DENSE: DISTILLING AGENT TRAJECTORIES INTO EVIDENCE-GROUNDED SHORTCUT TREES FOR SELF-REFINEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Online agent deployments accumulate execution trajectories at massive scale and behavioral diversity, for which predefined annotation criteria hardly exist. Extracting useful evidence therefore demands costly manual annotation or verifier signals that fail to scale, leaving valuable evidence buried among redundant, incomplete, and failed executions. This raises a question: without post-execution rewards or correctness labels, how can reusable experience be distilled from the trajectories themselves? To address this challenge, we introduce **DENSE** (Distilling Evidence from Nested Subtask Executions), which organizes trajectory-derived evidence into *nested shortcut trees*. By consolidating redundant attempts, identifying resolved subtasks, and retaining useful steps alongside outstanding requirements, DENSE transforms noisy execution traces into structured and reusable task-solving feedback. To evaluate whether such feedback helps agents retry the same task, we design **REFIT**, which measures success-rate changes between the initial attempt and feedback-guided retries. Among feedback methods without external outcome supervision, DENSE achieves the highest strict pass rate across four agent models on Terminal-Bench 2.1, improving over initial attempts by **7.12–21.81 percentage points** with **19.0–43.6%** fewer agent tokens on retries. In addition, on hard tasks DENSE consistently outperforms self-reflection in cumulative pass rate across multiple feedback iterations on all four models, demonstrating its strong potential for continual agent self-improvement.

1 INTRODUCTION

Large language model agents increasingly operate in software engineering (Yang et al., 2024), web interaction (Zhou et al., 2023), deep research (Tongyi DeepResearch Team et al., 2025), and desktop automation (Xie et al., 2024). In offline evaluation, curated tasks can be assessed using task-specific verifiers or human expert annotations. Online deployment continuously generates long traces of actions, observations, failed attempts, and recovery across diverse tasks; providing comparable supervision for these traces is costly and difficult to scale (Figure 1). This gap raises a question: **without post-execution rewards or correctness labels, what useful experience can we extract from the trajectories themselves?**

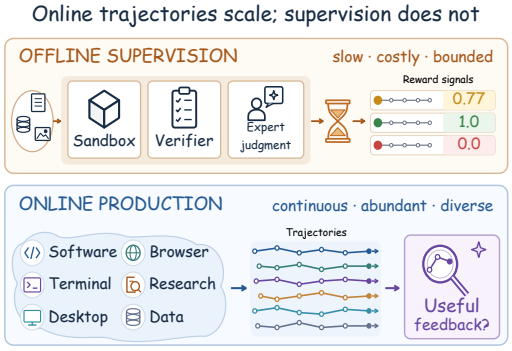


Figure 1: Execution traces accumulate, while external supervision is costly and bounded. Can agents reuse experience from these traces without external rewards or correctness labels?

Prior work demonstrates that trajectory experience can improve subsequent execution. Reflexion uses evaluator rewards to guide verbal reflection (Shinn et al., 2023); ExpeL extracts lessons by contrasting successful and failed attempts (Zhao et al., 2024); and DoVer tests failure hypotheses through interventions and outcome verification (Ma et al., 2025). These approaches improve task performance, but depend on outcome signals to

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.

guide reflection, experience selection, or diagnosis. However, how to systematically extract reusable experience from *unlabeled online trajectories*, without post-execution rewards, verifier outputs, or human correctness annotations, remains underexplored.

Our central insight is that **trajectories still contain reusable evidence, even without outcome labels**. For example, a tool call may fail because of a missing argument, then return the requested data after correction. This sequence supports reusing the corrected call, even when overall task completion is uncertain and the requested analysis remains unverified. The challenge is to connect useful steps to supporting observations and outstanding requirements. We call this process *trajectory evidence distillation*: constructing feedback for subsequent execution using only the task description, recorded observations, and its execution trace, without post-execution rewards or correctness labels.

To efficiently distill this evidence into reusable feedback, we propose **DENSE** (Distilling Evidence from Nested Subtask Executions), which constructs *evidence-grounded nested shortcut trees*. Each shortcut records attempted approaches, useful steps, supporting observations, and unfinished requirements within a subtask. DENSE groups actions into nested subtasks and condenses repeated attempts. When combining subtasks, it updates unresolved issues if later observations show that they have been resolved. It then summarizes completed subtasks and retains more detail for unfinished ones, preserving the actions needed to recreate earlier progress.

To evaluate the value of experience extracted from online trajectories by different methods, we introduce **REFIT**, which compares feedback constructed from the same initial trajectory (run0) without access to post-execution outcome information. The same agent then makes a separate attempt (run1) with each method's feedback, starting from the initial environment state and a fresh model context. Only feedback carries information between attempts. We compare strict pass rates and changes from run0 under matched task, tool, and execution-budget conditions. Following critique evaluation through subsequent answers (Tang et al., 2025), we assess feedback by task outcomes, keeping verifier results separate from feedback generation.

Our contributions are:

1. **The DENSE method.** We introduce evidence-grounded nested shortcut trees that preserve useful steps and unfinished requirements, using subtask structure to track issue resolution and prioritize feedback detail according to each subtask's remaining requirements.
2. **The REFIT framework.** We introduce a protocol that compares feedback methods using shared initial trajectories, reset environments and contexts, and separate outcome evaluation, while excluding post-execution outcome signals from feedback generation.
3. **Systematic evaluation and mechanism analysis.** Using REFIT, we compare DENSE with alternative feedback methods on Terminal-Bench 2.1 (Merrill et al., 2026). DENSE achieves the highest strict pass rate among the tested non-privileged methods[1] across four agent models, improving over run0 by **7.12–21.81 pp** and reducing observed agent token use in run1 by **19.0–43.6%**. Ablations and case studies probe the mechanisms; an exploratory hard-task extension shows higher cumulative pass rates than Self-reflection after three feedback iterations, suggesting DENSE's potential to support agent refinement over multiple rounds.

2 DENSE: Evidence-Grounded Shortcut Trees

DENSE organizes scattered actions and observations into reusable task experience in two stages. It first identifies semantic boundaries in ordered queues to build a nested subtask tree. It then compresses attempts into shortcuts, reconciles issues across levels, and expands feedback according to issue state. The following sections explain these steps through one CSV report example.

2.1 Problem Setting and Overview

Given task x and a finished attempt's trajectory $\tau = (a_1, o_1, \ldots, a_T, o_T)$, DENSE produces bounded feedback distinguishing useful progress, recovered local errors, and unfinished requirements (Figure 2). It reads only the task and visible trajectory; Section 3 defines the information boundary and reset conditions.

[1] By privileged feedback, we mean exposing to the model post-hoc reward information and verifier outputs that are otherwise hidden during execution.

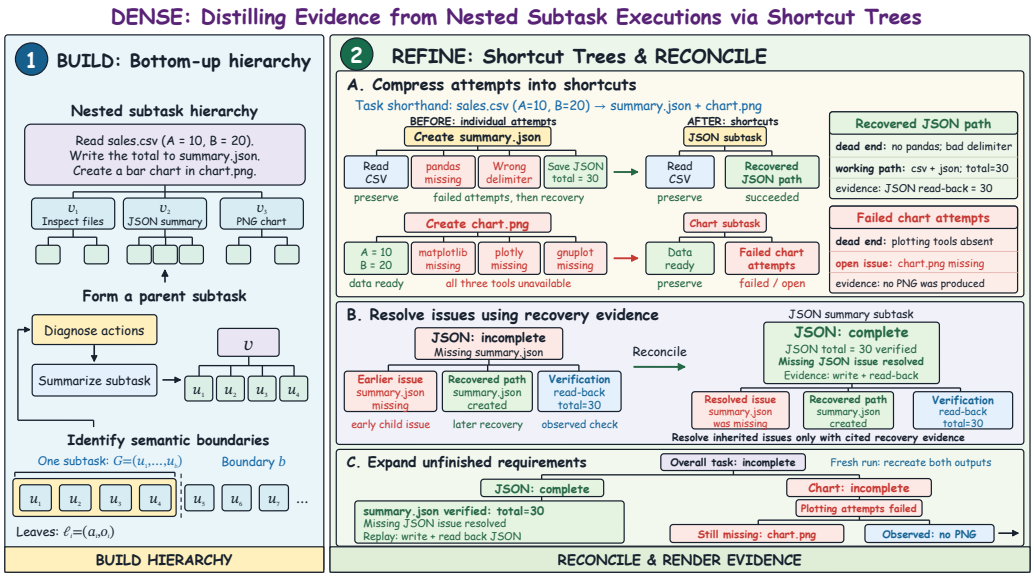


Figure 2: DENSE has two stages. Stage 1 identifies a semantic boundary for a consecutive prefix of queue Q_k, diagnoses its actions, summarizes a parent node, and places it in Q_{k+1} to build the hierarchy. Stage 2 has three steps: A compresses sibling attempts into shortcuts retaining failed attempts, working paths, and evidence; B reconciles inherited issues using later recovery evidence; C summarizes completed branches and expands unfinished requirements with their evidence. The constructed CSV report example has a completed JSON summary and an unfinished PNG chart.

2.2 Building Nested Subtasks

Local outcomes must be interpreted within their goals. DENSE pairs actions and observations as leaves $\ell_i = (a_i, o_i)$ in a temporally ordered queue Q_k. The analysis model identifies a prefix $G = (u_1, \ldots, u_b)$ serving one objective, expanding its visible range when evidence is insufficient. After selecting boundary b, it groups these nodes under parent v, diagnoses their actions, summarizes the subtask, and places v in Q_{k+1}. Repetition builds the root hierarchy. Each level preserves temporal order and source coverage, allowing local outcomes to be reinterpreted within higher-level goals.

In Figure 2, the task is to read `sales.csv` and produce a total in `summary.json` and a bar chart in `chart.png`. After encountering missing `pandas` and an incorrect delimiter, the JSON branch uses the standard library to write the summary and read back a total of 30. The plotting branch tries three unavailable tools and produces no PNG. Separate subtasks represent both the completed summary and the unfinished chart.

2.3 Compressing Attempts with Their Evidence

DENSE processes the tree bottom-up, fusing consecutive attempts under one parent into a shortcut. Each shortcut retains failed approaches, a compact working path supported by observations, the final local outcome, and source references. Unresolved shortcuts retain unmet requirements and relevant observations. Fusion never crosses parent boundaries or reorders sources.

In the CSV example, the JSON shortcut compresses the missing-`pandas`, incorrect-delimiter, and successful-write attempts. It records the first two as failed approaches and retains the working `csv`/`json` path with the observation confirming a total of 30. The plotting shortcut instead records the three unavailable tools and the missing `chart.png`. The same structure thus supplies historically supported paths and experience that helps the next attempt avoid repeating unproductive exploration.

2.4 Reconciling Issues through Recovery Evidence

An early issue may be repaired by actions in a later subtask, so a parent must revisit issues inherited from its children. For each compressed subtree, the analysis model produces a final-state summary and any unresolved or fatal issues. The parent combines child summaries, issues, and recovery

evidence:

$$(\mathcal{T}_v^\star, \sigma_v, \mathcal{I}_v) = \text{Reconcile}\Big(\widetilde{\mathcal{T}}_v, \{(\sigma_u, \mathcal{I}_u)\}_{u \in \text{child}(v)}\Big)\,. \tag{1}$$

Here $\widetilde{\mathcal{T}}_v$ is the compressed local subtree, σ_v its state summary, $\mathcal{I}_v$ the remaining issues, and $\mathcal{T}_v^\star$ the reconciled subtree passed upward.

Closing an inherited issue requires *concrete recovery evidence*. In the CSV example, early inspection reports a missing `summary.json`; a later subtask writes and reads it back. The JSON parent closes that issue, while the root retains the unfinished `chart.png` requirement. Recovery, subtask completion, and overall completion are judged at their respective scopes. Language models make semantic judgments; deterministic checks constrain source coverage, evidence references, issue updates, and key-action preservation.

2.5 Rendering Unfinished Requirements

Feedback combines a root overview with local expansion. Completed branches receive short summaries; incomplete, failed, or unsupported branches expose shortcuts, issues, and observations. The CSV example summarizes the JSON result and retains its key write operation, while expanding the missing PNG and failed plotting evidence. A fresh execution can recreate the JSON and focus further exploration on the PNG.

Selected critical write operations and final-response information retain their original content. Task state determines which processes can be summarized and which issues need expansion, producing high-density feedback of bounded length for long-horizon tasks. Length budgets and token accounting appear in Appendix A; Appendix C shows the final feedback format. Appendix F specifies the complete algorithm, module interfaces, and validation rules.

3 REFIT: Evaluating Trajectory Feedback

REFIT evaluates trajectory feedback by comparing task success on new attempts with the initial execution. It fixes the agent, tasks, and execution budget; methods extract feedback from the same raw trajectory, and each method's subsequent results are compared against this common starting point.

3.1 Paired Reruns

Let $\mathcal{D}$ be a task distribution, $x \sim \mathcal{D}$ a task, A the agent that receives and uses feedback, and $y(x, \tau) \in \{0, 1\}$ a strict pass indicator. An initial attempt (run0) produces a raw trajectory τ_0. For a fair comparison, all trajectory-feedback methods analyze this same τ_0. Each generator g produces feedback f_g for its own subsequent attempt (run1):

$$\tau_0 = A(x; \emptyset, s_0), \qquad f_g = g(x, \tau_0), \qquad \tau_1^g = A(x; f_g, s_0), \qquad y_1^g = y(x, \tau_1^g). \tag{2}$$

Each execution independently recreates the initial environment state s_0 from the task specification; run1 also uses a fresh model context. Feedback is the only channel for transferring information between executions. Figure 3 summarizes the procedure.

3.2 Available Information

Feedback generators receive task x and the messages, actions, and observations in source trajectory τ_0. Under *post-hoc outcome blindness*, they cannot access post-hoc rewards, hidden tests or verifier outputs, reference answers or trajectories, or human correctness labels. Verifier outcomes serve only outcome evaluation, except in the explicitly privileged Verifier Feedback reference.

3.3 Measuring Feedback Effectiveness

We measure feedback effectiveness by the change in strict pass probability from the initial execution. For a fixed agent A and feedback generator g, we define this change as *feedback utility*:

$$U(g \mid A, \mathcal{D}) = \mathbb{E}_{x \sim \mathcal{D}}\left[y(x, \tau_1^g) - y(x, \tau_0)\right]. \tag{3}$$

Comparisons fix the recipient, tasks, prompt, tools, and execution budget. Shared initial executions give gains and subsequent pass rates the same ranking. Run0 provides a common reference for measuring these changes for a given agent.

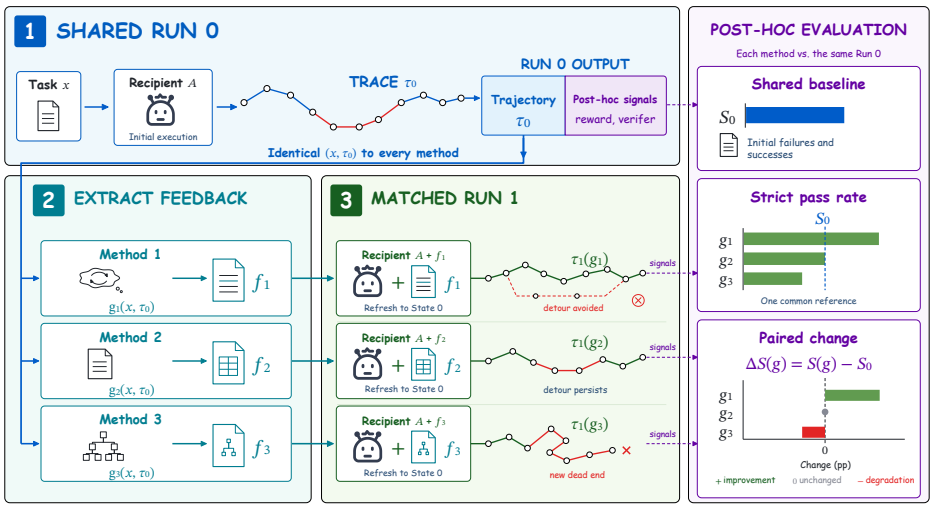


Figure 3: REFIT uses a shared run0 to construct feedback for each method's separate run1, starting from initial environment state s_0 and a fresh model context. Performance changes are measured against this common initial execution. Feedback generation is separated from post-hoc outcome evaluation.

4 Experiments

We evaluate task performance, token use, and subsequent behavior on Terminal-Bench 2.1. The comparisons test complete feedback methods and two core DENSE designs; an exploratory extension tests repeated feedback, and the case analysis examines how feedback relates to the recipient's actions.

4.1 Experimental Setup

Tasks. We use Terminal-Bench 2.1 (Merrill et al., 2026) to study trajectory reuse in complex, long-horizon tasks. Its containerized terminal environments require sustained tool interaction across multiple steps, approximating practical agent workflows. The 89 tasks span software engineering, file and data processing, and scientific computing, providing varied skills and failure modes within one benchmark.

Recipient models. We evaluate MiniMax-M2.7 (Chen et al., 2026a), DeepSeek V4 Pro (DeepSeek-AI et al., 2026), GPT-5.5 (OpenAI, 2026a), and Kimi K2.6 (Moonshot AI, 2026) as agents receiving and using feedback (*recipients*). These include open-weight and closed models to compare their use of trajectory feedback. MiniMax, GPT, and Kimi use all 89 tasks; DeepSeek uses the 81 supported by its text endpoint, excluding eight tasks requiring direct image or video input.

Execution settings. We follow REFIT (Section 3) with three paired repetitions per task and recipient. Within each repetition, trajectory-feedback methods share one fixed run0 and provide feedback for separate run1 executions. For each recipient, run0 and run1 use the same task instruction, tools, and execution budget.

Feedback generation model. All-at-once, Step-by-step, DENSE, and Advisor use Gemini 3 Flash Preview to hold the feedback model fixed while comparing information inputs and extraction procedures. All conditions use the same complete-feedback length limit; configurations and prompts appear in Appendices A and E.

Metrics. Strict pass requires verifier reward exactly 1. Let y_0 and y_1^g be the strict pass indicators for initial and subsequent executions. We define

$$P_0 = \overline{y_0}, \qquad P(g) = \overline{y_1^g}, \qquad \Delta P(g) = P(g) - P_0. \tag{4}$$

For the single-step comparisons, averages first combine the three repetitions within each task, then weight tasks equally. Rates are percentages; absolute changes from baseline are in percentage points (pp), computed as the difference between the two rates.

4.2 Comparison Methods

All-at-once analyzes the complete trajectory globally; Step-by-step scores actions from decision-time prefixes. Advisor tests the feedback model's task priors; Self-reflection leaves analysis of the raw trajectory to the next-round agent. DENSE (ours) extracts nested shortcut trees. Verifier Feedback (VF) adds post-hoc signals as a privileged reference. Table 1 describes each method; run0 is the initial-execution reference.

Table 1: Feedback supplied to the subsequent execution. Only Verifier Feedback (VF) uses post-hoc reward or verifier output. Advisor denotes the instruction-only advisor.

Method	Feedback and construction	Role
Advisor	Advice based only on the task description, without the execution trajectory.	Task-prior control
Self-reflection	Raw trajectory for the next-round agent to summarize past experience and reflect on it.	Self-reflection baseline
All-at-once	One call analyzes the full trajectory and returns action-level assessments and rationales.	Global baseline
Step-by-step	One call per action assesses history through that action, excluding its result and later events.	Sequential baseline
DENSE (ours)	Nested subtasks compress redundant attempts while retaining key evidence and unresolved issues.	Proposed method
Verifier Feedback	The initial trajectory, its final reward, and verifier test output.	Privileged ref.

All-at-once and DENSE analyze complete-trajectory action outcomes; Step-by-step scoring excludes the target action's result and later events. Delivered feedback combines source information with varied analyses, covering self-reflection, action assessment, and hierarchical extraction.

4.3 Task Performance

Table 2 compares strict pass rates. All-at-once and Step-by-step outperform Advisor on all four recipients and Self-reflection on MiniMax, GPT, and Kimi. All-at-once ranks second among non-privileged feedback methods on MiniMax and Kimi; the two methods tie for second on GPT. On DeepSeek, Step-by-step reaches 65.43%, above Self-reflection's 63.79%, while All-at-once reaches 62.14%. Analyzing actions thus yields higher strict pass rates than providing the raw trajectory alone in most comparisons, with outcomes also depending on the extraction procedure.

DENSE achieves the highest strict pass rate among the tested non-privileged methods on **all four recipients**: 52.43%, 73.25%, 75.66%, and 50.94% on MiniMax, DeepSeek, GPT, and Kimi, respectively. These improve on the common initial execution by 7.87, 21.81, 7.12, and 11.61 pp.

Table 2: Strict pass rate (%) / change from baseline (pp), computed before rounding. Baseline aggregates the initial executions shared by all methods. Verifier Feedback (VF)† has privileged outcome information. Bold / underline mark the highest / second-highest rates per recipient, including Baseline and excluding VF; ties share a mark.

Method	MiniMax-M2.7	DeepSeek V4 Pro	GPT-5.5	Kimi K2.6
Baseline	<u>44.57</u>	51.44	68.54	39.33
Advisor	41.20 / -3.37	53.91 / +2.47	69.29 / +0.75	43.82 / +4.49
Self-reflection	38.58 / -5.99	63.79 / +12.35	72.28 / +3.75	46.07 / +6.74
All-at-once	<u>44.57 / +0.00</u>	62.14 / +10.70	<u>73.03 / +4.49</u>	<u>47.94 / +8.61</u>
Step-by-step	42.70 / -1.87	<u>65.43 / +13.99</u>	<u>73.03 / +4.49</u>	46.82 / +7.49
DENSE (ours)	**52.43 / +7.87**	**73.25 / +21.81**	**75.66 / +7.12**	**50.94 / +11.61**
Verifier Feedback†	44.94 / +0.37	66.67 / +15.23	83.90 / +15.36	53.93 / +14.61

On MiniMax, Self-reflection and Step-by-step fall below baseline by 5.99 and 1.87 pp, while All-at-once matches it. DENSE is the only non-privileged feedback method with a positive gain, exceeding even Verifier Feedback by 7.49 pp. Appendix D.4 decomposes this gap and examines how feedback guides further tuning and delivery on `tune-mjcf`. Verifier Feedback exceeds DENSE on GPT and

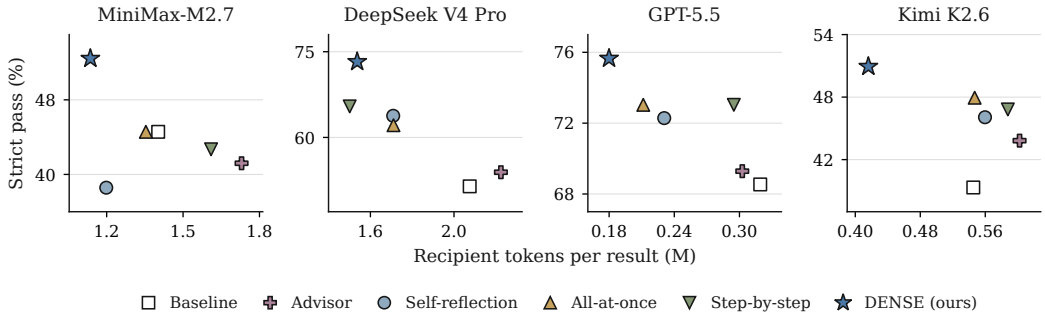


Figure 4: Strict pass rate versus observed recipient tokens per result for non-privileged conditions. Means use three repetitions and equal task weights: 89 tasks, or 81 for DeepSeek. Baseline is run0; other points count run1 execution only, excluding feedback generation and source run0. Axes vary by model; upper left is better. Token accounting follows Appendix A.

Table 3: Observed recipient token use: mean execution tokens (thousands) / percentage change from the baseline row. Negative changes indicate reduced use. †Privileged reference. Bold / underline mark the lowest / second-lowest token use per recipient, excluding VF; ties share a mark.

Method	MiniMax-M2.7	DeepSeek V4 Pro	GPT-5.5	Kimi K2.6
Baseline	1,402.2	2,075.0	319.1	<u>545.3</u>
Advisor	1,729.6 / +23.4%	2,224.4 / +7.2%	302.6 / -5.2%	602.2 / +10.4%
Self-reflection	<u>1,198.3 / -14.5%</u>	1,709.0 / -17.6%	230.7 / -27.7%	559.7 / +2.6%
All-at-once	1,353.9 / -3.4%	1,710.5 / -17.6%	<u>211.4 / -33.7%</u>	546.9 / +0.3%
Step-by-step	1,609.8 / +14.8%	**1,500.9 / -27.7%**	294.8 / -7.6%	587.8 / +7.8%
DENSE (ours)	**1,136.0 / -19.0%**	<u>1,536.8 / -25.9%</u>	**179.9 / -43.6%**	**416.1 / -23.7%**
Verifier Feedback†	1,718.5 / +22.6%	1,835.0 / -11.6%	260.2 / -18.5%	682.0 / +25.1%

Kimi but trails it by 6.58 pp on DeepSeek, illustrating how the benefit of outcome information varies across agents.

4.4 Computational Cost

Feedback methods emphasize different reusable experience from the same run0, guiding exploration in run1. To compare resource use across differently priced models, we measure tokens: execution input, output, and cache reads and writes. Appendix A details aggregation and attribution.

DENSE reduces observed recipient tokens relative to the displayed baseline by 19.0%, 25.9%, 43.6%, and 23.7% for MiniMax, DeepSeek, GPT, and Kimi, respectively (Table 3). Together with the pass-rate gains, these results show that subsequent executions complete more tasks with fewer observed tokens.

Figure 4 jointly compares task performance and recipient token use. DENSE combines the highest non-privileged pass rate with the lowest execution-token use on MiniMax, GPT, and Kimi. On DeepSeek, Step-by-step uses fewer execution tokens, while DENSE achieves a higher pass rate.

4.5 Ablation Study

Using GPT-5.5 on all 89 tasks with three paired repetitions, we compare two ablations under the same REFIT settings. Without S&R, we retain the nested subtask tree but omit shortcut construction and issue reconciliation. Without hierarchy, we replace nested analysis with a single-root, whole-trajectory summary.

Table 4: GPT-5.5 ablations: mean execution tokens (K) and strict pass (%).

Method	Token cost (K)↓	Strict pass (%)↑
run0	319.1	68.54
w/o S&R	228.6	72.28
w/o hierarchy	199.2	72.66
DENSE	**179.9**	**75.66**

The ablations reduce strict pass by 3.37 and 3.00 pp, respectively, relative to full DENSE, which also uses fewer observed recipient tokens (Table 4). These results support combining both designs. Appendix A.2 provides implementation details and a verifier sensitivity check.

4.6 Iterative Agent Refinement

An exploratory extension compares DENSE with Self-reflection under repeated execution on the hard subset of Terminal-Bench 2.1,[2] using the same four recipients. Appendix A.5 details the execution protocol and paired coverage.

Following the cumulative solved-task reporting in Reflexion (Shinn et al., 2023), we report *cumulative pass rate*: the fraction of task–repetition pairs solved at least once within the allowed attempts.

After three feedback iterations, DENSE exceeds Self-reflection by 3.3, 7.8, 6.7, and 10.3 pp on GPT-5.5, MiniMax-M2.7, Kimi K2.6, and DeepSeek V4 Pro, respectively (Figure 5). These results suggest that structured trajectory distillation can support iterative agent refinement beyond a single feedback step.

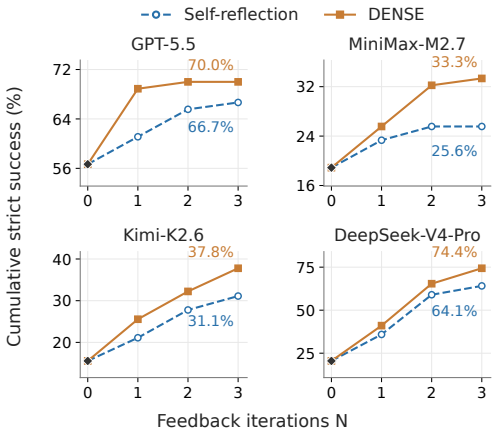


Figure 5: Cumulative strict success on Terminal-Bench 2.1 hard tasks through three feedback rounds.

5 Discussion

Recipient differences, feedback comparisons, and an illustrative case clarify what experience online trajectories contain and how DENSE makes it useful.

5.1 Differences in Recipients' Use of Feedback

DENSE's gain over run0 ranges from 7.12 pp on GPT to 21.81 pp on DeepSeek. MiniMax gains 7.87 pp and exceeds Verifier Feedback (VF) by 7.49 pp. VF adds post-hoc reward and verifier output to raw history. The paired analysis in Appendix D.4 shows that DENSE's advantage mainly reflects more retained successes in fresh executions.

On `tune-mjcf`, DENSE distinguishes prior tuning progress from the remaining speed target while retaining the physics-accuracy requirement. The recipient tests configurations, writes a tuned model, and verifies it in 25 tool calls, reaching approximately 55% of the reference simulation time. The VF recipient also makes 22 tool calls but continues exploring parameters without producing the required model file. This contrast illustrates how trajectory feedback can connect prior exploration to remaining goals and carry execution through to a verified deliverable.

5.2 Useful Experience in Unsupervised Online Trajectories

All-at-once, Step-by-step, and DENSE outperform task-prior-only Advisor on all four recipients, showing that analyzed traces are more useful than general advice here. The first two also improve on Self-reflection in most comparisons; DENSE leads it on MiniMax, DeepSeek, GPT, and Kimi by 13.86, 9.47, 3.37, and 4.87 pp, respectively. Thus, trajectories without post-hoc labels still contain useful experience, and selecting, connecting, and organizing it can help recipients use it more effectively than self-reflection on raw trajectories alone. Considerable room remains to improve how trajectory evidence is selected, connected, and organized, and how its presentation is adapted to different recipients.

The iterative extension (Section 4.6) suggests that structured trajectory distillation could provide useful feedback for broader recursive self-improvement (RSI) systems. Here, the recipient and distillation procedure remain fixed; improving the feedback mechanism itself and transferring gains across tasks remain open questions.

5.3 From Local Discoveries to Task Completion

The `password-recovery` task requires recovering a password from a data image and writing it to a specified file. The initial trajectory contains two password fragments, but execution continues exploring without writing the file. DENSE separates failed reconnaissance from successful fragment discovery into subtasks, retaining failed approaches and reusable evidence in shortcuts. The parent

[2]Tasks labeled `hard` in the official Terminal-Bench 2.1 difficulty annotations.

connects fragment concatenation to the unfinished file-writing requirement, turning local discoveries into a concrete delivery step.

In a matched comparison with Kimi K2.6, the DENSE-guided execution checks the environment, verifies the fragments, and writes the result in **three tool calls**, passing verification. The other four feedback conditions use **25–47 calls** and leave the file absent; All-at-once also assembles the same candidate but does not write it. This contrast illustrates the value of connecting available discoveries to unfinished task requirements so that reusable evidence supports concrete completion steps. Appendix D provides full feedback, action records, and outcomes across repetitions.

6 Related Work

6.1 Trajectory Reuse and Refinement

Prior work uses trajectories to guide later attempts through reflection and debugging (Shinn et al., 2023; Zhu et al., 2025; 2026; Ma et al., 2025), retrieve experience for future tasks (Fang et al., 2026; Zhao et al., 2024; Liu et al., 2024), and reduce execution context (Ren et al., 2026; Xiao et al., 2026). Reward-verified retries also support training (Shi et al., 2026), while successful rollouts inform planning (Gonzalez-Pumariega et al., 2026). DENSE extracts reusable evidence without post-hoc supervision by organizing trajectories hierarchically and summarizing subtask states.

6.2 Process Evaluation and Feedback Effectiveness

Process evaluation assesses execution quality (Fan et al., 2026; Lù et al., 2025; Li et al., 2026; Zhai et al., 2026) and attributes failures to agents or steps (Zhang et al., 2025; Liu et al., 2026; Barke et al., 2026; Chen et al., 2026b). Critique evaluation measures whether feedback improves subsequent answers (Tang et al., 2025; Yu et al., 2025; Sui & Hooi, 2026), a criterion that can diverge from explanation ratings (Kunz et al., 2022). Recovery studies examine restarts with carried-over changes or reconstructed failure states (Wang et al., 2026; Tan et al., 2025); self-correction studies highlight the need to control resampling and privileged information (Kamoi et al., 2024; İşcan, 2026). REFIT compares feedback methods using a shared initial trajectory and matched rerun conditions, measuring changes in strict pass rate without exposing post-execution outcomes to feedback generation.

7 Conclusion

We study how execution traces guide new attempts at the same task without external outcome labels. DENSE organizes useful steps, evidence of resolved errors, and unfinished requirements in nested shortcut trees. REFIT compares task success rates against initial executions, using shared source trajectories and reset environments and model contexts. On Terminal-Bench 2.1, DENSE achieves the highest strict pass rate among tested non-privileged methods across four agent models, improving over initial executions by 7.12–21.81 pp with 19.0–43.6% fewer observed agent tokens in new attempts. GPT-5.5 ablations support combining nested subtask analysis with shortcut construction and issue reconciliation. On MiniMax, organized evidence outperforms raw history augmented with verifier signals. An exploratory hard-task extension shows higher cumulative pass rates than Self-reflection after three feedback iterations across all four models. Our work points to promising research directions in the analysis of unlabeled online trajectories, with potential implications for agent recursive self-improvement.

Limitations

Our evaluation is limited to one benchmark, Terminal-Bench 2.1. Nevertheless, its diverse tasks span software engineering, data processing, and scientific computing while approximating practical agent workflows. Gains across four recipients demonstrate DENSE's effectiveness in this varied setting and support our central insight: trajectories without post-hoc supervision can yield useful experience. Future work can validate these findings across more benchmarks and environments.

DENSE's multi-stage distillation adds computation, and feedback may increase rerun context. A lower-priced generator can offset this overhead: with Gemini 3 Flash Preview and GPT-5.5, DENSE feedback costs about $0.14, while feedback plus rerun costs 36.6% less than run0 and strict pass rate is 7.12 pp higher (Appendix B.1). This tradeoff depends on model prices and execution costs; reducing generation overhead remains a direction for improvement.

AI USE STATEMENT

Generative AI tools assisted with drafting sections of the manuscript, revising and polishing the text, and retrieving and discovering relevant literature and research materials. They also assisted with organizing case-study evidence, preparing tables and figures, LaTeX formatting, code implementation, and quality review of code and research artifacts. The authors reviewed all final code, experimental procedures, and resulting data and artifacts, personally ran the experiments, and collected and organized the results. The authors checked the case-study descriptions against the underlying execution traces and verified the reported numerical results against the experimental records. All AI-assisted content was reviewed by the authors, who take full responsibility for the final text, claims, code, and artifacts.

REPRODUCIBILITY STATEMENT

The anonymous supplementary materials provide code, prompts, configurations, and data for reproducing the reported experiments. Sections 3 and 4 describe the evaluation protocol; Appendices A, E, and F provide experimental settings, metric definitions, prompts, and algorithm details.

ETHICS STATEMENT

This study evaluates agents in the containerized environments of Terminal-Bench 2.1 and does not involve human-subject experiments. Execution traces may contain sensitive information in real-world deployments; applying DENSE to such traces would require appropriate access controls and removal of sensitive content. Distilled feedback may preserve errors or unsafe actions from the source trajectory, so improved benchmark performance should not be interpreted as a guarantee of safe deployment.

REFERENCES

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing AI agent failures from execution trajectories, 2026. URL https://arxiv.org/abs/2602.02475.

Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, et al. The MiniMax-M2 series: Mini activations unleashing max real-world intelligence, 2026a. URL https://arxiv.org/abs/2605.26494.

Jing Chen, Yang Sun, Li Zhang, Lin Xu, and Jie Shi. Long-horizon agent trajectory attribution: A unified benchmark and fine-grained annotation framework, 2026b. URL https://arxiv.org/abs/2608.06909.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Shengda Fan, Xuyan Ye, Yupeng Huo, Zhi-Yuan Chen, Yiju Guo, Shenzhi Yang, Wenkai Yang, Shuqi Ye, Jingwen Chen, Haotian Chen, Xin Cong, and Yankai Lin. AgentProcessBench: Diagnosing step-level process quality in tool-using agents, 2026. URL https://arxiv.org/abs/2603.14465.

Gaodan Fang, Vatche Isahagian, K. R. Jayaram, Ritesh Kumar, Vinod Muthusamy, Punleuk Oum, and Gegi Thomas. Trajectory-informed memory generation for self-improving agent systems, 2026. URL https://arxiv.org/abs/2603.10600.

Gonzalo Gonzalez-Pumariega, Saaket Agashe, Jiachen Yang, Ang Li, and Xin Eric Wang. On the reliability of computer use agents, 2026. URL https://arxiv.org/abs/2604.17849.

Google. Gemini developer API pricing, 2026. URL https://ai.google.dev/gemini-api/docs/pricing. Accessed September 9, 2026.

Mehmet İşcan. Falsification, not exposure: An internally preregistered placebo-controlled decomposition of self-repair feedback in frozen small code models, 2026. URL https://arxiv.org/abs/2606.31511.

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. *Transactions of the Association for Computational Linguistics*, 12:1417–1440, 2024. doi: 10.1162/tacl_a_00713. URL https://aclanthology.org/2024.tacl-1.78/.

Jenny Kunz, Martin Jirenius, Oskar Holmström, and Marco Kuhlmann. Human ratings do not reflect downstream utility: A study of free-text explanations for model predictions. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 164–177. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.blackboxnlp-1.14. URL https://aclanthology.org/2022.blackboxnlp-1.14/.

Dawei Li, Yuguang Yao, Zhen Tan, Huan Liu, and Ruocheng Guo. ToolPRMBench: Evaluating and advancing process reward models for tool-using agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 12378–12391. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.602. URL https://aclanthology.org/2026.findings-acl.602/.

Jiale Liu, Huajun Xi, Shaokun Zhang, Yifan Zeng, Tianwei Yue, Chi Wang, Jian Kang, Qingyun Wu, and Huazheng Wang. Who&When Pro: Can LLMs really attribute failures in AI agents?, 2026. URL https://arxiv.org/abs/2607.09996.

Zhiwei Liu, Weiran Yao, Jianguo Zhang, Rithesh Murthy, Liangwei Yang, Zuxin Liu, Tian Lan, Ming Zhu, Juntao Tan, Shirley Kokane, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. PRACT: Optimizing principled reasoning and acting of LLM agent. In *Proceedings of the 28th Conference on Computational Natural Language Learning*, pp. 442–446. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.conll-1.33. URL https://aclanthology.org/2024.conll-1.33/.

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

Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. DoVer: Intervention-driven auto debugging for LLM multi-agent systems, 2025. URL https://arxiv.org/abs/2512.06749. arXiv preprint, revised January 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Moonshot AI. Kimi K2.6: Advancing open-source coding, April 2026. URL https://www.kimi.ai/blog/kimi-k2-6. Accessed September 17, 2026.

OpenAI. Introducing GPT-5.5, April 2026a. URL https://openai.com/index/introducing-gpt-5-5/. Accessed September 17, 2026.

OpenAI. GPT-5.5 Model: Pricing, 2026b. URL https://developers.openai.com/api/docs/models/gpt-5.5. Accessed September 9, 2026.

Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, and Chenghua Lin. A self-evolving framework for efficient terminal agents via observational context compression, 2026. URL https://arxiv.org/abs/2604.19572.

Weijie Shi, Yanxi Chen, Zexi Li, Xuchen Pan, Yuchang Sun, Jiajie Xu, Xiaofang Zhou, and Yaliang Li. R^3L: Reflect-then-retry reinforcement learning with language-guided exploration, pivotal credit, and positive amplification, 2026. URL https://arxiv.org/abs/2601.03715.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Yuan Sui and Bryan Hooi. Conversation for non-verifiable learning: Self-evolving LLMs through meta-evaluation. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026. URL https://arxiv.org/abs/2601.21464.

Shangyin Tan, Kevin Lin, Koushik Sen, and Matei A. Zaharia. Recovery-Bench: Evaluating agentic recovery from mistakes. In *NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling*, 2025. URL https://openreview.net/forum?id=8FZRnDgDxq.

Zhengyang Tang, Ziniu Li, Zhenyang Xiao, Tian Ding, Ruoyu Sun, Benyou Wang, Dayiheng Liu, Fei Huang, Tianyu Liu, Bowen Yu, and Junyang Lin. RealCritic: Towards effectiveness-driven evaluation of language model critiques, 2025. URL https://arxiv.org/abs/2501.14492.

Tongyi DeepResearch Team et al. Tongyi DeepResearch technical report. *arXiv preprint arXiv:2510.24701*, 2025. URL https://arxiv.org/abs/2510.24701.

Chenyu Wang, Yunbo Lyu, Junda He, Zhou Yang, Chenxing Zhong, Yaniv Harel, and David Lo. Fail-Fast, Restart-Smart: Early failure prediction and restart for SWE agentic tasks, 2026. URL https://arxiv.org/abs/2608.03222.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3(FSE):1241–1263, 2026. doi: 10.1145/3797084. URL https://doi.org/10.1145/3797084.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024. URL https://arxiv.org/abs/2404.07972.

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. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793.

Tianshu Yu, Chao Xiang, Mingchuan Yang, Pei Ke, Bosi Wen, Cunxiang Wang, Jiale Cheng, Li Zhang, Xinyu Mu, Chuxiong Sun, and Minlie Huang. Training language model to critique for better refinement. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 26760–26804. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1373. URL https://aclanthology.org/2025.findings-acl.1373/.

Yuwen Zhai, Runze Li, Liang Wang, Nian Shi, Liwu Xu, Wei Zhang, Ran Lin, Bo Xu, and Benlei Cui. GUIDE: Interpretable GUI agent evaluation via hierarchical diagnosis, 2026. URL https://arxiv.org/abs/2604.04399.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76583–76599. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhang25cq.html.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. URL https://arxiv.org/abs/2307.13854.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM agents fail and how they can learn from failures, 2025. URL https://arxiv.org/abs/2509.25370.

Kunlun Zhu, Xuyan Ye, Zhiguang Han, Yuchen Zhao, Bingxuan Li, Weijia Zhang, Muxin Tian, Xiangru Tang, Pan Lu, James Zou, Jiaxuan You, and Heng Ji. AgentDebugX: An open-source toolkit for failure observability, attribution, and recovery in LLM agents, 2026. URL https://arxiv.org/abs/2607.18754.

A Experimental Configuration and Accounting

A.1 Feedback Configuration and Budget

All generated-feedback conditions use Gemini 3 Flash Preview (API model identifier: `gemini-3-flash-preview`). DENSE builds its initial subtask tree from the full trajectory, scoring actions before producing subtask summaries. It then compresses consecutive attempts under the same parent, reconciles issue states from children to parents, and organizes the final feedback around unresolved issues. Tree feedback is limited to 180 KiB within a total feedback budget of 200 KiB. Other methods use the same total budget. When shortening feedback, DENSE operates on complete semantic blocks to avoid cutting through evidence. Appendix E presents the prompt roles, input views, and instructions used to construct and deliver these feedback representations.

A.2 DENSE Ablation Details

The ablations in Section 4.5 use GPT-5.5 with high reasoning effort and Gemini 3 Flash Preview as the feedback model. The hierarchical control keeps the full-trajectory tree builder and its score-then-summarize procedure; only the subsequent shortcut and reconciliation stages are removed. The single-root control instead makes one logical analysis call over the task and all recorded action–observation text, requesting completion status, evidence, lessons, and next steps. Both use only the task and visible trajectory, without post-hoc reward or verifier output.

The analysis output limit is 8,192 tokens. One single-root analysis (`dna-assembly`, repetition 3) required a 16,384-token limit to obtain a valid response with the same prompt. Retries address unscored analysis or execution failures; the first valid execution result is retained, including zero and partial rewards. All four conditions have complete coverage of the same 267 task–repetition pairs.

Token cost in Table 4 counts recipient execution only, using the accounting in Appendix A.3; feedback generation is excluded. All 1,068 results have recorded usage; 64 report completed calls only, omitting unfinished calls. The table therefore reports observed token use.

Historical verifier logs for `torch-pipeline-parallelism` contain port binding conflicts in ten scored runs across the four conditions, including two DENSE runs. We retain those recorded results in the main table and check sensitivity by excluding all three repetitions of this task from every condition. On the remaining 264 pairs, strict pass rates are 69.32% for run0, 72.73% without shortcut/reconcile, 73.48% for the single-root summary, and 76.14% for full DENSE. The DENSE gaps are 3.41 and 2.65 pp, respectively.

A.3 Recipient Token Accounting

Recipient tokens sum uncached input, cache-read input, cache-creation input, and output. Relative change is $(\overline{T}_g/\overline{T}_0 - 1)$, reported as a percentage, where $\overline{T}_0$ is the mean in the displayed baseline row and $\overline{T}_g$ is the method mean. Both average repetitions within tasks and then give applicable tasks equal weight. Recipient usage is recorded for all 7,308 outcomes: 5,554 records report execution totals and 1,754 sum completed calls only. The latter omit unfinished calls, and unknown usage is not replaced with zero.

A.4 Feedback Cost Attribution

Combined cost attributes one feedback generation to each subsequent execution and adds its recipient usage. It excludes the initial execution unless explicitly stated. Reused feedback is attributed to each result without implying new generation spending, so these per-result estimates are not total experimental expenditure.

A.5 Iterative Refinement Protocol and Coverage

The exploratory analysis in Section 4.6 reports run0 through run3 on the 30 tasks with official difficulty label `hard`. These comprise the initial execution and three feedback-conditioned attempts. DeepSeek uses the 26 supported by its text endpoint. Each task has three repetitions. Task instructions, tools, and per-attempt execution limits are matched between methods. DENSE uses Gemini

3 Flash Preview for distillation; Self-reflection supplies the corresponding raw action–observation history for the next-round agent to summarize experience and reflect on previous attempts.

Recipient models and feedback procedures remain fixed across iterations. Every attempt resets the environment and model context. The explicit memory contains feedback from up to the three preceding attempts ($K = 3$); each new item uses only the immediately preceding execution. Previously injected feedback is removed from the trajectory view used to construct new feedback, avoiding recursive inclusion of older memory. Post-hoc rewards and verifier outputs are excluded from feedback and recipient inputs. Cumulative outcomes are computed retrospectively; no verifier-based early stopping is required to obtain them.

For recipient m and method g, let P_m be a fixed set of matched task–repetition pairs and $y^g_{m,t,r,n}$ indicate strict success at iteration n. We compute

$$C_{m,g}(N) = \frac{1}{|P_m|} \sum_{(t,r)\in P_m} \mathbf{1}\left[\max_{0\le n\le N} y^g_{m,t,r,n} = 1\right], \qquad N \in \{0,1,2,3\}. \tag{5}$$

Each repetition is evaluated separately; successes are not pooled across repetitions.

All supported hard-task pairs are included: 90 for GPT, MiniMax, and Kimi, and 78 for DeepSeek. All executions through run3 are complete for both methods. The figure weights task–repetition pairs equally; with all three repetitions available per task, this is equivalent to averaging repetitions within each task and then weighting tasks equally, as in the main comparisons.

These are descriptive comparisons within a three-round refinement horizon. Equal attempt limits do not imply equal token or monetary costs. Cumulative success measures whether a task has been solved within that horizon, not the accuracy of the latest attempt. Without an independent-retry control, the increase over run0 cannot be attributed entirely to feedback; the comparison here concerns DENSE relative to Self-reflection.

B Supplementary Cost Analysis

We report a priced cost breakdown for GPT-5.5. Aggregation and feedback attribution follow Appendix A.

B.1 Estimated Feedback and Rerun Costs for GPT-5.5

We estimate GPT-5.5 execution and Gemini 3 Flash Preview feedback costs using standard base text rates. Input/output prices per million tokens are \$5/\$30 for GPT-5.5 (OpenAI, 2026b) and \$0.50/\$3 for Gemini (Google, 2026). Input and output usage are priced separately and added. These standardized estimates use uncached rates, without long-context surcharges or unrecorded usage; actual charges may differ.

Table 5: Mean estimated USD costs over 89 tasks with three repetitions each. Baseline is the initial GPT-5.5 execution; other rows show the GPT-5.5 rerun, Gemini feedback, and their sum. Δ is the sum's percentage change from baseline, computed before rounding. Dashes mean no feedback generation. Bold / underline mark the best / second-best values: lower cost and higher pass rate are better; feedback is ranked among generators.

Method	Execution (\$)	Generator (\$)	Total (\$) / Δ	Pass (%)
Baseline	1.8085	—	1.8085	68.54
Advisor	1.7067	**0.0024**	1.7091 / -5.5%	69.29
Self-reflection	1.2596	—	1.2596 / -30.4%	72.28
All-at-once	<u>1.1565</u>	<u>0.0138</u>	<u>1.1704 / -35.3%</u>	<u>73.03</u>
Step-by-step	1.5947	0.1207	1.7154 / -5.2%	<u>73.03</u>
DENSE (ours)	**1.0075**	0.1396	**1.1470 / -36.6%**	**75.66**

DENSE feedback costs \$0.14, or 7.7% of the \$1.81 baseline and 13.9% of its \$1.01 rerun. Feedback plus rerun costs \$1.15, **36.6% below baseline**, while strict pass rate increases from 68.54% to

75.66%. In this configuration, a small additional feedback cost accompanies lower rerun spending and higher task success. Even with feedback generation included, the estimated cost of this rerun remains below the initial-execution reference. Figure 11 visualizes this decomposition with an explicit run0 baseline; its gray segments represent the feedback generator.

Including the initial execution gives $2.96 for the complete DENSE workflow; cost attribution follows Appendix A.4.

C Anatomy of DENSE Feedback

C.1 Feedback Interface and Reading Guide

DENSE presents an execution as a hierarchy of task progress, reusable evidence, and remaining problems. Whereas Section 2 describes its construction, this appendix examines the artifact delivered to the recipient. The feedback begins with the fresh-environment reminder reproduced in Appendix E: files and processes mentioned in the report belong to the previous attempt, and required outputs must be recreated. The report then gives a root summary, nested tasks, compressed attempts, and selected action–observation evidence.

Table 6 summarizes how to read these components. Task completion and shortcut outcome describe different scopes. A shortcut can succeed at recovering a local path while its parent task remains incomplete. Moreover, `complete` is a judgment supported by the historical trajectory, not a statement that the hidden verifier passed. Numeric action scores used during construction are omitted from the selected final feedback.

Table 6: Reading the recipient-visible DENSE artifact. Field names follow the emitted report; the explanation describes their role rather than adding information to the recipient's input.

Visible component	Interpretation
Root `summary`, `state`, and `completion-scope`	Overall historical progress and the scope of the completion judgment.
Nested `task` blocks	Subtask boundaries, summaries, and completion states.
`shortcut`, `outcome`, and `source`	A compressed group of attempts, its local result, and source nodes.
`dead-end` / `working-path`	Unproductive attempts and useful procedures or intermediate results.
`open-issue` / `evidence`	A remaining problem and the historical evidence supporting it.
`key-action`, `tool-call`, and `observation`	Selected executable details and their observed consequences.

C.2 Hierarchy, Shortcuts, and Unresolved Issues

The `largest-eigenval` task requires a dominant eigenpair solver for small real matrices, including cases with complex eigenpairs, that outperforms a NumPy reference. We inspect Kimi K2.6, repeat 2. The initial execution established timing baselines and compared analytical formulas, iterative methods, and library calls. Some local approaches were useful, but manually reconstructing complex eigenvectors from LAPACK's real-valued representation left a performance bottleneck for larger sizes.

Figure 6 relates the source phases, reconciled tree, and delivered feedback. The baseline-and-constraints branch, n30, is complete; the solver-optimization branch, n34, remains incomplete. Under n34, `shortcut_5` combines n31–n33 with a `partial` outcome, retaining both useful computation paths and the unresolved reconstruction overhead. The complete branch is rendered compactly: its internal `shortcut_0` is not displayed separately, although the selected baseline action a2 remains visible. The incomplete branch exposes its issue and supporting actions, including a28. Thus, the reconciled tree and the final feedback are related representations with different levels of detail.

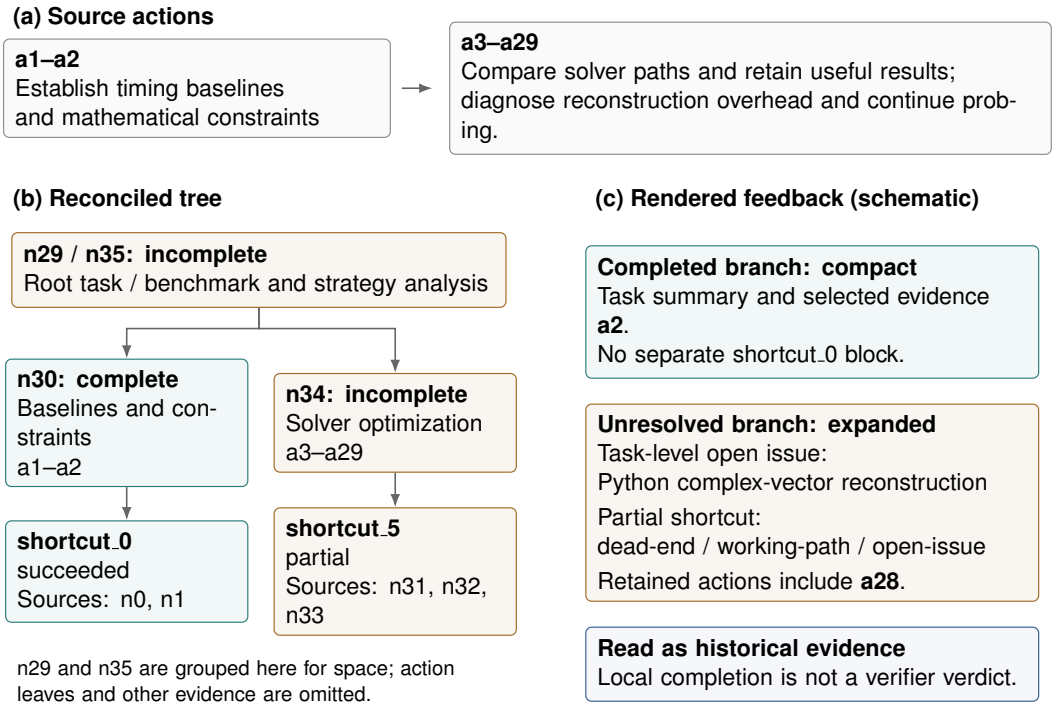


Figure 6: Structure and rendering for `largest-eigenval`, Kimi K2.6, repeat 2. This is a simplified view of the actual artifact, with source IDs retained and node descriptions shortened. Colors and explicit state labels distinguish completed content from unresolved content. The right panel paraphrases the rendered organization; Figure 7 provides the original wording.

The excerpt in Figure 7 shows how these distinctions appear in the input itself. An open issue states the bottleneck at the task level, while the partial shortcut records both unsuccessful approaches and reusable results. Evidence identifiers connect this synthesis to the retained actions. These identifiers are references within the source analysis; a1, a2, and so on enumerate serialized actions, each of which may contain multiple tool invocations. They are not counts of shell commands or elapsed time.

DENSE feedback excerpt: largest-eigenval

```
<shortcut-tree state="incomplete" [...]
  completion-scope="trajectory_local" [...]>
  [...]
  <task state="incomplete" [...]>
    [...]
    <task state="complete" [...]>
      <key-action source="a2" reason="critic-selected">
        [...]
      </key-action>
    </task>
    <task state="incomplete" [...]>
      <open-issue evidence="n27">Python-level overhead for manual
↪  complex eigenvector reconstruction (extracting pairs from real
↪  columns) makes the implementation slower than np.linalg.eig for
↪  6x6 to 10x10 matrices.</open-issue>
      <shortcut outcome="partial" source="n31,n32,n33">
```

DENSE feedback excerpt: largest-eigenval (continued)

```
        <dead-end>Pure-Python iterative methods (power/inverse
↪  iteration) and manual complex eigenvector reconstruction for
↪  matrices &gt; 4x4, which introduces enough Python-level overhead
↪  to negate the speed gains from bypassing high-level NumPy/SciPy
↪  wrappers.</dead-end>
        <working-path>Implement a 2x2 analytical solver using the
↪  quadratic formula (n9) and utilize direct LAPACK dgeev calls to
↪  bypass NumPy wrapper overhead for 4x4 matrices
↪  (n27).</working-path>
        [...]
        <key-action source="a28" reason="critic-selected">
          [...]
        </key-action>
        [...]
      </shortcut>
    </task>
  </task>
</shortcut-tree>
```

Figure 7: Selected original spans of the delivered feedback. Indentation and wrapping are adjusted for readability; `[...]` explicitly marks omitted attributes, blocks, or action contents. The retained wording is unchanged. The excerpt is abbreviated markup, not a replacement feedback template.

C.3 Local Recovery and Remaining Obligations

The `gpt2-codegolf` case illustrates why completion must be scoped to a subtask. In GPT-5.5 repeat 0, the original attempt implemented a compact C inference program, but generated repeated tokens. One local detour had already been recovered: unavailable inspection tools were replaced with available commands. The corresponding `shortcut_1` is marked `succeeded`, yet its parent n13 is incomplete because the checkpoint layout is still unverified. A separate shortcut covering implementation and inference remains `failed`.

The feedback explicitly preserves the distinction: "The specific internal layout and ordering of the GPT-2 weights within the binary checkpoint file are not yet verified, and the current implementation's assumptions are failing." Successful inspection and compilation therefore remain useful intermediate results without closing the inference problem. Appendix D examines how the recipient subsequently uses this information.

D Feedback and Subsequent Agent Behavior

D.1 Matched Cases and Information Views

We compare the information supplied by each feedback condition with the recipient's subsequent tool actions. Each comparison fixes the task, recipient, repetition, and source run0; feedback is consumed in a fresh run1 under the protocol in Section 3. We examine Advisor, Self-reflection, All-at-once, Step-by-step, and DENSE. In these five-condition comparisons, verifier results are used only to evaluate the completed executions. Section D.4 separately examines Verifier Feedback as a privileged-information condition.

Table 12 reports all three repetitions.

These views differ in more than the amount of context available to the extractor. Global access can still produce local action judgments; a prefix judgment can use outcomes of earlier actions; and a recipient can read the observations reproduced in a flat feedback table. We therefore distinguish facts that are present, connections explicitly made by the extractor, and remaining goals made actionable by the feedback.

Table 7: Input views and information processing in the five compared conditions. The Step-by-step feedback in these cases identifies its view as `causal_prefix`; the restriction applies to each scoring call.

Arm	Extractor input and unit	Information delivered to the recipient
Advisor	Task instruction only; task-level advice.	General strategy, risks, and a completion checklist.
Self-reflection	Serialized source trajectory; no externally generated analysis.	Raw actions and observations for the next-round agent to summarize and reflect on.
All-at-once	One joint analysis of the full trajectory.	Action-level assessments and rationales alongside the trajectory.
Step-by-step	One decision-time prefix per target action, excluding its result and later events.	Action-level assessments with a "Known at the time" basis. The delivered table also contains the source observations.
DENSE (ours)	Nested subtasks, grouped attempts, and issue reconciliation.	Task states, reusable paths, unresolved issues, and selected evidence.

D.2 Shared Evidence, Different Feedback

In `password-recovery`, Kimi K2.6 repeat 0, the task is to recover a 23-character string beginning with `8XD` and ending with `W54`, and write a candidate to `/app/recovered_passwords.txt`. The source trajectory already contains two fragments from a data image: `8XDP5Q2RT9Z` and `K7VB3BV4WW54`. Their lengths are 11 and 12; concatenating them yields a candidate satisfying the stated length and format. The initial execution nevertheless continued inspecting archive structures and byte offsets and did not create the required output file.

Table 8 compares how this shared situation is expressed. Self-reflection preserves the fragments as observations. Both flat extractors discuss useful evidence and endorse further inspection at particular steps. In Step-by-step, the basis for OA table row 23 explicitly names both fragments; their presence is not unique to DENSE. Advisor also explicitly requires writing the output, but cannot connect that requirement to evidence from an attempt it has not observed.

Table 8: Feedback excerpts for the same `password-recovery` run0, with code formatting normalized. OA row numbers refer to the feedback's action–observation table. The interpretation column is our analysis, not additional recipient input.

Arm / source	Retained feedback span	Information expressed
Advisor Strategy 5	"Format the discovered matches (one per line) and write them to /app/recovered_passwords.txt."	Explicit delivery requirement, without the discovered fragments.
Self-reflection Observation 22	`PASSWORD=8XDP5Q2RT9Z` `... K7VB3BV4WW54`	Both fragments are available; no extractor adds a synthesis.
All-at-once Reasoning 29	"Utilizing byte offsets to locate strings is a high-quality forensic approach that helps pinpoint the exact location of the target data."	A positive assessment of locating byte offsets within the search.
Step-by-step Reasoning 29	"Using grep -boa to find offsets is the correct forensic step to then examine the raw bytes around that location and reconstruct the full 23-character password."	Prefix evidence motivates further byte inspection to reconstruct the string.
DENSE Root issue, n26	"The identified password '8XDP5Q2RT9ZK7VB3BV4WW54' must be written to /app/recovered_passwords.txt."	A concrete assembled candidate is bound to the outstanding deliverable.

DENSE's root summary explicitly assembles the candidate and explains that the task remains incomplete because it has not been written out. Its tree separates unsuccessful reconnaissance from successful fragment discovery while retaining the delivery obligation above that local success. The

salient change is thus a task-level synthesis: what the available fragments jointly support, and what still has to be done. A generic inherited search issue also remains in this artifact; the example does not imply that reconciliation removes every irrelevant issue. Nor do positive flat action labels assert that the whole task has succeeded.

D.3 From Feedback to Subsequent Actions

Table 9 records the subsequent execution. DENSE's recipient lists the fresh environment, reruns `strings` on the known image, then writes and reads back the assembled candidate. The run uses three tool invocations and passes both output-file and password-match checks. The four comparison runs use 25–47 invocations and leave the required file absent.

Table 9: Observed behavior in `password-recovery`, Kimi K2.6, repeat 0. Call indices count tool invocations in recorded order; a call can contain several shell operations, and parallel calls are counted separately. "File" means the required output exists at verification.

Arm	Recorded action excerpts	Calls	File	Reward
Advisor	Searches filenames (2); inspects image context (15); continues to archive-directory signatures (47).	47	No	0
Self-reflection	Searches for a full contiguous match (3); carves and inspects archives; attempts a loopback device (25).	25	No	0
All-at-once	Scans known image patterns (2); inspects archives and offsets; prints the assembled candidate and length 23 (26).	26	No	0
Step-by-step	Searches full patterns (1–3); inspects fragments, extracts archives, and parses the central directory (35).	35	No	0
DENSE (ours)	Lists the environment (1); rechecks fragments with `strings` (2); writes the candidate and reads it back (3).	3	Yes	1

The All-at-once trace provides a useful qualification: its final invocation prints the *same* assembled candidate and confirms its length as 23. This recipient can infer the answer, but the recorded run ends without the required file. The contrast concerns the timing of the transition from investigation to delivery, not an absolute inability of the other feedback conditions to reconstruct the string. DENSE's explicit remaining obligation corresponds to that transition occurring immediately after a short check in the fresh environment. The trace establishes this correspondence; it does not reveal the recipient's internal attention or isolate the effect of the issue's position.

Carrying forward an unfinished optimization. In `largest-eigenval`, the retained reconstruction bottleneck corresponds to a subsequent implementation using `scipy.linalg.eig` with `overwrite_a=True` and `check_finite=False`. The recipient selects the largest-magnitude eigenvalue and its corresponding vector. This implementation follows from more than the issue summary: the feedback already retains an a29 experiment with SciPy and `overwrite_a=True`, whose observation records termination with exit code 143. The additional `check_finite=False` setting is absent from the feedback and appears in run1. The recipient therefore completes and extends an available experimental direction rather than receiving a finished patch.

The flat feedback also contains overhead-related evidence and library comparisons. In the representative run, DENSE passes 27/27 parameterized checks, All-at-once 26/27, and Step-by-step 25/27. Only DENSE meets the strict pass criterion in this repetition. These results support the observed implementation outcome; the performance checks remain sensitive to timing variation.

D.4 MiniMax Feedback Use and Outcome Transitions

On MiniMax-M2.7, DENSE achieves a strict pass rate of 52.43%, compared with 44.94% for Verifier Feedback (VF), a gap of $+7.49$ pp computed before rounding. We first decompose this difference over all 89 tasks and three repetitions, then examine how the two feedback conditions guide further optimization and delivery on `tune-mjcf`.

D.4.1 Retaining Success and Repairing Failures

Of the 267 source executions, 119 strictly succeed and 148 do not. We classify a subsequent execution as retaining success when both run0 and run1 succeed, regressing when only run0 succeeds, and repairing when only run1 succeeds. These are paired outcome descriptions; run1 must recreate the solution under the fresh-environment protocol, so a regression need not involve editing an existing artifact.

Table 10: MiniMax outcome transitions relative to the shared source run0. Retained and regressed partition the 119 successful sources. Repaired counts successes among the 148 unsuccessful sources. Net gain is repaired minus regressed. Each method has 267 subsequent executions.

Method	Retained	Regressed	Repaired	Net gain	Successes
Self-reflection	83	36	20	−16	103
DENSE (ours)	110	9	30	+21	140
VF	82	37	38	+1	120

VF repairs more unsuccessful sources than DENSE (38 versus 30), but loses success on many more previously successful sources (37 versus 9). Consequently, DENSE's advantage is 28 additional retained successes offset by 8 fewer repairs, yielding 20 additional successes overall. Among the 119 initially successful executions, DENSE's regression rate is 7.56%, compared with VF's 31.09%. This lower regression rate shows greater stability when reusing historical experience to recreate successful outcomes in a fresh environment.

Actual tool-use records show that 32 of VF's 37 regressions contain no tool calls; DENSE has no zero-tool execution among its 267 runs. Self-reflection also exhibits this behavior in 27 of its 36 regressions. Differences also arise during active execution: among the 49 pairs where DENSE passes and VF does not, VF uses tools in 17, including 13 with an unsuccessful run0. The following task illustrates how the recipients differ in turning further exploration into a verified deliverable.

D.4.2 Case: Completing a Constrained Simulation Optimization

Task and shared history. The `tune-mjcf` task asks the agent to tune a MuJoCo model so that simulating two seconds takes at most 60% of the reference execution time. The full final physics state must remain within an absolute tolerance of 10^{-5}, without NaN or Inf. The reference file `/app/model_ref.xml` must remain unchanged; the tuned model must be saved as `/app/model.xml`. A provided `eval.py` script supports local evaluation.

The shared source execution explores solver parameters, timestep changes, and simulation flags. Its delivered model passes the reference-integrity, file-existence, and physics-correctness checks, but fails the speed check: the measured simulation time is approximately 110% of the reference time. Both feedback conditions use this history, and their recipients start from fresh environments under the same execution limits.

Feedback distinguishes partial progress from the remaining target. VF supplies the source trajectory, its reward of 0.75, and the verifier output identifying the unmet speed requirement. DENSE instead organizes the recorded exploration around the accuracy–speed tradeoff. Its feedback retains a locally tested Newton configuration with reduced solver iterations, the accuracy violations observed in earlier parameter changes, and profiling evidence pointing to `mj_step2`. The root remains `incomplete`, with an explicit outstanding requirement:

> The target of $\leq$60% simulation time (40% speedup) has not been achieved; the best valid configuration reached only $\sim$80–84% of reference time.

This records useful intermediate progress while preserving the need to find a configuration that satisfies both constraints. The 80–84% figures refer to local trials in the history, whereas the 110% figure above comes from verification of the source execution's delivered file.

VF continues exploring without producing the required file. The VF recipient makes 22 tool calls. It reads the evaluation script and reference model, benchmarks parameter settings, and checks

XML parsing and simulation flags. Execution ends while it is still exploring configurations, without writing `/app/model.xml`. The verifier reports:

> Tuned model file does not exist

Only the unchanged-reference check passes. The file-existence check fails, and the correctness and speed checks cannot load the missing model, giving one of four checks passed.

DENSE carries parameter exploration through to delivery. The DENSE recipient makes 25 tool calls. It compares integrators and solver settings, then tests combinations of `implicitfast`, the PGS solver, and disabled contact processing. After refining the iteration count, it checks candidate settings across ten seeds. Eight iterations give a maximum observed state difference of 1.23×10^{-5}, just above the tolerance; nine reduce it to 5.66×10^{-6} while retaining approximately 55% of the reference simulation time in this local check.

The recipient writes the tuned file on its nineteenth tool call, using:

```
<option integrator="implicitfast" solver="PGS"
  iterations="9" jacobian="auto">
  <flag contact="disable"/>
</option>
```

It then runs `eval.py`, repeats the evaluation, and checks that the reference file is unchanged. The post-execution verifier passes all four checks and records simulation time at approximately 55% of the reference. The retained history supplies intermediate findings and outstanding constraints; the recipient completes the remaining search, writes the model, and validates the resulting artifact.

Table 11: MiniMax on `tune-mjcf`. Checks and tool calls describe the executions discussed above; calls are counted from actual transcript records. The final column reports strict passes across all three executions of this task for each condition.

Condition	Verifier checks passed	Tool calls	Strict passes across the task
Baseline	3/4	44	0/3
Self-reflection	3/4	32	1/3
DENSE (ours)	**4/4**	25	**3/3**
VF	1/4	22	1/3

From exploration to a tested artifact. Across the three executions of this task, DENSE passes three, while VF and Self-reflection each pass one (Table 11). The detailed comparison shows active parameter exploration under both DENSE and VF. DENSE's feedback makes partial progress and the remaining target explicit; the subsequent execution tests new combinations and completes the required artifact. The case illustrates how trajectory-derived evidence can support continued optimization through verification and delivery.

D.5 Complementary Outcomes and Overall Interpretation

For `gpt2-codegolf`, the recipient investigates checkpoint offsets and changes the mapping from logical layers to physical weight blocks to $\{0,1,4,5,6,7,8,9,10,11,2,3\}$, together with corrections to embedding and final LayerNorm offsets. The resulting 3,578-byte C program passes the verifier. The feedback identifies unverified layout assumptions and repeated token output; it does not supply this mapping. Step-by-step also succeeds in this repetition, using 11 tool calls compared with DENSE's 24. This case illustrates a useful representation of nested progress, while showing that another feedback view can support a successful and shorter execution.

The three trajectory-feedback cases show how historical observations can be organized as reusable progress and explicit remaining goals, with subsequent actions that correspond to those goals. The three-repeat results also limit a stronger interpretation: DENSE fails one `password-recovery` repetition, and alternative feedback succeeds in `gpt2-codegolf`. The five-condition comparisons

Table 12: Three-repeat outcomes. Entries report strict pass rate (%), with strict passes out of three in parentheses. The process analysis uses `password-recovery` repeat 0, `largest-eigenval` repeat 2, and `gpt2-codegolf` repeat 0. Within each case, bold / underline mark the highest / second-highest rates, including ties.

Task / recipient	Advisor	Self-reflection	All-at-once	Step-by-step	DENSE (ours)
`password-recovery`					
Kimi K2.6	0.0 (0/3)	0.0 (0/3)	0.0 (0/3)	0.0 (0/3)	**66.7 (2/3)**
`largest-eigenval`					
Kimi K2.6	0.0 (0/3)	33.3 (1/3)	33.3 (1/3)	33.3 (1/3)	**100.0 (3/3)**
`gpt2-codegolf`					
GPT-5.5	33.3 (1/3)	0.0 (0/3)	0.0 (0/3)	33.3 (1/3)	**66.7 (2/3)**

jointly change synthesis, hierarchy, length, and prominence; this analysis does not identify the causal contribution of hierarchy alone. For example, the concrete concatenated answer in the password feedback is part of the intervention, not merely a formatting change. A content-matched comparison would be needed to separate that synthesis from its hierarchical presentation.

The MiniMax `tune-mjcf` case in Section D.4 extends this analysis to privileged feedback: both recipients explore configurations, while DENSE-guided execution proceeds to a tuned model that passes verification.

E Prompts Used in the Experiments

This section organizes the prompts by their role in feedback delivery, baseline extraction, and DENSE construction and reconciliation. The boxes reproduce the English instructions from the implementation. **System message** denotes a complete system message; **excerpt** denotes a selected passage, with internal omissions marked explicitly. **Required output** gives the original output constraints. The surrounding text describes inputs, invocation points, and intended behavior. Content slots in braces and output types in angle brackets are instantiated for each example; line wrapping and indentation are adjusted for readability. `Shortcut-Tree` in the original prompts is the implementation name of DENSE. Appendix F connects these prompts through complete control flow, structured interfaces, and deterministic validation rules.

Table 13 maps each prompt to its input view. Except for the privileged Verifier Feedback condition, generators do not receive post-hoc run0 rewards, hidden verifier outputs, or reference answers. Public tests already observed within the trajectory remain admissible evidence under the boundary in Section 3.

Table 13: Prompt roles and input views. The recipient wrapper is static text; the remaining rows identify analysis-model roles.

Prompt	Role	Visible input
P1	Recipient wrapper	Generated feedback, followed by the original task
P2	Task-only Advisor	Original task instruction only
P3	Global action scoring	Task, complete OA sequence, and action indices
P4	Prefix action scoring	Task and prefix through the target action; its result is hidden
P5	Subtask boundaries	Root task, current level, and consecutively numbered queue
P6	Local action scoring	Root task, newly formed subtask, target action and observation
P7	Subtask summaries	Actions, observations, local scores, or child summaries
P8	Hierarchy termination	Root task and candidate top-level nodes with summaries
P9	Shortcut compression	Ordered direct children of one parent and evidence IDs
P10	Issue reconciliation	Cleaned subtree, inherited issues, and available evidence IDs

E.1 Recipient-Side Feedback Delivery

The run1 task message contains the feedback, a separator, and the original task instruction, in that order. Prompt P1 requires the recipient to recreate deliverables in the fresh environment and to treat historical paths, files, and processes as evidence. The slot {content} is replaced by the feedback for the selected condition. This wrapper is used for trajectory feedback and Verifier Feedback; Advisor instead uses the separate recipient template in Prompt P2. Run0 receives the original task instruction without this feedback prefix.

Prompt P1 Feedback for a fresh execution

Recipient-side template (complete)

```
# Previous Attempt Analysis

This rerun starts in a fresh isolated environment. Do not assume that files,
processes, temporary directories, or session state from the previous attempt
still exist. Paths and artifacts mentioned below are historical evidence only.
Recreate every required deliverable in the current environment. Reuse the
approach or content, not previous filesystem state.

Below is an annotated analysis of a previous attempt at this task. Use it as
evidence about what to preserve, correct, or recreate in this attempt.
<report>
{content}
</report>
```

E.2 Baseline Prompts: Task Priors, Full Trajectories, and Prefixes

Task-only Advisor. The advisor receives only the task description and returns a strategy, instruction-inferable risks, and a verification checklist. Its prompt prohibits presenting general advice as an error or observation that has already occurred. The recipient is also told that the advisory was generated without observing a previous execution.

Prompt P2 Task-only Advisor

System message

```
You are a task-only advisor for an AI agent before it begins a task. You
receive only the original task instruction. You have not observed any previous
execution, reasoning, action, tool call, tool result, test output, reward,
verifier feedback, reference answer, or hidden state. Base your advice only on
general knowledge and the task instruction. Never claim or imply that a
particular event, error, file, command result, or failure has already occurred.
Do not fabricate observations.
```

User message (requested-advisory excerpt)

```
Provide concise, actionable guidance for an agent that has not attempted the
task yet. Use these Markdown sections:
1. `### Recommended Strategy` - a practical sequence of steps;
2. `### Likely Risks` - pitfalls inferable from the instruction alone;
3. `### Verification Checklist` - checks the agent should perform before
finishing.
Do not discuss a previous attempt and do not claim access to execution
evidence. Return only the advisory Markdown.
```

Recipient-side template (complete)

```
# Task-Only Advisory

The following advisory was generated before execution from the original task
instruction alone. The advisor did not observe any previous attempt or
outcome.
<advisory>
```

Prompt P2 Task-only Advisor (continued)

```
{content}
</advisory>
```

All-at-once. One call receives the complete OA sequence and the indices of all target actions, and returns a label and rationale for every action. Later trajectory events may establish that an earlier action was redundant, undone, or invalidated. This within-trajectory hindsight does not provide access to the external verifier. Prompt P3 includes the scoring rubric and output instructions.

Prompt P3 All-at-once trajectory scoring

System message

```
You are an expert evaluator of AI agent trajectories. You are given a task
description and an agent's COMPLETE execution trajectory as an OA
(observation-action) sequence. Score EVERY action in a single pass. Because
you see the whole trajectory and all of your own verdicts at once, keep the
labels mutually consistent: judge each action by what it contributed to the
trajectory as a whole, not in isolation. An action that looked reasonable
locally but was made redundant, undone, or invalidated by what followed should
be labelled accordingly.
```

User message: scoring criteria

```
label ranges from -2 to 0, where:
- 0: an effective action that helps complete the task
- -1: an ineffective or failed action, but not severe enough to fail the task
- -2: a severe, fatal error that may directly cause the task to fail
```

User message: required-output excerpt

```
Score all of the actions listed above in ONE pass - exactly one entry per
action_index, no more and no less. Output ONLY a JSON array, nothing else:
[{"action_index": <int>, "reasoning": "<str>", "label": <int>}, ...]
Note: within each object, output reasoning first (the complete analysis), then
give label based on that reasoning. Keep each reasoning to a couple of
sentences so every action fits in one response.
```

Step-by-step. Each action is assessed in a separate call using the prefix ending at the action itself, excluding its immediate observation and all later events. The `known_basis` field cites evidence already established at that point or states what should have been checked first. A negative label therefore concerns the justification of the decision, rather than directly identifying execution failure.

Prompt P4 Step-by-step decision scoring

System message

```
You are an expert evaluator of AI agent trajectories, scoring one action of an
AI agent in real time. You are given a task description and the trajectory
PREFIX up to and including the target action - exactly what the agent itself
had seen when it decided to act. Everything that happened afterwards is
deliberately hidden from you: you do NOT know whether the action succeeded.
Judge the DECISION, not the outcome. Ask: given only the information
available at this point, was this the right move? A well-grounded action that
later fails for reasons nobody could foresee is still a good decision; a guess
that happens to work is still a bad one. Concretely, weigh whether the action:
follows from evidence already in the prefix rather than from an unverified
assumption; is the most direct available next step toward the task rather than
a detour or a repeat of work already done; is well-formed and correctly
parameterized given what is known; and gathers the information it needs before
committing to an irreversible change.
```

Prompt P4 Step-by-step decision scoring (continued)

User message: scoring criteria

```
label ranges from -2 to 0, where:
- 0: a well-justified decision given what was known at this point
- -1: a poorly justified decision - an unverified assumption, an avoidable
detour, or a repeat of work already done - but not fatal
- -2: a severely misguided decision that a careful agent could have avoided
with the information already available, and that may derail the task
```

User message: evidence and output excerpt

```
known_basis: one sentence naming the concrete evidence ALREADY IN THE PREFIX
that justifies this action (cite the index, e.g. "the file list from index
4"), or - if there is none - what the agent would have needed to check first.
Do NOT speculate about what happens next; you have not been shown it.
Output ONLY JSON, nothing else: {"action_index": <int>, "known_basis":
"<str>", "reasoning": "<str>", "label": <int>}
Note: output known_basis and reasoning before label, so the conclusion follows
from the evidence.
```

Self-reflection supplies the serialized actions and observations for the next-round agent to summarize past experience and reflect on the previous attempt. Verifier Feedback also serializes the reward and test output. Neither condition uses a separate analysis-model prompt to generate advice.

E.3 DENSE Prompts for Subtask-Tree Construction

Contiguous semantic boundaries. The boundary prompt identifies a complete sub-phase starting at the queue head; the same operation is repeated at higher levels. In Prompt P5, `action_index` denotes the current queue ordinal, rather than the original trajectory action index. Zero selects the head alone, and -1 requests more elements. At the final segment of a level, the user message explicitly forbids -1 because no further elements will arrive.

Prompt P5 DENSE: subtask boundaries

System message

```
You are an expert at structuring AI agent trajectories, reconstructing an
execution trajectory into a hierarchical subtask tree in a [bottom-up] manner.
You will see the root task description, the current tree-building level, and a
[contiguous, numbered] view of actions/sub-phases (a queue). Your task is to
decide: from the head of the queue up to which element exactly forms one
[complete and self-contained sub-phase/subtask] - i.e. those elements together
accomplish a relatively independent thing, while the element immediately
following clearly opens a new purpose. Like identifying function boundaries,
split only at natural semantic breaks; prefer to wait for more actions to enter
the view rather than forcibly merging elements that clearly span multiple
different purposes into one sub-phase. [Prefer split] when the queue
transitions from exploration/debugging to committing irreversible state
(overwriting files, opening a DB that may checkpoint/delete WAL, deleting
artifacts, installing packages, starting long-lived processes), or from
discovery to recipe execution - do not glue the trial-and-error stretch and the
final commit into one oversized sub-phase.
```

User message: required output

```
Output ONLY JSON, nothing else: {"reasoning": "<first analyze the queue view:
which elements belong to the same thing, where it breaks, or why it cannot
break yet>", "action_index": <int>}
Rules for action_index (the number is the [ordinal] labeled on each line of the
view, increasing consecutively from the head element's ordinal, NOT a 0-based
array index):
- output some ordinal i (queue-head ordinal <= i <= queue-tail ordinal): means
the elements from the queue head up to ordinal i [together form one complete
```

Prompt P5 DENSE: subtask boundaries (continued)

```
sub-phase], and ordinal i+1 onward clearly opens a new purpose. These elements
will be aggregated into an upper-level node.
- output 0: means the [single] element at the queue head is itself already a
complete sub-phase; aggregate just it alone.
- output -1: means the elements currently in view are not yet enough to form a
complete sub-phase; wait for more actions to enter the queue before deciding.
Note: reasoning must come before action_index, so the conclusion is driven by
the analysis.
```

Scoring before summarization. DENSE scores actions within each newly formed local subtask before generating its summary. The XML input marks the subtask with `focus="current-subtask"` and the action with `focus="TARGET"`; scoring also uses the observation immediately following that action. Prompt P6 grounds the assessment in both the local objective and the root task, while recognizing necessary exploration, debugging, and correction. These labels are internal analysis signals, not verifier verdicts, and are omitted from the final DENSE feedback.

Prompt P6 DENSE: local action scoring

System message

```
You are an expert evaluator of AI agent trajectories, scoring an agent
trajectory action by action inside a [just-formed sub-phase]. You will see:
the root task (instruction), an XML mini-tree whose only subtask is the current
sub-phase (focus="current-subtask"), and one [target action] marked with
focus="TARGET". Your scoring [MUST] be grounded in the root instruction and
this sub-phase's local goal:

1. Infer the sub-phase's local goal from the actions it contains
(subtitle/summary may be empty at this stage);

2. Judge whether the target action advances that local goal, is
ineffective/repetitive/a detour, or damages progress; necessary exploration,
debugging, and error-correction are valuable and should not be judged
ineffective merely because they did not immediately produce a result;

3. Align with the root task's final goal: even if locally reasonable, reflect
clear deviation or harm faithfully;

4. Combine with the environment feedback (observation) immediately following
the target action. Always keep your scoring basis on [the root task
requirements + this sub-phase's local role].
```

User message: scoring-rubric excerpt

```
label ranges from -2 to 0, where:
- 0: an effective action that helps complete the task, e.g. correctly
advancing task progress, successfully building the environment, achieving a
milestone, or advancing the local goal of its subtask / the final goal of the
root task
- -1: an ineffective or failed action (repetition, detour, no progress, a
failed tool call, or no help to the task) that leaves the task no worse off -
the agent can still recover from here. A failed attempt that is later
corrected by other means belongs to -1, not -2
- -2: an action that actively damages the task outcome, i.e. the delivered
result is wrong (or is wrongly believed to be right) because of it. Typical
cases:
```

[… excerpt omitted …]

User message: required output

```
Output ONLY JSON, nothing else: {"reasoning": "<scoring reasoning>", "label":
<int>}
Note: reasoning must come before label, so the conclusion is driven by the
reasoning.
```

Summarizing reusable evidence. Prompt P7 asks for the shortest observed working path while retaining the verification that supports it. Reusable failures belong in `dead_ends`, and unverified results belong in `open_issues`. The output fields are `subtitle`, `summary`, `artifacts`, `final_state`, `key_values`, `key_mechanisms`, `critical_order`, `dead_ends`, and `open_issues`; fields without relevant content contain `none`. Parent summaries also reconcile conflicting child values and preserve ordering requirements.

Prompt P7 DENSE: evidence-preserving summaries

System message

```
You are an expert evaluator of AI agent trajectories. A group of consecutive
actions/sub-phases has just been identified as a higher-level sub-phase. Give
it a short subtitle, and based on the sequence of actions it contains and their
execution results (and per-action labels / score reasoning when present),
distill the key information that an agent rerunning this task later can
directly reuse. Focus on: which files/artifacts were finally produced (and
their paths), what cleanup was done to the environment when the sub-phase ended
(restore/cleanup/leftovers), the key values and mappings extracted or computed,
critical ordering constraints, and whether there are any unresolved issues. In
addition, specifically distill the load-bearing key implementation mechanisms
of this sub-phase - the required step order, irreversible state changes,
correct API/parameter mappings, environment prerequisites, and (only when
present) concurrency/async/daemon design whose omission on rerun would cause
functional regression.

[Shortcut rule] The summary is what a rerun agent reads INSTEAD of the
step-by-step detail. Write it as the [shortest path that actually worked], not
as a chronicle of the search. If the agent tried tools/approaches A, B, C and
only D worked, the summary says "used D to do X" - it does NOT narrate trying
A, B, C first. [Exception - never shortcut away verification]: if the subtask
ran a check, test, or validation that substantiates its result, that step MUST
stay in the summary (state what was verified and how), because a rerun agent
that skips verification will silently ship a wrong result. Failed attempts are
not discarded: put them in `dead_ends` so the rerun agent can avoid them, but
keep them out of the summary narrative.

[Evidence-grounded values] Put a value into `key_values` only when a later
action in this sub-phase verifies it (re-read file, query/test, cross-check
against the instruction). If the agent only computed or wrote a result without
verification, put `unverified result: ...` in `open_issues` - do NOT state it
as a reusable fact in `key_values`.

[Critical ordering] Extract must-do-before constraints for
irreversible/stateful steps (overwrite, DB open that may checkpoint, delete,
install, start daemon) into `critical_order` and also mention them in the
summary when they are load-bearing.

[Mine failures] Scan execution results (and negative labels when present) for
reusable dead ends (command not found, traceback, permission errors, 404,
incompatible versions). Put those in `dead_ends`; leave one-off typos the
agent immediately corrected out.

[Parent reconcile] When children are already sub-phases with conflicting
`key_values`, keep the later verified value; mark superseded values as dead
ends - never present two contradictory authoritative parameter sets.
```

Stopping the hierarchy construction. Prompt P8 tests whether the candidate nodes form a small, coherent set of top-level phases. A `can_mount_all=true` response means that hierarchy construction can stop; it does not assert that the original task succeeded. Further natural groupings require another aggregation level.

Prompt P8 DENSE: hierarchy termination

System message

```
You are an expert at structuring AI agent trajectories, deciding whether a
hierarchical subtask tree is already [fully built]. There is currently a set
of candidate top-level nodes, plus the root task description. Only when [every
one of these nodes is exactly one top-level major phase of the root task]
(coarse-grained enough, with clear boundaries between them, together fully
covering the whole task), and the number of nodes is already few enough (a
handful of trunk phases, not a long list of fine-grained nodes that could still
be merged), should you judge the tree building complete and mount them all
directly under the root node. If these nodes are still too many, uneven in
granularity, or several of them clearly could be further aggregated into larger
phases, you [MUST] judge it not yet complete and aggregate one more level
upward. [Important] Do NOT judge it complete just because 'these nodes can all
technically be mounted under the root' - almost any node can be mounted under
the root, that is not the criterion; the criterion is whether they are already
the [coarsest-grained, no-longer-naturally-aggregatable] top-level phase
division. If candidate summaries still mix unresolved exploration with the
final recipe, or present conflicting key values that a further aggregation
could reconcile, lean toward not-yet-complete. First [node by node] briefly
state whether it holds up as a direct sub-phase of the root task and whether it
could still be aggregated with adjacent nodes, then give the overall
conclusion.
```

User message: required output

```
Output ONLY JSON, nothing else: {"reasoning": "<node-by-node analysis of
whether each is already a top-level major phase of the root task, whether it
can be aggregated further, and whether the overall count and granularity are
appropriate>", "can_mount_all": <true|false>}
Note: reasoning must come before can_mount_all. Lean strict: as long as
further aggregation is clearly still possible, output false.
```

E.4 DENSE Prompts for Compression and Issue Reconciliation

Compressing sibling attempts. The cleaner receives the ordered direct children of one parent and returns a complete partition in the same order. A `keep` group contains one source node; a `shortcut` group contains at least two consecutive source nodes. Each group records the reusable failure, shortest working path, observed local outcome, and remaining obligation, citing supplied evidence and key-action IDs. If every attempt fails, the failure and open issue must remain visible.

Prompt P9 DENSE: contiguous shortcut compression

System message

```
You are the path cleaner for Shortcut-Tree, an evidence-preserving
representation of an AI-agent execution trace. You receive the ordered direct
children of exactly one parent subtask. Partition those siblings without
reordering them. Replace a contiguous try/fail/recover stretch with one
shortcut node that records the reusable dead end, the shortest working path,
and the observed result. If every attempt failed, say so and retain an open
issue. Do not replay the search chronicle.

Numeric action labels are hints about where to inspect, never proof of success.
Success requires cited observation, verification, or an already-criticized
child subtask. Paths and artifacts are historical evidence from an earlier
attempt; never imply that they exist in the rerun environment. Never invent a
path, value, artifact, result, or verification without a supplied
source/evidence ID. Retain decisive correct or incorrect actions and any key
action identified in the input. Output a strict, complete, contiguous
partition.
```

Prompt P9 DENSE: contiguous shortcut compression (continued)

User message: required output

```
Output ONLY JSON:
{
  "groups": [
    {
      "mode": "keep|shortcut",
      "source_node_ids": ["direct-child-id", "..."],
      "title": "short factual title",
      "dead_end": "reusable failed approach and observed reason, or none",
      "working_path": "shortest evidence-backed path that worked, or empty",
      "outcome": "succeeded|failed|partial|unknown",
      "open_issue": "unresolved requirement, or empty",
      "evidence_node_ids": ["supplied-id"],
      "key_action_ids": ["supplied-action-node-id"]
    }
  ]
}
Rules: a keep group contains exactly one source; a shortcut contains at least
two contiguous sources. Every direct child must occur exactly once and in the
original order.
```

Reconciling completion and inherited issues. The critic reads the cleaned subtree and assesses coherence separately from completion. It may close an inherited issue only when a later node provides concrete recovery evidence, with supplied IDs recorded in `resolution_evidence`. The completion-scope instruction in Prompt P10 is also supplied to the cleaner: a `complete` verdict describes what the trajectory evidence supports.

Prompt P10 DENSE: evidence-based issue reconciliation

System message

```
You are the subtree critic for Shortcut-Tree. Judge only the cleaned subtree
and its cited evidence. Decide whether the local theme is internally coherent
and whether its subtask was actually completed. Identify unresolved ordinary
issues and fatal issues. A confident statement or a numeric action label is
not evidence. Success requires an observation, verification, or a coherent
completed child verdict. Historical artifacts do not persist into a rerun
environment.

You may resolve an issue inherited from a child only when a later cleaned node
contains concrete recovery evidence. For every resolved issue, return
non-empty resolution_evidence containing supplied IDs. Select key actions by
ID; never rewrite their original content. Do not output a broken flag: status
is derived deterministically after validation.
```

User message: completion scope (also used by the cleaner)

```
## Completion scope No reward or verifier output was supplied. A complete
verdict means only that the supplied trajectory evidence is internally coherent
and appears to satisfy the task. It is not verifier-confirmed correctness.
```

User message: required output

```
Output ONLY JSON:
{
  "coherence": "coherent|inconsistent|insufficient_evidence",
  "completion": "complete|incomplete|failed|unknown",
  "summary": "shortest evidence-grounded account of the final state",
  "open_issues": [
    {"issue": "description", "evidence_node_ids": ["supplied-id"]}
  ],
```

Prompt P10 DENSE: evidence-based issue reconciliation (continued)

```
  "fatal_issues": [
    {"issue": "description", "evidence_node_ids": ["supplied-id"]}
  ],
  "resolved_issue_ids": ["inherited-issue-id"],
  "resolution_evidence": {"inherited-issue-id": ["supplied-id"]},
  "key_action_ids": ["supplied-action-node-id"]
}
```

Turning structured outputs into feedback. The implementation validates partition coverage, node references, and issue states before rendering the feedback described in Section 2. Completed branches are folded, unresolved issue paths remain expanded, and selected key actions retain their original content. Rendering is deterministic and does not call a model to rewrite the final feedback, so it has no additional generation prompt.

F DENSE ALGORITHM AND VALIDATION RULES

This appendix specifies the control flow and structured interfaces of DENSE. Algorithms 1–3 connect the prompts in Appendix E: P5–P8 build the hierarchy, and P9–P10 compress and reconcile it. Model judgments determine semantic grouping, useful working paths, and recovery; deterministic operations maintain source order, check structured outputs and references, derive states, and render the feedback.

F.1 INPUTS, OUTPUTS, AND EXECUTION ORDER

The inputs are the task x and its visible trajectory $\tau = (a_1, o_1, \ldots, a_T, o_T)$. Each leaf binds an action to its immediately following observation, if present. Node IDs identify tree elements; global action indices identify source actions; queue ordinals identify positions in the current aggregation level. These three identifiers have distinct roles. The outputs are a dense source tree $\mathcal{T}$, a reconciled shortcut tree $\mathcal{T}^\star$, and the recipient-visible feedback f. Original action and observation text remains available in the indexed source tree.

Algorithm 1 DENSE: from a visible trajectory to recipient feedback

Require: Task x, trajectory τ, chunk size Δ, round cap L, byte budgets B_t, B_f
Ensure: Dense source tree $\mathcal{T}$, reconciled tree $\mathcal{T}^\star$, recipient feedback f

function DENSE($x, \tau, \Delta, L, B_t, B_f$)
  $Q \leftarrow$ ordered leaves, each binding an action to its following observation
  if $|Q| > 1$ **then**
    for $k = 1, \ldots, L$ **do**
      $Q' \leftarrow$ BUILDLEVEL(x, Q, k, Δ) ▷ Algorithm 2
      if $|Q'| = 1$ or $|Q'| = |Q|$ **then**
        $Q \leftarrow Q'$; **break**
      end if
      $s \leftarrow$ TOPLEVELCHECK($x, Q', |Q|$) ▷ P8; false on call/parse failure
      $Q \leftarrow Q'$
      if $s = \text{true}$ **then**
        break
      end if
    end for
  end if
  $\mathcal{T} \leftarrow$ ROOT(x, Q)
  Score any remaining unscored leaves in singleton local contexts ▷ P6
  Index source nodes and their original action–observation contents
  $\mathcal{T}^\star \leftarrow$ RECONCILE($x, \mathcal{T}$) ▷ Algorithm 3
  Select recipient-visible key actions from retained evidence ▷ Section F.5
  $f \leftarrow$ RENDER($\mathcal{T}^\star, B_t$) ▷ States, issues, summaries, and selected actions
  Apply the recipient wrapper and the complete-feedback budget B_f
  return $(\mathcal{T}, \mathcal{T}^\star, f)$
end function

The configured chunk size is $\Delta = 999$ nodes and the maximum number of aggregation rounds is $L = 20$. The tree and complete-feedback budgets are $B_t = 180$ KiB and $B_f = 200$ KiB, respectively. P6 uses the labels $\{-2, -1, 0\}$ internally. Each new multi-node subtask is scored before its summary is generated; leaves already scored at a lower level keep their labels. Singletons are promoted unchanged and any still-unscored actions are scored after hierarchy construction. The pseudocode presents dependencies sequentially: independent local scores, summaries, and sibling subtrees can run concurrently. All summaries for a level finish before the next-level boundary or termination calls read them.

F.2 SUBTASK BOUNDARY DECISIONS

Interface. P5 receives the root task, current level, and a consecutively numbered queue. At the leaf level it reads action text; at higher levels it reads subtask titles and summaries. Its JSON output contains `reasoning` and `action_index`. The latter identifies the end of a coherent prefix beginning at the queue head. A value of -1 requests more queue elements, and zero selects the head

alone. The prompt asks for a complete local objective at level one and progressively broader phases at higher levels.

Algorithm 2 Ordered prefix grouping and boundary normalization

Require: Task x, ordered level Q, level index k, chunk size Δ
Ensure: Ordered next level H, covering every member of Q exactly once

function BUILDLEVEL(x, Q, k, Δ)
  $W \leftarrow [\,]$; $H \leftarrow [\,]$; $h \leftarrow 1$ ▷ h is the current head ordinal in Q
  for each consecutive chunk C of at most Δ nodes from Q **do**
    Append C to W; $last \leftarrow$ (C is the final chunk)
    repeat
      $d \leftarrow$ BOUNDARY($x, W, k, h, last$)
      if $d =$ WAIT **then**
        break
      end if
      Remove prefix G of length $d - h + 1$ from W; $h \leftarrow d + 1$
      if $|G| = 1$ **then**
        Append the unchanged node in G to H
      else
        $v \leftarrow$ SUBTASK(G)
        Score unscored action leaves in the local tree rooted at v ▷ P6
        Populate v's title, summary, and structured facets ▷ P7
        Append v to H
      end if
    until $\neg last$ or $W = [\,]$ ▷ One cut per nonfinal chunk; drain the final chunk
  end for
  return H
end function

function BOUNDARY($x, W, k, h, last$)
  $t \leftarrow h + |W| - 1$
  Query P5 with x, k, ordinals $h{:}t$, the queue view, and $last$
  Parse `action_index` as integer r; on call/parse failure **return** h
  if $r = -1$ **then**
    if $last$ **then**
      return t
    else
      return WAIT
    end if
  end if
  return $\max(h, \min(r, t))$ ▷ Zero selects the head; normalize out-of-range ordinals
end function

Coverage and stopping. Removing only prefixes and appending their replacements in order preserves every source element exactly once at each level. A singleton introduces no extra parent. P8 receives the candidate top-level nodes and returns `can_mount_all`; this is a hierarchy-stopping decision. Construction also stops if a round leaves the node count unchanged, produces one node, or reaches the round cap. The remaining nodes are attached to the task root. Boundary normalization in Algorithm 2 ensures progress even when a boundary response cannot be used directly.

F.3 SHORTCUT COMPRESSION AND ISSUE RECONCILIATION

Processing is bottom-up: each parent receives its children's already reconciled representations. Table 14 specifies the two model interfaces. The full JSON field definitions appear in P9 and P10.

Table 14: Inputs and outputs at a parent node v. Source and evidence IDs always refer to supplied trajectory-derived records.

Module	Input	Output
Compression (P9)	Task, parent title/summary, ordered direct children, child verdicts, available evidence, internal action-label hints, and deterministic key-action IDs.	Ordered groups: mode, source IDs, title, failed approach, working path, local outcome, open issue, evidence IDs, and selected key-action IDs.
Reconciliation (P10)	Task, cleaned subtree, prior summary, child verdicts, inherited issue IDs, and retained evidence/actions.	Coherence, completion, final-state summary, open/fatal issues, resolved inherited issue IDs, recovery-evidence map, and key-action IDs.

Algorithm 3 Bottom-up compression, reconciliation, and validated model calls

Require: Task x, source node v, indexed original evidence
Ensure: Reconciled node $v^\star$ with source references, retained actions, and validated verdict

function RECONCILE(x, v)
  if v is a leaf **then**
    return identity copy of v
  end if
  $C \leftarrow [\textsc{Reconcile}(x, c) : c \in \text{children}(v)]$ in source order
  $K \leftarrow \textsc{DeterministicKeys}(C)$; mark these actions for original-content retention
  $C' \leftarrow C$
  if $|C| \geq 2$ **then**
    $q \leftarrow \textsc{CleanerInput}(x, v, C, K)$ ▷ P9
    $G \leftarrow \textsc{CheckedCall}(q, \text{ValidatePartition}(\cdot, C), \perp)$
    if $G \neq \perp$ **then**
      $C' \leftarrow \textsc{CompileGroups}(G, C, K)$ ▷ Keep singletons; compile shortcut groups
    end if
  end if
  $u \leftarrow$ root/subtask copy of v with children C' and their source-action indices
  $q \leftarrow \textsc{CriticInput}(x, u, \text{IssueIDs}(C'))$ ▷ P10
  $z \leftarrow \textsc{CheckedCall}(q, \text{ValidateVerdict}(\cdot, u), \text{UnknownVerdict})$
  Attach z to u; derive u's state by Equation 6
  Use z's summary if nonempty; otherwise retain u's prior summary
  Mark the original actions selected by z's valid key-action IDs
  return u
end function

function CHECKEDCALL($q, V, z_{\text{fallback}}$)
  $q' \leftarrow q$
  for $j = 1, 2$ **do**
    Attempt $r \leftarrow \text{ParseJSON}(\text{Model}(q'))$ and $z \leftarrow V(r)$
    if the call, parsing, and validation succeed **then**
      return z
    end if
    $q' \leftarrow q+$ previous response + validation error
    Append the instruction to return a corrected JSON object
  end for
  return z_{fallback}
end function

Compiling a shortcut. A `keep` group retains its single source node. A `shortcut` group replaces at least two consecutive siblings, recording their node IDs, source-action indices, failed approach, working path, local outcome, and remaining issue. It also carries inherited issue IDs from those sources. Key actions are copied from source records, including their observations. The retained set combines model-selected IDs with deterministic keys: recognized write/edit operations, shell operations that write or change files or run tests, and the last action under each parent. Thus, choosing a shortcut does not ask the model to regenerate executable action text.

Reconciling an issue. An issue is represented by its description, evidence references, and a unique identifier linking it to subsequent recovery evidence. P10 assesses whether a later cleaned node provides concrete recovery evidence for an inherited issue. The response names that issue in `resolved_issue_ids` and supplies its evidence in `resolution_evidence`. The checks below validate those identifiers before accepting the update. The renderer collects accepted resolved IDs and omits matching inherited issue entries. Local shortcut outcome and parent completion remain separate fields: a recovered path can support a still unfinished parent task.

F.4 Validation Rules and State Derivation

For an ordered child list $C = (c_1, \ldots, c_m)$, let $R(C)$ be the supplied node, source, and evidence references reachable from that representation, $A(C)$ its retained action-node IDs, and $I(C)$ its inherited issue IDs. Validation checks each reference against these supplied sets.

Table 15: Deterministic checks on compression and reconciliation outputs.

Checked object	Acceptance rule
Sibling partition	Concatenating all `source_node_ids` lists must equal $(c_1, \ldots, c_m)$ exactly. Groups are nonempty, contiguous, and ordered; `keep` has one source and `shortcut` has at least two.
Shortcut fields	Mode is `keep` or `shortcut`; outcome is `succeeded`, `failed`, `partial`, or `unknown`. Every cited evidence ID belongs to $R(C)$.
Key actions	Selected IDs belong to $A(C)$; cleaner-selected actions also belong to their own source group. Compilation unions these IDs with deterministic keys and copies original records.
Critic fields	Coherence is `coherent`, `inconsistent`, or `insufficient_evidence`; completion is `complete`, `incomplete`, `failed`, or `unknown`.
New issues	Issue lists contain nonempty descriptions; every supplied evidence ID is valid. Each issue has a unique identifier.
Resolved issues	Every resolved ID belongs to $I(C')$. Each has a nonempty recovery-evidence list, all of whose references are valid in the cleaned subtree.

Let c and p denote validated coherence and completion, and let O and F be the reported open- and fatal-issue lists. The final state is computed in the following priority order:

$$\text{state}(c, p, O, F) = \begin{cases} \texttt{broken}, & c = \texttt{inconsistent} \lor p = \texttt{failed} \lor F \neq \emptyset, \\ \texttt{incomplete}, & p = \texttt{incomplete} \lor O \neq \emptyset, \\ \texttt{unknown}, & c = \texttt{insufficient_evidence} \lor p = \texttt{unknown}, \\ \texttt{complete}, & \text{otherwise.} \end{cases} \tag{6}$$

The first satisfied condition applies. The model supplies the evidence-based judgments; the program computes this state from validated fields. Both cleaner and critic are instructed to ground success in observations, verification visible in the trajectory, or completed child verdicts. Numeric action labels serve as diagnostic hints. Under the DENSE information boundary, the root completion scope is `trajectory_local`.

Handling an invalid response. Compression and reconciliation each allow one corrective call with the original input, the previous response, and the validation error. If both attempts fail, compression retains the original child list; reconciliation retains the prior summary and returns an `insufficient_evidence`/`unknown` verdict. Algorithm 3 specifies these handling rules.

Bounded module views. Module inputs use length-limited views of actions, observations, and summaries. Original action–observation records remain available for feedback rendering.

F.5 Selecting Evidence and Producing Feedback

Recipient-visible evidence. From the retained actions, the deterministic selector includes critic-selected actions and available action references supporting unresolved issues or their resolution. It also selects branch representatives for product creation, verification, final responses, and terminal actions. Shortcuts retain available failure and working-path representatives; their reusable failure/recovery lessons are collected at the root. Selection introduces no additional model call.

Required evidence representatives are selected first. Additional retained actions are prioritized by their roles in delivery, verification, and model-selected evidence, subject to an adaptive byte budget. The renderer applies the overall tree budget B_t to the final feedback.

State-guided rendering and budget handling. The initial rendering presents the root summary, scope, and unresolved issues. Subtasks marked `complete` are folded to summaries and selected key actions; `incomplete`, `broken`, and `unknown` subtasks expand recursively. Numeric action labels are omitted. If the result exceeds B_t, the renderer first condenses succeeded shortcuts, then folds non-root subtask detail while retaining the top-level overview, issues, and selected actions. If necessary, it removes entire selected-action blocks in reverse traversal order. A final compact root-and-scope record handles any remaining overflow. Retained key-action bodies are serialized from source text without model rewriting. The complete-feedback budget is then applied, retaining the structured tree while dropping optional surrounding sections as whole blocks. Appendix C shows the resulting interface.

G Full-Result Visualizations

Table 2 reports strict pass rates and their paired changes; Table 3 reports recipient token use and changes from the baseline row. The figures below show the same results across methods and tasks. All methods use a fixed order, with the initial execution as Baseline and Verifier Feedback (VF) distinguished by its additional outcome information.

Figure 10 reports execution-token changes. Figure 11 shows estimated USD costs for GPT-5.5 execution and Gemini feedback, using the prices in Appendix B.1. Figure 12 retains token units and compares Baseline, Advisor, Self-reflection, All-at-once, Step-by-step, and DENSE across all four recipients, with equal task weights. Token totals combine execution and feedback generation per result, following the usage accounting and cost attribution in Appendix A. Scatter points are model–method aggregates with independent per-model axes and no fitted trend: left uses fewer tokens, up is better.

G.1 AGGREGATE TASK PERFORMANCE

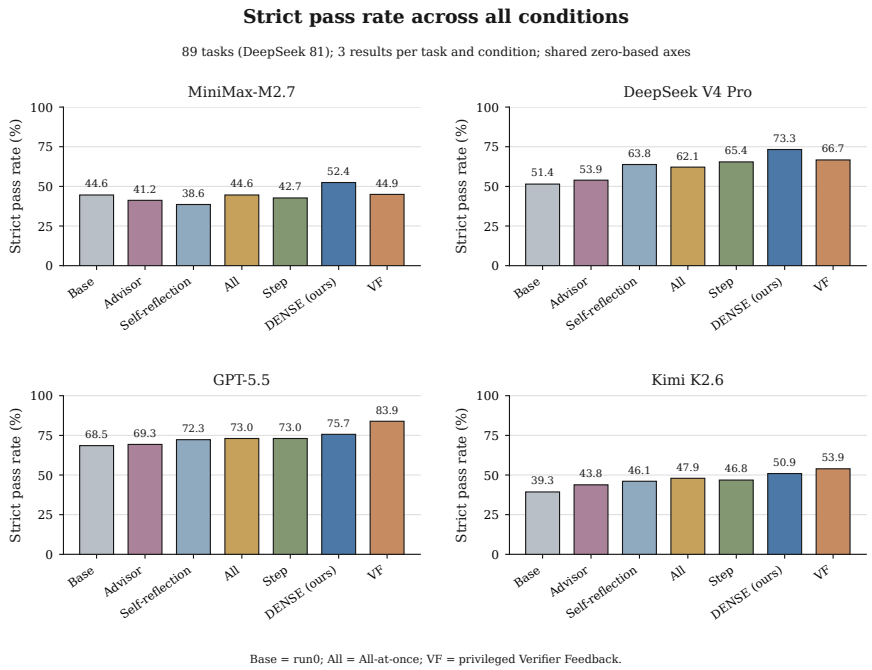


Figure 8: Strict pass rates across all seven conditions, using the outcomes in Table 2. Strict pass requires reward exactly one; rates average three repetitions within each task and then weight tasks equally. MiniMax, GPT, and Kimi use 89 tasks each, and DeepSeek uses 81. Shared zero-based axes show absolute performance on the same percentage scale. DENSE leads the tested non-privileged methods for every recipient. VF uses additional outcome information and exceeds DENSE on GPT and Kimi, but not on MiniMax or DeepSeek, so it serves as a privileged reference rather than a guaranteed upper bound.

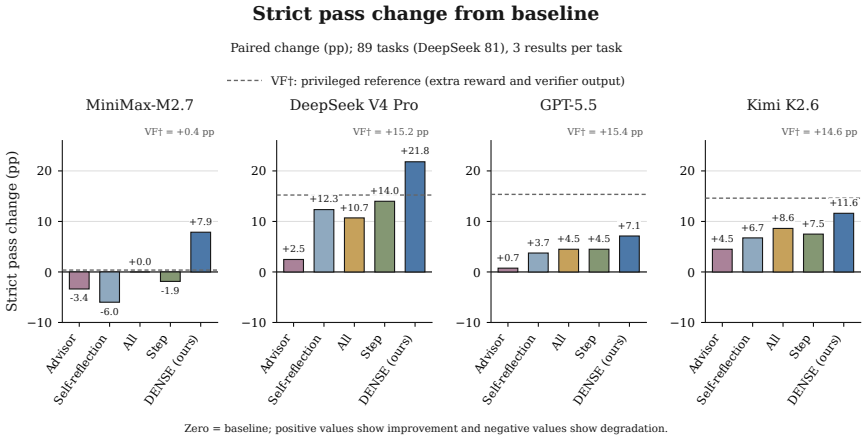


Figure 9: Paired strict pass change from the initial execution, using Table 2. Each value averages the run1-minus-run0 binary success difference over matched repetitions and equally weighted tasks, expressed in percentage points rather than relative percentages. Zero marks unchanged performance; negative bars indicate degradation. Bars show methods without post-hoc verifier information, and gray dashed lines show VF. DENSE gains 7.87, 21.81, 7.12, and 11.61 pp on MiniMax, DeepSeek, GPT, and Kimi, respectively. On MiniMax, it is the only non-privileged feedback method with a positive change.

G.2 Execution Costs, Token Use, and Performance

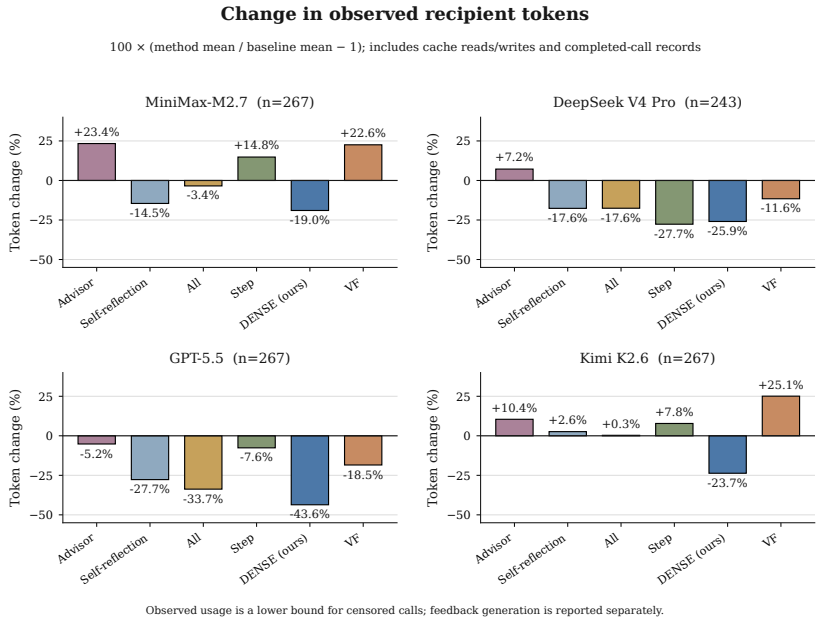


Figure 10: Observed recipient-token change, computed from Table 3, excluding feedback generation and following Appendix A.3. Negative bars indicate lower execution use. DENSE reduces it by 19.0–43.6% across recipients, while Step-by-step achieves the largest reduction on DeepSeek.

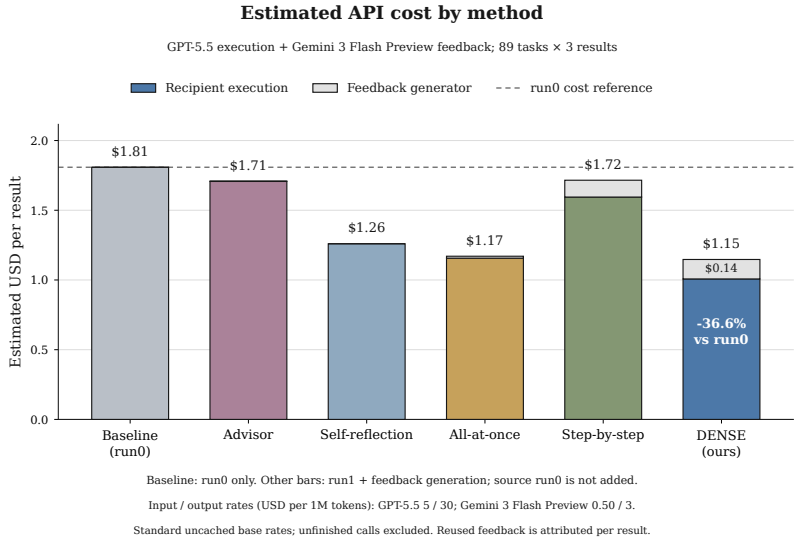


Figure 11: Estimated USD per result for GPT-5.5, using the recorded input/output usage and fixed model-specific prices in Appendix B.1. Costs average three repetitions within each of 89 tasks, then weight tasks equally. Method-colored segments denote recipient execution; gray segments denote Gemini feedback generation. Baseline and the dashed line show run0; other bars combine run1 and feedback. DENSE totals $1.15, including $0.14 for feedback, versus $1.81 for run0 (Table 5). The 36.6% reduction includes feedback generation under this pricing configuration; attribution follows Appendix A.4.

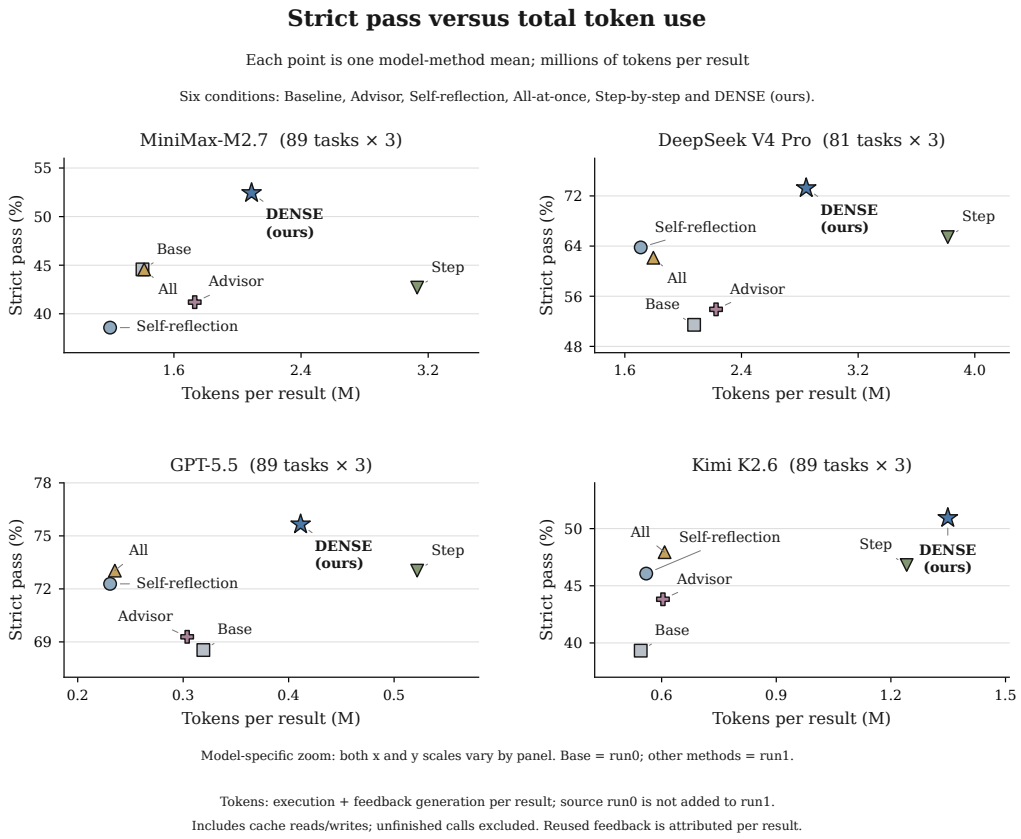


Figure 12: Strict pass rate versus observed execution plus feedback-generation tokens per result. Each point aggregates one model–method condition over three repetitions per task with equal task weights; VF is excluded. Horizontal values sum recipient and attributed generator tokens in millions, while vertical values match Table 2. Accounting follows Appendix A. Upper-left positions combine higher success with fewer tokens, but panel scales differ. DENSE has the highest pass rate in every panel, while its combined token use exceeds Baseline for all four recipients. Including generation thus exposes overhead absent from the execution-only comparison in Figure 4; token totals do not apply the model-specific dollar prices used in Figure 11.

G.3 Task-Level Results

Each task heatmap cell reports strict pass rate over three repetitions: 0, 33.3, 66.7, or 100%. Gray N/A cells are excluded from DeepSeek's denominator.



Figure 13: MiniMax-M2.7: all 89 tasks and seven conditions, three results per cell. Each cell reports strict pass rate (%) over three repetitions.

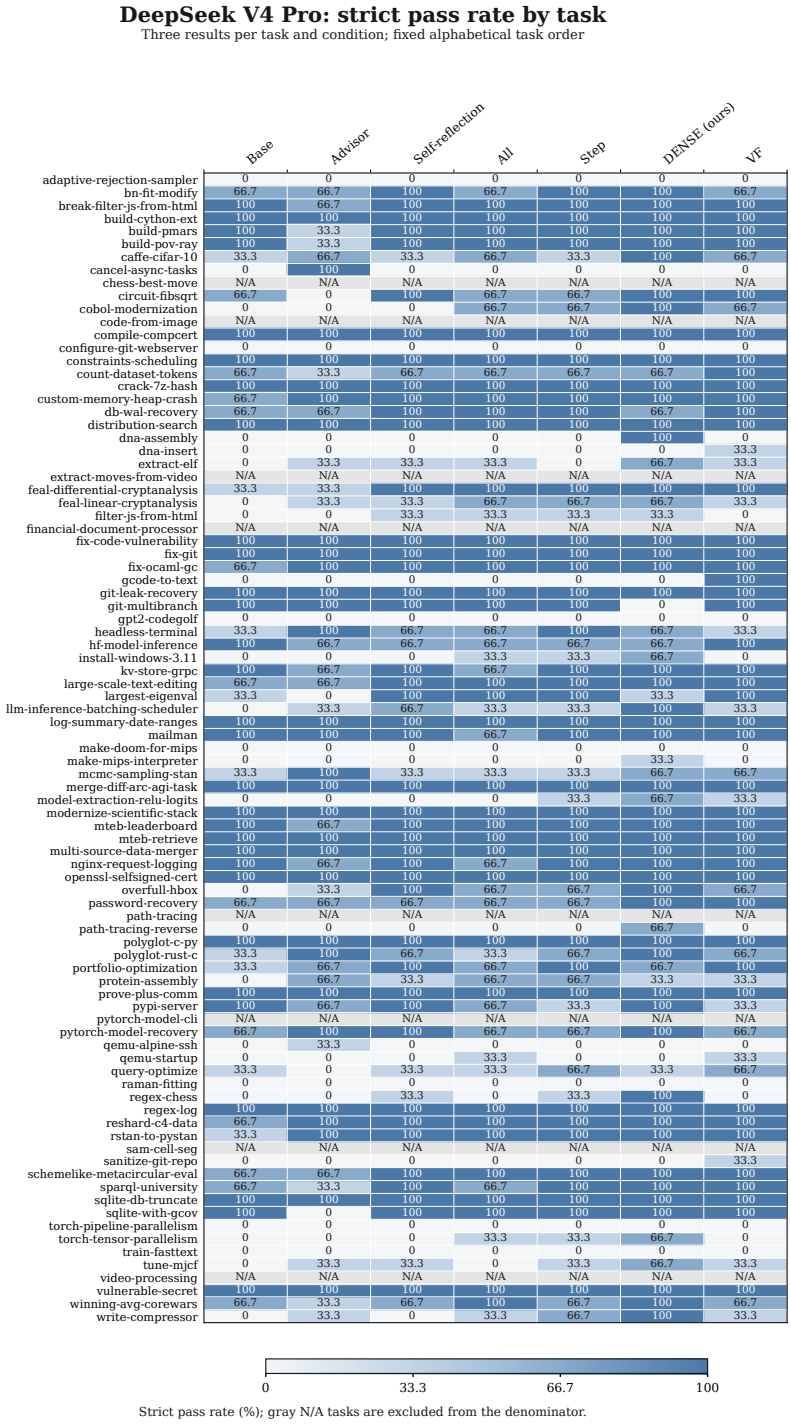


Figure 14: DeepSeek V4 Pro: 81 applicable tasks and eight visual tasks marked N/A. Each cell reports strict pass rate (%) over three repetitions.

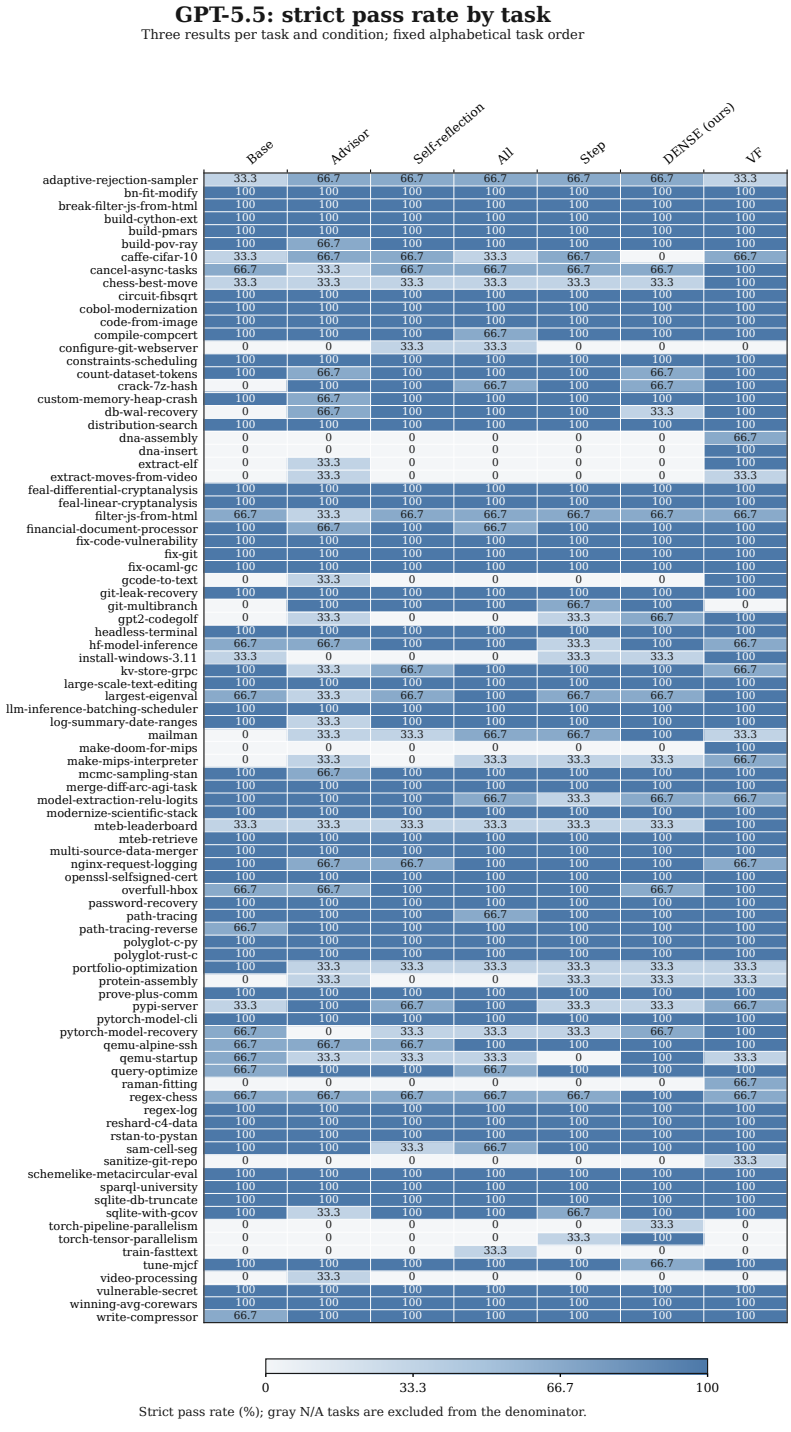


Figure 15: GPT-5.5: all 89 tasks and seven conditions, three results per cell. Each cell reports strict pass rate (%) over three repetitions.

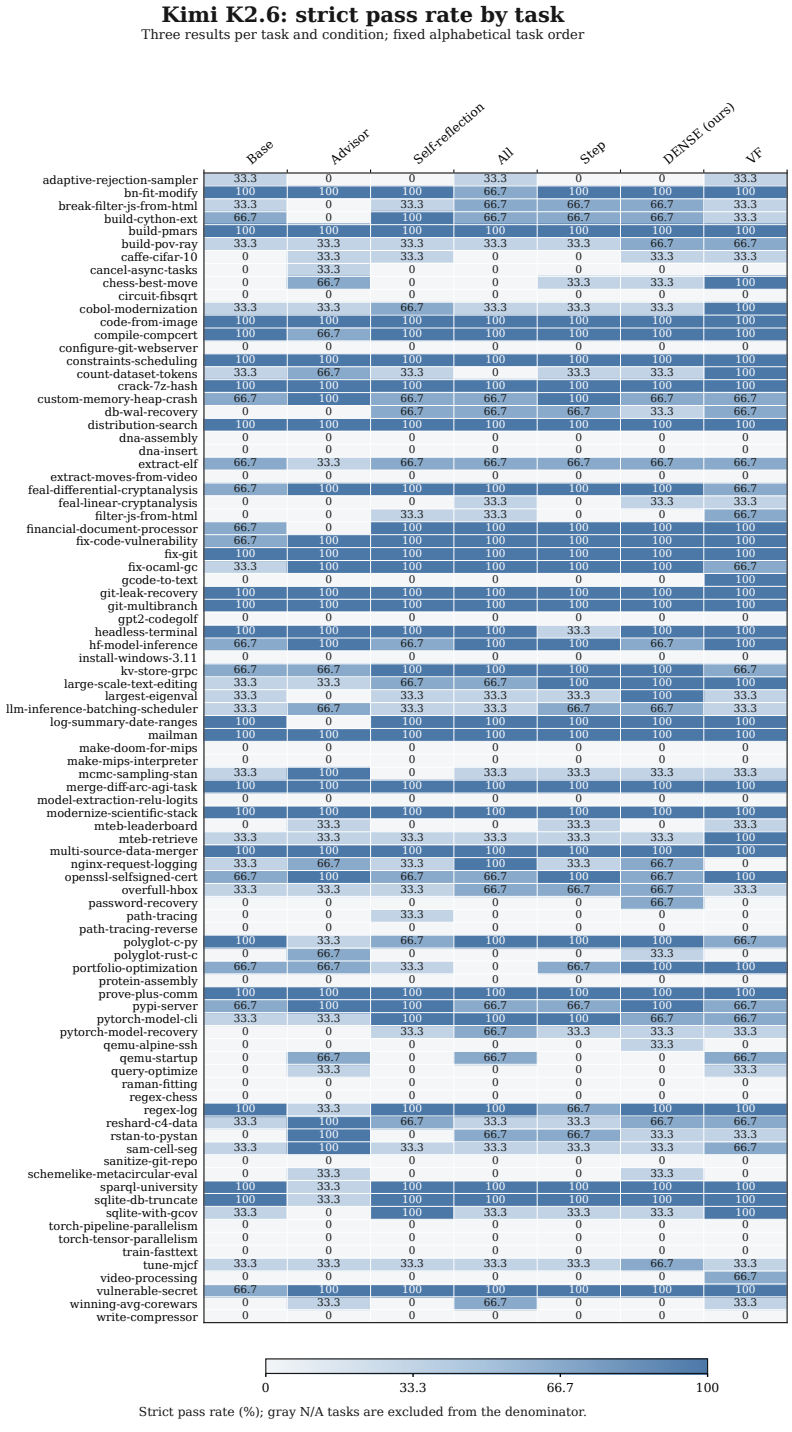


Figure 16: Kimi K2.6: all 89 tasks and seven conditions, three results per cell. Each cell reports strict pass rate (%) over three repetitions.