RECOMPACT: LEARNING TO ACT AND COMPACT FROM GRAPH-GUIDED TRAJECTORY SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon software agents compress interaction histories to work within a finite context window. A mistaken edit is a poor imitation target, yet it remains part of the working tree after a history reset. *ReCompact* separates action weighting from state retention through graph-guided trajectory supervision. A supervised neural graph module assigns contribution labels that weight action targets and prioritize key and supporting information in compact training targets and online compaction. Two training views jointly supervise compact generation and continuation after a history reset. On 500 SWE-bench Verified tasks, ReCompact achieves 54.8% resolution, versus 29.0% for standard supervised fine-tuning, 43.4% for step-aware training, and 39.4% for compact-only training. ReCompact and Compact only achieve similar median input reductions (62.4% and 61.2%, respectively), while ReCompact reduces repeat candidates for previously failed commands from 7.64 to 4.01 per 100 post-compaction tool actions. Removing continuation training or retention-label guidance reduces resolution by 9.6 or 8.4 percentage points; disabling online compaction and reset reduces it by 10.8 points. ReCompact also uses 10.1% fewer total inference tokens per task than Compact only.

1 INTRODUCTION

Software agents address repository issues through cycles of code inspection, editing, and executable feedback (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2024b). As these interactions grow, the agent must allocate a finite context window between past evidence and continued execution. Keeping the full history preserves detail but also retains repeated reads, superseded edits, and verbose tool output. A compact state must preserve enough of that history to support the next decision.

Action imitation and state retention serve different purposes. The schematic in Figure 1 illustrates an edit that adds a guard for `None` when the task input is an empty list. The edit receives low imitation weight, yet remains part of the working tree at the compact boundary. The compact records the attempted guard and its untested status; the later test result belongs to the continuation. A repeated file read adds no new state once its code evidence is retained. The compact therefore records workspace changes even when the actions that produced them receive low imitation weight.

Summarization and learned context management address the growth of agent histories (Wu et al., 2025; Ye et al., 2025; Sun et al., 2025). Training must distinguish *which actions to learn from* from *which execution facts to preserve*. The model must also learn to select its next action from the compact state after reset.

ReCompact trains action selection, compact generation, and post-reset continuation with step-aware supervised fine-tuning (SFT). A supervised neural graph module assigns contribution labels that control action weights and prioritize key/supporting information during training-target construction and online compaction. Two training views teach compact generation from a prefix and continuation from the resulting state. At inference, a validated compact replaces visible history while preserving the task, repository, shell, and tools.

We evaluate a 2×2 design: Standard, Step-aware only, Compact only, and ReCompact. ReCompact improves resolution by 25.8 percentage points over Standard and by 15.4 points over Compact only. At similar input reduction, it produces fewer post-compact repeated-failure candidates and uses

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.

fewer cumulative inference tokens than Compact only. Three targeted ablations remove continuation training, retention-label guidance, and online compaction/reset; each lowers resolution. We also compare 4B implementations of OpenHands, SWE-agent, CaT/SWE-Compressor, and SWE-MeM to assess system-level task resolution.

2 Background and Problem Formulation

Let H_t denote the model-visible message history before a model request and E_t the persistent repository and runtime state. An ordinary assistant action a_t can produce a tool observation and change E_t. A *compact* C is a standalone assistant message describing the state needed for continued work. With the original system message S and task message U, an accepted reset is

$$(H_t, E_t) \xrightarrow{C} ([S, U, C], E_t). \tag{1}$$

Reset leaves E_t unchanged: it discards visible interaction history without undoing edits or restarting tools. Subsequent messages extend $[S, U, C]$. Time and usage counters remain monotonic. The compact is inert text; tool execution is a separate action.

Training assigns imitation weights to actions and constructs compact targets from their observed consequences. For a training boundary b, compact C_b describes facts supported by the corresponding prefix. Offline action weighting uses trajectory hindsight; compact content and online labels use information available at the boundary. Later observations and sealed grading outcomes are excluded from C_b. Continuation training then supervises actions conditioned on this retained state.

3 Method

3.1 Graph-guided action weighting and state retention

The supervised neural graph module produces contribution labels for action weighting and compact construction. Its trajectory representation includes observations, hypotheses, edits, tests, and recoveries as typed nodes. Evidence-use, test-validation, and failure–recovery relations describe dependencies across the trajectory. Annotations link these dependencies to their source spans.

For each original assistant action a_i, the neural module predicts a contribution label z_i: key, supporting, or redundant. The label maps to a positive coefficient:

$$w(a_i) = \alpha_{z_i}, \qquad \alpha_{\text{key}} > \alpha_{\text{supporting}} > \alpha_{\text{redundant}} > 0. \tag{2}$$

The coefficients for key, supporting, and redundant actions are 1.5, 1.0, and 0.25, respectively. All original actions retain positive supervision weight. Contribution labels reflect an action's role in the trajectory, including diagnostic evidence from failed tests. Action-contribution labels apply to assistant actions; system, task, and tool messages are masked.

For compact content, let $P_b \subseteq \{1, \ldots, b\}$ denote the prefix steps whose state or evidence is retained in C_b. Compact construction uses contribution labels directly, prioritizing key and supporting information both for training targets and online. Consequences of redundant actions, such as attempted edits or failed tests, remain eligible for retention when they describe the live environment.

3.2 Compact generation and reset-conditioned continuation

The model learns both to generate compact state and to act from it after reset. Let $A_{1:T}$ denote the sequence of original assistant actions, with intervening tool observations included as context. For a boundary after A_b and its observations, compact training constructs two views:

$$\begin{aligned} \text{generation:} \quad & [S, U, A_{1:b}, C_b], && \text{targets: } A_{1:b}, C_b, \\ \text{continuation:} \quad & [S, U, C_b, A_{b+1:T}], && \text{targets: } A_{b+1:T}. \end{aligned} \tag{3}$$

The first view supervises the original prefix actions and compact generation. The second supervises suffix actions in their original order, conditioned on S, U, C_b instead of the discarded prefix. Each original action is a target in exactly one training view. Compact generation receives coefficient

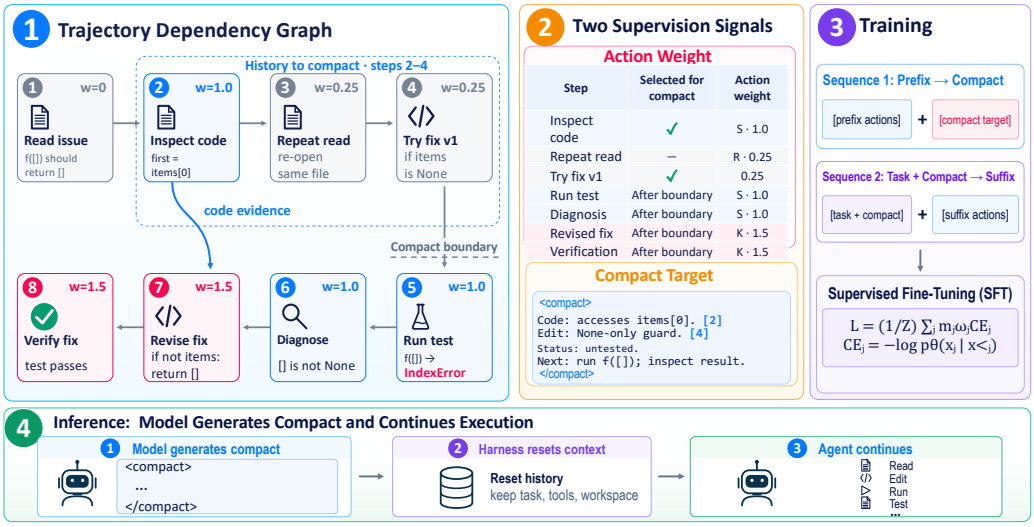

Figure 1: **ReCompact schematic.** An ineffective edit receives low imitation weight but remains in the compact because it changes the workspace. Only prefix-observed facts enter the compact; the later test and repair belong to the continuation. Two training views connect compact generation to execution after reset. K/S/R denote key/supporting/ redundant actions; the task input has no action target ($w = 0$). Repository and runtime state persist across reset. Enlarged panels appear in Appendix F.

$\lambda_C > 0$; the same compact is masked when used as continuation context. System, task, and tool messages are also masked.

Compact targets describe current edits, observed results, unresolved questions, next actions, and relevant failed attempts using prefix-supported information.

3.3 Step-aware training objective

Action weights determine each action's contribution to the training loss; the two views specify its conditioning context. For token x_j at position j in a serialized training view, let $m_j \in \{0, 1\}$ indicate a supervised assistant target. Its coefficient ω_j is $w(a_i)$ for a token in original action a_i, and λ_C for a compact-generation target. With model parameters θ and preceding tokens $x_{<j}$, the objective is

$$\mathcal{L}(\theta) = -\frac{1}{Z} \sum_j m_j \omega_j \log p_\theta(x_j \mid x_{<j}), \qquad Z = \sum_j m_j \omega_j, \tag{4}$$

where Z is the sum of weights over effective supervised tokens. The loss is a weighted mean of token-level negative log-likelihoods. Tokens with $m_j = 0$ contribute to neither the numerator nor the denominator. The prefix $x_{<j}$ follows the corresponding view in Eq. (3).

3.4 Model-authored compact and atomic reset

At each online compact boundary, the graph module labels the observed prefix. These labels prioritize key/supporting evidence in a complete, model-authored `<compact>` state `</compact>` assistant turn. After validating the compact, the harness applies Eq. (1): it replaces the interaction history with the original system and task messages and the exact compact. The same model resumes tool actions from this state and subsequent observations. Appendix B.1 specifies the validation rules.

Rejected compacts leave history unchanged. Accepted resets preserve E_t. The harness permits at most three compacts per task (Appendix B.1).

Repeated resets apply the same transition to the history accumulated since the preceding reset.

Table 1: Component controls and training counts. Compact combines generation/continuation training with online compaction.

Condition	Action weights	Compact	Train rows	Trajectories
Standard	uniform	off	4,617	4,617
Step-aware only	step-aware	off	4,617	4,617
Compact only	uniform	on	9,234	4,617
ReCompact	step-aware	on	9,234	4,617

4 Experimental Setup

4.1 Data and component controls

The 2×2 design tests graph-derived action weighting and compaction (Table 1). The four main conditions use Qwen3.5-4B-Base, 4,617 training trajectories, 392 development trajectories, and seed 42. Compact conditions use Eq. (3); non-compact conditions use the original trajectories. Standard and Compact only assign unit weight to original-action targets, whereas Step-aware only and ReCompact use Eq. (2). Compact-generation targets receive $\lambda_C = 1.5$.

Compaction combines target generation, continuation training, label-guided retention, and online reset. Compact only uses these components with uniform original-action weights. All four conditions share the source trajectories and training schedule.

Targeted ablations. Three additional conditions start from the ReCompact design. *No continuation* removes compact-conditioned continuation training while retaining compact generation, graph-derived action weights, and online reset. *No retention labels* removes contribution-label guidance from both training compact targets and online compaction, retaining action weighting, continuation training, and reset under a matched compact-length budget. *No compaction/reset* evaluates the complete ReCompact checkpoint with online compaction and reset disabled. All three ablations use the full model's 500-task evaluation set and budgets.

Training implementation. All four main conditions use AdamW with a learning rate of 2×10^{-5}, betas $(0.9, 0.95)$, weight decay 0.1, and a cosine schedule. Training makes one pass over the trajectories, with 1,155 optimizer steps per condition. Four composable-FSDP ranks process one trajectory per rank per optimizer step, using bfloat16 mixed precision and float32 reductions. Serialized training windows have a 32,768-token capacity and 4,096-token overlap. Supervision covers the assistant body through <|im_end|>; the target-ownership rule in Section 3.2 assigns each original action to one view. Appendix A summarizes these settings and the weighted-token totals.

4.2 Task evaluation

All four conditions are evaluated on the same 500 SWE-bench Verified tasks with identical evaluation budgets, a 32,768-token context window, and an OpenHands XML interface. The evaluated groups are v43 Standard and Step-aware only, and mixed-v4 Compact only and ReCompact. Compact is enabled only for the latter two conditions, with at most three compacts per task.

Let $\ell(\cdot)$ count serialized tokens, y_t denote the output of request t, and M_t its output allowance. Let B be the context capacity and $T_{\max}$ the total per-task inference budget. The evaluation budget satisfies

$$\ell(H_t) + M_t \le B, \qquad \sum_t \big(\ell(H_t) + \ell(y_t)\big) \le T_{\max}, \tag{5}$$

with $B = 32{,}768$ and $T_{\max} = 3{,}000{,}000$ tokens per task. Input is counted on every request, including repeated context. Requests reserve at least 4,096 tokens for output; the output allowance is the smaller of 16,384 tokens and the remaining context capacity. Execution uses temperature 0, top-p 1, and sampling seed 42, with limits of 100 iterations, 28,800 seconds per task, and 300 seconds per request. Two endpoint workers serve each checkpoint. These settings hold across the four conditions; Appendix B.1 specifies the interface and history-transition rules.

The process analysis uses the same evaluation runs for Compact only and ReCompact: 500 observable tasks per method, with 246 and 281 compact boundaries. Each task contributes its observed boundaries to its method's process summary.

4.3 Outcome and process metrics

Resolution is solved tasks divided by 500. We report component contrasts in percentage points and their factorial decomposition. Each main condition uses one training seed (42); contrasts are computed from the aggregate resolved counts.

For each boundary, we summarize input length before compact, compact text length, and input length of the first post-reset request. Writing the pre-compact and first post-reset lengths as L_{before} and L_{after}, input reduction is $1 - L_{\text{after}}/L_{\text{before}}$. We report the 25th percentile, median, and 75th percentile across boundaries.

Post-compact behavior is measured in windows of at most ten tool actions, truncated at the next compact or the end of observed execution. A repeat candidate re-executes an identical pre-compact failed command without an intervening change to the relevant code, input, dependencies, or environment. Tests rerun to validate a repair are excluded. We report command and test candidates per 100 tool actions and the fraction of boundaries with any candidate. Definitions appear in Appendix D.

Execution costs are mean cumulative input tokens, output tokens, tool calls, and elapsed time over all 500 tasks per method. Amortized token cost is total evaluation input-plus-output tokens divided by the number of resolved tasks; its numerator includes unsuccessful executions.

5 Results and Analysis

We compare task resolution across 4B software agents and the controlled component study, then examine how continuation training, retention guidance, and online reset affect resolution, repeat candidates, and execution cost.

5.1 Task resolution

Table 2: System-level comparison of 4B software agents on 500 SWE-bench Verified tasks each. Advantage is ReCompact minus each method in total resolved tasks and percentage points (pp).

			ReCompact advantage	
Method	Resolved	Rate (%)	Tasks	Rate (pp)
OpenHands (CodeAct)	160	32.0	+114	+22.8
SWE-agent	175	35.0	+99	+19.8
CaT / SWE-Compressor	235	47.0	+39	+7.8
SWE-MeM-SFT	255	51.0	+19	+3.8
SWE-MeM-RL	270	54.0	+4	+0.8
ReCompact	**274**	**54.8**	—	—

Table 2 compares ReCompact with OpenHands/CodeAct and SWE-agent (Wang et al., 2024b;a; Yang et al., 2024), and with learned context-management systems CaT/SWE-Compressor and SWE-MeM (Liu et al., 2025; Gao et al., 2026). ReCompact resolves 274 tasks (54.8%), exceeding CaT/SWE-Compressor, SWE-MeM-SFT, and SWE-MeM-RL by 7.8, 3.8, and 0.8 percentage points, respectively.

Within the controlled component study (Table 3), ReCompact resolves 129 more tasks than Standard, a 25.8-point gain. It also exceeds Step-aware only by 11.4 points and Compact only by 15.4 points.

Table 3: Task resolution and component ablation on SWE-bench Verified. All conditions use the same 500 tasks and evaluation budgets. Compact denotes generation/continuation training together with online compaction. Gains are percentage-point differences from Standard.

	Components		SWE-bench Verified		
Method	Step weights	Compact	Resolved	Rate (%)	Gain (pp)
Standard	—	—	145 / 500	29.0	—
Step-aware only	✓	—	217 / 500	43.4	+14.4
Compact only	—	✓	197 / 500	39.4	+10.4
ReCompact	✓	✓	**274 / 500**	**54.8**	**+25.8**

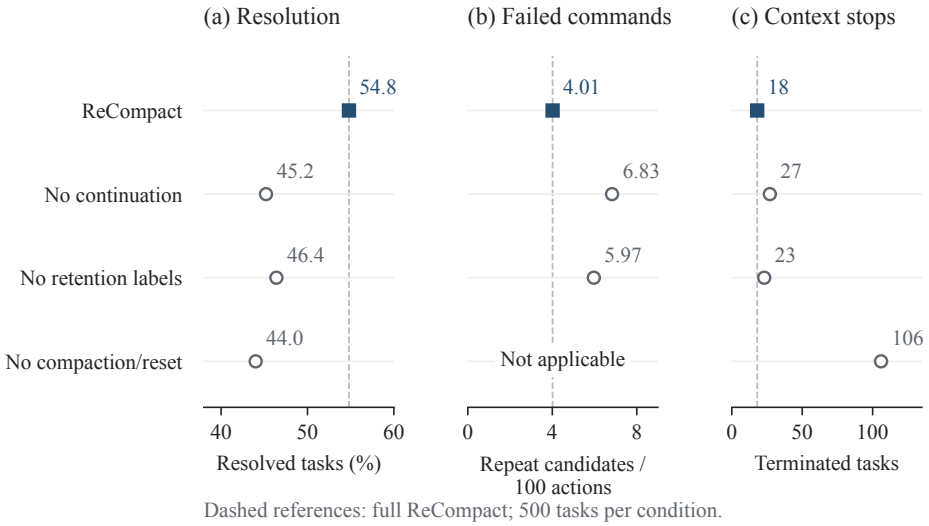

Figure 2: **Resolution, repetition, and context exhaustion under component ablations.** Aligned rows show resolution, post-compact command-repeat candidates per 100 actions, and context-exhaustion terminations for the same 500-task evaluation. Filled blue squares mark ReCompact; open circles mark conditions with one component removed. Dashed lines give the full-model references. Post-compact repetition is inapplicable when online compaction is disabled. Table 9 gives counts and denominators.

5.2 Component ablation

Step-aware weighting. Step-aware weighting improves resolution both without compact (+14.4 pp) and with compact (+15.4 pp; Table 3). With compact enabled, both conditions retain label-guided compact construction, isolating the change in original-action weights.

Gains beyond either component. Compact improves resolution under uniform (+10.4 pp) and step-aware weighting (+11.4 pp). The joint gain over Standard is 25.8 points, compared with 24.8 points for the sum of the standalone gains, giving a +1.0-point interaction. Both components improve resolution when added to the other. Table 10 gives the complete decomposition.

5.3 Targeted training and execution ablations

Figure 2 compares the full model with each ablation on resolution, failed-command repeat candidates, and context-exhaustion terminations.

Learning to continue from compact state. Removing compact-conditioned continuation training lowers resolution from 54.8% to 45.2% (274 to 226 tasks). The median compact length remains 2.2K tokens, while command-repeat candidates increase from 4.01 to 6.83 per 100 actions and test-repeat candidates from 1.15 to 2.08. The fraction of follow-up boundaries with any repeat candidate rises from 16.4% to 28.4%.

Guiding what the compact retains. Removing contribution-label guidance from compact construction lowers resolution to 46.4% (232 tasks). The compact-length quartiles remain 1.7/2.2/2.7K tokens, matching the full model, while command and test candidate rates rise to 5.97 and 1.82 per 100 actions. Affected boundaries increase to 25.3%. This ablation changes retention guidance while keeping graph-derived action weights enabled.

Replacing history during execution. Disabling online compaction and reset in the full checkpoint lowers resolution to 44.0% (220 tasks), with context-exhaustion terminations rising from 18 to 106. The full system resolves 54 more tasks and has 88 fewer context terminations with the same checkpoint.

The continuation and retention-label ablations produce 288 and 285 compact boundaries, respectively, compared with 281 for ReCompact.

The label-related comparisons separate two uses of graph supervision. ReCompact versus Compact only changes action-loss weighting while retaining label-guided compact construction. The no-retention-label ablation changes compact construction while preserving action weights. Both comparisons favor the full system: graph supervision contributes through action training as well as state retention.

5.4 Compression and post-reset continuation

Figure 3 relates context lengths and repeat candidates to task outcomes and execution costs.

Context lengths at reset. Median input reduction is 61.2% for Compact only and 62.4% for ReCompact, with overlapping interquartile ranges. ReCompact's median compact is longer (2.2K versus 2.0K tokens), while its first post-reset input is shorter (8.4K versus 8.6K). Post-reset input comprises the compact and the original system/task messages. Appendix C gives the full boundary distributions.

Fewer repeat candidates after reset. ReCompact has 4.01 repeated-failure command candidates per 100 actions, compared with 7.64 for Compact only, a 47.4% relative reduction. Failed-test candidate rates fall from 2.42 to 1.15, a 52.4% reduction. At the boundary level, 16.4% of ReCompact follow-up windows contain a repeat candidate, compared with 30.6% for Compact only. Appendix E provides the complete counts and denominators.

ReCompact contributes more observed follow-up actions, yet has fewer repeat candidates per action and fewer affected boundaries. Both normalizations show lower repetition within post-reset windows, beyond differences in the number of observed actions.

Appendix E.1 shows successive resets in two execution segments, with request-level curves and exact before/after transitions.

5.5 Execution cost and resolved-task efficiency

ReCompact averages 742K input and 18.6K output tokens per task, or 760.6K in total, compared with 846.4K for Compact only. Its resolution is 15.4 percentage points higher and its total token use is 10.1% lower. Mean tool calls fall from 56.8 to 50.7 and elapsed time from 1,530 to 1,380 seconds, reductions of 10.7% and 9.8%, respectively. Figure 3(c–d) plots token costs against resolution; Table 8 includes tool calls and elapsed time.

Across the four main methods, ReCompact achieves the highest resolution and lowest amortized token cost per resolved task: approximately 1.39M versus 2.40M, 1.67M, and 2.15M for Standard, Step-aware only, and Compact only. Mean tokens per task are 696.8K for Standard and 724.1K for

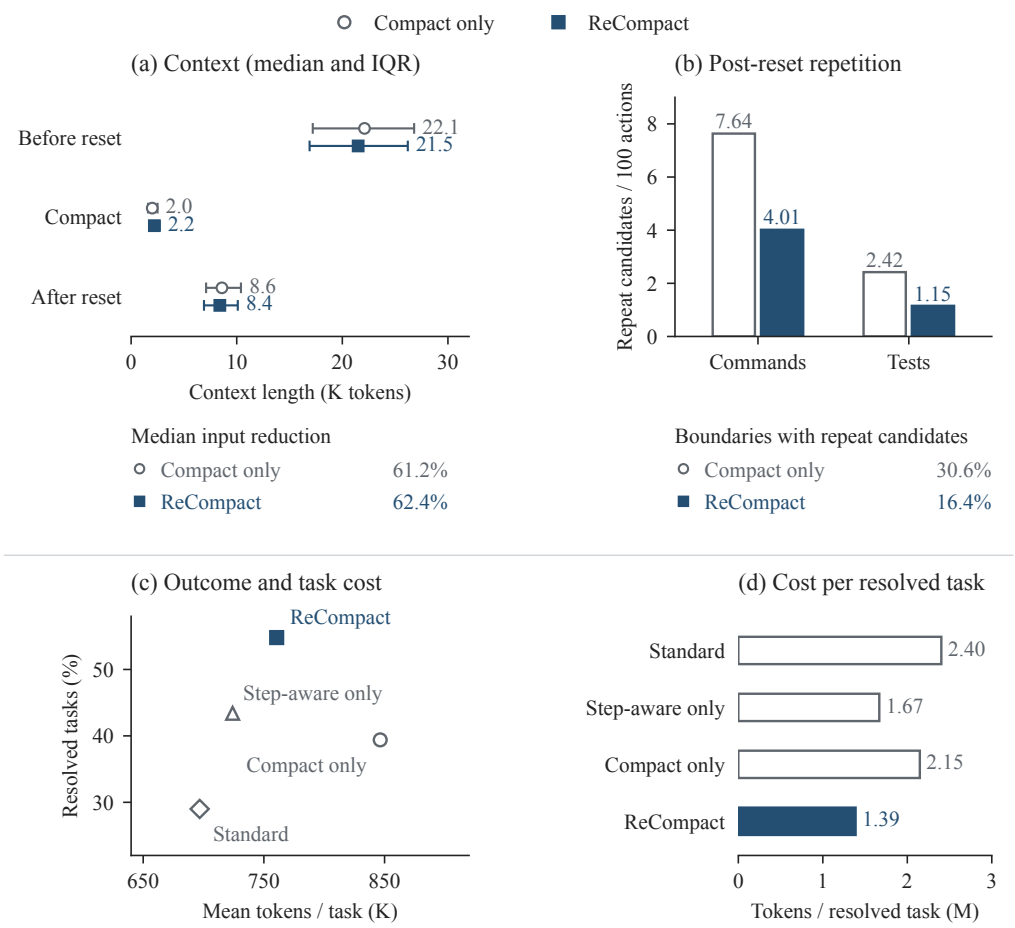

Figure 3: **Context length, post-reset repetition, and inference cost.** (a) Context-length medians and P25–P75 intervals over 246 Compact-only and 281 ReCompact boundaries. (b) Repeat-candidate rates over 2,148 and 2,516 follow-up actions. Affected-boundary percentages use 232 and 269 windows. (c) Resolution versus mean input-plus-output tokens per task. (d) Total evaluation tokens divided by resolved tasks, including tokens from all 500 executions. K and M denote 10^3 and 10^6 tokens. Tables 7, 11, and 8 give the full aggregates.

Step-aware only, compared with 760.6K for ReCompact. Relative to Step-aware only, ReCompact uses 5.0% more tokens per task, increases resolution by 11.4 percentage points, and reduces amortized token cost by 16.8%. Disabling compaction/reset reduces mean tokens per task to 710.9K and raises amortized cost to 1.62M.

Both training ablations increase cumulative token use (Table 9). Without continuation training, mean input and output rise to 801K and 19.8K tokens; without retention labels, they rise to 784K and 19.4K. Both conditions resolve fewer tasks than the full model while consuming more tokens. Disabling compaction/reset instead reduces both token use and resolution, with more runs ending in context exhaustion.

6 Related Work

Memory and learned context management. Agent memory and context management include persistent-memory approaches such as MemGPT and Reflexion (Packer et al., 2023; Shinn et al.,

2023), summarization and folding (Wu et al., 2025; Ye et al., 2025; Sun et al., 2025), and context management for coding agents (Liu et al., 2025; Gao et al., 2026). The system comparison in Table 2 includes CaT/SWE-Compressor and both the SFT and RL variants of SWE-MeM. Training under context eviction addresses the change in conditioning history after information is removed (Zinco J et al., 2026). ReCompact trains compact generation from the trajectory prefix and action selection from the retained state. Graph labels weight action targets and guide compact content.

Software-agent execution. SWE-bench, SWE-agent, and OpenHands provide benchmarks and interfaces for repository-level software tasks (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2024b). CodeAct and Agentless explore additional action and repair workflows (Wang et al., 2024a; Xia et al., 2024a). AutoCodeRover retrieves code through structure-aware search and incorporates test-based fault localization when tests are available (Zhang et al., 2024). These systems organize how agents inspect and modify a repository. Table 2 includes 4B OpenHands/CodeAct and SWE-agent implementations as execution-system baselines. ReCompact uses execution histories to supervise actions and construct the state retained across resets.

Learning from software execution. SWE-Gym, SWE-smith, R2E-Gym, and AgentTrek provide resources for agent training (Pan et al., 2025; Yang et al., 2025; Jain et al., 2025; Xu et al., 2024). SWE-rebench automates repository-task collection, environment installation, and execution-based validation (Badertdinov et al., 2025). SWE-RL learns from software evolution using patch-similarity rewards (Wei et al., 2025); Golubev et al. (2025) train multi-turn SWE policies with test-based rewards and trajectory-length penalties. ReCompact uses a fixed trajectory corpus to supervise action importance, compact content, and continuation after history replacement.

Dependencies and training-data utility. Dependency-structured memory and graph-based credit assignment motivate representing relations between trajectory steps (Wu et al., 2026; Cheng et al., 2026). LESS and DoReMi study data utility and allocation (Xia et al., 2024b; Xie et al., 2023). ReCompact uses contribution labels to weight action imitation and select execution facts for retention across resets.

Recursive self-improvement and agent design. STOP recursively applies a code-improvement scaffold to its own implementation (Zelikman et al., 2024). ADAS searches agent programs with a meta-agent and an archive of evaluated designs (Hu et al., 2025). SICA selects the best archived agent to modify its own code (Robeyns et al., 2025), while the Darwin Gödel Machine maintains a population of self-modifying coding agents (Zhang et al., 2026). The Huxley–Gödel Machine allocates self-modification and evaluation through tree search (Wang et al., 2026). These methods revise agent implementations across iterations. ReCompact trains the policy to preserve and use execution state across context resets within a task.

7 Discussion

At the same median compact length of 2.2K tokens, the full model resolves more tasks and produces fewer post-reset repeat candidates than either training ablation. These comparisons link compact-content guidance and continuation training to the actions taken after reset.

The cost comparison distinguishes context compression from execution efficiency. Retained-state length determines part of the next request's input, whereas cumulative token use also reflects the number of requests and the history processed at each one. ReCompact combines similar post-reset input lengths with fewer tool calls and lower cumulative token use than Compact only. Figure 3 connects these local reset measurements to the cost of completing the evaluation.

8 Conclusion

ReCompact combines graph-derived action weighting, label-guided compaction, and two-view continuation training. It resolves 54.8% of SWE-bench Verified tasks, the highest rate among the compared 4B systems, and achieves the lowest amortized token cost in the controlled study. Removing retention guidance or continuation training lowers resolution while leaving median compact length unchanged. Disabling online compaction and reset increases context-exhaustion terminations.

REPRODUCIBILITY STATEMENT

The appendices specify training settings, reset semantics, process metrics, and aggregate experimental tables. The accompanying Python script reproduces the result figures from these summaries and request sequences. Appendix G describes the available artifacts.

ETHICS STATEMENT

This work concerns repository code, agent traces, and generated patches. Relevant risks include license restrictions, sensitive information in logs, insecure changes, and benchmark contamination. The design excludes sealed grader information from model prompts and compact construction. Patch deployment calls for security and correctness review beyond benchmark tests. Trace and patch release requires license and privacy review.

AI USE STATEMENT

During the preparation of this article, the authors used ChatGPT for language editing. AI assistance also included manuscript drafting, literature organization, figure-code preparation, suggestions for experimental and ablation design, and interpretation of aggregate results. Experimental values were supplied by the authors. The authors reviewed and revised the resulting text and take full responsibility for the content of the publication.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Xin Cheng, Shuo He, Lang Feng, HaiYang Xu, Ming Yan, Lei Feng, and Bo An. Beyond trajectory-level attribution: Graph-based credit assignment for agentic reinforcement learning. *arXiv preprint arXiv:2605.26684*, 2026.

Shuzheng Gao, Wenhao Zeng, Zhaojian Yu, Jianqiao Wangni, Chaozheng Wang, Kai Cai, Shilin He, and Michael R. Lyu. SWE-MeM: Learning adaptive memory management for long-horizon coding agents. *arXiv preprint arXiv:2606.28434*, 2026.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, Sergei Skvortsov, and Boris Yangel. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

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

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon SWE-agents. *arXiv preprint arXiv:2512.22087*, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning*, 2025.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. In *ICLR Workshop on Scaling Self-Improving Foundation Models without Human Supervision*, 2025.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon LLM agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Wenyi Wang, Piotr Piękos, Nanbo Li, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley–Gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations*, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024b.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. ReSum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Yating Wu, Yuhao Zhang, Sayan Ghosh, Sourya Basu, Anoop Deoras, Jun Huan, and Gaurav Gupta. ContextWeaver: Selective and dependency-structured memory construction for LLM agents. *arXiv preprint arXiv:2604.23069*, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024a.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. LESS: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024b.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V. Le, Tengyu Ma, and Adams Wei Yu. DoReMi: Optimizing data mixtures speeds up language model pretraining. *arXiv preprint arXiv:2305.10429*, 2023.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. AgentTrek: Agent trajectory synthesis via guiding replay with web tutorials. *arXiv preprint arXiv:2412.09605*, 2024.

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

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. AgentFold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024. doi: 10.1145/3650212.3680384.

Zinco J, Xunjie Zhu, Shen Huang, Zhenyi Wang, Pengjun Xie, and Jieping Ye. MemoryWalker: Stop training agents on contexts they never saw. *arXiv preprint arXiv:2609.00865*, 2026.

A Training Configuration and Target Accounting

Tokenization and loss implementation. Training uses the native Qwen chat template and supervises the assistant body through <|im_end|>. Compact targets are standalone assistant messages rendered with the permitted empty-think prefix. Target-preserving windows overlap by 4,096 tokens. The ownership rule supervises each original target in exactly one view, including when context windows overlap. The token masks and coefficients follow Eq. (4). The tables below list the training settings and weighted-token totals.

Table 4: Training configuration for the four main conditions. Trajectory counts refer to unique source trajectories; Compact only and ReCompact export two views per trajectory.

Setting	Value
Initialization / seed	Qwen3.5-4B-Base / 42
Training / development trajectories	4,617 / 392
Sequence capacity / overlap	32,768 / 4,096 tokens
Optimizer / learning rate	AdamW / 2×10^{-5}
Adam betas / weight decay	$(0.9, 0.95)$ / 0.1
Schedule / trajectory passes	Cosine / one
Parallelism	Four composable-FSDP ranks
Precision / reductions	bfloat16 mixed / float32
Trajectory batch	One trajectory per rank per optimizer step
Optimizer steps per arm	1,155
Uniform original-action coefficient	1.0
$(\alpha_{\text{key}}, \alpha_{\text{supporting}}, \alpha_{\text{redundant}})$	$(1.5, 1.0, 0.25)$
Compact target λ_C / continuation-context weight	1.5 / 0
Target span	Assistant body through <|im_end|>

Table 5: Sum of supervised token weights across the training examples.

Condition	Effective token weight
Standard	58,648,518.00
Step-aware only	34,324,584.25
Compact only	64,871,533.50
ReCompact	40,547,599.75

B Training-Data Audit

Table 6: Compact audit across 5,009 unique training and development trajectories.

Audit item	Count or content tokens
Standalone compact with opening and exactly one closing tag	5,009 / 5,009
Strict system + task + compact + suffix continuation layout	5,006 / 5,009
Continuation layouts containing a user reminder	3 / 5,009 (0.060%)
Compact length median / p95 / p99	732 / 1,828 / 3,161
Maximum compact content length	14,392
Compact content exceeding 4,096 tokens	23 / 5,009 (0.459%)
Compact containing inert quoted tool syntax	1

The raw compact began with <compact>; after chat-template rendering its target prefix was <think>\n\n</think>\n\n<compact>.

B.1 Execution settings and transition implementation

Evaluation uses an OpenHands 0.53.0 fork, temperature 0, top-p 1, sampling seed 42, XML actions, and matched prompts. It disables browsing, the LLM editor, native tool calling, and the external

condenser. For serialized input length L, it defines

$$A = 32768 - L, \qquad M = \min(16384, A), \qquad \text{send only if } A \geq 4096. \tag{6}$$

The 4,096-token threshold reserves output space; the allowance M varies with the remaining context capacity. Execution limits are 100 iterations, 3,000,000 total inference tokens, 28,800 seconds per task, 300 seconds per request, and two endpoint workers per checkpoint. These settings are shared across the four conditions. Compact-enabled conditions permit at most three compacts per task.

The transition contract accepts a complete assistant turn with the permitted empty-think prefix, a nonempty `<compact>` state, and explicit `</compact>`. Structured tool calls or executable suffixes in that turn are rejected; quoted tool syntax inside compact is inert. Invalid output leaves history unchanged. An accepted transition preserves system and task messages, inserts the exact compact, and excludes the removed prefix. Persistent state follows Eq. (1). Non-compact conditions retain the accumulated history.

C Context Lengths and Resource Interpretation

Table 7: Compact-boundary distributions. Each distribution cell is P25 / median / P75. K denotes 1,000 tokens. Reductions are summarized from individual boundary ratios.

Metric	Compact only	ReCompact
Observable tasks	500	500
Tasks with compact / 500 (%)	193 (38.6)	212 (42.4)
Compact boundaries	246	281
Boundaries without subsequent actions	14	12
Pre-compact input (K tokens)	17.2 / 22.1 / 26.8	16.9 / 21.5 / 26.2
Compact text (K tokens)	1.6 / 2.0 / 2.5	1.7 / 2.2 / 2.7
First post-reset input (K tokens)	7.1 / 8.6 / 10.4	6.9 / 8.4 / 10.1
Input reduction (%)	48.7 / 61.2 / 70.5	50.1 / 62.4 / 71.8

Table 7 and Figure 3 characterize request lengths at compact boundaries. Total inference accounting sums input and output over all requests, including compact generation, continuation, failed tasks, and counted retries.

For n resolved tasks and aggregate token totals $T_{\text{in}}, T_{\text{out}}$, resolved tasks per million tokens are

$$\eta_{\text{total}} = \frac{10^6 n}{T_{\text{in}} + T_{\text{out}}}, \qquad \eta_{\text{input}} = \frac{10^6 n}{T_{\text{in}}}. \tag{7}$$

The inverse of η_{total} is the amortized token cost in millions of tokens per resolved task, as plotted in Figure 3. Input and output tokens each contribute one token to this total.

Table 8: Cumulative execution costs across the four main methods. Each mean uses all 500 tasks, with 500 valid observations for each of input tokens, output tokens, tool calls, and elapsed time. K denotes 1,000 tokens.

Method	Resolved	Input (K)	Output (K)	Tool calls	Time (s)
Standard	145	682	14.8	43.2	1,180
Step-aware only	217	708	16.1	46.5	1,260
Compact only	197	826	20.4	56.8	1,530
ReCompact	274	742	18.6	50.7	1,380

C.1 Targeted ablation aggregates

Table 9 expands the three ablations defined in Section 4.1. Dashes denote inapplicable post-compact metrics. All four conditions have 500 evaluated tasks and 500 valid observations for each cumulative input/output mean.

Table 9: Targeted ablations on 500 tasks per condition. NC: no continuation training; NL: no retention labels; NR: no online compact/reset. Lengths and cumulative task-token means are in K tokens. Length cells are P25/median/P75; repeat windows contain at most ten tool actions.

Metric	ReCompact	NC	NL	NR
Resolved tasks	274	226	232	220
Resolution (%)	54.8	45.2	46.4	44.0
Drop from full model (pp)	—	9.6	8.4	10.8
Context-exhaustion stops	18	27	23	106
Compact length (K)	1.7 / 2.2 / 2.7	1.7 / 2.2 / 2.8	1.7 / 2.2 / 2.7	—
Valid compact lengths	281	288	285	—
Compact boundaries	281	288	285	0
Boundaries with follow-up	269	275	273	—
Window tool actions	2,516	2,548	2,531	—
Command-repeat candidates	101	174	151	—
Test-repeat candidates	29	53	46	—
Commands / 100 actions	4.01	6.83	5.97	—
Tests / 100 actions	1.15	2.08	1.82	—
Boundaries with any candidate	44	78	69	—
Affected boundaries (%)	16.4	28.4	25.3	—
Mean task input (K)	742	801	784	695
Mean task output (K)	18.6	19.8	19.4	15.9

Table 10: Component contrasts derived from Table 3. Net changes are differences in total resolved counts.

Comparison	Net resolved change	Difference (pp)
Step-aware only − Standard	72	+14.4
ReCompact − Compact only	77	+15.4
Compact only − Standard	52	+10.4
ReCompact − Step-aware only	57	+11.4
ReCompact − Standard	129	+25.8
Average weighting contrast	—	+14.9
Average compact contrast	—	+10.9
Sum of single-component gains	—	+24.8
Factorial interaction	—	+1.0

D METRIC DEFINITIONS

Outcome contrasts. Let p_0, p_S, p_C, p_R be the percentage resolution rates for Standard, Step-aware only, Compact only, and ReCompact. The average compact contrast D_C, average weighting contrast D_W, and interaction I are

$$\begin{aligned} D_C &= \tfrac{1}{2}[(p_C - p_0) + (p_R - p_S)] = 10.9 \text{ pp}, \\ D_W &= \tfrac{1}{2}[(p_S - p_0) + (p_R - p_C)] = 14.9 \text{ pp}, \\ I &= p_R - p_C - p_S + p_0 = +1.0 \text{ pp}. \end{aligned} \tag{8}$$

Boundary distributions. Pre-compact length is the serialized input of the compact-emitting request; post-reset length is the input to the first subsequent request. Compact text length measures the retained compact. Post-reset input also includes the original system/task messages and their serialization. Reduction quantiles summarize the distribution of individual boundary ratios, $r_b = 1 - L_{\text{after},b}/L_{\text{before},b}$. The cohorts contain 246 and 281 boundaries. All 281 ReCompact boundaries have valid compact-text lengths; the corresponding length counts for the continuation and retention-label ablations are 288 and 285.

Windows and denominators. Each window begins after compact, includes at most ten tool actions, and ends early at the next compact or end of observed execution. Fourteen Compact-only and

twelve ReCompact boundaries have no observed follow-up actions and are excluded from repeat-rate denominators. The event-rate denominator is observed tool actions; the affected-boundary denominator is boundaries with follow-up.

Aggregation. Process summaries are unpaired method-level aggregates of observed windows. Each execution determines its compact timing and contributes all observed windows; boundaries from the same task share an execution history.

E Repeated Work and Request-Level Examples

Table 11: Post-compact repeated-failure candidates. Windows contain at most ten tool actions and end at the next compact or the end of execution. Command and test categories can overlap. Rates are derived from the displayed counts.

Metric	Compact only	ReCompact
Boundaries with follow-up	232	269
Observed tool actions	2,148	2,516
Failed-command repeat candidates	164	101
Failed-test repeat candidates	52	29
Boundaries with any repeat candidate	71	44
Boundaries without a repeat candidate	161	225
Command candidates / 100 actions	7.64	4.01
Test candidates / 100 actions	2.42	1.15
Affected follow-up boundaries (%)	30.6	16.4
Follow-up boundaries without candidates (%)	69.4	83.6
Derived comparisons: ReCompact vs. Compact only		
Command candidate rate reduction (%)	47.4	
Test candidate rate reduction (%)	52.4	
Repeat-free boundary difference (pp)	+14.2	

Repeat-candidate rule. A command is a candidate if a pre-compact execution showed a nonzero exit code, a failed test, or an explicit failure log; an identical command string is then executed after compact without a relevant intervening change to code, inputs, dependencies, or environment. Tests rerun to validate a repair are excluded. Failed-test and command-candidate categories overlap.

Rate calculation. Event rates are counts per 100 observed tool actions; boundary percentages use boundaries with follow-up. For either event category, relative reduction is $100(1-r_R/r_C)\%$, where r_R and r_C are the unrounded ReCompact and Compact-only rates. Table 11 gives the counts and rounded rates for both methods.

E.1 REQUEST-LEVEL EXECUTION SEGMENTS

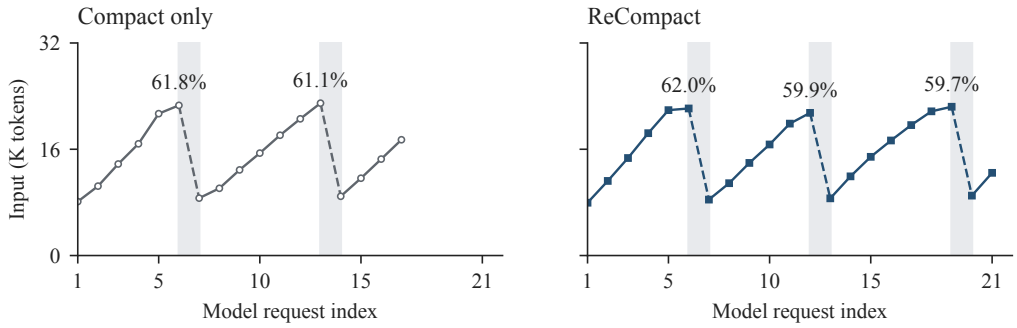

Figure 4: **Input growth and reset in two execution segments.** Shaded intervals and dashed drops mark reset; labels give input reduction at each event. Each panel shows a segment from a different task, selected by compact length near the method's median, independently of final outcome. Table 12 gives the exact transitions.

Figure 4 shows 17 requests for Compact only and 21 for ReCompact, with compact events at requests 6/13 and 6/12/19. Indices denote positions within each displayed sequence. Across the five resets, input length falls by 59.7–62.0% and then grows as execution continues.

Table 12: Transitions in the execution segments. Reductions are calculated from each displayed before/after pair.

Method	Request transition	Before	After	Reduction (%)
Compact only	$6 \rightarrow 7$	22,610	8,640	61.8
Compact only	$13 \rightarrow 14$	22,940	8,920	61.1
ReCompact	$6 \rightarrow 7$	22,140	8,420	62.0
ReCompact	$12 \rightarrow 13$	21,480	8,610	59.9
ReCompact	$19 \rightarrow 20$	22,400	9,020	59.7

After the displayed segments, the Compact-only execution ends in context-budget failure and ReCompact passes. The curves show the preceding cycles of input reduction and renewed history growth.

F ENLARGED METHOD DIAGRAM

Figure 5 enlarges the trajectory and training panels of Figure 1.

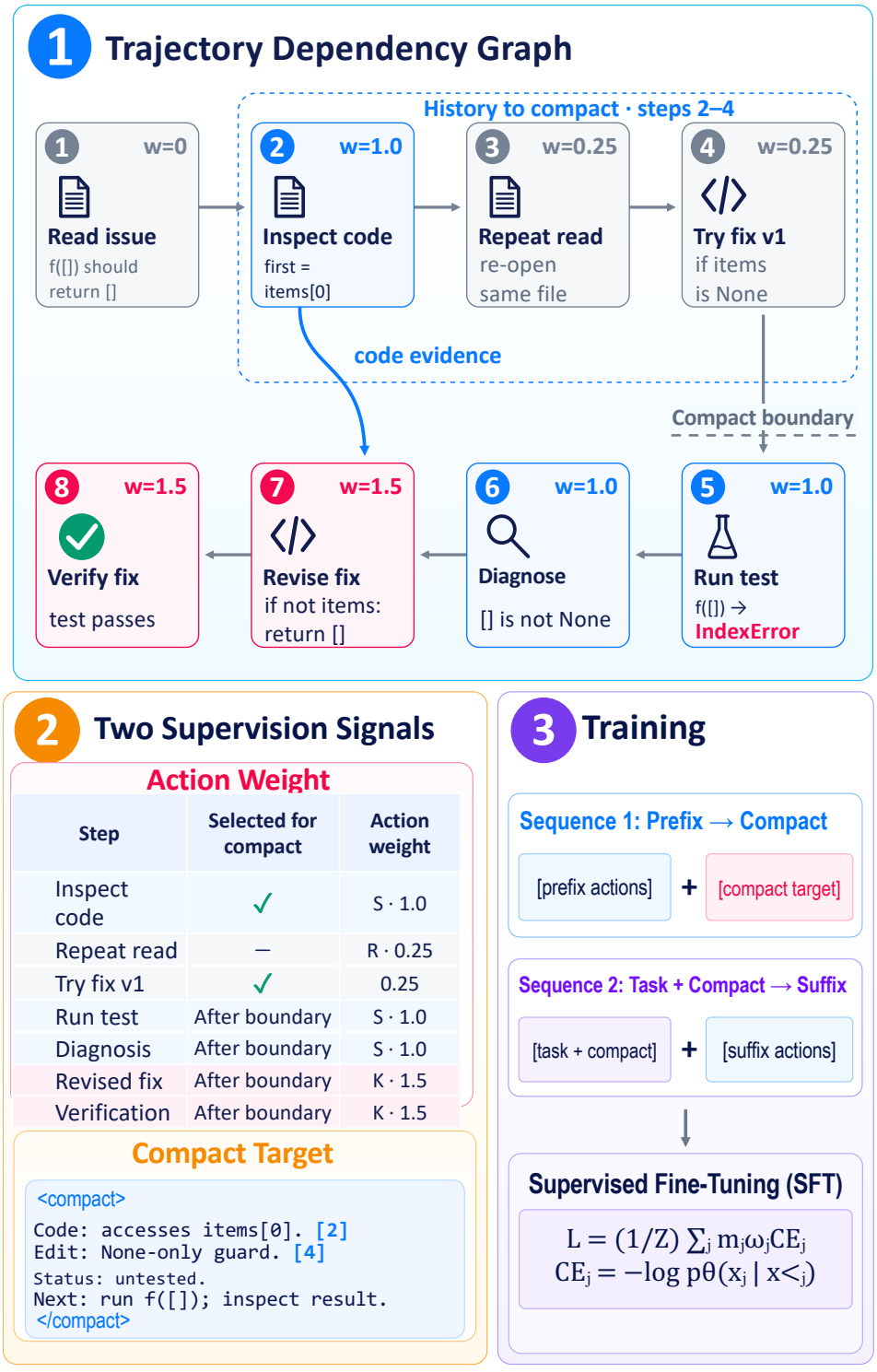

Figure 5: **Enlarged panels from Figure 1.** Top: trajectory dependencies. Bottom: supervision signals and training.

G SUPPORTING ARTIFACTS AND DATA AVAILABILITY

The accompanying plotting package contains the aggregate task outcomes, compact-boundary quantiles, bounded-window event counts, cumulative cost means, and request sequences used in the tables and figures.

The figure source, `figures/results/plot_results.py`, uses NumPy and Matplotlib to generate eight experimental figure sets as vector PDF/SVG and PNG previews, three of which appear in the manuscript. PDF fonts are embedded. The method schematic is a separate vector asset. The companion `plotted_values.json` records the numerical inputs and derived rates. The package reproduces figures from aggregate data and request sequences; it does not include raw trajectories or training and evaluation implementations.