AUTOCOMPACT: LEARNING WHEN TO COMPACT CONTEXT IN LONG-HORIZON CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents solve repository-level software engineering tasks through long trajectories of code inspection, search, editing, and testing. As a task progresses, earlier exploration becomes stale, so managing context is more than avoiding overflow: an agent must decide when to compact, what working state to preserve, and how to continue from it. We introduce AutoCompact, which trains a coding agent to make these decisions as part of its policy. To collect training data, we run the base agent on coding tasks and use a judge to review its compaction decisions, summaries, and actions after compaction. Flawed outputs are replaced with corrected ones before being executed in the environment, so each trajectory continues from the corrected decisions. We use these trajectories for supervised fine-tuning, then jointly optimize coding and compaction through reinforcement learning with task-success rewards. Experiments on SWE-bench Verified and SWE-PolyBench Verified show that AutoCompact improves pass rates over the base model by an absolute 9.2% and 5.0%, respectively. The improvements hold across all evaluated inference budgets, with a 256K context window that never overflows and with a 16K window whose overflow triggers fallback compaction.

1 INTRODUCTION

Large language model (LLM) agents have demonstrated strong capabilities on repository-level software engineering tasks by inspecting code, searching the repository, editing files, and running tests (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025). Resolving complex issues often requires investigating multiple candidate causes and iterating between implementation and validation, so trajectories accumulate code snippets, tool outputs, and intermediate findings. As the agent moves from one stage of a task to the next, much of this information, such as exploratory hypotheses, failed attempts, and detailed tool outputs, becomes stale. Keeping the full trajectory thus fills the context with outdated details, whereas the next stage needs only a compact *working state*: the conclusions reached so far, the relevant code and workspace status, and the remaining actions. Effective context management therefore requires an agent to decide *when* to compact, *what* information to preserve in the working state, and *how* to continue execution from it.

Existing approaches address only part of this problem. Length-triggered methods such as CompactionRL (Li et al., 2026b) compact only when the remaining context budget falls below a fixed threshold, which ties compaction to context length rather than task progress: stale exploration accumulates until the threshold is reached, and compaction may then occur in the middle of an unresolved stage whose evidence is still needed. Proactive methods instead let the agent decide when to compact, either through an inference-time rubric (Li et al., 2026a) or through fine-tuning on trajectories into which compression calls are inserted offline (Liu et al., 2026). However, deciding when to compact is not sufficient: the working state may omit or misrepresent key information, and even when it is correct, the agent may fail to follow it, revisiting completed exploration or ignoring the intended next action. Neither kind of proactive method addresses these failures: a rubric provides no training signal, and offline insertion keeps the original actions after each inserted call.

We introduce AutoCompact, which trains a coding agent to decide when to compact, what to preserve, and how to continue as part of its policy. To collect training data, we run the base agent on training tasks and use a judge to review its compaction decisions, the summaries it writes, and its actions after compaction. The judge replaces flawed outputs with corrected ones before they are

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.

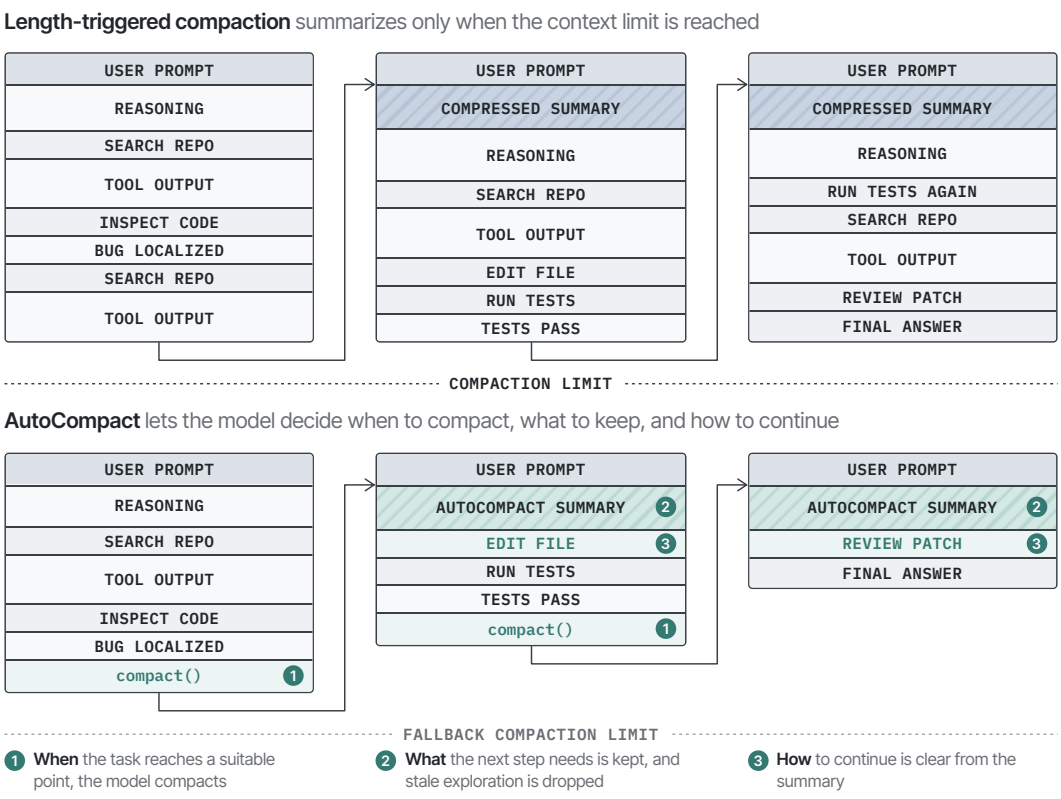


Figure 1: Length-triggered compaction versus AutoCompact. Under length-triggered compaction, the agent keeps accumulating context after the bug is localized, such as repeated searches with long outputs, and summarizes only when the context limit is reached. AutoCompact lets the model decide (1) when to compact, (2) what to keep, rewriting useful findings into a working-state summary while dropping stale exploration, and (3) how to continue from it.

executed in the environment, so each trajectory continues from the corrected decisions and shows how to act after compaction. We use these trajectories for supervised fine-tuning (SFT), and then apply outcome-based reinforcement learning (RL), which jointly optimizes coding and compaction with binary task success as the only reward signal.

We evaluate AutoCompact on SWE-bench Verified (Jimenez et al., 2024; OpenAI, 2024) and SWE-PolyBench Verified (Rashid et al., 2025), achieving pass rates of 39.6% and 24.5%, respectively, with improvements of 9.2% and 5.0% over the base model. We further examine cost efficiency on SWE-bench Verified under two complementary context regimes: a 256K setting in which no trajectory reaches the forced-compaction threshold, and a constrained 16K setting in which all methods share the same length-triggered fallback. The trained policies consistently improve performance across all evaluated budgets from \$0.10 to \$4.00 per task, with outcome-based RL providing further gains over SFT, especially at low budgets. Comparing the same trained checkpoint with and without executing compaction further shows that compaction is particularly beneficial under tight inference budgets, supporting its contribution to efficient task completion.

Our main contributions are as follows. (1) We introduce AutoCompact, which corrects compaction decisions, summaries, and subsequent actions during data collection and executes the corrected outputs in place of the original ones. (2) We collect 1,052 judge-corrected trajectories on SWE-rebench (Badertdinov et al., 2025) to initialize the policy via SFT, followed by outcome-based RL that jointly optimizes coding and compaction without compaction-specific reward shaping. (3) We demonstrate improved task success on SWE-bench Verified and SWE-PolyBench Verified, with gains across inference budgets and context settings. To rule out that these gains stem from training alone, we ignore the compaction calls of the same trained model; the resulting drop in pass rates confirms that executing compaction itself contributes.

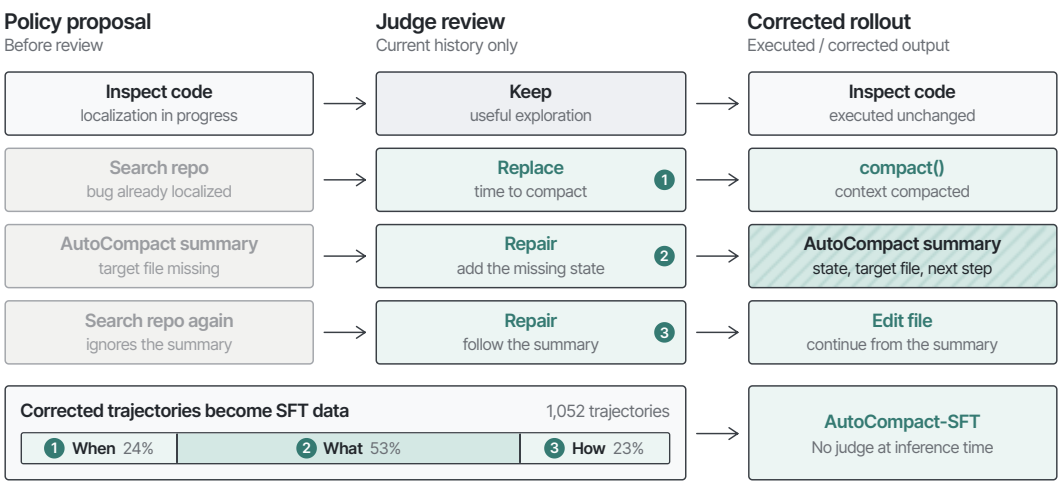


Figure 2: Judge-guided on-policy data collection. At each step, the judge reviews the policy's proposal using only the current execution history. Useful exploration is kept, while corrections target (1) when to compact, e.g., replacing a search after the bug is localized with `compact()`; (2) what to keep, e.g., repairing a summary that omits the target file; and (3) how to continue, e.g., redirecting a repeated search toward the edit planned in the summary. The environment executes the corrected outputs, and the resulting trajectories supervise SFT; the trained agent needs no judge.

2 METHOD

AutoCompact equips a coding agent with a `compact()` action (Section 2.1) and trains the policy in two stages. We first collect on-policy trajectories with judge-guided corrections to compaction timing, the generated working state, and the actions immediately following compaction (Section 2.2). These corrected trajectories provide the training data for SFT. We then apply outcome-based RL to jointly optimize coding and compaction using final task success as the reward (Section 2.3).

2.1 PROACTIVE CONTEXT COMPACTION

We augment the agent with a `compact()` action that can be invoked proactively during execution. The agent otherwise operates normally by inspecting files, editing code, and running tests. When it determines that the current stage has been sufficiently resolved and the accumulated history can be summarized, it may invoke `compact()` before reaching the context limit. For example, the agent may compact after localizing the cause of an issue and before proceeding to implementation.

Compaction replaces earlier interaction history with a model-generated working-state summary headed by `# Auto Context Summary`, while retaining the original task unchanged. The summary is intended to preserve established findings, relevant code and workspace state, and remaining actions, while omitting unnecessary details of prior exploration. The agent then resumes execution from the resulting context, as illustrated in Figure 1.

This mechanism makes compaction a model decision rather than solely a response to context-window pressure. Its effectiveness therefore depends on three behaviors: choosing an appropriate compaction point, constructing an accurate working state, and continuing reliably from that state. We next describe how we supervise these behaviors through judge-guided on-policy data collection.

2.2 JUDGE-GUIDED DATA COLLECTION AND SFT

Prompting alone does not induce proactive compaction. In our preliminary experiments, the base model rarely invoked `compact()` before the context limit, even when compaction rules were specified in the agent prompt. Proactive compaction introduces a behavior that is uncommon in the base policy: the model must interrupt an otherwise valid execution trajectory and replace its accumulated context before any context-limit constraint forces it to do so. We therefore collect training trajecto-

ries with online judge corrections and use them for SFT. These corrections provide demonstrations of compaction and subsequent execution that the base policy rarely produces on its own. The judge is used only during data collection; the trained agent runs without the judge at inference time.

To collect such demonstrations, we roll out the base policy on software engineering tasks and use a judge to review each proposed action as execution proceeds. The judge receives only the trajectory history available at the current step and determines whether the proposed action is satisfactory, following a predefined annotation protocol that specifies appropriate compaction points, the information that should be preserved in the compacted working state, and how post-compaction behavior should continue. Figure 2 provides an overview of this online corrective data-collection process, with one concrete example of each of the three types of correction described in the next paragraph.

We supervise three types of decisions corresponding to the main failure modes of compaction. *Trigger correction* checks whether the current point is appropriate for compaction: the agent should compact when the current stage has been sufficiently resolved, but continue exploring when intermediate evidence is still needed. *Working-state correction* checks whether the generated `# Auto Context Summary` accurately preserves the information needed for subsequent execution, including relevant conclusions, code and workspace state, and remaining actions. *Continuation correction* checks whether the first actions after compaction follow the resulting working state, rather than revisiting completed exploration or ignoring the intended next step, such as re-running a search whose result is already recorded in the working-state summary.

Corrections are applied before execution, so each one shapes the rest of the trajectory. When the judge identifies an output unsatisfactory, it writes a corrected output, which the environment executes in place of the original; the policy then generates the following actions from this corrected history. This differs from post-hoc annotation, where corrections do not influence later states.

Finally, we fine-tune the base policy on the corrected trajectories with the standard next-token prediction objective. The resulting policy, AutoCompact-SFT, learns both ordinary coding behavior and the three compaction behaviors introduced above: selecting appropriate compaction points, constructing actionable working states, and reliably continuing execution afterward. SFT provides a behavioral initialization for proactive compaction, but it optimizes the locally corrected decisions rather than their eventual effect on task completion. We then further optimize the policy using outcome-based RL, which directly targets task success.

2.3 Joint RL for Task Solving and Compaction

Starting from the SFT checkpoint, we further train the agent on SWE-Gym (Pan et al., 2025) with end-to-end multi-turn RL (Xue et al., 2026), using group relative policy optimization (GRPO) (Shao et al., 2024) and a fully asynchronous RL system. Each rollout receives a binary outcome reward based on whether the final patch passes the task tests. Within each rollout group, GRPO converts these outcomes into relative advantages, which provide a shared learning signal for all model-generated tokens in a trajectory, including regular coding actions, the decision to invoke `compact()`, the generated context summary, and the continuation after compaction.

Compaction therefore requires neither a separate auxiliary objective nor compaction-specific reward shaping. Instead, task solving and context management are trained jointly under the final task reward, allowing the policy to learn when compaction is beneficial, what information should be preserved, and how execution should continue afterward.

Compaction also changes how trajectories are converted into training sequences, because each call to `compact()` rewrites the context. As a result, a trajectory with compaction no longer forms a single sequence in which each model turn extends the previous context. We therefore split each trajectory into segments wherever the context prefix is rewritten, such that within each segment every model turn is conditioned on a prefix that only grows. All segments of a trajectory share its advantage, so the decision to compact, the summary, and the actions after compaction receive the same outcome-based signal even though they fall into different segments. The policy loss is averaged at the token level over all segments in a batch, without a KL penalty or an entropy bonus, so the binary outcome reward is the only training signal for both coding and compaction.

Table 1: Full-run pass rates without an inference-cost cap, grouped by compaction trigger. All methods share the same base model and scaffold. In the Context column, 256K denotes the model context window, while † marks a 16K forced-compaction threshold.

Method	Trigger	Optimization	Context	SWE-bench Verified (%)	SWE-PolyBench Verified (%)
Full-history baseline					
Base	—	—	256K	30.4	19.5
Length-triggered compaction					
Fixed Compaction	Length	—	16K†	28.8	18.6
CompactionRL (Li et al., 2026b)	Length	RL	16K†	32.7	19.8
Proactive compaction					
SelfCompact (Li et al., 2026a)	Rubric	—	256K	31.7	20.6
SWE-Compressor (Liu et al., 2026)	Learned	SFT	256K	31.0	20.1
AutoCompact-SFT	Learned	SFT	256K	32.2	21.7
AutoCompact	Learned	SFT → RL	256K	**39.6**	**24.5**

3 EXPERIMENTS

Our experiments address three questions. First, does AutoCompact improve task success over full-history execution and over existing length-triggered and proactive compaction methods (Section 3.2)? Second, how do proactive compaction, outcome-based RL, and the execution of compaction itself contribute across inference budgets and context settings (Section 3.3)? Third, how do the summaries produced after SFT and after RL differ (Section 3.4)?

3.1 EXPERIMENTAL SETUP

Models and scaffold. We use Qwen3-Coder-30B-A3B-Instruct (Yang et al., 2025a) as the base model throughout our experiments and evaluate all model variants on two repository-level software-engineering benchmarks: SWE-bench Verified (Jimenez et al., 2024; OpenAI, 2024) and SWE-PolyBench Verified (Rashid et al., 2025). All variants use the same terminal-based read-eval-print loop (REPL) scaffold (Lin & Liu, 2025; Du et al., 2025), augmented with the model-invocable `compact()` action described in Section 2.1. During supervised data collection, we additionally use GPT-5.5-Codex as the judge for online correction; the judge is removed during evaluation.

Training. For SFT, we roll out the base model on 379 SWE-rebench tasks (Badertdinov et al., 2025) and collect 1,052 judge-corrected trajectories using the procedure in Section 2.2, after filtering out malformed requests, summary loops, and off-track continuations. Of these trajectories, 24% supervise compaction triggers, 53% supervise working-state construction, and 23% supervise post-compaction continuation. We train for two epochs with a learning rate of 5×10^{-7} and a batch size of 8. Starting from AutoCompact-SFT, we further optimize the policy with GRPO on SWE-Gym (Pan et al., 2025). For each task, we sample 8 trajectories and assign a binary outcome reward according to whether the final patch passes the task-specific test suite. RL uses a learning rate of 1×10^{-6} and a batch size of 64, with rollouts capped at 50 environment steps and 32K tokens.

Evaluation. We report pass rates on SWE-bench Verified and SWE-PolyBench Verified using full agent runs without an inference-cost cap. All reported quantitative evaluation results are averaged over three runs. To assess cost efficiency, we additionally evaluate SWE-bench Verified under six per-task inference budgets from \$0.10 to \$4.00 (Figure 3). Each rollout is terminated when its cumulative inference cost reaches the specified budget. We estimate inference costs from token usage using the official Alibaba Cloud Model Studio pricing for Qwen3-Coder-30B-A3B-Instruct (Alibaba Cloud, 2026). To account for prefix caching, we price cached tokens at 20% of the standard input-token rate, following Alibaba Cloud Model Studio's implicit-caching pricing, and price uncached input and output tokens at their respective standard rates.

Baselines. We compare AutoCompact against several context-management strategies. Base uses the same underlying model and scaffold without learned proactive compaction. Fixed Compaction

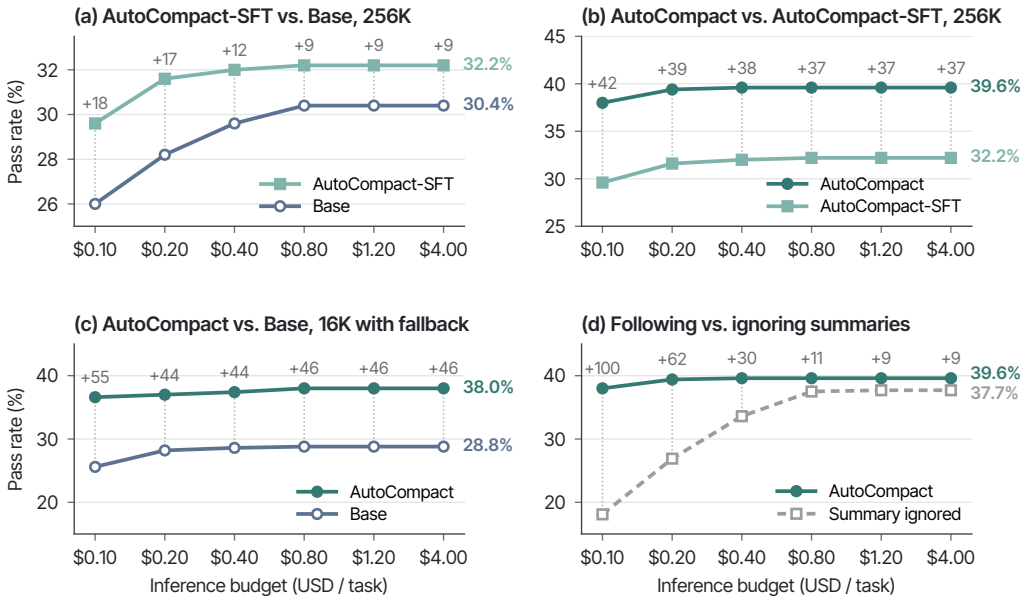


Figure 3: Pass rate versus inference budget on SWE-bench Verified. (a) AutoCompact-SFT versus Base at 256K, where no trajectory reaches the forced-compaction threshold. (b) AutoCompact (after RL) versus AutoCompact-SFT at 256K. (c) AutoCompact versus Base at 16K, with a shared forced-compaction fallback. (d) The same checkpoint following versus ignoring summaries. Budget categories are equally spaced; gains denote additional solved tasks per 500 tasks.

triggers compaction when the remaining context budget falls below a predefined threshold. SelfCompact (Li et al., 2026a) uses an inference-time rubric to guide compaction decisions, while CompactionRL (Li et al., 2026b) jointly trains task execution and summary generation under a length-triggered compaction mechanism. SWE-Compressor (Liu et al., 2026) learns to invoke a context-management tool through supervised fine-tuning on trajectories reconstructed offline by inserting compression calls. For this SFT comparison, we collect trajectories from comparable SWE-rebench task sets and use similar numbers of training trajectories for SWE-Compressor and AutoCompact-SFT. Unlike these baselines, AutoCompact corrects the agent's compaction decisions and subsequent actions during data collection, directly supervising how the agent uses the compacted state.

3.2 Main Results

As shown in Table 1, AutoCompact achieves the highest full-run pass rates on both benchmarks, improving over Base by 9.2% on SWE-bench Verified and 5.0% on SWE-PolyBench Verified.

Length-triggered compaction yields limited gains. Fixed Compaction underperforms Base by 1.6% and 0.9% on the two benchmarks, indicating a performance cost in this constrained-context setting. CompactionRL improves upon Fixed Compaction by 3.9% and 1.2% through RL. However, its gains over Base remain limited, leaving a substantial gap to AutoCompact.

Proactive compaction outperforms full-history execution. Under the 256K setting, all evaluated proactive methods, including SelfCompact, SWE-Compressor, and the AutoCompact variants, outperform Base on both benchmarks. SelfCompact achieves improvements through inference-time guidance alone, while SWE-Compressor learns compaction-tool use through SFT. These results suggest that proactive context management helps even when the full history fits in context.

Online judge-guided SFT and outcome-based RL both add gains. AutoCompact-SFT exceeds SWE-Compressor by 1.2% and 1.6% on the two benchmarks. Since both use the same base model, scaffold, and comparable training data, this comparison suggests that online judge-guided correction is more effective than offline insertion into completed trajectories. AutoCompact-SFT learns from trajectories in which the corrected outputs, including corrected actions after compaction, were ac-

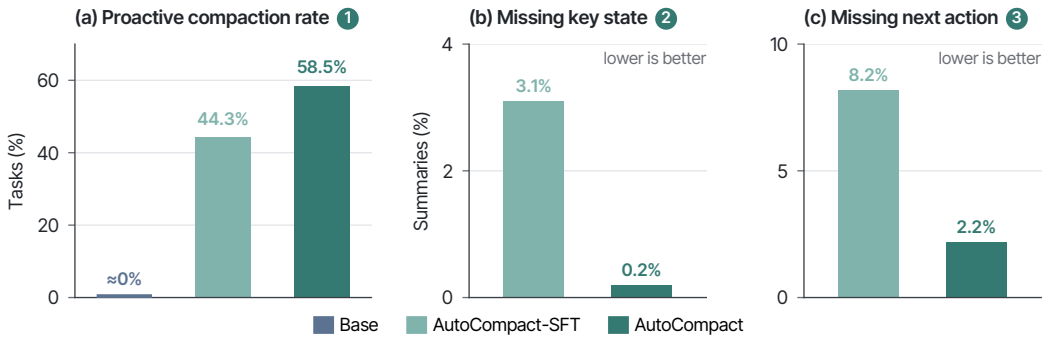


Figure 4: Compaction behavior, with (1), (2), and (3) marking when to compact, what to keep, and how to continue (Figure 1). (a) Fraction of tasks in which the model proactively invokes `compact()`. (b) Fraction of summaries that omit relevant task or workspace state. (c) Fraction of summaries that omit a concrete next action. Lower is better in (b) and (c).

tually executed, whereas SWE-Compressor inserts compaction calls into completed trajectories and keeps their original subsequent actions. Outcome-based RL then improves pass rates by a further 7.4% and 2.8% over AutoCompact-SFT on the two benchmarks.

3.3 Further Analysis

Proactive compaction helps before context overflow. We first compare Base with AutoCompact-SFT under the 256K long-context setting. Because no trajectory reaches the forced-compaction threshold, this comparison tests whether supervised training for proactive compaction improves agent performance even when the full interaction history fits within the context window. As shown in Figure 3 (a), AutoCompact-SFT achieves a higher pass rate than Base across all evaluated inference budgets. This result shows that retaining the complete interaction history is not necessarily optimal, even when the context window is sufficiently large. Over long trajectories, obsolete hypotheses, failed attempts, and verbose tool outputs can accumulate and distract the model from the current working state. By replacing stale history with a concise working state, AutoCompact-SFT improves performance before context becomes a bottleneck.

Outcome-based RL improves beyond supervised imitation. Under the same 256K protocol, we compare AutoCompact-SFT with AutoCompact. As shown in Figure 3 (b), AutoCompact consistently outperforms AutoCompact-SFT across all inference budgets, with the largest gains in the low-budget regime. These results suggest that SFT establishes the basic compaction behavior, while outcome-based RL further improves how this behavior is used to maximize task success. Since low budgets cut rollouts short, the larger gains at these budgets indicate that RL makes the agent more cost-efficient rather than simply relying on longer trajectories.

RL improves compaction behavior without compaction-specific rewards. Figure 4 compares compaction behavior before and after RL. While Base rarely invokes `compact()`, AutoCompact-SFT already uses it on 44.3% of tasks, indicating that supervised training establishes the basic behavior. RL extends its use to 58.5% of tasks while reducing both types of summary omission: key-state omissions fall from 3.1% to 0.2%, and next-action omissions fall from 8.2% to 2.2%. These statistics are computed on trajectories for all 500 SWE-bench Verified tasks using keyword-based screening, supplemented by random manual spot checks. The two trends are complementary: compaction is used across more tasks, yet fewer summaries are flagged for missing state or next-action information. This suggests that task-success rewards encourage the agent to preserve information needed to continue the task from the compacted state.

Proactive compaction complements a length-triggered fallback. We evaluate Base and AutoCompact under a shared 16K forced-compaction threshold. This setting represents a context-constrained runtime in which both systems use forced compaction as a fallback, while AutoCompact can also invoke `compact()` proactively before reaching the threshold. As shown in Figure 3 (c),

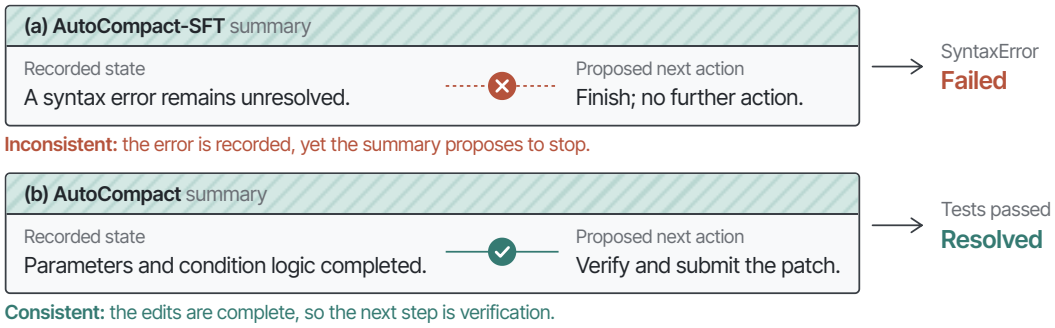


Figure 5: Summary self-consistency on SWE-bench Verified task `django-13809`. This qualitative example contrasts paraphrased summaries from AutoCompact-SFT and AutoCompact with their final evaluation outcomes. Each summary records a state and proposes a next action; the marker between them indicates whether the action is consistent with the recorded state.

AutoCompact achieves a higher pass rate than Base across all evaluated inference budgets. This result shows that learned proactive context management remains effective when deployed together with a length-triggered fallback, allowing the agent to reorganize its context before compaction becomes mandatory. It also indicates that compaction based on task progress is more effective than compaction based solely on the length of the context.

Compaction contributes to the gains of the trained agent. The gains of AutoCompact may reflect both better agent behavior learned during training and the use of compaction during inference. To assess the latter contribution, we compare normal execution with a Summary-ignored variant of the same trained checkpoint. In this variant, `compact()` calls are skipped: no summary is generated, and execution continues with the existing history. Both conditions retain all learned parameters, allowing us to test whether executing compaction provides a benefit beyond the improvements encoded in those parameters. Figure 3 (d) shows that normal execution achieves higher pass rates, with the largest advantage under tight inference budgets. This indicates that the observed gains cannot be attributed solely to better agent behavior learned during training: part of the benefit depends on actually executing compaction. The advantage is 19.9% at \$0.10 and remains 1.9% at \$4.00, so executing compaction improves cost efficiency without reducing final performance.

3.4 Case Study: Summary Self-Consistency under SFT and RL

Preserving the relevant state and specifying a next action do not by themselves make a summary usable. The summary-content metrics in Figure 4 check both, yet a summary can satisfy them while proposing a next action incompatible with the state it records, violating what we call *summary self-consistency*. Figure 5 illustrates this gap through paraphrased summaries from AutoCompact-SFT and AutoCompact on the same SWE-bench Verified task.

The two summaries in Figure 5 differ in exactly this respect. The AutoCompact-SFT summary records an unresolved syntax error but still proposes finishing without further action: the error is retained in the summary, yet it fails to constrain the proposed continuation. In contrast, the AutoCompact summary reports that the parameter and conditional-logic changes are complete and proposes verification before submission. This next step is consistent with the recorded implementation state. The corresponding AutoCompact-SFT run fails at evaluation with a `SyntaxError`, while the AutoCompact run passes the tests and resolves the task.

This contrast highlights summary self-consistency as a dimension of summary quality beyond information coverage. The difference may stem from the training objectives: SFT imitates demonstrated summaries, which does not ensure that the recorded state constrains the next action, whereas RL rewards a summary only through the success of the actions that follow it.

4 Related Work

Coding agents. Research on repository-level coding improves task-solving capabilities through both harness design and model training. Work on harness design organizes repository exploration, code editing, and validation through agent-computer interfaces, tool execution environments, and structured repair workflows (Yang et al., 2024; Wang et al., 2025; Xia et al., 2025). Work on model training improves coding and problem-solving capabilities through executable task environments, expanded task and trajectory datasets, and supervised or reinforcement learning (Pan et al., 2025; Yang et al., 2025b; Wei et al., 2025). AutoCompact combines the two as model-harness co-design: the harness exposes a `compact()` action, and the model learns to use it jointly with task execution rather than through a separate compression module.

Memory and context management. Long-horizon agents manage growing interaction histories through memory hierarchies (Packer et al., 2023), abstraction of raw observations (Zheng et al., 2024), summaries of accumulated observations and interactions (Kang et al., 2026; Wu et al., 2025; Liu et al., 2026), or compression of sub-trajectories and selected history spans (Sun et al., 2025; Ye et al., 2025; Gao et al., 2026). These works mainly design how retained context is represented. AutoCompact uses a simple representation, a single working-state summary alongside the original task and recent turns, and focuses on when to compact and how to continue afterward.

Learning context-management policies. Rubric- and guideline-based methods (Li et al., 2026a; Kang et al., 2026) steer compaction through instructions rather than training the agent's own compaction behavior. Supervised methods learn from constructed trajectories (Liu et al., 2026; Ye et al., 2025; Gao et al., 2026), for example by inserting context-management actions into completed trajectories (Liu et al., 2026), which keeps the original actions after each insertion, or by synthesizing them, as in concurrent work SWE-MeM (Gao et al., 2026), where a stronger model decides when to compress a span of steps and writes the summary but generates no task actions. RL methods (Wu et al., 2025; Sun et al., 2025; Li et al., 2026b; Gao et al., 2026) optimize context management through rewards, in SWE-MeM with additional step-level masks for memory-management failures, but do not directly supervise the actions that follow compaction. Unlike these methods, AutoCompact corrects the agent's own compaction decisions, summaries, and subsequent actions during data collection, executes the corrected outputs, and refines this behavior with outcome-based RL alone.

5 Conclusion

AutoCompact trains coding agents to manage context as part of task execution. Judge-corrected trajectories provide supervision for compaction decisions, summary construction, and the actions that follow compaction, followed by outcome-based reinforcement learning that jointly optimizes coding and compaction. Experiments on SWE-bench Verified and SWE-PolyBench Verified show improved task success across all evaluated inference budgets, and ignoring the compaction calls of the same trained model lowers pass rates, showing that the gains do not come from training alone. These findings highlight the value of training not only the construction of compacted states but also the agent's behavior that follows from these states, which prior methods leave unsupervised.

Limitations and future work. Due to resource constraints, RL training uses 32K-token sequences, shorter than the 256K context window used at evaluation. Nevertheless, the gains across all evaluated budgets in both the 256K and 16K settings indicate that this training setting suffices to learn useful proactive compaction. More broadly, AutoCompact is a form of model-harness co-design: the harness provides the compaction mechanism, while the model learns when to invoke it, what to preserve, and how to continue afterward. We study this co-design within a single scaffold. Widely used agent harnesses such as Codex and Claude Code currently compact the context automatically when it approaches the window limit, which is the length-triggered design discussed in Section 1. Our 16K results suggest that combining such mechanisms with learned proactive compaction could further improve these systems, which we leave for future work.

AI Use Statement

In this work, we used generative AI tools for language editing and trajectory correction for training. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Alibaba Cloud. Alibaba Cloud Model Studio: Model inference pricing. https://www.alibabacloud.com/help/en/model-studio/model-pricing, 2026.

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*, 2025.

Cunxiao Du, Tianduo Wang, Longxu Dou, Shenggui Li, Tianjie Zhang, Tianyu Liu, Xianwei Chen, and Min Lin. On policy annotation: How minimal human edits unlock massive gains in LLM agents. https://terminal-agent.github.io/blog/annotation/, 2025. Blog post.

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.

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.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. ACON: Optimizing context compression for long-horizon LLM agents. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

Tianjian Li, Jingyu Zhang, William Jurayj, Xi Wang, Chuanyang Jin, Mehrdad Farajtabar, Eric Nalisnick, and Daniel Khashabi. Self-compacting language model agents. *arXiv preprint arXiv:2606.23525*, 2026a.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*, 2026b.

Min Lin and Zichen Liu. Terminal: LLM's last tool. https://terminal-agent.github.io/blog/tool/, 2025. Blog post.

Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon SWE-agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026.

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

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 *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025.

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*, 2025.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. ReSum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. SimpleTIR: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. In *International Conference on Learning Representations*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

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. In *Advances in Neural Information Processing Systems*, 2025b.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, et al. AgentFold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, 2024.