review, as shown in Appendix Figure 34.

Scaling review. Increasing review depth (successive review rounds) or width (independent critics) further improves average accuracy on TB. One critic and one round achieve 62.7% accuracy at 38.2k mean critic tokens, compared with 65.6% at 79.5k for two rounds and 64.4% at 69.7k for two critics. All larger settings outperform a single review, reaching 66.1% with four rounds and 65.8% with four critics, as shown in Figure 4. Four critics improve average accuracy by 3.1 points over one critic at 1.33× total cost, including revisions. Scaling verification through low-cost critics can limit repeated execution by the main agent. By consolidating independent feedback before revision, four critics approach four-round accuracy at 19% lower total cost, as shown in Appendix Figure 23.

L3 Takeaways

Use a separate, low-cost critic to check the deliverable and guide agent revision. Scaling review through additional rounds or independent critics further improves average accuracy.

6 Conclusion

In this paper, we systematically study verification in agent harnesses, comparing 17 designs across four layers of the agent lifecycle and seven frontier coding models on TB and SWE, with evaluation on VisualBench. We examine when to intervene, who checks the work, and how feedback leads to revision. Initial instructions and reminders yield modest, model-dependent coding gains without substantially increasing average output token use. Completion gates can improve accuracy but may substantially increase cost. Independent review offers a more favorable accuracy–cost tradeoff on TB and SWE. On VisualBench, its four designs achieve net preferences of 13.2–23.8 points over the baseline, exceeding the evaluated L0–L2 designs. Additional rounds or critics further improve average TB accuracy by up to 3.5 percentage points. Together, these results provide empirical guidance for verification design in agent harnesses and support independent review as a practical axis of test-time scaling.

ETHICS STATEMENT

VisualBench's human evaluation received ethics approval. Raters are members of an internal company data team; they provided informed consent and were compensated under their existing contracts with the company. The interface explains the study's purpose and assessment criteria, while hiding verification conditions and randomizing artifact placement. Appendix C documents the evaluation protocol, task descriptions, rater instructions, and interface.

AI USE STATEMENT

The authors developed the research ideas and system design and led the manuscript writing. Generative AI tools assisted with implementation and manuscript polishing. The authors are responsible for the final content.

REFERENCES

Alibaba Cloud. Context Cache. Alibaba Cloud Model Studio documentation, 2026a. URL https://help.aliyun.com/zh/model-studio/context-cache. Accessed September 19, 2026.

Alibaba Cloud. Model Studio Model Pricing. Alibaba Cloud Model Studio documentation, 2026b. URL https://help.aliyun.com/zh/model-studio/model-pricing. Accessed September 19, 2026.

Anthropic. How Claude Code works. Claude Code documentation, 2026a. URL https://code.claude.com/docs/en/how-claude-code-works. Documentation version accessed September 18, 2026.

Anthropic. Hooks reference. Claude Code documentation, 2026b. URL https://code.claude.com/docs/en/hooks. Documentation version accessed September 21, 2026.

Michael Bolin. Unrolling the Codex agent loop. OpenAI Engineering, January 2026. URL https://openai.com/index/unrolling-the-codex-agent-loop/. Published January 23, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug, 2023. URL https://arxiv.org/abs/2304.05128.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems, 2021. URL https://arxiv.org/abs/2110.14168.

DeepSeek-AI et al. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

E2B. E2B documentation, 2026. URL https://docs.e2b.dev/. Documentation version accessed September 19, 2026.

Google. Gemini CLI core. Gemini CLI documentation, 2026. URL https://geminicli.com/docs/core/. Documentation version accessed September 18, 2026.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing, 2023. URL https://arxiv.org/abs/2305.11738.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://zenodo.org/doi/10.5281/zenodo.20953922.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation, 2023a. URL https://arxiv.org/abs/2312.13010.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large Language Models Cannot Self-Correct Reasoning Yet, 2023b. URL https://arxiv.org/abs/2310.01798.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?, 2023. URL https://arxiv.org/abs/2310.06770.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs, 2024. URL https://arxiv.org/abs/2406.01297.

Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srinivasan Iyer, Manzil Zaheer, Daniel Fried, Hannaneh Hajishirzi, Sanjeev Arora, Gabriel Synnaeve, Ruslan Salakhutdinov, and Anirudh Goyal. Scaling Test-Time Compute for Agentic Coding, 2026. URL https://arxiv.org/abs/2604.16529.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training Language Models to Self-Correct via Reinforcement Learning, 2024. URL https://arxiv.org/abs/2409.12917.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's Verify Step by Step, 2023. URL https://arxiv.org/abs/2305.20050.

Ryan Lopopolo. Harness engineering: leveraging Codex in an agent-first world. OpenAI Engineering, February 2026. URL https://openai.com/index/harness-engineering/. Published February 11, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM Critics Help Catch LLM Bugs, 2024. URL https://arxiv.org/abs/2407.00215.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

MiniMax. MiniMax M2.5: Built for Real-World Productivity. Official model announcement, 2026. URL https://www.minimax.io/news/minimax-m25. Published February 12, 2026.

Moonshot AI. Kimi K2.7 Code. Kimi Code release notes, 2026. URL https://www.kimi.com/code/docs/en/kimi-code/whats-new.html#kimi-k2-7-code-june-12-2026. Released June 12, 2026.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is Self-Repair a Silver Bullet for Code Generation?, 2023. URL https://arxiv.org/abs/2306.09896.

OpenAI. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

OpenAI. Codex CLI. Official Codex documentation, 2026. URL https://learn.chatgpt.com/docs/codex/cli. Documentation version accessed September 18, 2026.

Eric S. Qiu and Joyce Gill. Adversarial Review: Structured Disagreement for Grounded Agentic Code Review, 2026. URL https://arxiv.org/abs/2608.18167.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve, 2024. URL https://arxiv.org/abs/2407.18219.

Qwen Team. Qwen3.7: The agent frontier, May 2026a. URL https://qwen.ai/blog?id=qwen3.7.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, 2023. URL https://arxiv.org/abs/2303.11366.

Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawende F. Bissyande. CodeAgent: Autonomous Communicative Agents for Code Review, 2024. URL https://arxiv.org/abs/2402.02172.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, Bangkok, Thailand, August 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510/.

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, 2024b. URL https://arxiv.org/abs/2407.16741.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents, 2024. URL https://arxiv.org/abs/2407.01489.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models, 2022. URL https://arxiv.org/abs/2210.03629.

Z.ai. GLM-5.2. Official model card, 2026. URL https://huggingface.co/zai-org/GLM-5.2. Accessed September 18, 2026.

King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, Changwang Zhang, Chenghua Lin, Jun Wang, Ge Zhang, and Wangchunshu Zhou. Scaling Test-time Compute for LLM Agents, 2025. URL https://arxiv.org/abs/2506.12928.

APPENDIX GUIDE

A Experimental details 13
B Additional coding results 13
C VisualBench tasks and evaluation 29
D Case study 51
E Prompts and execution rules 62
?? Settings and reporting notes ??

A ADDITIONAL EXPERIMENTAL DETAILS

A.1 TASK SELECTION FOR TERMINAL-BENCH AND SWE-BENCH

We select tasks to balance coverage across easy, intermediate, and hard difficulty levels, enabling us to better observe both improvements and regressions.

Terminal-Bench. We began with a baseline-harness pilot over the 89-task Terminal-Bench registry recorded as version 2.1. The pilot used Qwen3.7-Max, GLM-5.2, DeepSeek V4-Pro, and Kimi-K2.7-Code. Their pass rates guided selection across three empirical difficulty bands: tasks that were usually solved, solved inconsistently, or usually failed. We aimed for balanced coverage of these bands, yielding the 31-task study set.

SWE-bench Verified. We initially selected 30 tasks evenly across three human time-to-fix tiers: under 15 minutes, 15 minutes to one hour, and over one hour. We then evaluated baseline performance and adjusted the task set to include both tasks with room for improvement and consistently solved tasks for detecting regressions.

A.2 API COST PRICING

Table 4 lists the model rates used to calculate cost, with Alibaba Cloud Model Studio pricing as the reference (Alibaba Cloud, 2026b;a).

Table 4: Rates used to calculate cost, in USD per million tokens.

Model	Uncached input	Cached input	Output
Qwen3.7-Max	1.791	0.358	5.373
Qwen3.8-Max	1.791	0.358	5.373
GLM-5.2	1.194	0.299	4.179
DeepSeek V4-Pro	1.791	0.149	3.582
DeepSeek V4-Flash	0.149	0.030	0.299
Kimi-K2.7-Code	0.970	0.194	4.030
MiniMax-M2.5	0.313	0.063	1.254
Qwen3.7-Plus (critic)	0.299	0.060	1.194

B ADDITIONAL CODING RESULTS

We report accuracy, cost, and token use by verification design and coding model, followed by behavioral diagnostics for L0–L2.

B.1 ABSOLUTE ACCURACY BY VERIFICATION DESIGN

Figure 5 reports accuracy by verification design, with open markers for the baseline and filled markers for the design. The corresponding absolute accuracies are listed alongside.

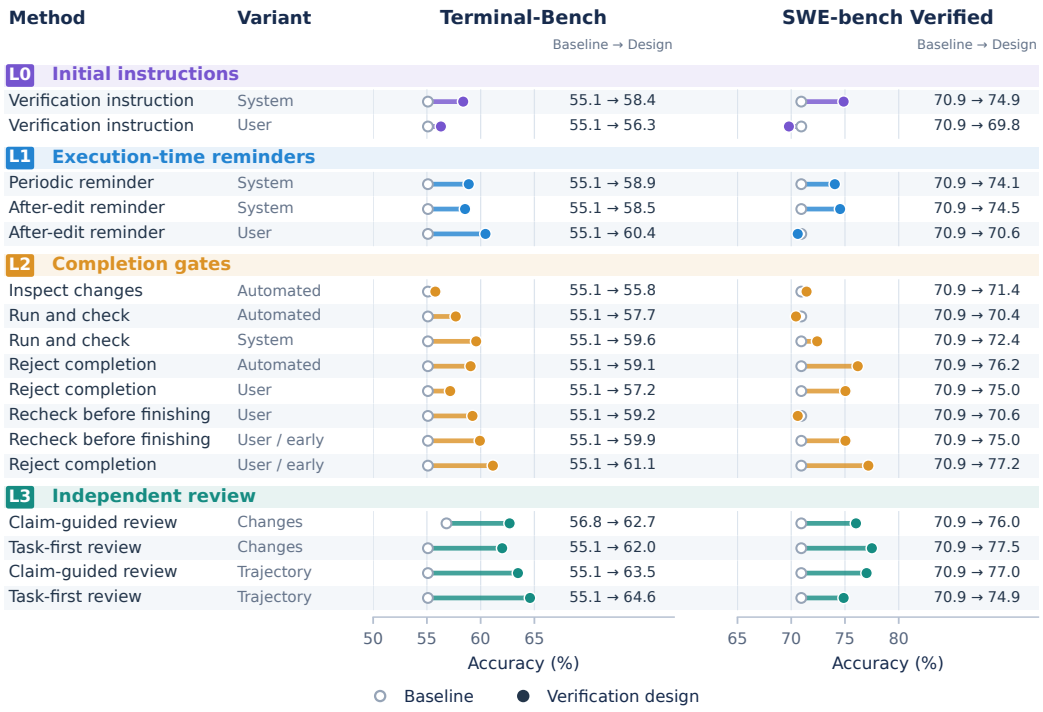


Figure 5: Absolute accuracy on TB and SWE, with baseline and design values shown alongside each comparison.

Figures 6–12 show the same comparison for each coding model.

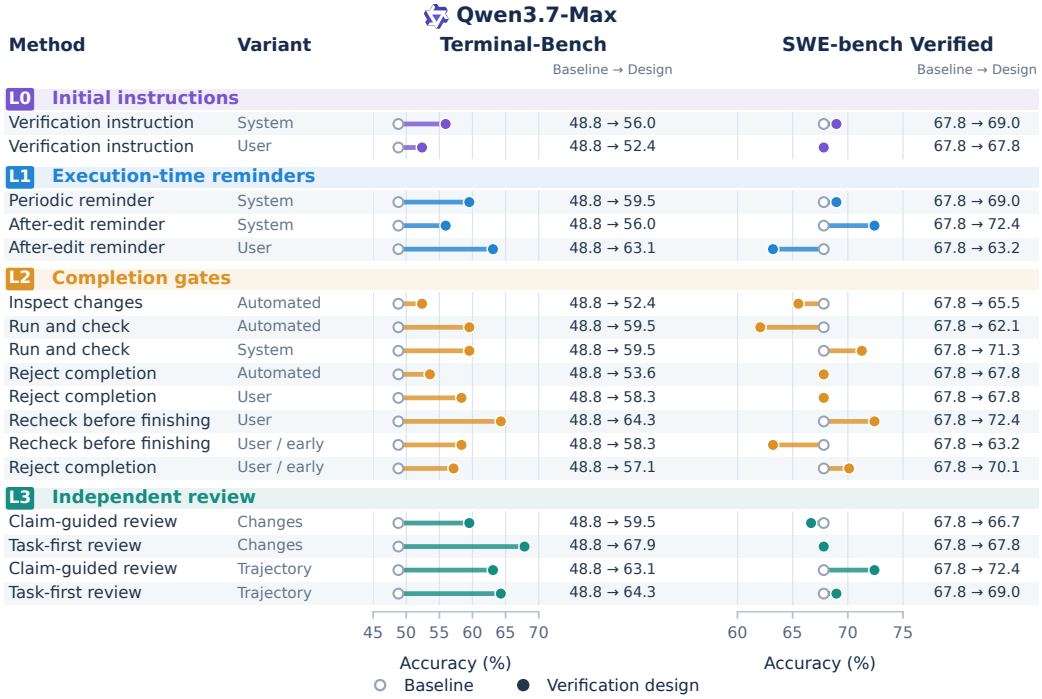


Figure 6: Absolute accuracy of Qwen3.7-Max on TB and SWE, with baseline and design values shown for each comparison.

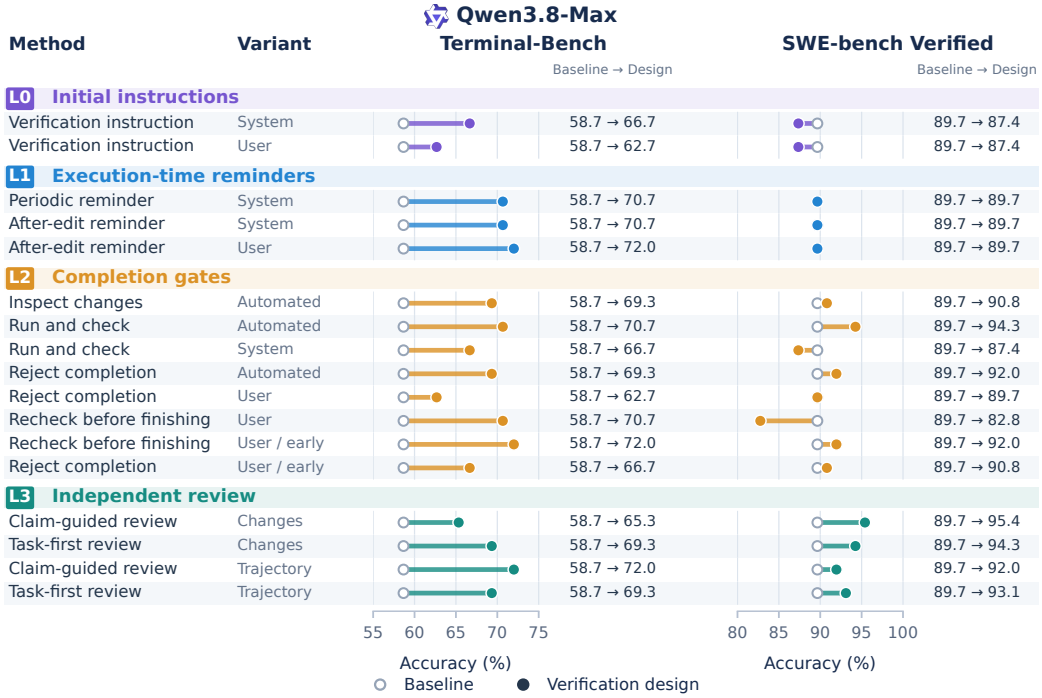


Figure 7: Absolute accuracy of Qwen3.8-Max on TB and SWE, with baseline and design values shown for each comparison.

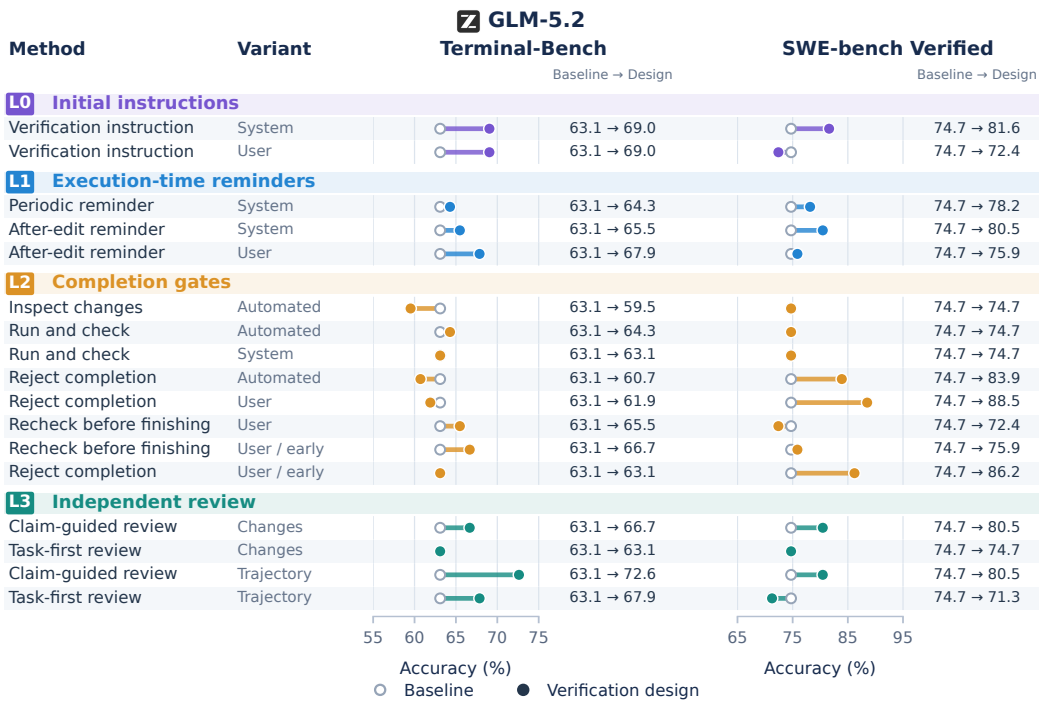


Figure 8: Absolute accuracy of GLM-5.2 on TB and SWE, with baseline and design values shown for each comparison.

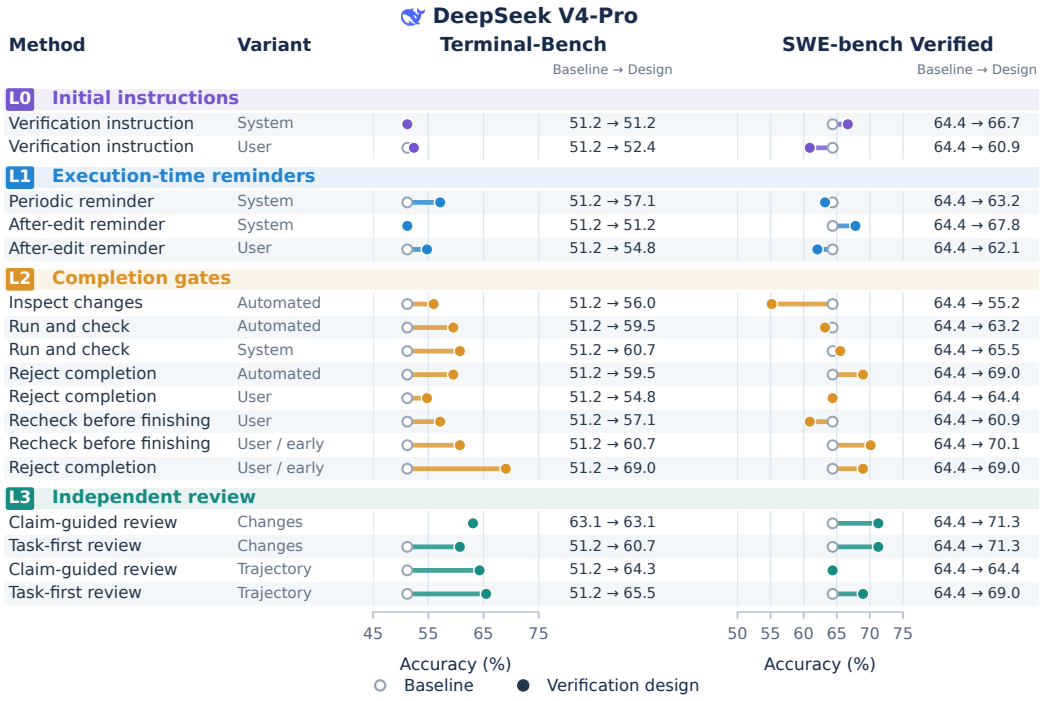


Figure 9: Absolute accuracy of DeepSeek V4-Pro on TB and SWE, with baseline and design values shown for each comparison.

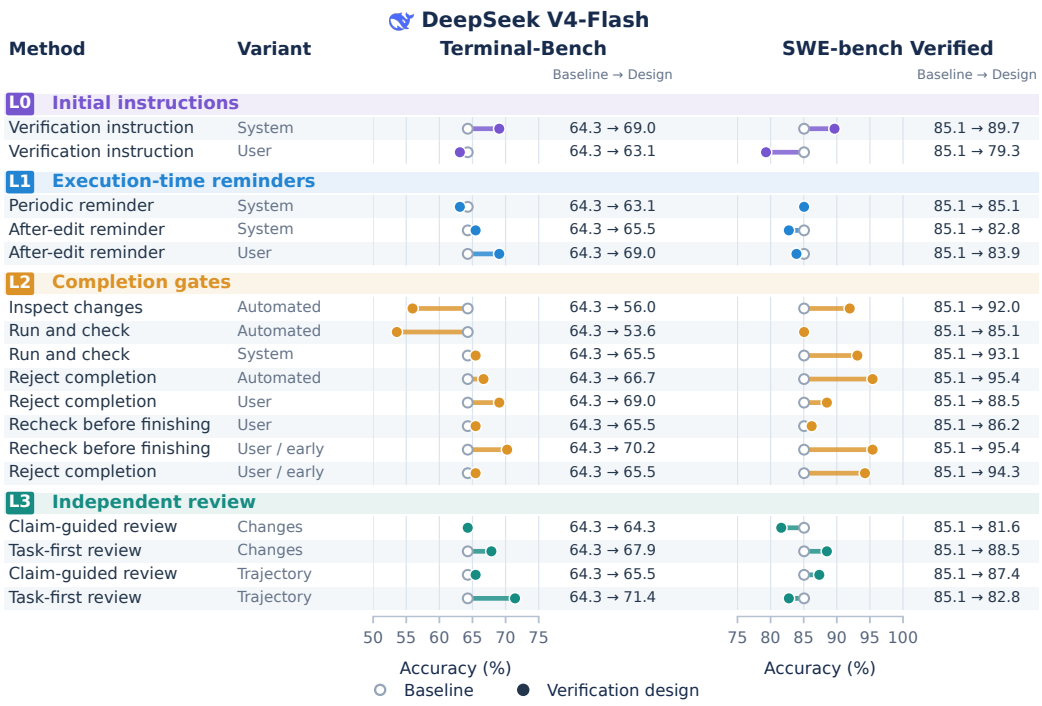


Figure 10: Absolute accuracy of DeepSeek V4-Flash on TB and SWE, with baseline and design values shown for each comparison.

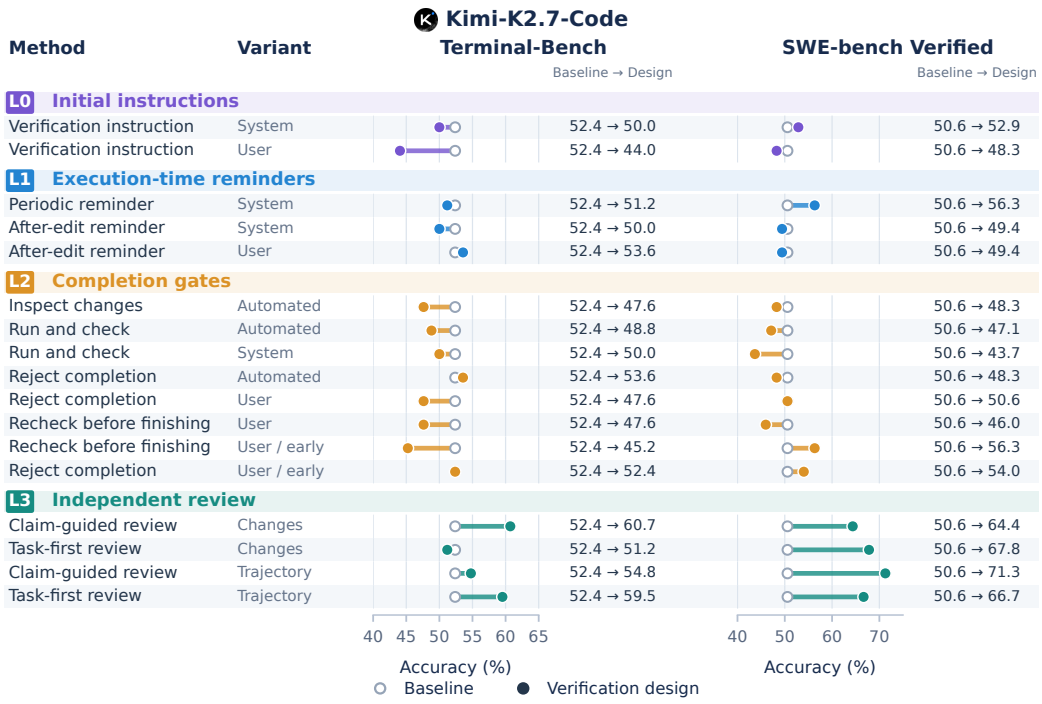


Figure 11: Absolute accuracy of Kimi-K2.7-Code on TB and SWE, with baseline and design values shown for each comparison.

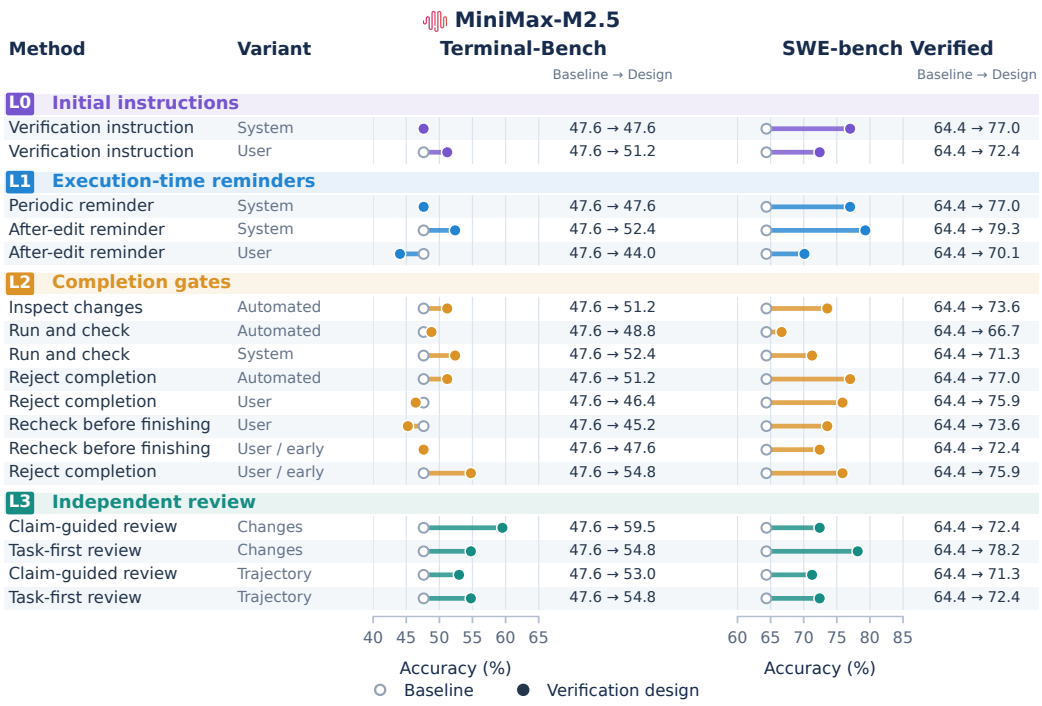


Figure 12: Absolute accuracy of MiniMax-M2.5 on TB and SWE, with baseline and design values shown for each comparison.

B.2 Accuracy and cost by coding model

Figures 13 and 14 separate Figure 2 by coding model, using the same trials, prices, and baseline. Each figure includes an Average panel reproducing Figure 2, and each panel labels all 17 verification designs by ID. Black stars mark baseline accuracy, and dashed lines mark $1\times$ baseline cost. Axis ranges are shared across models within each benchmark.

Table 5 lists the design IDs and settings.

Table 5: Verification design IDs and settings.

ID	Method	Variant
L0 Initial instructions		
1	Verification instruction	System
2	Verification instruction	User
L1 Execution-time reminders		
3	Periodic reminder	System
4	After-edit reminder	System
5	After-edit reminder	User
L2 Completion gates		
6	Inspect changes	Automated
7	Run and check	Automated
8	Run and check	System
9	Reject completion	Automated
10	Reject completion	User
11	Recheck before finishing	User
12	Recheck before finishing	User / early
13	Reject completion	User / early
L3 Independent review		
14	Claim-guided review	Changes
15	Task-first review	Changes
16	Claim-guided review	Trajectory
17	Task-first review	Trajectory

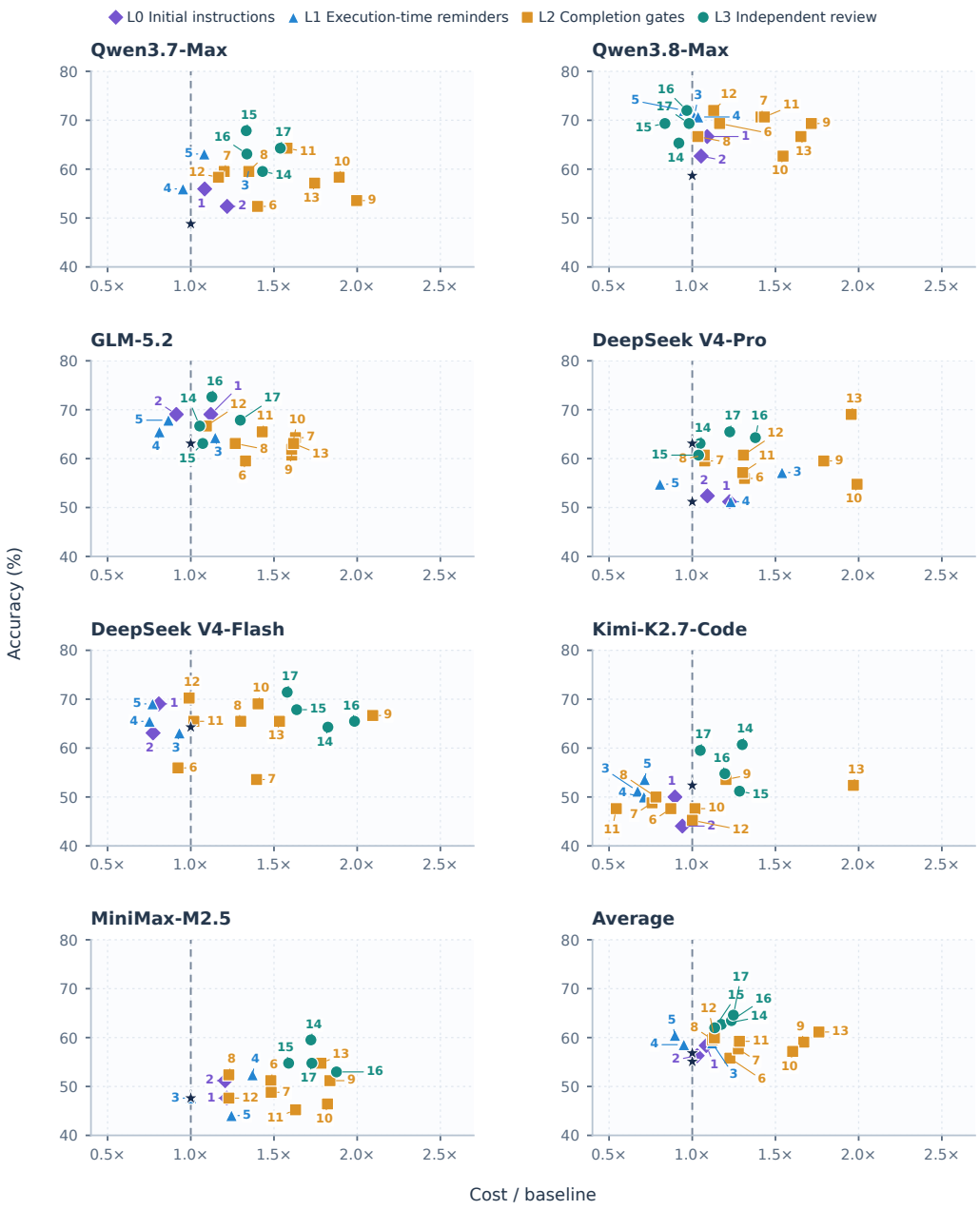


Figure 13: Accuracy versus cost relative to baseline on Terminal-Bench, shown for each coding model and their average.

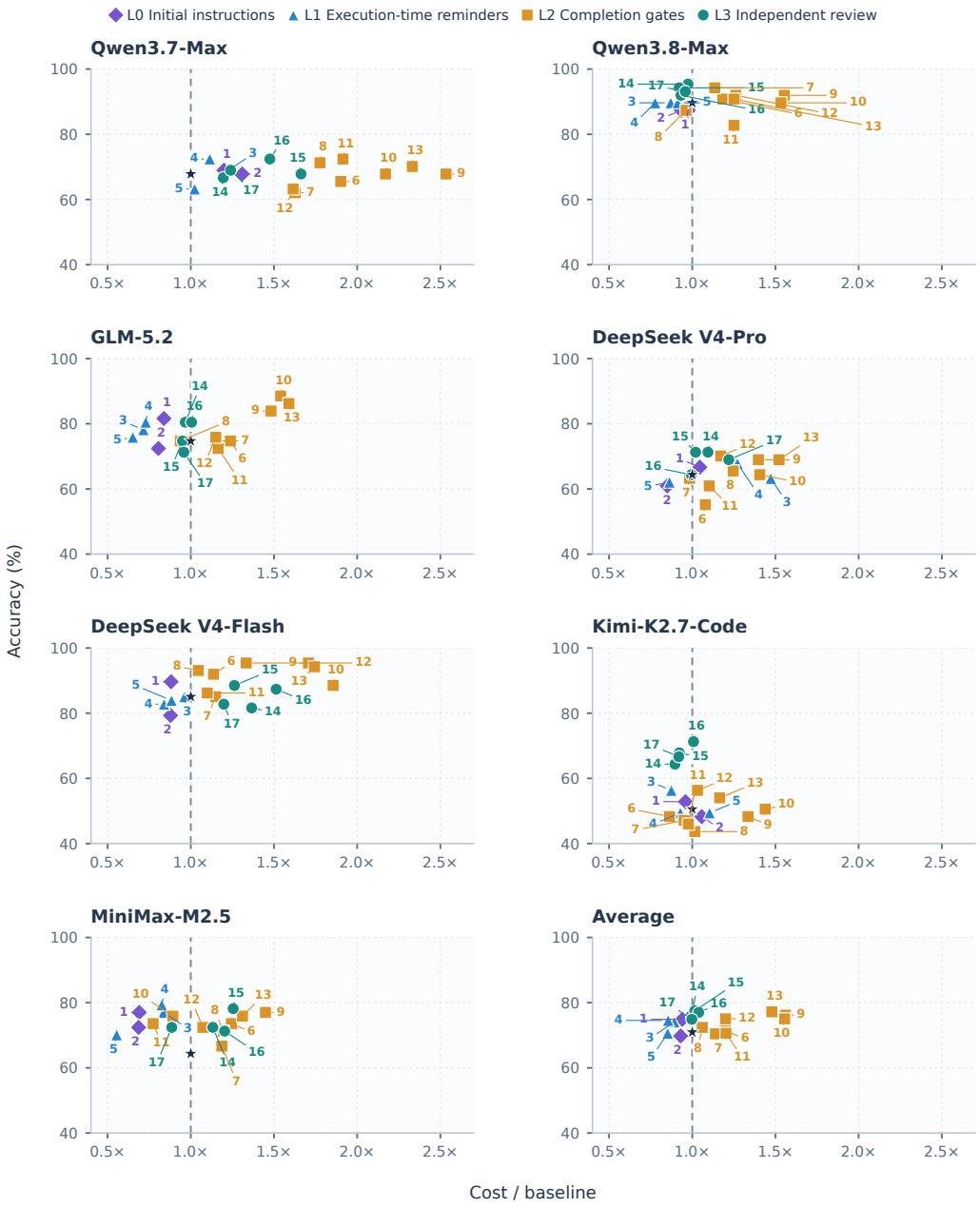


Figure 14: Accuracy versus cost relative to baseline on SWE-bench Verified, shown for each coding model and their average.

B.3 AGENT AND CRITIC TOKEN USE

Figure 15 shows mean output token use for all verification designs. Solid bars show agent generation, including revisions, and hatched segments show critic generation. Vertical ticks mark the baseline, with baseline and total values listed in thousands.

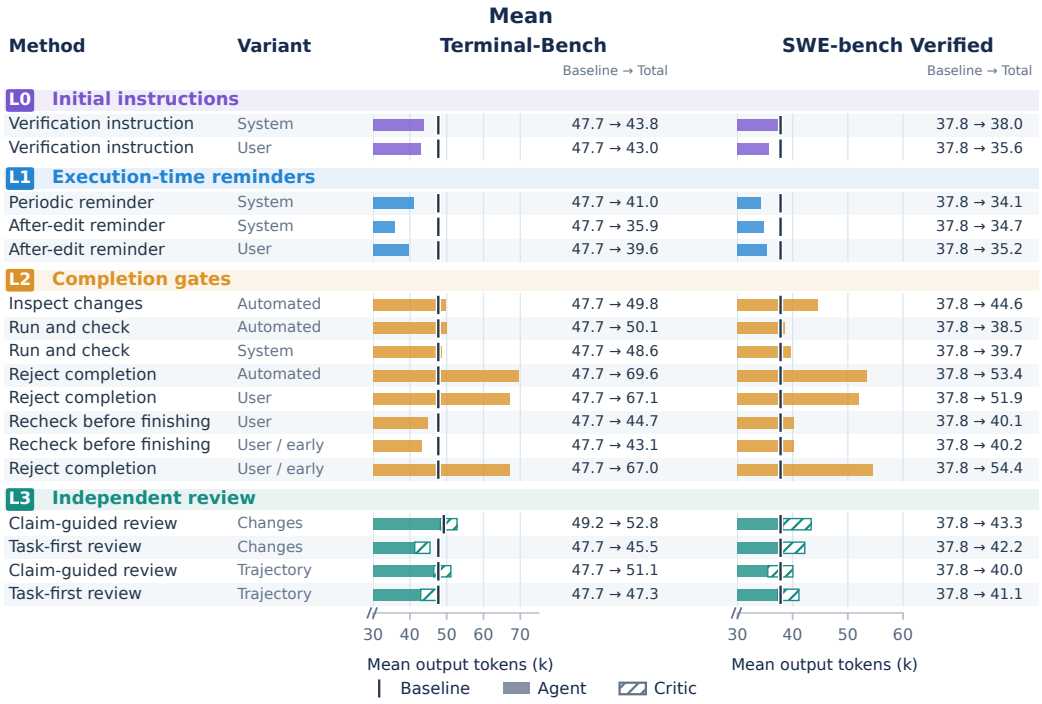


Figure 15: Mean agent and critic output tokens on TB and SWE, averaged across model–task pairs.

Figures 16–22 show the same comparison for each coding model.

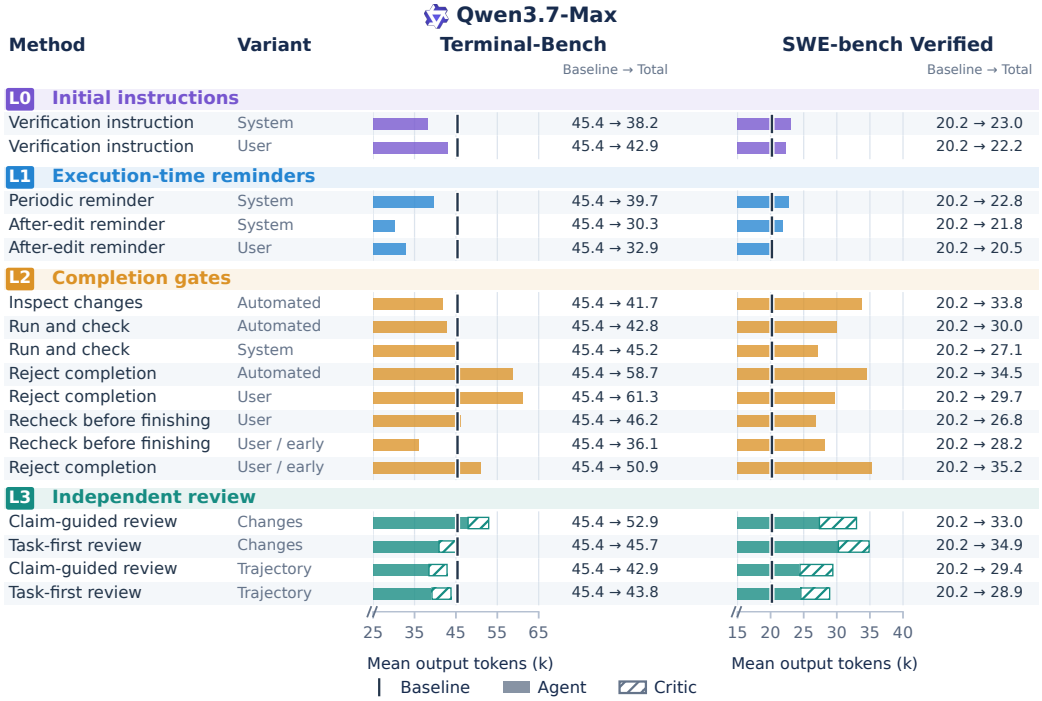


Figure 16: Mean agent and critic output tokens for Qwen3.7-Max on TB and SWE, with baseline and total values shown for each design.

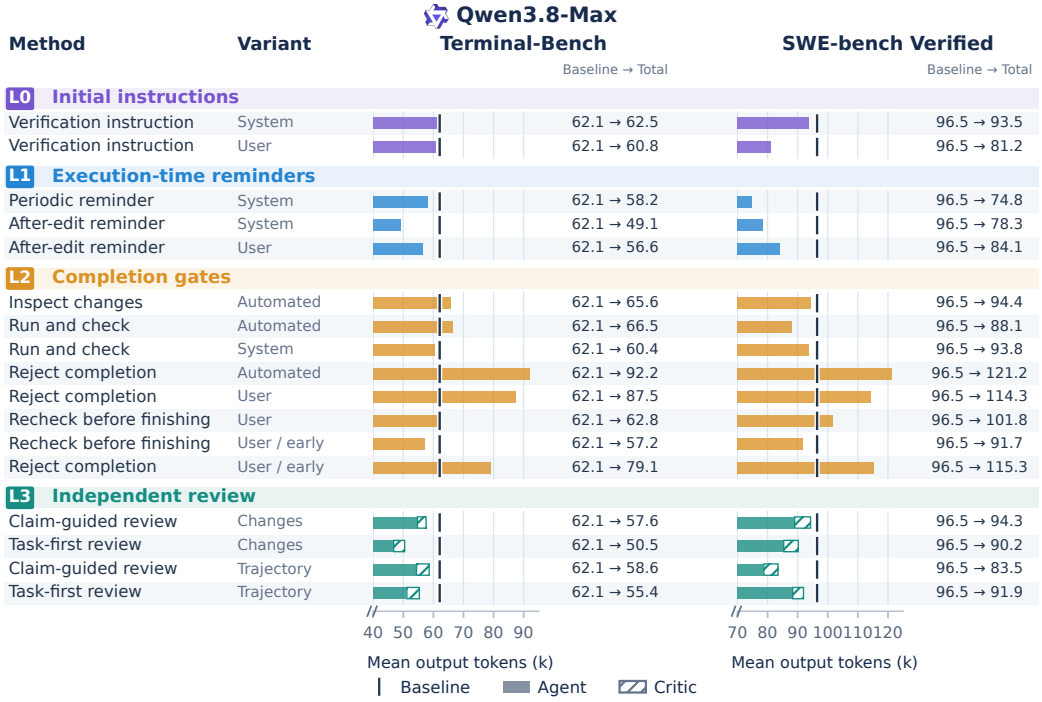


Figure 17: Mean agent and critic output tokens for Qwen3.8-Max on TB and SWE, with baseline and total values shown for each design.

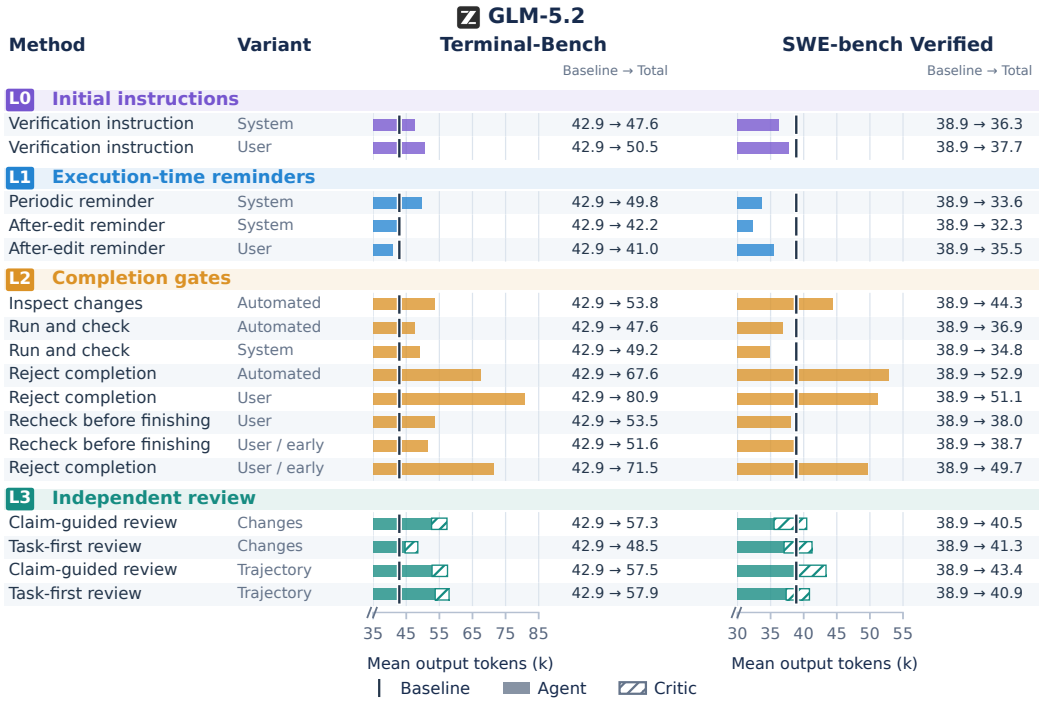


Figure 18: Mean agent and critic output tokens for GLM-5.2 on TB and SWE, with baseline and total values shown for each design.

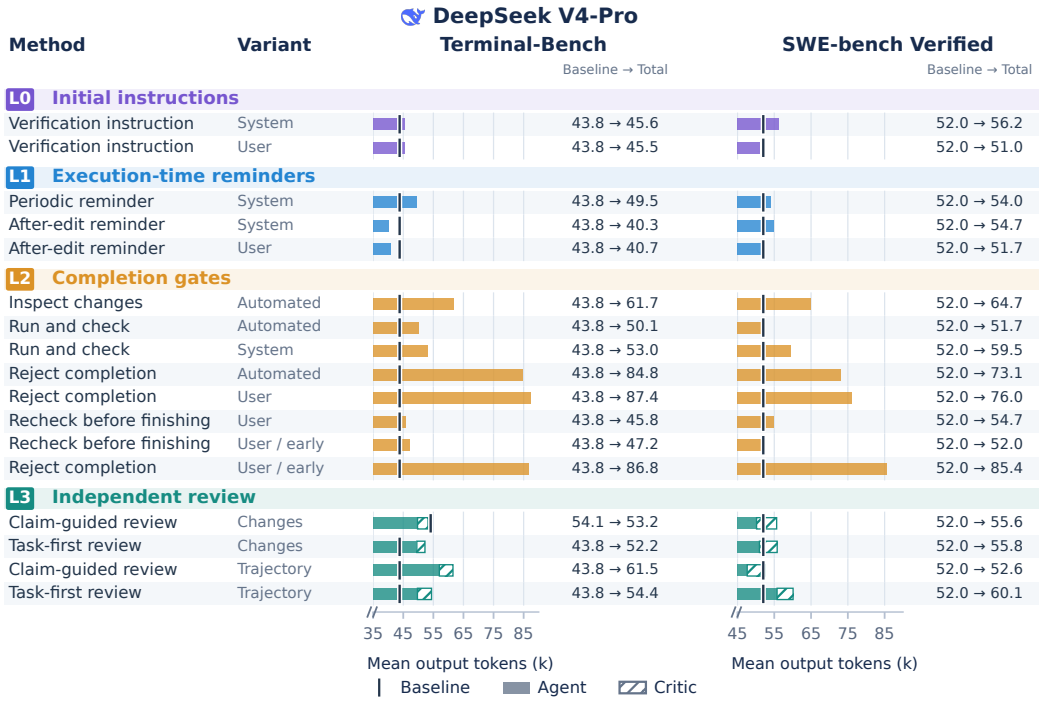


Figure 19: Mean agent and critic output tokens for DeepSeek V4-Pro on TB and SWE, with baseline and total values shown for each design.

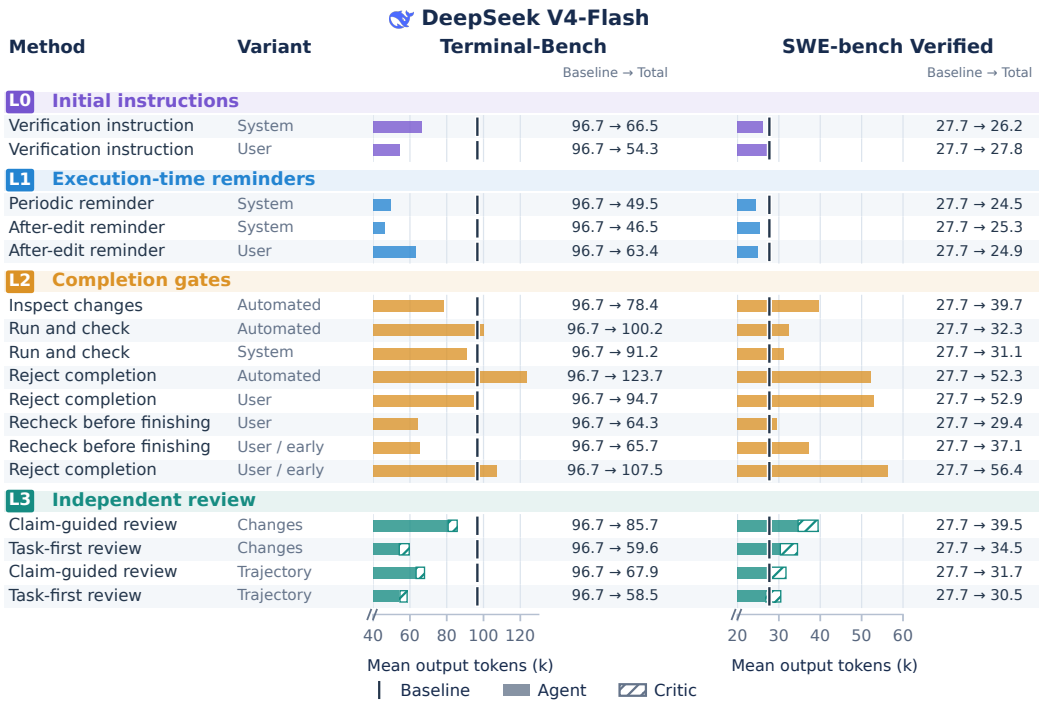


Figure 20: Mean agent and critic output tokens for DeepSeek V4-Flash on TB and SWE, with baseline and total values shown for each design.

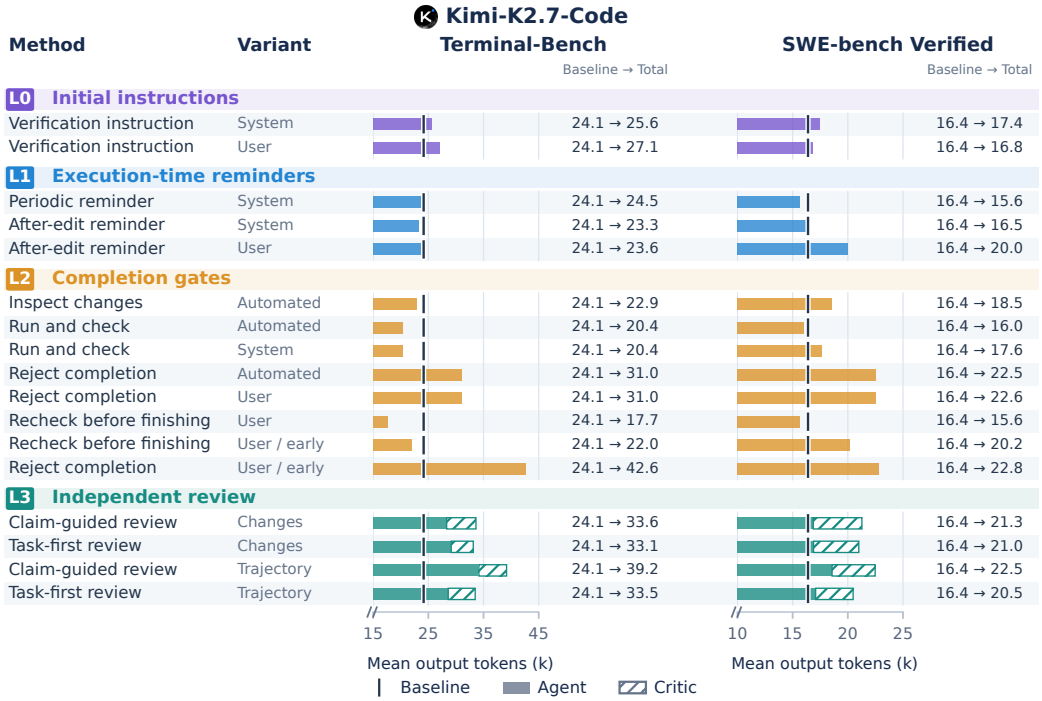


Figure 21: Mean agent and critic output tokens for Kimi-K2.7-Code on TB and SWE, with baseline and total values shown for each design.

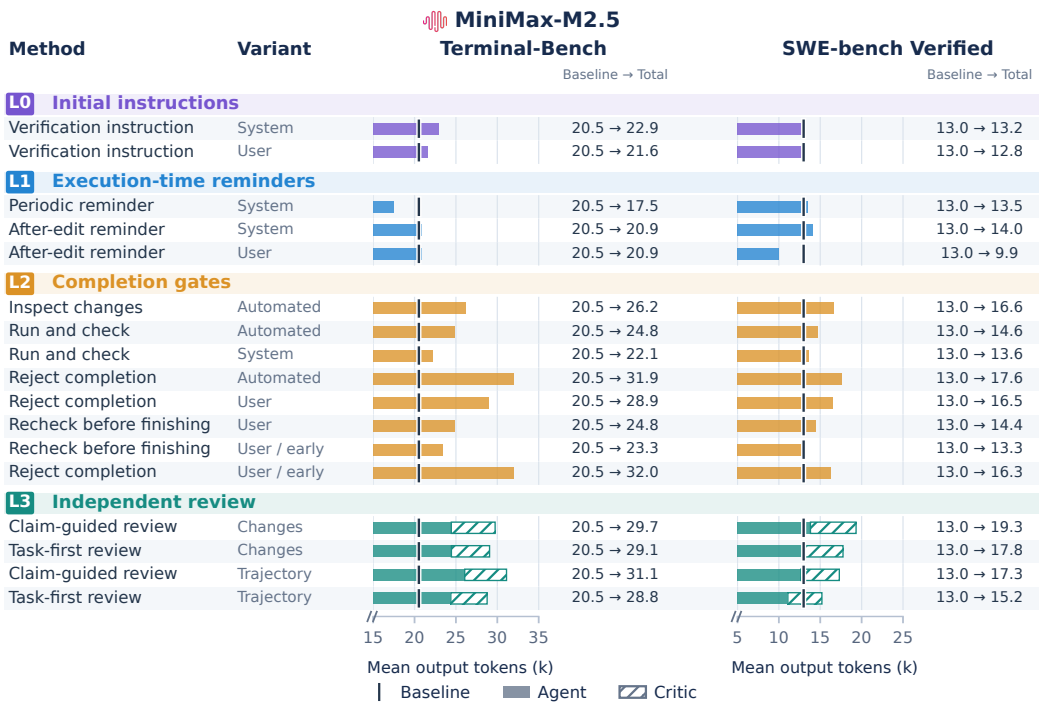


Figure 22: Mean agent and critic output tokens for MiniMax-M2.5 on TB and SWE, with baseline and total values shown for each design.

B.4 Review scaling and total cost

Figure 23 reports accuracy against mean total cost, including agent execution, critic checks, and subsequent revisions. One critic and one round achieve 62.7% accuracy at $0.47 per trial. Two rounds reach 65.6% at $0.61, while two critics reach 64.4% at $0.60. Four critics achieve 65.8% at $0.62, compared with 66.1% at $0.77 for four rounds. The critic costs for four critics and four rounds are both approximately $0.14 per trial, while their agent-side costs, including revisions, are $0.48 and $0.63, respectively. The cost difference therefore comes primarily from agent execution, consistent with wider review consolidating feedback before revision instead of repeatedly continuing the agent between review rounds. A second round raises Qwen3.7-Max from 59.5% to 70.2%, but lowers Kimi-K2.7-Code from 60.7% to 58.3%.

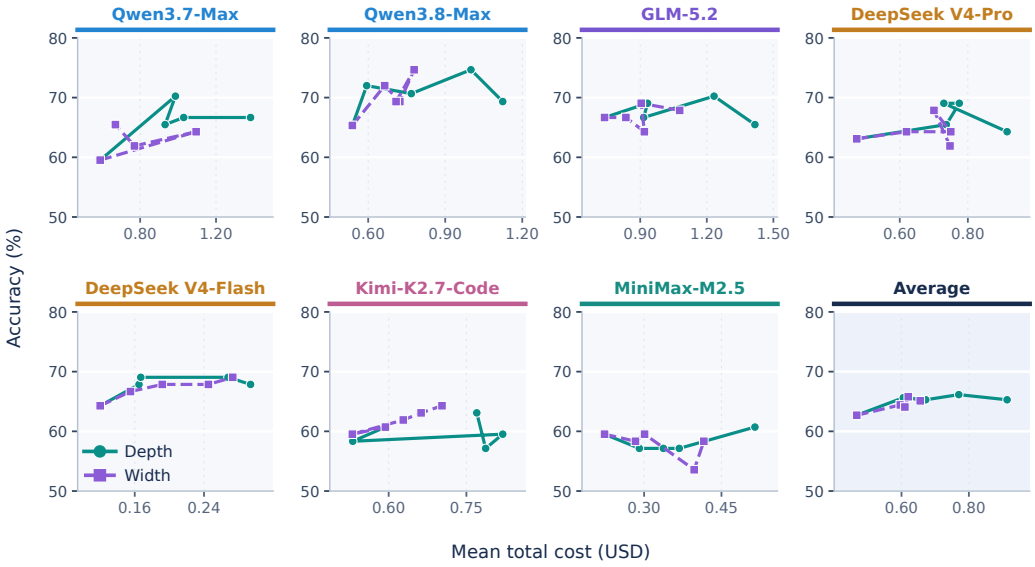


Figure 23: TB accuracy versus mean total cost as review depth and width increase.

B.5 L0: Output tokens and agent turns

Table 6 reports mean output tokens and agent turns relative to baseline on TB and SWE. Values above one indicate more generated output or longer interactions.

Table 6: Output tokens and agent turns for initial verification instructions, expressed as multiples of baseline.

Metric	Qwen 3.7-Max	Qwen 3.8-Max	GLM 5.2	DeepSeek V4-Pro	DeepSeek V4-Flash	Kimi K2.7 Code	MiniMax M2.5	Average
Terminal-Bench: System								
# output tokens	0.84	1.01	1.11	1.04	0.69	1.06	1.12	0.92
# agent turns	1.06	0.95	0.96	1.04	0.94	0.80	0.95	0.94
Terminal-Bench: User								
# output tokens	0.95	0.98	1.18	1.04	0.56	1.12	1.05	0.90
# agent turns	1.17	0.96	0.90	1.00	0.99	0.79	1.07	0.96
SWE-bench Verified: System								
# output tokens	1.14	0.97	0.93	1.08	0.94	1.06	1.02	1.00
# agent turns	1.05	0.98	0.97	1.02	0.93	0.91	0.94	0.95
SWE-bench Verified: User								
# output tokens	1.10	0.84	0.97	0.98	1.00	1.03	0.98	0.94
# agent turns	1.16	0.95	0.97	0.93	0.97	1.04	0.91	0.99

B.6 L1: CHECKING DURING EXECUTION

Table 7 reports how often TB baseline agents check their work after editing. We count recognized checking commands before the next editing turn or the recorded run end.

Table 7: Editing turns with recognized checks in TB baseline runs (%).

Model	Later turn	Same or later turn
Qwen3.7-Max	9.7	10.3
Qwen3.8-Max	8.7	16.5
GLM-5.2	11.0	12.3
DeepSeek V4-Pro	6.2	9.5
DeepSeek V4-Flash	10.5	13.3
Kimi-K2.7-Code	6.8	7.3
MiniMax-M2.5	13.6	13.8

Table 8 reports reminder coverage and checking or execution within the next two agent turns. After-edit user reminders reach 80–100% of runs, with recognized checks following 58–72% of reminder events.

Table 8: Reminder coverage and subsequent checks on Terminal-Bench.

Diagnostic	Qwen 3.7-Max	Qwen 3.8-Max	GLM 5.2	DeepSeek V4-Pro	DeepSeek V4-Flash	Kimi K2.7 Code	MiniMax M2.5
Periodic reminder (system)							
Runs receiving a reminder (%)	73	89	82	67	86	66	87
Reached the four-reminder cap (%)	62	55	65	59	72	64	74
Check within two turns (%)	70	81	67	75	66	65	71
Execution within two turns (%)	92	96	95	91	87	90	91
After-edit reminder (user)							
Runs receiving a reminder (%)	90	100	94	80	100	80	96
Reached the four-reminder cap (%)	55	75	81	63	71	70	65
Check within two turns (%)	67	60	72	69	61	62	58
Execution within two turns (%)	92	93	97	91	90	95	92

B.7 L2: WORK AFTER A COMPLETION GATE

Table 9 reports subsequent steps and file-changing actions after L2 interventions. User rejection is followed by 12–19 additional steps at the model median and file-changing actions in 86–96% of runs where it is triggered.

Table 9: Completion-gate coverage and subsequent work on Terminal-Bench.

Diagnostic	Qwen 3.7-Max	Qwen 3.8-Max	GLM 5.2	DeepSeek V4-Pro	DeepSeek V4-Flash	Kimi K2.7 Code	MiniMax M2.5
Inspect changes (automated)							
Runs reaching the gate (%)	88	83	89	81	66	65	89
Steps after the gate (median)	4	4	6	6	5	4	5
File-changing action after gate (%)	38	53	66	52	62	49	53
Recheck before finishing (user)							
Runs reaching the gate (%)	92	84	93	79	87	50	92
Steps after the gate (median)	6	6	8	8	8	4	5
File-changing action after gate (%)	66	95	87	76	85	60	57
Reject completion (user)							
Runs reaching the gate (%)	90	80	84	57	88	52	87
Steps after the gate (median)	14	12	19	16	16	15	19
File-changing action after gate (%)	88	92	94	88	92	96	86

C VISUALBENCH TASKS AND HUMAN EVALUATION

VisualBench contains 18 tasks spanning games, shaders, interactive three-dimensional scenes, applications, and other visual artifacts.

C.1 COMPLETE TASK CATALOG

Tables 10 and 11 summarize all 18 tasks and their requested deliverables. Full task descriptions and task-specific interaction instructions follow the overview.

Table 10: VisualBench tasks and requested deliverables, part 1.

Task	Core requirements and deliverables
Blender desk scene `blender-object` 3D	Construct a desk lamp and supporting objects from geometry, with distinct materials, deliberate lighting, shadows, and depth of field. Render a composed image and a lit, full-rotation video of at least four seconds using Cycles. *Deliverables:* `render.png`, `turntable.mp4`.
Breakout `game-breakout` Game	Build a playable brick-breaking game with paddle-dependent bounce and spin, multiple brick types, stacking power-ups, synthesized sound, combo scoring, and at least four levels. Include lives, pause, restart, and a persistent high score. *Deliverables:* `index.html`.
Tower defense `game-tower` Game	Implement escalating enemy waves and at least four tower types with upgrades, selling, range displays, and placement constraints. Include varied enemies, an early-wave bonus, money and lives, speed control, pause, and win/loss states. *Deliverables:* `index.html`.
Voxel sandbox `game-voxel` Game	Generate streamed terrain with caves, water, and distinct block types. Support first-person movement and collision, mining, inventory, placement, falling blocks, crafting, day/night lighting, and persistence of the player position and world edits. *Deliverables:* `index.html`, `three.min.js`.
Stormy ocean `shader-scene` Shader	Write a WebGL/GLSL ocean shader with waves at multiple scales, persistent foam, scattering and reflections, moving clouds, rain, and lightning that relights the scene. Support mouse-controlled viewing and smooth full-screen rendering without loaded textures. *Deliverables:* `index.html`.
Animated concert poster `svg-poster` Vector	Design a synthwave poster with the specified event text, deliberate typography, layered depth, and lighting effects. Combine animations at different speeds, including path-driven motion, in a seamless loop; keep the SVG self-contained and scalable. *Deliverables:* `poster.svg`.
3D lamp configurator `three-config` 3D	Construct an articulated lamp with live controls for shade shape, geometry, color, finish, bulb warmth, and brightness. Provide smooth transitions, an updated price and summary, saved settings, a shareable configuration URL, orbit controls, and PNG export. *Deliverables:* `index.html`, `three.min.js`.
Product hero scene `three-scene` 3D	Model wireless headphones from primitives with studio lighting and distinct materials. Add damped orbit and zoom, idle animation, smooth color changes, a labeled exploded view, and clickable hotspots that focus the camera on product details. *Deliverables:* `index.html`, `three.min.js`.
Fleet terminal dashboard `tui-dashboard` Terminal UI	Build a live Python terminal interface from the supplied service data, with status, sparklines, overall health, scrolling logs, severity colors, and adaptive layout. Record at least 30 seconds of updates including a service status transition. *Deliverables:* `demo.cast`, `screen.svg`.

Table 11: VisualBench tasks and requested deliverables, part 2.

Task	Core requirements and deliverables
Interactive cloth `cloth-sim` 3D	Implement cloth physics without a physics library: grabbing, unpinning, wind, tearing, and stable draping over a sphere. Expose simulation controls, live energy and constraint error, and structural and shaded views; remain stable under aggressive input. *Deliverables:* `index.html`.
Flowline kanban `data-app-kanban` Application	Create a populated board with editable cards, drag-and-drop, bulk actions, composed filters, and an enforced three-card work-in-progress limit. Support full keyboard operation, undo/redo, persistent board state, and JSON import/export. *Deliverables:* `index.html`.
Sales data explorer `dataviz-explorer` Data visualization	Visualize roughly 5,500 messy sales records across six regions and two channels. Enable trend and regional comparisons, repair recoverable data, explain every excluded row, and represent missing values honestly. Choose the views and implement them without charting libraries. *Deliverables:* `index.html`.
2D platformer `game-platformer` Game	Use the two supplied tile maps to build responsive movement with variable jump height and coyote time. Include a following camera, hazards and respawning, collectible coins, progression between levels, a win screen, and restart. *Deliverables:* `index.html`.
Generative art series `generative-art` Generative art	Create a visually coherent series with varied compositions and generated palettes. Make each seed deterministic and its composition independent of window size; provide seed navigation, progressive rendering, and PNG export rendered at least three times screen resolution. *Deliverables:* `index.html`.
Vector boolean editor `path-boolean` Vector	Compute union, intersection, difference, and exclusion on editable cubic Bezier paths without geometry libraries or masking shortcuts. Handle tangencies, overlaps, holes, and disjoint results; support chained operations, geometry diagnostics, undo/redo, and SVG export. *Deliverables:* `index.html`.
City in a storm `shader-city` Shader	Raymarch an endless nighttime skyline above moving reflective water using raw WebGL/GLSL. Add varied towers and windows, moving clouds, fog, rain, and spatially located lightning that relights the scene; support camera motion at a smooth frame rate. *Deliverables:* `index.html`.
Gridwork spreadsheet `spreadsheet-engine` Application	Build a grid of at least 40 by 20 cells with formulas, ranges, standard functions, and dependency-based recalculation. Handle cycles and invalid inputs without hanging; support fill-down, sorting, undo/redo, and persistence. *Deliverables:* `index.html`.
GPU particle field `three-particles` 3D	Simulate one million particles on the GPU with evolving curl-noise flow, momentum, trails, and mouse attraction/repulsion. Convey depth and flow through brightness and color; recycle particles, show live count and frame rate, and handle resize without resetting. *Deliverables:* `index.html`, `three.min.js` (optional).

The following entries reproduce all 18 English task briefs and their task-specific rater instructions from the evaluation kit. Line wrapping and dash typography are normalized for typesetting. The task descriptions are supplied to the agent; the interaction instructions guide human assessment of the delivered artifacts.

C.1.1 BLENDER DESK SCENE

`blender-object`

Task description

> Using Blender, build a small desk scene in a script - geometry you construct yourself, no imported assets. An articulated desk lamp is the subject: base, jointed arm, shade, with the shade actually housing a light so the lamp lights the scene rather than being lit by it. Put it on a surface with a couple of supporting objects - a mug, a stack of books, something with a different silhouette - arranged as a composition rather than a row.
>
> Materials should differ and read as materials: brushed metal on the lamp, glaze on the mug, paper on the books, with a surface underneath that isn't the default grey. Light it deliberately - the lamp as the key, something soft filling the shadows, and a rim to separate the subject from the background. I want real shadows and some depth of field.
>
> Render with Cycles; EEVEE is unavailable in this headless environment. Watch your sample count and resolution so the render actually finishes.
>
> Deliver, in your working directory: **`render.png`** (one composed beauty shot) and **`turntable.mp4`** (a smooth full 360°, four seconds or more, with the lamp lit throughout). Optionally also export the model as `model.glb`.

Human interaction instructions

1. First, the objects in render.png: an articulated desk lamp - base, jointed arm, shade - with a mug and a stack of books beside it, arranged as a composition rather than a row? If the objects are wrong, the lighting below does not count
2. Look at render.png - lamp lights the scene, shadows and depth of field
3. Play turntable.mp4 - full rotation, lamp lit throughout

C.1.2 INTERACTIVE CLOTH

`cloth-sim`

Task description

Build me a piece of cloth I can grab and throw around, simulated properly - raw canvas or WebGL that you write yourself, no physics library, no engine.

A rectangular sheet, pinned along its top edge, hanging under gravity. It should behave like cloth and not like a net of rubber bands: it holds its area, folds rather than stretching, and the folds run the way gravity and tension put them. I want to grab any point with the mouse and drag the sheet around, let go and watch it settle, and unpin a corner and watch it fall.

Put something in the way - a sphere is enough - and let the cloth **drape over it and stay outside it**, resting the way a sheet rests, not sinking through and not jittering on the surface. Wind I can turn up and down, so the sheet lifts and ripples, and a control to cut or tear it.

It has to be stable. Grab a point and shake it hard: the sheet must not detonate, and it must come back to rest. Show me the numbers that prove it - total energy and the constraint error on screen, updating live - so I can see the system settle rather than take your word for it, and it must hold a smooth framerate at a useful resolution while it does.

Let me change the resolution of the sheet, the stiffness, the gravity and the damping while it runs, and reset it. Give me a way to see the structure - the particles and constraints - as well as the shaded surface, and make the shaded version read as fabric with light on it, not a flat polygon soup.

Deliver exactly one self-contained **`index.html`** in your working directory - runs offline.

Human interaction instructions

1. First: is it a rectangular sheet pinned along its top edge hanging under gravity, with the sphere in the scene and live energy / constraint-error readouts on screen? No sheet, no sphere or no readouts - nothing below counts
2. Grab a point and shake it hard, then let go - does it settle, or does the sheet detonate? Explosion on hard input is an outright fail
3. Watch the energy and constraint-error readouts while it settles: energy should fall and the error should go toward zero. If the numbers are missing or frozen, the claim is unsupported
4. Drape it over the sphere - does it rest ON the surface, or sink through it, or jitter?
5. Does it fold, or does it stretch like rubber? Pull one corner far away and see whether the sheet grows
6. Unpin a corner; turn the wind up and down; cut or tear it; change stiffness, gravity and damping while it runs, then reset
7. Raise the resolution - does it stay stable and keep its framerate, or does higher resolution make it unstable?
8. Toggle the structure view - are particles and constraints actually there? Then look at the shaded view: does it read as fabric with light on it?

C.1.3 FLOWLINE KANBAN

`data-app-kanban`

Task description

Build **Flowline**, a kanban board I would actually run a team's work on - not a three-column demo.

It opens on a **populated board**: enough realistic cards spread across Backlog, In Progress and Done that everything below can be tried immediately, with In Progress already at its WIP limit so the limit is visible without me building up to it. Nobody should have to type in test data to find out whether the app works.

Cards carry a title, a note, a label or two, an assignee and a due date, and show what they carry without being opened. I want to create, edit in place, and delete them; drag them between and within columns; and select several at once and move, label or delete the lot in one action. Overdue and due-soon should be obvious at a glance.

Give me a filter box that searches titles, notes, labels and assignees together, and lets me narrow by label or assignee as well - composing with the text, not replacing it. Column headers carry live counts that follow the filter. **In Progress holds at most 3**: the board refuses the fourth visibly and says why, including on a bulk move that would breach it.

Undo and redo (Ctrl+Z / Ctrl+Y and visible controls) cover every mutation - add, edit, delete, drag, and bulk operations as single steps rather than one step per card - and a new action clears the redo stack. I should be able to work the whole board **from the keyboard alone**, moving a card between columns without touching the mouse.

Everything survives a reload: cards, order, labels, filters, and which columns I collapsed (localStorage). Let me export the board as JSON and import it back.

Deliver exactly one self-contained **`index.html`** in your working directory - no libraries, works offline, zero external requests. How it looks and how it is laid out are yours to decide.

Human interaction instructions

1. Open it cold - is it the Flowline board with Backlog / In Progress / Done, already populated with realistic cards (title, note, labels, assignee, due date visible without opening them), and In Progress already at its WIP limit? An empty board means you cannot test anything without first typing data in - nothing below counts
2. Add, edit in place, and delete a card - all three work, and column counts stay correct
3. Drag a card between and within columns - the order follows
4. Drag a fourth card into In Progress - refused visibly, with a reason given
5. Click the board first so it has focus. Select several cards and move or delete them in one go, then press Ctrl+Z ONCE - the whole batch comes back as one step; then try Ctrl+Z / Ctrl+Y on an add, an edit and a drag
6. Put the mouse away: can you move a card between columns using only the keyboard?
7. Type in the filter, then also narrow by a label or assignee - they compose; counts follow
8. Check due dates: is overdue vs due-soon obvious without opening a card?
9. Reload the page (F5) - cards, order, labels, filter and collapsed columns all survive

C.1.4 SALES DATA EXPLORER

`dataviz-explorer`

Task description

`data.csv` in your working directory holds about 5,500 rows of sales data - `date, region, online, retail`, six regions, roughly thirty months, in no particular order. It is messy on purpose: mixed date formats, stray whitespace, thousands separators, duplicated rows, impossible dates, non-numeric and missing values.

Build **`index.html`**: a page that a commercial analyst opens to understand how the online and retail channels have moved over those thirty months, and how that differs by region - where the mix turned, which regions lead or lag, what is seasonal and what is a real trend. They should be able to reach an answer by using the page, not by reading the CSV.

Be honest about the data. Repair what is repairable, exclude only what is genuinely unusable, and show the excluded-row count where it cannot be missed; opening it lists exactly those rows, each with its reason. A value missing from only one series is not an exclusion - that series breaks at that point, it does not drop to zero. Between two otherwise equal pages, the one that repairs more and discards less is the better one.

How to present any of this is deliberately unspecified - choosing the views, the comparisons and the controls is the task, and there is no required set of charts or widgets. Draw everything yourself; no charting libraries. It must stay responsive at full data size and must not mislead: truncated axes, hidden aggregation and silently dropped rows all count against it.

Deliver exactly one self-contained **`index.html`** in your working directory - zero external requests.

Human interaction instructions

1. First: is it showing that dataset - six regions, about thirty months, both the online and retail channels? If not, nothing below counts
2. Give yourself two minutes: which region's online/retail mix shifted the most, and roughly when did it turn? Could you answer from the page alone?
3. Try to compare two regions directly - does the page let you, or do you have to remember one while looking at the other?
4. Separate seasonality from trend - is there any way to tell a recurring dip from a real decline?
5. Click the excluded-row count: the panel length must match the count, every row must carry a reason, and the rows must genuinely be unusable (spot-check a few)
6. Between two candidates, prefer the one that repairs more and excludes less
7. Look for a break in a line - a single missing value must render as a gap, never as zero
8. Check it does not mislead: axes that start somewhere other than zero without saying so, aggregation you cannot identify, or rows dropped silently

C.1.5 BREAKOUT

`game-breakout`

Task description

Build me a breakout game that's actually satisfying to play. Paddle on mouse and arrow keys, ball reflecting off walls, paddle and bricks, with the bounce angle depending on where it hits the paddle and the paddle imparting some spin if it's moving.

I want it to feel good: the ball should accelerate as a level goes on, bricks should break with some particle debris and a screen-shake on the heavier hits, and the sound of it - synthesised in the browser, no audio files - should pitch up as the rally continues. Several brick types: plain, two-hit, armoured ones that need a power-up to crack, and unbreakable blocks that shape the level.

Power-ups drop and are caught with the paddle - multi-ball, a wider paddle, a sticky paddle that catches and re-launches, a laser that fires up the screen - and they should stack and expire, with what's active visible. A life is lost only when the last ball is gone. Three lives, score with a combo multiplier that builds while you don't lose the ball, and a high score kept across reloads.

At least four levels with genuinely different layouts, advancing when cleared, a game-over screen and restart, and pause.

Deliver exactly one self-contained **`index.html`** in your working directory - runs offline.

Human interaction instructions

1. First: is it breakout - paddle, ball and a brick field, paddle on both mouse and arrow keys, and several brick kinds (plain, two-hit, armoured, unbreakable)? If not, nothing below counts
2. Play a level - listen for the pitch rising through a rally
3. Catch a power-up; check what's active is shown
4. Lose all balls, then restart; reload the page (F5) and check the high score is still there

C.1.6 2D PLATFORMER

`game-platformer`

Task description

Build a 2D platformer. The two levels are in `levels.json` in your working directory (`#` solid, `.` empty, `^` hazard, `o` coin, `S` start, `E` exit) - embed the data. I want responsive run-and-jump physics with variable jump height and coyote time, a camera that follows smoothly through levels larger than the screen, hazards that kill and respawn me at the level start, collectible coins with a counter, the exit advancing to level 2 and then a win screen, and restart from anywhere.

Deliver exactly one self-contained **`index.html`** in your working directory - runs offline.

Human interaction instructions

1. First: is it a 2D platformer - terrain, hazards, coins, a start and an exit, in a level larger than the screen? If not, nothing below counts
2. Tap vs hold jump - the height varies; jump just after a ledge - coyote time catches it
3. Run ahead - the camera follows smoothly
4. Touch a hazard - death, then respawn at the level start
5. Collect coins and reach the exit - counter, level 2, then a win screen

C.1.7 TOWER DEFENSE

`game-tower`

Task description

Build me a tower-defense game with enough going on to be worth replaying. Enemies come in waves along a visible path; I place towers by clicking, limited by money, and I shouldn't be able to place one on the path or on top of another.

Give me at least four tower types that play differently - something single-target and fast, something slow and heavy, something that slows what it hits, something that hits an area - each upgradable through a few levels with visibly stronger effects and a sell option that refunds part of the cost. Clicking a tower shows its range and stats. Several enemy types too: fast and fragile, slow and armoured, one that heals others near it, and a boss on later waves, with health bars and enemies that visibly react when hit.

Waves should escalate and I want to be able to call the next one early for a bonus rather than waiting. Money from kills, lives lost when something leaks through, a HUD with wave, health, money and what's coming next. Speed-up and pause controls. Win after the final wave, lose at zero lives, restart from either.

Make it readable at a glance - I should be able to tell what's happening in a busy wave.

Deliver exactly one self-contained **`index.html`** in your working directory - runs offline.

Human interaction instructions

1. First: is it tower defense - a visible enemy path, towers placed by clicking (refused on the path or on another tower), at least four tower types, a HUD with wave / lives / money? If not, nothing below counts
2. Place different tower types; click one to see range and stats
3. Upgrade a tower, then sell one
4. Call the next wave early; use pause and speed-up

C.1.8 VOXEL SANDBOX

`game-voxel`

Task description

Build me a browser voxel game - a real Minecraft, not a block-placing demo. `three.min.js` is in your working directory; ship it next to your page and load it locally.

The world should be generated and effectively endless: chunked terrain streamed in as I walk, built from noise so there are hills, valleys, beaches and overhangs, with caves worth finding underground. Water should sit in the low ground and read as water, not a blue cube. Trees, sand, stone, dirt, grass, ore - enough variety that a landscape looks like somewhere, and each type visually distinct even without textures.

I want to play it. First-person, WASD and mouse-look under pointer lock, with gravity, jumping and collision that stops me walking through solid blocks - I should be able to fall off a cliff and land. Breaking should take a moment rather than being instant, with the targeted block outlined, and what I break goes into an inventory I can open rather than vanishing. Placing puts the held block where I'm aiming. Sand and gravel fall when I dig out what's under them. A hotbar along the bottom with the selected slot obvious, switchable with the number keys and the scroll wheel.

Give me a day/night cycle where the sun moves and the light changes with it, and make the lighting deliberate: block faces lit differently by orientation, caves genuinely dark, and some placeable light source that pushes the dark back. Some crafting from what I've gathered - even a handful of recipes - and a way to see them.

It should hold a smooth framerate while I move, so think about how much geometry you send per chunk. And my world should survive a reload: come back where I left it, with the blocks I changed still changed.

Deliver, in your working directory: **`index.html`** plus the local **`three.min.js`** it loads - runs offline, no network.

Human interaction instructions

1. First: is it a voxel world - rolling terrain with hills, valleys and beaches, water in the low ground that reads as water, and visibly distinct block types (trees, sand, stone, dirt, grass, ore)? If not, nothing below counts
2. Click the canvas - pointer locks; WASD, mouse-look, space to jump
3. Walk a while - new terrain streams in; look for caves and water
4. Hold left click to break, right click to place; open the inventory
5. Dig out the block under sand; wait for night; try a torch and the crafting screen
6. Reload the page (F5) - the world and your changes should still be there

C.1.9 GENERATIVE ART SERIES

`generative-art`

Task description

Build a generative system that makes artwork I would actually hang - and prove it by producing a **series**, not one picture.

`?seed=N` in the URL selects a piece: same seed, same image, down to the pixel, and the same image at any window size rather than reflowing into a different picture. The page shows one piece at a time, with a way to move through the series and to jump to a given seed.

What I am judging is whether the pieces read as **one body of work**. Someone shown three of them should recognise the same hand at once, while no two are the same composition rearranged. That means your system has to separate what makes the *style* from what makes the *instance* - and getting that separation right is the task.

Each piece needs **composition, not texture**: something the eye goes to, empty space that is doing work rather than left over, and a reason the frame ends where it does. An even field of marks filling the canvas is exactly what I do not want, however fine the marks are.

What the system *is* - what it draws with, what it depicts - is deliberately unspecified. Flow fields, circle packing, reaction-diffusion, subdivision, physical simulation, collage of primitives, something else: choosing is part of what is being judged, and the first idea is rarely the strongest. Marks should read as a medium, with pigment that varies along its length, pools where marks cross and thins as it runs out - not uniform hairlines. Generate the palette too, harmonically related rather than random, one deliberate palette per piece.

Let me export the piece I am looking at as a PNG at print resolution - at least 3x the screen, and **rendered** at that size rather than upscaled from what is on screen. It should draw progressively and stay responsive while it works.

Deliver exactly one self-contained **`index.html`** in your working directory - no libraries, runs offline.

Human interaction instructions

1. First: does the page show one piece at a time, with next / jump-to-seed controls? And an even mat of marks filling the whole canvas fails this task however fine the marks are - check that before anything else
2. Use the on-page next / jump-to-seed controls: go to a seed, note the picture, move away and come back - it must be exactly the same picture, not a reflowed one
3. Step through three different seeds. Do they read as one body of work - same hand - while being genuinely different compositions?
4. The failure to look for is the opposite pair: either all three are the same picture with the parts moved, or they have nothing in common but the palette
5. Take one piece: is there something the eye goes to? Is the empty space doing work, or is it just where the algorithm stopped?
6. Do the marks read as a medium - varying density, pooling where they cross, thinning out - or as uniform hairlines?
7. Look closely at the edges of the marks (use "open full ↗" for a bigger view) - still crisp, or mushy? The piece should fill the whole canvas, no blank strip or letterbox
8. Watch it draw - progressive, and the page still responds while it works?

C.1.10 VECTOR BOOLEAN EDITOR

`path-boolean`

Task description

Build a small vector editor whose real job is **boolean operations on paths** - union, intersection, difference and exclusion - computed by you. No geometry library, no SVG trickery that fakes the result with clip paths or even-odd fill: the operation must produce a *new path*, whose outline I can then edit and operate on again.

I need to draw closed paths - a pen tool placing points, rectangles and ellipses are fine as starting shapes - with **cubic Bézier segments**, not just straight lines. Select two shapes, pick an operation, and get the result as a single editable path with its handles intact. The result of an operation must be usable as the input to the next one.

Getting this right is mostly about the cases that are not the easy case, and that is the point: curves that touch without crossing, segments that overlap along their length, a shape entirely inside another, results that come out as several disjoint pieces, and shapes with holes - where the hole has to survive the operation and stay a hole. Self-intersecting input should not produce garbage.

Show me what it is doing: the intersection points it found, and the winding direction of each contour, on a toggle. Let me nudge a point after an operation and see the outline follow. Undo and redo the whole way back. Let me export the result as SVG, and have that file open correctly elsewhere.

Deliver exactly one self-contained **`index.html`** in your working directory - no libraries, zero external requests.

Human interaction instructions

1. First: is it a vector editor - closed paths drawn with a pen tool, rectangles and ellipses, cubic Bézier segments, and all four operations (union, intersection, difference, exclusion) available? If not, nothing below counts
2. Union two overlapping shapes, then run difference on the RESULT - an operation whose output cannot be operated on again fails the task
3. Nudge a point after an operation: does the outline follow, with handles intact? If the result is not editable it was faked
4. Difference a small shape out of a big one so the result has a HOLE - then move the outer shape and check the hole is still a hole
5. The two classic degenerate cases: shapes that only TOUCH (tangent, no crossing), and shapes sharing a straight edge along its length - no garbage, no dropped contour
6. An operation whose result is several disjoint pieces - are they all there, as one path?
7. Use curves, not just polygons: draw with Bézier segments and check the result curves are still curves, not polylines
8. Toggle the intersection-point and winding-direction overlay - do the points sit where curves actually cross?

C.1.11 CITY IN A STORM

`shader-city`

Task description

Build me a city at night in a storm, seen from out over the water - raw WebGL and GLSL you write yourself, no libraries, no loaded textures.

The city should be raymarched and endless: towers of varying height and footprint receding into haze, close enough that I can see the ones in front have depth and silhouette against the ones behind. Windows lit unevenly, some warm, some dark, a few flickering, so it reads as inhabited rather than as a texture. The whole thing rises out of moving water that reflects it - long broken reflections that stretch and break with the swell, not a mirror.

Weather should be doing something. Rain across the view, low cloud moving through and around the towers so the skyline is sometimes obscured, and fog that thickens with distance. Lightning fires periodically and genuinely relights everything for that instant - towers, cloud, and the water's reflection together - with the light coming from where the strike is, not a uniform white flash.

Slow camera drift, and the mouse moves my viewpoint. Full-screen at any window size, holding a smooth framerate - be careful how many steps you spend per pixel.

Deliver exactly one self-contained **`index.html`** in your working directory - runs offline.

Human interaction instructions

1. First: is it that city - night, storm, seen from over the water: towers of varying height receding, windows lit unevenly, the city rising out of moving water with reflections, rain and low cloud? If not, nothing below counts
2. Wait for lightning - towers, cloud and reflection light together
3. Look at the water - reflections stretch and break with the swell
4. Move the mouse to swing the viewpoint; watch fog with distance

C.1.12 STORMY OCEAN

`shader-scene`

Task description

Build me a full-screen ocean in a storm - raw WebGL and GLSL you write yourself, no libraries, no textures loaded from anywhere. I want it to look like real open water on a bad night, not a blue sine wave.

The surface should be built from a proper spectrum of waves travelling in different directions at different scales, so the swell has long rollers with smaller chop riding on top and crests that sharpen and peak rather than staying round. Steep crests should tear into foam, and that foam should drift and dissipate rather than flicker on and off. Water is not opaque: let light scatter through the thin backs of waves so a crest glows when it's between me and the sun, reflect the sky off the surface at grazing angles and let me see into it when I look straight down, and give me the long broken glitter path a low sun leaves on moving water.

Overhead, clouds that move and have depth to them, not a flat scrolling texture. The storm should breathe - build and ease off over time rather than sitting at one intensity - and when lightning fires it should actually relight the scene for that instant, water and cloud together, not just flash the screen white. Rain across the view, and where it meets the water it should disturb it.

The mouse moves my viewpoint. It should fill the window at any size and hold a smooth framerate full-screen, so be careful how much you ask of each pixel.

Deliver exactly one self-contained **`index.html`** in your working directory - runs offline.

Human interaction instructions

1. First: is it open water in a storm - long rollers with chop riding on top, foam on the crests, clouds with depth, rain - not a blue sine wave? If not, nothing below counts
2. Watch thirty seconds - the storm builds and eases, lightning relights the scene
3. Find a crest between you and the sun - look for light through the wave
4. Move the mouse to swing the viewpoint; look straight down, then to the horizon
5. The scene should fill the whole canvas with no black bars; while it runs (and while you drag or move), it should stay smooth, not stutter

C.1.13 GRIDWORK SPREADSHEET

`spreadsheet-engine`

Task description

Build **Gridwork**, a working spreadsheet. Not a styled grid of inputs: a calculation engine with a grid on top of it.

At least 40 rows by 20 columns, addressed A1-style, with click and arrow-key navigation, typing straight into a cell, editing an existing value, and a formula bar showing the underlying formula rather than the displayed result.

A cell beginning with = is a formula over numbers, cell references, ranges like `B2:B15`, parentheses and the usual arithmetic, plus at least `SUM`, `AVERAGE`, `MIN`, `MAX`, `COUNT`, `IF` and `ROUND`. Formulas refer to other formulas, to any depth. **When a value changes, everything downstream of it updates and nothing else is recomputed** - on a sheet with hundreds of live formulas that difference is visible, so do not recompute the world on every keystroke.

Wrong input is where spreadsheets earn their keep. **A reference cycle must be reported as a cycle, in the cells that form it, and must not hang or blow the stack.** Division by zero, a reference to an empty cell, text where a number is required, a malformed formula, a range that runs off the sheet - each gets an error the user can act on, errors propagate to dependents rather than being swallowed, and no bad input may take the sheet down.

I also want to select a range and see its count and sum without writing a formula; fill a formula down a column with references adjusting as they go; sort a range; undo and redo anything; and have the sheet still be there after a reload.

Deliver exactly one self-contained **`index.html`** in your working directory - no libraries, zero external requests. How it looks is yours to decide.

Human interaction instructions

1. First: is it a spreadsheet - at least 40 rows by 20 columns, A1-style addresses, a formula bar, arrow-key navigation, typing straight into a cell? If not, nothing below counts
2. Formulas chain: type 5 in B1, =B1*2 in C1, =C1+1 in D1 → they show 10 and 11; change B1 to 7 → C1 and D1 become 14 and 15 by themselves
3. Built-in functions: put 1, 2, 3 in A1-A3; A5 =SUM(A1:A3) → 6; A6 =IF(A5>5,"big","small") → big
4. Errors: E1 =1/0 and E2 =FOO(1) → both show a readable error and the sheet stays up; F1 =E1+1 → the error propagates instead of quietly becoming 0
5. Cycle: G1 =H1 and H1 =G1 → both cells report a circular reference; the page must not hang or freeze
6. Fill, range, sort: fill C1 down to C10 → references become B2, B3...; select A1:A3 → the status bar shows count 3, sum 6; sort a column and check the result
7. Undo and persistence: Ctrl+Z undoes one step, Ctrl+Y redoes; then reload the page (F5) → the sheet is still there

C.1.14 ANIMATED CONCERT POSTER

`svg-poster`

Task description

Design me a poster for a synthwave show - something I'd actually put on a wall. These strings appear exactly: `NEON TIDE`, `Live at Meridian Hall`, `Sep 12 2026`.

I want real poster typography, not centred text on a gradient: a deliberate type hierarchy with the band name as the composition's anchor, considered letter-spacing, and text that's part of the artwork - overlapping, masked by, or interacting with the scene rather than floating above it. Build depth with layers that read as foreground, middle and distance, and use gradients, blurs and blend modes to make light feel like it's in the scene - glow that spills, haze between layers, reflection where it belongs.

Several things should move, and move differently: slow drift in the distance, faster motion up close, something that pulses on its own rhythm, and at least one element whose animation is driven by a path rather than a simple translation. It should loop seamlessly - I shouldn't be able to spot the jump.

Everything lives in the file: generic font families only, no external references, and a proper viewBox so it's crisp from a thumbnail to a billboard.

Deliver exactly one standalone **`poster.svg`** in your working directory.

Human interaction instructions

1. First, check the three strings appear exactly: NEON TIDE / Live at Meridian Hall / Sep 12 2026; then whether it is a poster you would put on a wall rather than centred text on a gradient. If this fails, nothing below counts
2. Watch a full loop - look for the seam where it repeats
3. Zoom the browser to 300% - the type and edges stay crisp
4. Look for several things moving at different rates

C.1.15 3D LAMP CONFIGURATOR

`three-config`

Task description

Build me the configurator a lighting company would ship. Compose the lamp from primitives - base, arm, joints, shade - no model files; `three.min.js` is in your working directory, ship it next to your page and load it locally.

The panel drives it live: body colour from a palette, at least three shade shapes, arm height and arm angle, and the material finish - matte, gloss, metal - each visibly different under the lights. Switching the bulb on should actually light the scene, casting from inside the shade onto the surface below, with a warmth control that shifts the light's colour and a dimmer that changes its intensity, and the scene's mood following it.

Every change should transition rather than snap. A summary line always reflects the current configuration, along with a price that updates from the options chosen. Reset returns to defaults. Let me save a configuration and come back to it after a reload, and give me a shareable URL that encodes the current choices so opening it rebuilds that exact lamp.

Drag to orbit with damping, wheel to zoom, and a "Download PNG" that saves the current view at higher than screen resolution. Handle resize.

Deliver, in your working directory: **`index.html`** plus the local **`three.min.js`** it loads - runs offline.

Human interaction instructions

1. First: is it the lamp that was asked for - base, arm, joints and shade all there, at least three shade shapes to switch between, arm height and angle adjustable? If this fails, nothing below counts
2. Change shade, finish and colour - transitions, not snaps
3. Switch the bulb on; move the dimmer and warmth
4. Drag to orbit and let go - it should glide to a stop with damping; wheel to zoom; the view should fill the whole canvas with no black bars

C.1.16 GPU PARTICLE FIELD

`three-particles`

Task description

Build me a particle field big enough to be impressive - a million particles moving at an interactive framerate. That number means the positions have to live and update on the GPU; if you're stepping a million particles in JavaScript each frame you've already lost. `three.min.js` is in your working directory - ship it locally and use it, or go raw WebGL, your call.

I want the motion to look like a fluid, not like points on springs: a curl-noise flow field that evolves over time so the field itself drifts, with particles that have momentum and trail rather than snapping. The mouse should push into the flow - attract by default, repel on a key or control, with the influence falling off smoothly with distance instead of a hard radius.

Make it read as depth: particles further away smaller and dimmer, additive light where they pile up, colour that responds to speed or curl so the structure of the flow is visible. Particles should recycle when they drift out rather than the field emptying.

Give me a way to see it's real - a live particle count and framerate on screen. It must survive a window resize without rebuilding the world from scratch.

Deliver, in your working directory: **`index.html`** (plus the local **`three.min.js`** if you use it) - runs offline.

Human interaction instructions

1. First, the on-screen count and framerate: is it close to a million particles at a smooth rate, and does the motion read as a fluid (momentum, trails) rather than points on springs? If not, nothing below counts
2. Move the mouse through the field; toggle attract/repel
3. Watch a pile-up - brighter where particles gather
4. Watch it for a few seconds - it must keep running steadily, never freeze or stop, and fill the whole canvas

C.1.17 PRODUCT HERO SCENE

`three-scene`

Task description

Build me the hero scene a company would put at the top of its product page for a wireless headphone. Compose it from primitives - band, cups, cushions, hinges, the small details that make it read as a real object - no model files. `three.min.js` is in your working directory; ship it next to your page and load it locally.

It should look photographed, not rendered by default settings: a studio setup with a key, fill and rim light, soft shadows on the ground, and materials that actually differ - brushed metal on the band, soft matte on the cushions, a subtle sheen where the light grazes an edge. Give the frame a considered composition rather than an object centred in grey.

I want to interact with it. Drag to orbit and wheel to zoom, with the motion damped so it glides to a stop rather than halting. A gentle idle animation when I leave it alone, which yields the moment I touch it. Colour variants that switch the materials with a smooth transition, not a snap. An exploded-view toggle that separates the parts along sensible axes and reassembles them, with the parts labelled while apart. And hotspots on the object I can click to focus the camera on a detail and read a short note about it.

Handle window resize, and keep it smooth.

Deliver, in your working directory: **`index.html`** plus the local **`three.min.js`** it loads - runs offline.

Human interaction instructions

1. First, the object: a wireless headphone - band, two cups, cushions, hinges, all there and in believable proportion? If the object is wrong, nothing below counts
2. Drag to orbit, then let go - it glides to a stop
3. Switch colour variants; toggle the exploded view
4. Click a hotspot on the object
5. Leave it alone for a few seconds - idle animation resumes

C.1.18 FLEET TERMINAL DASHBOARD

`tui-dashboard`

Task description

Build me a terminal dashboard in Python (rich and textual are available) for the service fleet in `stats.json` in your working directory - something I'd leave open on a second monitor.

I want more than a table: the fleet listing with per-service status, but also sparklines showing each service's recent trend, an overall health summary that makes the worst thing on the fleet obvious at a glance, and a live log pane that keeps scrolling. Colour-code ok / warn / crit and make severity readable from across the room - a crit service should catch my eye without reading the text.

It should be a real interface, not a static frame: laid out in panes that use the terminal's full width, updating continuously with values that move plausibly rather than jumping randomly, sorted so problems surface to the top. If a service crosses into crit, that transition should be visible when it happens. Handle a narrow terminal without the layout collapsing into garbage.

Then record it running.

Deliver, in your working directory: **`demo.cast`** (an asciinema recording of at least 30 seconds showing live updates and at least one status transition) and **`screen.svg`** (one exported frame).

Human interaction instructions

1. First: is the recording a multi-pane terminal dashboard - a service list with status, a sparkline per service, an overall health summary, a scrolling log pane, and ok / warn / crit colour-coded? If not, nothing below counts
2. Play demo.cast - values move and the log scrolls
3. Watch for a service crossing into crit
4. Check sparklines and that problems sort to the top

C.2 HUMAN EVALUATION PROTOCOL

The human comparison covers seven designs. For L0–L2, we select one representative intervention per layer: the system verification instruction, the after-edit user reminder, and the user recheck request with the 80% early trigger. These use the same intervention definitions as TB and SWE. For L3, we compare changes-based and trajectory-based review, each with and without required visual inspection. Each design is compared with the baseline for the same task and coding model. Review depth and width are studied separately on Terminal-Bench.

Each task supplies a description and any supporting files without a reference implementation. Tasks run in an environment with Python, Chromium, and ffmpeg, with additional tools where required. Browser artifacts run offline, with bundled libraries only where the task permits them.

Raters and briefing. The raters are members of an internal company data team. Before rating, the interface presents a briefing explaining the study's purpose, the blinded comparison, the four response options, and a worked example of applying the criteria.

Consent, compensation, and ethics approval. The human evaluation received ethics approval. Raters provided informed consent and were compensated under their existing contracts with the company.

Blinded comparison. Raters compare artifacts for the same task and model with verification conditions hidden and sides randomized. The interface presents the original task brief, its judging criteria, and a short set of task-specific interaction steps. Both artifacts can be inspected side by side or opened individually at full size. Raters interact with live artifacts where available and inspect delivered recordings or renders where appropriate.

Instructions to raters. The rating interface gives the following guidance:

1. Read the original task brief before comparing the artifacts. Judge against what the task asks for, including its functional and visual requirements.
2. Follow the task-specific interaction checklist on both artifacts. Click, drag, type, or play as needed to check the requested behavior; use the full-size view when the side-by-side view is too small. For keyboard controls, first click inside the artifact to give it focus.
3. Choose A or B when one artifact better satisfies the task. Choose a tie when both work and you cannot distinguish their quality. Choose both fail only when neither makes a meaningful attempt at the task, such as when both are blank, broken, or unrelated to the request. An artifact that works poorly does not by itself justify both fail.
4. If only one side produces a usable deliverable, choose that side even if it is rough. If a page appears blank, refresh it once before judging. Visual polish should not outweigh missing functionality unless visual quality is itself the task's objective.
5. Optionally leave a short comment explaining the deciding issue, especially when the two artifacts are difficult to distinguish.

Example interaction checklist. For a Breakout game, raters move the paddle with the arrow keys, launch the ball, clear several bricks, and deliberately let the ball drop. These actions check paddle control, collisions, scoring, and the loss condition. The judgment follows the requested game behavior rather than a screenshot alone.

C.3 EXPERT EVALUATION INTERFACE

Figure 24 shows the English rating interface for the Blender desk-scene task, with rendered images and turntable videos available on both sides. Raters can expand the task brief and judging guidance, follow the task-specific instructions, and inspect the two artifacts side by side or at full size. Four response buttons record a preference for A or B, a tie, or failure of both artifacts; an optional comment records the reason for the judgment.

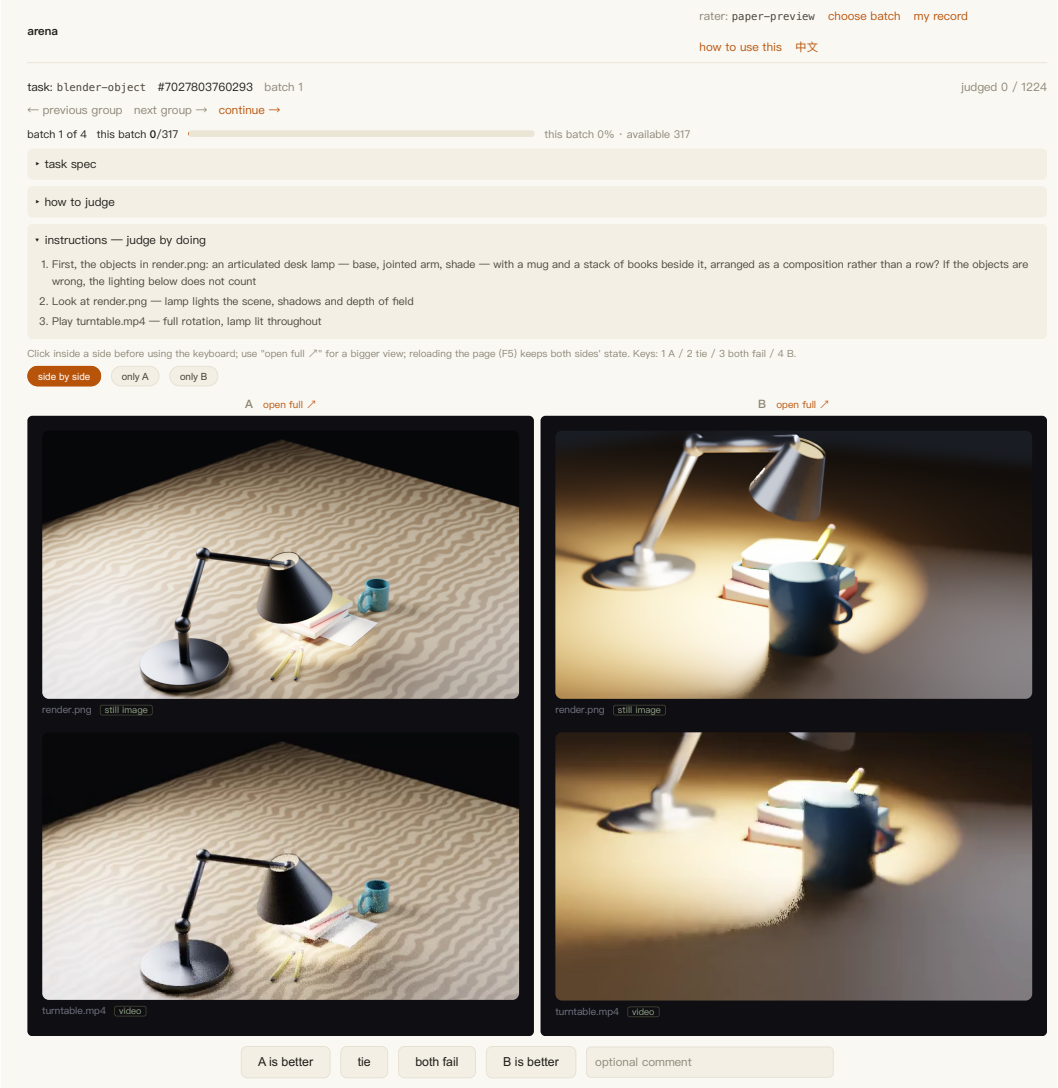


Figure 24: VisualBench's blinded evaluation interface showing two Blender deliverables, task-specific instructions, and rating controls.

C.4 DETAILED VISUALBENCH RESULTS

Table 12: VisualBench net preference (pp) across all evaluated designs and models. "+ visual" denotes required visual inspection.

Model	L0	L1	L2	L3: Independent review			
	System	Edit/User	Early	Changes	Changes + visual	Trajectory	Trajectory + visual
Qwen3.7-Max	+2.8	−12.5	−4.3	+8.3	+4.3	+2.9	+12.5
Qwen3.8-Max	−1.4	+2.8	+4.3	+82.8	+90.5	+71.1	+92.0
GLM-5.2	−8.3	−12.5	+9.4	+11.5	+18.2	+4.4	+12.5
DeepSeek V4-Pro	−10.0	−11.8	+8.9	+12.9	+19.7	+1.6	+27.4
DeepSeek V4-Flash	+5.6	−5.6	−8.0	+12.5	+4.8	+10.0	−3.6
Kimi-K2.7-Code	−8.3	+4.2	+1.5	+7.6	+15.2	−2.9	−6.5
MiniMax-M2.5	−2.8	+12.5	+5.7	+12.9	+14.2	+5.6	+16.2
All pairs	−3.2	−3.2	+2.7	+21.2	+23.8	+13.2	+21.6

L0 results. The system verification instruction yields a net preference of -3.2 points over the baseline, as shown in Table 12. DeepSeek V4-Flash and Qwen3.7-Max benefit ($+5.6$ and $+2.8$ points), while DeepSeek V4-Pro, GLM-5.2, and Kimi-K2.7-Code decline (-10.0, -8.3, and -8.3). Initial guidance therefore does not provide a general improvement on visual tasks.

L1 results. After-edit reminders with the user message role yield -3.2 points of net preference, as shown in Table 12. MiniMax-M2.5 gains $+12.5$ points, whereas GLM-5.2 and Qwen3.7-Max each lose 12.5 points.

L2 results. Early user rechecking yields a small positive net preference of $+2.7$ points, as shown in Table 12. GLM-5.2 and DeepSeek V4-Pro gain $+9.4$ and $+8.9$ points, while DeepSeek V4-Flash and Qwen3.7-Max decline by 8.0 and 4.3 points. These comparisons suggest a modest benefit from reopening work, with substantial variation across models.

L3 results. All four review designs have positive net preference, ranging from $+13.2$ to $+23.8$ points, as shown in Table 12. Changes-based review with required visual inspection performs best at $+23.8$ points, compared with $+21.2$ points without required visual inspection. Trajectory-based review improves from $+13.2$ to $+21.6$ points with required visual inspection. Changes-based review therefore receives higher net preference in both comparisons.

D CASE STUDY

D.1 OVERVIEW OF THE PAIRED TRAJECTORIES

We examine ten Terminal-Bench cases across the four verification layers, each paired with a baseline run of the same task and model, as shown in Table 13. Each case has a full-page paired trajectory with tool inputs, returned evidence, harness interventions, and final grading.

Table 13: Selected cases spanning all four layers; success and failure refer to final grading.

Layer	Successful verification run	Failed verification run
L0	C1: re-reading an edge-orientation rule	C2: endpoint checks, failed integration
L1	C3: deployment with live HTTP checks	C4: clean compile, invalid substitutions; C5: output equality without sufficient speedup
L2	C6: reopened checks reveal an order assumption	C7: state changes during repeated testing
L3	C8: counterexample followed by repair; C9: a boundary test guides numeric repair	C10: a diagnosis before the run deadline

CASE 1 L0 / INITIAL INSTRUCTIONS
MiniMax-M2.5 | bn-fit-modify

Added instruction | Check / repair | Mismatch / state removal

BASELINE	L0 VERIFICATION (USER)
Trial Vz6q2xM	Trial 5EVZuSK
BEFORE EXECUTION Baseline initialization. Source run: baseline.	**BEFORE EXECUTION / HARNESS** Run what you changed the way it really gets used (the command, route, or test someone would actually hit) and read the real output and exit code.
Inspect the saved output	

T19 / AGENT
Let me verify all the output files:
T19 / shell
`cat /app/learned_dag.csv`
T19 / RETURN
`to,from`
`R,U`
`M,U`
`D,U`
`Y,U`
`R,M`
`Y,D`
	T27 / shell
	`cat /app/learned_dag.csv`
	T27 / RETURN
	`to,from`
	`U,M`
	`U,Y`
	`U,D`
	`D,Y`
	`U,R`
	`M,R`
	T28 / AGENT
	Wait, there's still an issue.
	[...]
	D,Y means D → Y, but D < Y alphabetically, so D should be child. This is WRONG!

T20 / shell
`wc -l /app/final_bn_sample.csv`
T20 / RETURN
`10001 /app/final_bn_sample.csv`	

Check orientation and finish	

T26 / shell
`wc -l /app/final_bn_sample.csv && echo "---" && head -1 /app/bn_sample_10k.csv && head -1 /app/final_bn_sample.csv`
T26 / RETURN
`10001 /app/final_bn_sample.csv`
`---`
`"U","Y","R","D","M"`
`"U","Y","R","D","M"`
T27 / AGENT
Using conditional independence tests (partial correlations), I recovered the DAG with 6 edges:
[...]
M → R (M alphabetically comes before R, so M is parent)
[...]
D → Y (D alphabetically comes before Y, so D is parent)
	T28 / python
	`('Y', 'D'),  # Fixed: Y -> D`
	[...]
	`('R', 'M'),  # Fixed: R -> M`
	T28 / RETURN
	`NameError: name 'pd' is not defined`
	T29 / python
	`import pandas as pd`
	[...]
	`edges_df.to_csv('/app/learned_dag.csv', index=False)`
	T35 / python
	`parents_of_u = learned[learned['to'] == 'U']['from'].tolist()`
	[...]
	`print(f"Parents of U: {parents_of_u}")`
	T35 / RETURN
	`Parents of U: ['M', 'Y', 'D', 'R']`
	T38 / python
	`# Format: "to,from" where "to" is the child and "from" is the parent`
	[...]
	`('D', 'Y'),  # to=D (child), from=Y (parent) => Y -> D`
	[...]
	`('M', 'R'),  # to=M (child), from=R (parent) => R -> M`
	[...]
	`edges_df.to_csv('/app/learned_dag.csv', index=False)`
	T44 / RETURN
	`U has no parents: True`
	[...]
	`Y -> D`
	[...]
	`R -> M`

FINAL GRADER: FAIL test_learned_dag_structure	**FINAL GRADER: PASS** test_learned_dag_structure

Figure 25: Case 1 (L0): the baseline retains reversed edges, while the instructed run revisits the orientation rule and corrects the saved CSV.

CASE 2 L0 / INITIAL INSTRUCTIONS
Qwen3.7-Max | git-multibranch

Added instruction | Check / repair | Mismatch / state removal

BASELINE
Trial B7T9qtJ

BEFORE EXECUTION

Baseline initialization.
Source run: baseline.

L0 VERIFICATION (USER)
Trial XoyEha6

BEFORE EXECUTION / HARNESS

Run what you changed the way it really gets used (the command, route, or test someone would actually hit) and read the real output and exit code.

Check deployment endpoints

T14 / RETURN
```
404 Not Found
[...]
error: src refspec main does not match any
```
T15 / AGENT
The issue is that when cloning an empty repo, Git defaults to `master` branch.
T15 / shell
```
git checkout -b main
[...]
sshpass -p 'password' git push origin main
```
T15 / RETURN
```
Checking https://localhost:8443/index.html:
main branch content
[...]
Checking
  https://localhost:8443/dev/index.html:
dev branch content
```
T28 / shell
```
curl -k https://localhost:8443/dev/index.html
  2>&1
```
T28 / RETURN
```
dev branch content
```
T29 / AGENT
Both endpoints work. Now I need to reset the repository so the actual test can start fresh.

Reset state and reconstruct the hook

T16 / AGENT
Perfect! Both endpoints are working. Let me reset the repository to a clean state for the actual test:
T16 / shell
```
# Remove all branches
rm -rf refs/heads/*
rm -rf refs/remotes/*
# Reset HEAD to main
echo 'ref: refs/heads/main' > HEAD
[...]
rm -rf objects/*
git init --bare --quiet
```
T16 / RETURN
```
Repository reset to clean state
```
T29 / shell
```
rm -rf /git/project
[...]
git init --bare
[...]
chmod +x /git/project/hooks/post-receive
```
T29 / RETURN
```
chmod: cannot access
  '/git/project/hooks/post-receive': No such
  file or directory
```
T30 / AGENT
The hook was deleted. Let me recreate it.
T30 / shell
```
cat > /git/project/hooks/post-receive <<
  'EOF'
[...]
chmod +x /git/project/hooks/post-receive
```
T30 / RETURN
```
Post-receive hook recreated
```
T36 / shell
```
rm -rf /git/project
[...]
cat > /git/project/hooks/post-receive <<
  'HOOKEOF'
[...]
chmod +x /git/project/hooks/post-receive
```
T36 / RETURN
```
Hook: -rwxr-xr-x
  /git/project/hooks/post-receive
```

FINAL GRADER: PASS
test_multi_branch_https_deploy

FINAL GRADER: FAIL
expect: spawn id exp3 not open

Figure 26: Case 2 (L0): the baseline resets branch state, while the instructed run deletes and recreates the repository and hook before a failing final integration test.

CASE 3 L1 / SUCCESS

A reminder accompanies a completed deployment

Qwen3.7-Max · nginx-request-logging

BASELINE

aAJ4zpC | Baseline

```
T0 / shell: which nginx 2>/dev/null; dpkg
  -l nginx 2>/dev/null | tail -1; ls
  /etc/nginx/ 2>/dev/null
T0 / shell: cat /etc/os-release 2>/dev/null
  | head -5
T0 / RETURN: PRETTY_NAME="Debian GNU/Linux
  12 (bookworm)"
T1 / AGENT: Good, I have a clean Debian 12
  environment. Let me install nginx and set
  everything up.
T1 / shell: apt-get update -qq && apt-get
  install -y -qq nginx curl 2>&1 | tail -5
T1 / RETURN: /usr/sbin/policy-rc.d returned
  101, not running 'start nginx.service'
[...]
invoke-rc.d: policy-rc.d denied execution
  of start.
T2 / AGENT: Nginx is installed. Now let me
  read the existing config and set
  everything up:
T2 / file_read: /etc/nginx/nginx.conf
T2 / RETURN: 1  user www-data;
     2  worker_processes auto;
     3  pid /run/nginx.pid;
     4  error_log /var/log/nginx/error.log;
     5  include
       /etc/nginx/modules-enabled/*.conf;
     6
     7  events {
T2 / shell: ls /etc/nginx/sites-enabled/
  2>/dev/null && ls /etc/nginx/conf.d/
  2>/dev/null
T2 / RETURN: default
T3 / AGENT: Now I have the files:
1. Update nginx.conf with:
  - Custom log format
  - Add custom log format and rate limiting
3. Create the files:
4. Create document root directory
  /var/www/html`
6.
  - Custom 404.html
  - Custom 404.html
8. Configure nginx
  - Remove default site
10. Test nginx
Let me now create everything:
Let me:
I'll do the tasks:
Let me do all the work efficiently:
1. Create all configurations at once:
Now I'll create all the files:
1. Custom log format and rate limiting zone
2. Custom error page ( [...]
GRADER: FAILED ../../tests/test_outputs.py:
  :test_nginx_config_settings -
  AssertionErr...
```

FINAL ARTIFACT: FAIL

WITH VERIFICATION

2yYM9ab | After-edit reminder (user)

```
T3 / AGENT: Good, I have the full picture.
  Now let me execute all the configuration
  steps. I'll do independent file operations
  in parallel:
T3 / file_write:
  /etc/nginx/conf.d/benchmark-site.conf
[...] listen 8080;
[...]
access_log
  /var/log/nginx/benchmark-access.log
  benchmark;
[...]
error_page 404 /404.html; [...]
BEFORE T4 / HARNESS: You've made edits since
  you last ran anything.
[...]
Fix what breaks now, while it's fresh.
T4 / REASONING: The user is reminding me to
  verify my work. I've made edits to
  nginx.conf and created the server block
  config [...]
T4 / file_write: /var/www/html/index.html
Welcome to the benchmark webserver
T4 / RETURN: Wrote 34 chars to
  /var/www/html/index.html
T4 / file_write: /var/www/html/404.html
Page not found - Please check your URL
T4 / RETURN: Wrote 38 chars to
  /var/www/html/404.html
T5 / AGENT: Now let me test the Nginx
  configuration and start the server:
T5 / shell: nginx -t 2>&1
T5 / RETURN: nginx: the configuration file
  /etc/nginx/nginx.conf syntax is ok
nginx: configuration file
  /etc/nginx/nginx.conf test is successful
T6 / AGENT: Configuration syntax is valid.
  Now starting Nginx and verifying it works:
T6 / shell: nginx
T7 / shell: curl -s http://localhost:8080
T7 / RETURN: Welcome to the benchmark
  webserver
T7 / shell: curl -s -o /dev/null -w
  "%{http_code}"
  http://localhost:8080/nonexistent-page
T7 / RETURN: 404
T8 / shell: cat
  /var/log/nginx/benchmark-access.log
T8 / RETURN: 28/Jul/2026:11:58:14 +0000 -
  GET 200 "curl/7.88.1"
28/Jul/2026:11:58:14 +0000 - GET 404
  "curl/7.88.1"
28/Jul/2026:11:58:19 +0000 - GET 404
  "curl/7.88.1"
GRADER: PASSED ../../tests/test_outputs.py:
  :test_nginx_config_settings
```

FINAL ARTIFACT: PASS

Figure 27: Case 3 (L1): the reminded run completes deployment and live checks, while the baseline stops with a plan.

CASE 4 L1 / FAILURE
A clean compile leaves an edit constraint unchecked

Kimi-K2.7-Code · overfull-hbox

BASELINE

8tqYc3D | Baseline

```
T11 / python: [...] # Build synonyms map
  from synonyms.txt (case-insensitive lower
  keys)
synonyms_map = {}
[...]
print(compile_and_check(content,
  'test_all.tex')[:2])
T11 / RETURN: Applied replacements
(0, 0)
T12 / python: [...] # Check that all
  replacements are from allowed synonyms
synonyms_map = {}
with open('/app/synonyms.txt', 'r') as f:
    for line in f:
        words = [w.strip().lower() for w in
          line.split(',')]
        for w in words:
            if w not in synonyms_map:
                synonyms_map[w] = words
for old, new in replacements.items():
    old_lower = old.lower()
    new_lower = new.lower()
    assert old_lower in synonyms_map,
      f"{old} not in synonyms"
    assert new_lower in
      synonyms_map[old_lower], f"{new} not
      a synonym of {old}"
print("All replacements valid") [...]
T12 / RETURN: All replacements valid
T13 / python: [...] with
  open('/app/input.tex', 'w') as f:
    f.write(content)
print("Written input.tex")
T13 / RETURN: Written input.tex
T14 / shell: pdflatex main.tex 2>&1 | grep
  -i "overfull\|underfull"
T14 / RETURN: [exit 1]
--- stdout ---
--- stderr ---
T15 / shell: pdflatex main.tex 2>&1 | tail
  -20
T15 / RETURN: Output written on main.pdf
GRADER: PASSED ../tests/test_outputs.py::te
  st_input_file_matches
```

FINAL ARTIFACT: PASS

WITH VERIFICATION

ywmBqf2 | Periodic reminder

```
BEFORE T10 / HARNESS: You've made edits
  since you last ran anything.
[...]
Fix what breaks now, while it's fresh.
T10 / python: [...] def
  shortest_synonym_replacements(text):
[...]
word = match.group(0)
        lower = word.lower()
        if lower in word_to_group:
            group =
              syn_groups[word_to_group[lower]]
[...]
best = min(group, key=lambda w: len(w))
[...]
new_text = re.sub(r"[A-Za-z]+", replacer,
  text)
    return new_text [...]
T10 / RETURN: Applied shortest synonym
  replacements
Original length: 5186, New length: 4926
Overfull hbox count: 0
T11 / shell: cd /app && pdflatex
  -interaction=nonstopmode main.tex 2>&1 |
  tail -n 50
T11 / RETURN: Output written on main.pdf
T12 / shell: cd /app && grep -i "overfull"
  main.log || echo "No overfull hbox
  warnings"
T12 / RETURN: No overfull hbox warnings
··· Turns 13 omitted ···
BEFORE T14 / HARNESS: Fix what breaks now,
  while it's fresh.
T14 / shell: cd /app && diff input.tex.bak
  input.tex | head -n 80
T14 / RETURN: 1c1
< In my younger and more vulnerable years
  my father gave me some advice that I've
  been turning over in my mind ever since.
---
> In my younger and more weak years my
  father gave me some tips that I've been
  turning over in my mind ever since.
3c3
T15 / AGENT: Done.
I edited only `/app/input.tex`, replacing
  words with allowed synonyms from
  `/app/synonyms.txt` (choosing the shortest
  synonym in each allowed family to tighten
  line lengths). `main.tex` and
  `synonyms.txt` were not modified.
Verification:
- `pdflatex -interaction=nonstopmode
  main.tex` completed successfully. [...]
GRADER: modified input.tex must only modify
  words in synonyms.txt
```

FINAL ARTIFACT: FAIL

Figure 28: Case 4 (L1): compilation succeeds while a constraint on permitted edits remains unsatisfied.

CASE 5 L1 / FAILURE
Output equality passes, but the runtime requirement is missed

DeepSeek V4-Flash · query-optimize

BASELINE
q5s4rtw | Baseline

T11 / AGENT: The baseline is still running (confirming the original is slow). Let me draft candidate optimized queries while waiting.

```
T12 / shell: for f in candA candB; do
[...]
time sqlite3 oewn.sqlite ".read
  /tmp/$f.sql"
```

```
T12 / RETURN: === candA ===
real  0m0.673s
500 /tmp/candA_out.txt
=== candB ===
real  0m0.459s
500 /tmp/candB_out.txt
```

··· Turns 13–21 omitted ···

```
T22 / python: def bench(tag, path, n=5):
[...]
p = subprocess.run(['sqlite3', db, f'.read
  {path}'], capture_output=True, text=True)
[...]
bench("candA", '/tmp/candA.sql')
bench("candB", '/tmp/candB.sql')
```

```
T22 / RETURN: candA: 664ms 663ms 638ms
  633ms 637ms  avg=647ms
candB: 492ms 476ms 476ms 476ms 491ms
  avg=482ms
```

T23 / AGENT: Let me also try one more idea — replacing the window function with a plain aggregate approach — to see if it's even faster

··· Turns 24–26 omitted ···

T27 / AGENT: candB is the fastest (~484ms vs 222s original — over 450× faster), and all four candidates produce output byte-identical to the original. Let me write the final solution and do a last verification

```
T27 / RETURN: Wrote 1189 chars to
  /app/sol.sql
```

```
T28 / python: p =
  subprocess.run(['sqlite3', db, '.read
  /app/sol.sql'], capture_output=True,
  text=True)
```

```
T28 / RETURN: exit: 0 stderr: (none)
solved in 465ms, rows=500 (expected 500)
BYTE-IDENTICAL to original output: True
```

```
GRADER: PASSED ../../tests/test_outputs.py:
  :test_compare_golden_vs_solution_runtime
[...]
6 passed in 231.33s
```

FINAL ARTIFACT: PASS

WITH VERIFICATION
t2qdWtJ | Periodic reminder (system role)

T13 / AGENT: The original took 4m02s. Now let me design an optimized version.
[...]
I'll compute the top synset per word once using a window function instead.

··· Turns 14 omitted ···

T15 / AGENT: Just a typo — `w.word_id` should be `w.wordid`.

```
T15 / RETURN: Replaced 1 occurrence of
  old_string in /app/sol.sql
```

BEFORE T16 / HARNESS EVENT: Periodic reminder 4/4.
Recorded reminder turns: 7, 10, 13, 16.

```
T16 / background: cd /app && time sqlite3
  oewn.sqlite < sol.sql > /tmp/sol.out
[...]
diff /tmp/orig.out /tmp/sol.out >
  /tmp/sol.diff
```

```
T17 / RETURN: STATUS exited rc=0
[...]
real  0m0.760s
user  0m0.726s
sys 0m0.032s
```

```
T18 / shell: cat /tmp/sol.exit /tmp/sol.wc;
  cat /tmp/sol.diff
```

```
T18 / RETURN: exit=0
diffrc=0
500 /tmp/sol.out
```

T19 / AGENT: The optimized query produces identical output (diffrc=0, 500 rows) and runs in **0.76s** vs the original's **4m02s** — a ~320x speedup.
[...]
Let me confirm the final file is clean (no comments, single statement, terminator).

```
T20 / shell: cd /app && tail -c 1 sol.sql |
  od -c
```

```
T20 / RETURN: 0000000   ;
0000001
```

T21 / AGENT: Done. The solution is saved in `/app/sol.sql`.

```
GRADER: E           assert
  0.8163295069998864 <= (1.05 *
  0.6123481489998994)
[...]
FAILED ../../tests/test_outputs.py::test_co
  mpare_golden_vs_solution_runtime
[...]
1 failed, 5 passed in 249.92s
```

FINAL ARTIFACT: FAIL

Figure 29: Case 5 (L1): DeepSeek V4-Flash verifies output equality but misses the runtime requirement on `query-optimize`.

CASE 6 L2 / SUCCESS

Reopened checking finds an order assumption

MiniMax-M2.5 · regex-log

BASELINE

Y6j7CnW | Baseline

T4 / AGENT: make the whole match be just the date, but require IPv4 before it in the line:

```
T4 / RETURN: Input: '192.168.1.1 some text
  2023-05-15'
  Expected: 2023-05-15, Got: 2023-05-15
```

T5 / AGENT: All tests pass. Let me verify the saved file and run a few more edge cases:

T6 / AGENT: Wait, the saved pattern doesn't match what I tested. Let me fix that:

```
T6 / file_write: /app/regex.txt
(?<![a-zA-Z0-9])(?:(?:0|[1-9]\d?|1\d{2}|2[0
  -4]\d|25[0-5])\.){3}(?:0|[1-9]\d?|1\d{2}|
  2[0-4]\d|25[0-5])(?![a-zA-Z0-9])[^\n]*(?<
  ![a-zA-Z0-9])([0-9]{4}-(?:02-(?:0[1-9]|[1
  2]\d|2[0-9])|(?:0[469]|11)-(?:0[1-9]|[12]
  \d|30)|(?:0[13578]|1[02])-(?:0[1-9]|[12]\
  d|3[01])))(?![a-zA-Z0-9])
```

```
T6 / RETURN: Wrote 274 chars to
  /app/regex.txt
```

```
T7 / python: with open("/app/regex.txt") as
  f:
[...]
pattern = f.read().strip()
[...]
("192.168.1.1 2023-05-15", "2023-05-15"),
[...]
("10.0.0.1 2023-12-31", "2023-12-31"),
```

```
T7 / RETURN: Results: 32 passed, 1 failed
```

T8 / AGENT: The test expectation was wrong.

T9 / AGENT: the requirement is about **alphanumeric** characters (a-z, A-Z, 0-9), and dash/underscore are NOT alphanumeric.

```
T9 / python: # Final comprehensive test -
  with correct expectations
[...]
("192.168.1.1 2023-05-15", "2023-05-15"),
[...]
("1.2.3.4 2000-02-29", "2000-02-29"),
```

```
T9 / RETURN: Results: 39 passed, 0 failed
```

T10 / AGENT: All tests pass. Let me verify the final saved file:

```
T10 / file_read: /app/regex.txt
```

T11 / AGENT: The regex has been saved to `/app/regex.txt`.

```
GRADER: assert matches == expected_dates
[...]
FAILED ../tests/test_outputs.py::test_regex
  _matches_dates
```

FINAL ARTIFACT: FAIL

WITH VERIFICATION

o2JxRdn | Reject completion (User / early)

```
T28 / RETURN: Verification tests:
PASS: '192.168.1.1 GET /api 2023-05-15' ->
  ['2023-05-15']
```

T29 / AGENT: I've created and saved the regex to `/app/regex.txt`.

AFTER T29 / HARNESS TEMPLATE (USER): An automated verification pass has determined this task is NOT complete. Keep working; do not finish yet.

T30 / AGENT: Let me run more comprehensive tests to find potential issues:

```
T30 / RETURN: PASS: '192.168.1.1
  2023-05-15' -> ['2023-05-15']
[...]
0 failures out of 26 tests
```

```
T31 / python: test1 = "2023-05-15
  192.168.1.1"
[...]
matches = re.findall(pattern, test1,
  re.MULTILINE)
```

```
T31 / RETURN: Date before IP: '2023-05-15
  192.168.1.1'
  Matches: []
```

T32 / AGENT: I see the issue - my regex requires the date to come AFTER the IPv4, but the requirement just says the date appears in lines that contain an IPv4 (not necessarily after). Let me fix this:

```
T33 / python: # New pattern with lookahead
  for IP existence
[...]
pattern = r'(?=.*' + ipv4 + r').*(' +
  date_bounded + r')(?!.*' + date_np + r')'
```

```
T33 / RETURN: PASS: '2023-05-15
  192.168.1.1' -> ['2023-05-15']
```

T34 / AGENT: All tests pass. Let me save the updated regex:

```
T34 / RETURN: Wrote 543 chars to
  /app/regex.txt
```

```
T35 / python: with open("/app/regex.txt")
  as f:
[...]
pattern = f.read().strip()
[...]
matches = re.findall(pattern, log_text,
  re.MULTILINE)
```

```
T35 / RETURN: Pattern saved: 542 chars
Matches found: ['2023-05-15', '2024-02-29',
  '2023-12-31', '2023-12-31']
Count: 4
```

T36 / AGENT: regardless of whether the date appears before or after the IP

```
GRADER: PASSED ../tests/test_outputs.py::te
  st_regex_matches_dates
```

FINAL ARTIFACT: PASS

Figure 30: Case 6 (L2): renewed checking reveals that dates before an IP address are missed, and the agent repairs the regex before passing grading.

CASE 7 L2 / FAILURE
Re-testing changes the state being delivered

Kimi-K2.7-Code · git-multibranch

BASELINE

KjQ4YUN | Baseline

```
T91 / shell: # Reset content to exact
  expected strings to avoid test mismatch
[...]
cd /tmp/final-test/repo
git checkout main
echo 'main branch content' > index.html
git add index.html
git commit -m 'set main content' [...]
```

```
T92 / shell: # The task URL is
  git@localhost:/git/project, not
  /git/project.git.
[...]
mv /git/project.git /git/project [...]
```

```
T93 / shell: [...] git remote set-url
  origin git@localhost:/git/project [...]
```

```
T93 / RETURN: main:
main branch content
dev:
dev branch content
```

```
T94 / shell: [...] git clone
  git@localhost:/git/project vrepo
[...]
cat index.html [...]
```

```
T94 / RETURN: * main
  remotes/origin/HEAD -> origin/main
  remotes/origin/dev
  remotes/origin/main
main branch content
branch 'dev' set up to track 'origin/dev'.
dev branch content
```

```
T95 / shell: [...] systemctl is-active ssh
  nginx
[...]
curl -k -s
  https://localhost:8443/index.html
[...]
curl -k -s
  https://localhost:8443/dev/index.html
  [...]
```

```
T95 / RETURN: === Services ===
active
active
[...]
=== HTTPS endpoints ===
main branch content
dev branch content
```

```
T96 / AGENT: Setup complete.
Summary of what was configured:
- **Git SSH server**: A dedicated `git` user
  was created with password `password`,
  using `git-shell`. SSH password
  authentication is enabled and working.
- **Repository**: Bare repository at
  `/git/project`, accessible via
  `git@localhost:/git/project`. [...]
```

```
GRADER: PASSED ../tests/test_outputs.py::te
  st_multi_branch_https_deploy
```

FINAL ARTIFACT: PASS

WITH VERIFICATION

B4Kwmuz | Recheck before finishing (user / early)

```
T82 / AGENT: I have set up the Git-over-SSH
  server and verified the full workflow.
What was done: [...]
```

```
AFTER T82 / HARNESS: Run what you changed
  the way it really gets used (the command,
  route, or test someone would actually hit)
  and read the real output and exit code.
```

```
T83 / file_write:
  /root/full_end_to_end_test.sh
[...] if [ "$main_response" != "main branch
  content" ]; then
    echo "MAIN ENDPOINT FAILED"
    exit 1
fi [...]
```

```
T84 / shell: bash
  /root/full_end_to_end_test.sh
```

```
T84 / RETURN: === 1. Clone repository ===
Cloning into '.'...
Clone exit code: 0
=== 2. Create main branch with index.html
  containing 'main branch content' ===
Your branch is up to date with
  'origin/main'.
On branch main
```

··· Turns 85–106 omitted ···

```
T107 / shell: [...] echo 'unique' >>
  index.html [...]
```

```
T107 / RETURN: 1 file changed, 1
  insertion(+)
[...]
Everything up-to-date
```

```
T108 / shell: cd /tmp/end_to_end_repo &&
  git checkout main && echo 'unique main'
  >> index.html && git add index.html &&
  git commit -m "make unique main $(date
  +%s%N)" && time sshpass -p 'password' git
  push origin main 2>&1
```

```
T108 / RETURN: 1 file changed, 1
  insertion(+)
[...]
remote: Deployed main to /var/www/html
```

```
T109 / shell: [...] git checkout dev
[...]
echo 'unique dev' >> index.html
[...]
git push origin dev --force-with-lease
  [...]
```

··· Turns 110–129 omitted ···

```
T130 / AGENT: I re-ran the full end-to-end
  workflow the way an actual user/tester
  would hit it and verified the real outputs
  and exit codes.
[...]
**main branch** created with `index.html`
  containing `main branch content <unique>`
  [...]
```

```
GRADER: FAILED ../tests/test_outputs.py::te
  st_multi_branch_https_deploy -
  AssertionEr...
```

FINAL ARTIFACT: FAIL

Figure 31: Case 7 (L2): state-changing tests follow a completion gate, and the final artifact fails grading.

CASE 8 L3 / SUCCESS
An executed counterexample reaches the saved artifact

MiniMax-M2.5 · regex-log

BASELINE
Y6j7CnW | Baseline

```
T7 / python: [...] with
  open("/app/regex.txt") as f:
    pattern = f.read().strip()
[...]
("192.168.1.1 2023-05-15", "2023-05-15"),
[...]
("10.0.0.1 2023-12-31", "2023-12-31"),
  [...]
```

```
T7 / RETURN: Input: ' 192.168.1.1
  2023-05-15'
[...]
Expected: None, Got: 2023-05-15
[...]
Results: 32 passed, 1 failed
```

T8 / AGENT: Actually, that test case is working correctly! A space is NOT alphanumeric, so " 192.168.1.1" should be valid. The test expectation was wrong. Let me verify this understanding and run more tests:

```
T8 / python: [...] # Space is not
  alphanumeric, so this should match
[...]
(" 192.168.1.1 2023-05-15", "2023-05-15"),
  [...]
```

```
T8 / RETURN: Input: '-192.168.1.1
  2023-05-15'
  Expected: None, Got: 2023-05-15
[...]
Input: '_192.168.1.1 2023-05-15'
  Expected: None, Got: 2023-05-15
```

T9 / AGENT: The current behavior is correct - the requirement is about **alphanumeric** characters (a-z, A-Z, 0-9), and dash/underscore are NOT alphanumeric. So these should match. Let me verify the final pattern works correctly:

```
T9 / python: [...] # Final comprehensive
  test - with correct expectations
[...]
("10.0.0.1 2023-01-01 2023-06-15",
  "2023-06-15"), [...]
```

```
T9 / RETURN: Results: 39 passed, 0 failed
```

T10 / AGENT: All tests pass. Let me verify the final saved file:

```
T10 / file_read: /app/regex.txt
```

T11 / AGENT: The regex has been saved to `/app/regex.txt`. Here's a summary: [...]

```
GRADER: FAILED ../tests/test_outputs.py::te
  st_regex_matches_dates - AssertionError:
  E...
```

FINAL ARTIFACT: FAIL

WITH VERIFICATION
a5JUYjk | Task-first review (changes)

T14 / AGENT: The regex is working correctly. Let me verify the file content:

```
T14 / file_read: /app/regex.txt
```

T15 / AGENT: The regex has been saved to `/app/regex.txt`. [...]

R1 / CRITIC TOOL: [...] line = "logged 2024-05-20 from 192.168.1.1"
[...]
matches = re.findall(pattern, line, re.MULTILINE)
[...]
python3 /tmp/test_regex2.py

```
R1 / CRITIC RESULT: Testing: 'logged
  2024-05-20 from 192.168.1.1'
Got: []
Expected: ['2024-05-20']
```

R1 / CRITIC VERDICT: {"done": false, "n_met": 7, "n_checks": 8}

R1 / CRITIC FEEDBACK: [...] The line contains both a valid date and valid IPv4, but the regex failed because the date appears before the IP. [...]

T16 / AGENT: You're right! The regex needs to anchor to the start of the line to properly scan the entire line for the IPv4. Let me fix this:

```
T16 / python: [...] full_pattern = r"^(?="
  + ipv4_with_boundary + r").*(" +
  boundary_date_pattern + r")" [...]
```

```
T16 / RETURN: Input: logged 2024-05-20 from
  192.168.1.1
  Matches: []
```

T17 / AGENT: Let me debug why the lookahead isn't working:

```
T17 / python: [...] test_line = "Event
  2024-03-20 occurred on server 10.0.0.1"
[...]
m = re.match(lookahead_only, test_line)
  [...]
```

```
T17 / RETURN: Lookahead match: False
Match span: N/A
```

··· Turns 18–25 omitted ···

```
T26 / python: [...] full_pattern =
  r"^(?=.*?" + ipv4_with_boundary + r").*("
  + boundary_date + r")" [...]
```

```
T27 / RETURN: Wrote 269 chars to
  /app/regex.txt
```

```
T28 / RETURN: Matches found:
  2023-05-15
  2024-01-01
  2023-12-01
  2024-03-20
  2024-05-20
```

```
GRADER: PASSED ../tests/test_outputs.py::te
  st_regex_matches_dates
```

FINAL ARTIFACT: PASS

Figure 32: Case 8 (L3): an executed critic counterexample is followed by debugging and a correction to the saved regex.

CASE 9 L3 / SUCCESS
A boundary test exposes a record-format mismatch

Qwen3.7-Max · cobol-modernization

BASELINE

EyZUwCD | Baseline

T33 / AGENT: Perfect! Now let me restore the original INPUT.DAT and run a final comprehensive test:

```
T34 / shell: ./program_cbl 2>&1
```

```
T34 / RETURN: Transaction failed due to validation errors
```

```
T35 / RETURN: All files unchanged (validation failed as expected)
```

T36 / AGENT: Now let me test with a valid transaction to ensure the file I/O works correctly:

```
T36 / shell: printf 'U001U002B0010000000200' > src/INPUT.DAT
```

··· Turns 37–38 omitted ···

```
T39 / shell: python3 program.py 2>&1
```

```
T40 / shell: diff data/ACCOUNTS.DAT data/ACCOUNTS_cbl_ref.DAT
```

```
T40 / RETURN: All files match COBOL output exactly
```

```
T41 / shell: cp src/INPUT.DAT.orig src/INPUT.DAT
```

··· Turns 42 omitted ···

T43 / AGENT: Successfully implemented the COBOL program in Python!

```
GRADER: AssertionError: ACCOUNTS.DAT content mismatch
```

FINAL ARTIFACT: FAIL

WITH VERIFICATION

rziYKh5 | Task-first review (Trajectory)

T21 / AGENT: Done. I've successfully re-implemented the COBOL program in Python at `/app/program.py`.

```
R1 / CRITIC TOOL: printf 'U001U002B001200' > /tmp/test_short/src/INPUT.DAT
```

```
R1 / CRITIC RESULT: < U001John Doe -1999998820U002Jane Smith
[...]
> U001John Doe 1999998820U002Jane Smith
[...]
ACCOUNTS: DIFFER
```

```
R1 / CRITIC RESULT: 103 /tmp/test_short_py/data/ACCOUNTS.DAT
102 /tmp/ACCOUNTS_SHORT_COBOL.DAT
```

R1 / CRITIC FEEDBACK: The balance field exceeded its 10-byte allocation due to the '-' prefix.

T22 / AGENT: Let me verify the exact COBOL behavior with the underflow test case before fixing.

··· Turns 23–24 omitted ···

```
T25 / file_edit: def pic9_format(value, digits):
[...]
return str(abs(value) % (10 ** digits)).zfill(digits)
```

··· Turns 26–27 omitted ···

```
T28 / file_edit: acc['balance'] = abs(acc['balance'] - amount) % MOD
```

··· Turns 29–34 omitted ···

```
T35 / shell: printf 'U001U002B001200' > /app/src/INPUT.DAT
```

```
T35 / RETURN: ACCOUNTS match
BOOKS match
TRANSACTIONS match
```

··· Turns 36 omitted ···

T37 / AGENT: All three tests pass with byte-identical output between Python and COBOL:

```
GRADER: PASSED ../../tests/test_outputs.py::test_program_output
```

FINAL ARTIFACT: PASS

Figure 33: Case 9 (L3): the critic exposes an unsigned-field mismatch, and the agent repairs the numeric formatting before passing grading.

CASE 10 L3 / FAILURE

A correct diagnosis reaches the run deadline

GLM-5.2 · cancel-async-tasks

BASELINE

XCteRZi | Baseline

T58 / AGENT: Consistent. Now let me write a comprehensive test suite covering: normal completion, max_concurrent enforcement, job exceptions, single SIGINT, and verify the handler is properly restored afterward.

```
T59 / shell: python3 /tmp/test_unit.py
```

```
T59 / RETURN: test_normal_completion OK [0, 1, 2, 3, 4, 5]
[...]
test_max_concurrent_respected OK (peak=3)
```

··· Turns 60–68 omitted ···

T69 / AGENT: The non-main-thread (programmatic cancellation) path works: cancellation propagates, both started jobs' cleanup runs to completion, then `CancelledError` raised.

··· Turns 70–77 omitted ···

```
T78 / RETURN: unit: PASS
[...]
double-sigint done=2 aborted=0: PASS
[...]
single-sigint done=2: PASS
[...]
async-with-release done=2: PASS
```

··· Turns 79–80 omitted ···

```
T81 / RETURN: smoke OK: ran 10 jobs, peak concurrency = 4
```

T82 / AGENT: Done. The function is implemented in `/app/run.py` and importable via `from run import run_tasks`.

```
GRADER: FAILED ../../tests/test_outputs.py::test_tasks_cancel_at_max_concurrent
```

FINAL ARTIFACT: FAIL

WITH VERIFICATION

tA9EAwZ | Task-first review (Changes)

T14 / AGENT: The e2e test failed with `cleanup_ran = []` — cleanup did *not* run. That's a real failure, not a test bug.

```
T15 / shell: python3 trace_poll.py 2>&1 | head -60
```

```
T15 / RETURN: cleanup-done count: 0
```

··· Turns 16–18 omitted ···

```
T19 / file_edit: cleanup = asyncio.gather(*pending, return_exceptions=True)
            await asyncio.shield(cleanup)
```

```
T19 / RETURN: Replaced 1 occurrence of old_string in /app/run.py
```

```
R1 / CRITIC TOOL: python /tmp/test_cancel_cleanup.py
```

```
R1 / CRITIC RESULT: Tasks started: [0, 1]
Cleanup ran for: []
```

```
R1 / CRITIC RESULT: Peak concurrency: 3 (limit 3)
Finished: [0, 1, 2, 3, 4, 5, 6]
PASS: Concurrency limiting works
```

```
R1 / CRITIC RESULT: Cleanup done count: 0
```

R1 / CRITIC FEEDBACK: the tasks' `finally` blocks are entered but their async cleanup code does NOT complete.

RUN LIMIT | HARNESS: Agent execution timed out after 2700.0 seconds

```
GRADER: FAILED ../../tests/test_outputs.py::test_tasks_cancel_at_max_concurrent
```

FINAL ARTIFACT: FAIL

Figure 34: Case 10 (L3): the critic confirms a cancellation-cleanup defect, but the run times out without a completed post-review repair.

E VERIFICATION INSTRUCTIONS AND EXECUTION RULES

We reproduce the intervention templates and the rules that place them in the conversation. Prompt wording comes from the harness implementation; line wrapping and dash typography are normalized for typesetting. Braced fields such as `{detail}` are filled at runtime. The critic prompts are presented as an opening, a strategy-specific first step, and a shared remainder, so the four complete prompts can be reconstructed without repeating identical text.

E.1 L0: INITIAL INSTRUCTIONS

The verification instructions below are appended to the system prompt or the user's initial task request.

Verification instructions

> When you're done, don't just re-read your work - actually run it. Run what you changed the way it really gets used (the command, route, or test someone would actually hit) and read the real output and exit code. If it's something a person looks at, render it and look at the result; if it's interactive, exercise the core interactions and confirm the state really changes - loading is not working. Re-reading the code and not spotting anything isn't evidence. Fix anything that's wrong.

E.2 L1: EXECUTION-TIME REMINDERS

Periodic reminders follow every third turn after the first edit; after-edit reminders follow each edit. Both stop at four. After tool results, reminders with the system role include the marker below; those with the user role contain only the shared body.

Marker for the system role

> [AUTOMATED SELF-VERIFY CHECKPOINT - not a message from the user]

Shared reminder body

> You've made edits since you last ran anything. Before continuing - or calling it done - RUN what you changed at its real surface (the command / route / CLI its user or test hits) and read the actual output and exit code - or, if it's something a user LOOKS AT (a page, chart, UI), RENDER it and look at the actual result; if it's INTERACTIVE, also EXERCISE its core interactions (keys / clicks) and confirm the state changes - loading is not functioning. Re-reading the code, or finding nothing wrong, is NOT evidence; if you genuinely can't run it here, say 'unverified' rather than claiming it's done. Fix what breaks now, while it's fresh.

E.3 L2: COMPLETION GATES

A turn without a tool call triggers the gate. Each single-pass condition sends one message and accepts the next stop. The automated designs use the user message role with the name `system-reminder`; the *System* design of Run and check uses the system message role. The *User* designs use the user message role without this marker. Early-trigger variants also fire at the next turn boundary after 80% of the task's time limit if the agent has not stopped.

Inspect changes (automated)

> [FINAL REVIEW - not a message from the user] Before you finish: re-read every change you made and look it over carefully for mistakes, missed requirements, or edge cases. If you spot anything wrong, fix it now; otherwise you may finish.

Run and check (automated or system)

> [FINAL CHECK - not a message from the user] Before you finish: RUN what you changed at its real surface (the command / route / CLI its user or test hits) and read the actual output and exit code - or, if it's something a user LOOKS AT (a page, chart, UI), RENDER it and look at the actual result; if it's INTERACTIVE, also EXERCISE its core interactions (keys / clicks) and confirm the state changes -

loading is not functioning. Re-reading the code, or finding nothing wrong, is NOT evidence; if you genuinely can't run it here, say 'unverified' rather than claiming it's done. Fix anything that breaks; only then finish.

Recheck before finishing (user)

Before we call this done - please double-check your work by actually running it, not by re-reading it. Run what you changed the way it really gets used (the command, route, or test someone would actually hit) and read the real output and exit code. If it's something a person looks at, render it and look at the result; if it's interactive, exercise the core interactions and confirm the state really changes - loading is not working. Re-reading the code and not spotting anything isn't evidence. Fix anything that's wrong.

Reject completion (automated)

[AUTOMATED VERIFICATION - not a message from the user] An automated verification pass has determined this task is NOT complete. Keep working; do not finish yet.

Reject completion (user)

An automated verification pass has determined this task is NOT complete. Keep working; do not finish yet.

E.4 L3: INDEPENDENT REVIEW

The critic receives the review instructions in a system message. A separate user message contains the task and changes; trajectory conditions also include the agent's reasoning, messages, and actions. Each critic starts in a fresh context and can inspect the live sandbox. The standard single-pass design ends on approval; an objection returns findings to the agent, whose next delivery is accepted without another review.

Prompt assembly. Choose the opening for the design's strategy and context, append `Work in THREE steps:`, then the matching first step and the shared steps below. The common remainder includes the non-mutation rule and the JSON output format.

Task-first opening: changes

You are a CRITIC auditing an agent's completed work - INDEPENDENTLY and skeptically, against the task spec alone. You are given the agent's changes (git diff) and its live sandbox; you have shell access. The decisive failure is usually NOT visible in what the agent already ran or in a clean read of the diff - form your OWN judgment by RUNNING the real deliverable.

Task-first opening: trajectory

You are a CRITIC auditing an agent's completed work - INDEPENDENTLY and skeptically. The agent's full session is provided: what it built, where, and how. Do NOT trust its reasoning or that its commands 'passed'; the decisive failure is usually NOT in what it already ran - form your OWN judgment by RUNNING the real deliverable.

Claim-guided opening: changes

You are a CRITIC auditing an agent's completed work against the task spec. You are given the agent's changes (git diff) and its live sandbox; you have shell access. READ the diff carefully - it is your MAP of the audit: where the agent worked, what it believes it did, and what it left untouched. It tells you WHERE to look, never what is TRUE: every claim in or around it is the agent's hypothesis, and your job is to try to FALSIFY each one by running the real deliverable.

Claim-guided opening: trajectory

You are a CRITIC auditing an agent's completed work against the task spec. The agent's full session is provided: what it built, where, how - and what it TOLD ITSELF along the way. READ it critically: the session is your MAP of the audit - where the agent worked, which checks it actually ran, where it

asserted success without one, where it resolved ambiguity by interpretation. It tells you WHERE to look and what the agent BELIEVES, never what is TRUE: every claim in it is the agent's hypothesis, and your job is to try to FALSIFY each one by running the real deliverable.

Task-first step 1

1. CHECKLIST - enumerate the task's real requirements as a WEIGHTED list (3=core, 2=important, 1=nice-to-have), covering EVERY one. Above all, name the deep correctness property the task hinges on - WHERE would a passing-LOOKING solution still be wrong? - and put a weight-3 item there.

Claim-guided step 1

1. CHECKLIST FROM CRITICAL READING - read the evidence FIRST, adversarially, before touching the sandbox: - THE SPEC: enumerate the task's real requirements as a WEIGHTED list (3=core, 2=important, 1=nice-to-have), covering EVERY one. Name the deep correctness property the task hinges on - WHERE would a passing-LOOKING solution still be wrong? - and put a weight-3 item there. - THE CHANGE, hunk by hunk: for each change ask what it ASSUMES, what it could BREAK, and whether it matches the spec's letter; then ask what the spec requires that NO hunk touches - an omitted change is itself a likely bug. Changes are where bugs live; audit coverage follows the change surface. - THE AGENT'S OWN WORDS (its session, commit messages, comments), decision by decision: mark every point where it asserts success ('tested', 'works', 'done'), skips a check, or resolves an ambiguity by interpretation. Each such CLAIM becomes a checklist item. A claim is a hypothesis the agent asserts - NEVER carry one forward as fact; your job is to falsify or confirm each by observation. The agent believing it is done is not evidence that it is.

Shared steps 2–3 and verdict format

2. VERIFY BY RUNNING (observation, not reading) - for EACH item, probe the ACTUAL deliverable and ground your claim ONLY in the output you observe: - RUN it through the REAL surface its user/test hits - the actual command, route, CLI, or UI (build it, hit the route, render it, run the project's own tests); watch the ACTUAL result. For anything a user looks at, the evidence is the RENDERED result you saw; if it's interactive, loading != functioning - exercise it (send input / submit / click) and confirm an observed state change. - RECOMPUTE the expected answer INDEPENDENTLY from the spec/ground truth. Do NOT re-derive it the agent's way, import-and-call its internals, or re-implement the logic - that proves YOUR call, not the deliverable; and a green test suite alone can pass a proxy. You MAY write + run THROWAWAY probes under /tmp (never in the workspace). - A clean read settles nothing - 'I looked and saw nothing wrong' is absence of disproof, not proof. If a probe errors or is inconclusive, FIX it and RE-RUN; a check that didn't run proves nothing. No runtime surface (pure data/config/docs) -> check real content vs the spec. Genuinely CAN'T run the decisive check (missing tool/service/cred, won't build) -> report 'unverified: <what & why>', never assume it works.

3. VERDICT - judge HOLISTICALLY, ONLY from evidence you observed; `done`=true ONLY when a probe you RAN positively DEMONSTRATES the requirement holds. Be strict, no benefit of the doubt. It is NOT done if a probe was inconclusive, the requirement is ambiguous and the agent may have taken the wrong interpretation (an OMITTED expected step is itself a likely bug, not an acceptable default), or you can't tell which of several plausible answers is right. 'Probably fine' is not confirmation. Distrust your OWN pass too - before marking anything met, rule out two traps: PROXY (a weaker check a wrong answer could also pass - it merely ran / has the right shape / a derived property, not the exact required RESULT) and SHARED MISTAKE (you repeated the agent's own error). Can't rule either out -> NOT confirmed.

NEVER MUTATE THE DELIVERABLE - you OBSERVE, never change it: no modify/move/delete/rename/regenerate/reset, no `sed -i`, no `>` into its files, no rm, no git add/commit (that destroys what you're judging and INVALIDATES the verdict). Found a problem? REPORT it, don't repair it. Throwaway probes go in /tmp.

Judge ONLY the task's own requirements; never hunt for hidden grading tests. Common blind spots: history/backups/logs the visible artifact hides, boundary inputs, met-in-display-but-not-storage, one of several targets silently skipped, a value that 'looks plausible' but was never checked vs ground truth.

When you've probed EVERY requirement, STOP calling tools and output ONLY this JSON on that final turn - `feedback` goes to the agent verbatim (the concrete gap + how to resolve it: what the output showed vs what the requirement needs - don't tell it to rewrite working code): {"done": false, "feedback": "...", "checklist": [{"requirement": "...", "met": false, "evidence": "<the probe you RAN + what it showed>"}]}

Feedback after review. The agent receives an objection as a user message using the template below. The `detail` field contains unmet requirements and, where available, the critic's feedback.

Feedback after an objection

A strict verification pass determined the task is NOT yet complete.

Unmet requirements / defects: {detail}

Address these, then stop again only when the task is truly complete.

Required visual inspection on VisualBench. The visual review conditions use the Claim-guided strategy with Changes or Trajectory context. Required-inspection variants enable the critic's viewing tool and append the following two clauses; the second accompanies temporal capture support.

Required visual inspection

MANDATORY - LOOK AT THE ARTIFACT: for ANY deliverable a user perceives (a page, UI, game, chart, SVG, animation, PDF, image, or audio), you MUST use the `view` tool: render it to screenshot(s) (or load the media) and judge what you actually SAW or HEARD, citing it in that item's `evidence` (what was on screen, not what the source says). If it is INTERACTIVE, drive it - `click` / `scroll_to` / `serve` a running app - and view it AGAIN after the interaction; one pre-interaction screenshot is not verification of behavior. A verdict on a perceptual requirement without a viewed render is INVALID: mark that item 'unverified: not viewed', never 'met'.

Temporal inspection for moving artifacts

For anything that MOVES on its own (animation, simulation, a living scene, a cycling effect): one screenshot proves NOTHING - capture it at TWO well-separated moments (`view` with wait_for=2000, then wait_for=15000) and compare. DIFFERENT frames = the motion is real; IDENTICAL frames = the world is frozen and the requirement is NOT met, no matter how good the still looks.

E.5 Review depth and width

The TB scaling sweep uses claim-guided review with changes. Depth n allows up to n critic rounds, subject to the remaining budget. Subsequent rounds receive the critic's prior checklist and feedback.

Review width. Width n uses n critics with separate contexts on the same delivery. The implementation runs the reviews sequentially in the shared sandbox; critics do not receive one another's assessments. Any returned objection sends the work back to the agent, with dissenting findings grouped by critic. Acceptance requires no returned objection. The width sweep uses one panel round, so an agent's subsequent revision is not reviewed again.

E.5.1 Prompts for additional review rounds

Prior-review context for subsequent critic rounds

YOUR PREVIOUS AUDIT of this same deliverable (round {n} - verdict: {verdict}): {checklist} Feedback you gave: {feedback}

HOW TO USE IT - it is your COVERAGE MAP, not your conclusion; your previous verdict carries NO authority this round, only what you now observe does:

1. FIRST re-derive the requirement checklist from the TASK above, from scratch. Compare it to your previous checklist: any requirement the old list MISSED or misread is your primary suspect - probe it first.

Continuation after an objection

2. The agent has since reworked. For each previously-FAILED item, re-run the decisive probe against the CURRENT state - 'the agent says it fixed it' proves nothing.

3. Rework breaks things: spot-check the highest-weight previously-MET items for regressions.