TRAINING UNDER AN UNKNOWN FUTURE: LONG-HORIZON TASK AGENTS THAT LEAVE ROOM FOR WHAT COMES NEXT

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents are increasingly asked to implement successive requirements over long horizons, and a change focused solely on the current requirement may make later ones infeasible or substantially more expensive to implement. Existing training does not address this failure: rewards are assigned to the current task before downstream consequences become observable. On SlopCodeBench, which evaluates three to eight sequential requirements while requiring all earlier tests to keep passing, none of the fifteen evaluated frontier agents solves a complete problem.

We propose a training method built on *consequence chains* mined from real development histories. Each chain links an early structural decision to later requirements that expose its downstream effects. We train on three-requirement chains, a practical length for exposing these effects. The objective first requires whole-chain test correctness, then refines the reward with a model-based judge's assessment of downstream impact, paying the model for writing code future steps can build on.

On ChainSWE, our method improves the complete-chain success rate on the agent's own accumulated patches by at least 28% relative to the corresponding base models, across scales from 2B to 35B. Our 27B model achieves a sequential bug-resolution rate of 45.7% on ChainSWE, compared with 42.4% for Claude Opus 4.6 under the same harness. On SlopCodeBench, checkpoint-level improvements extend beyond the three-step training horizon: on sequences of up to eight steps, the trained 35B model improves core solve rate by 8.0 percentage points at steps four through eight, compared with 2.8 points over the first three.

1 INTRODUCTION

An agent is typically evaluated on a task that starts from a clean state and ends with a single verdict. Work in practice often has a different structure: requirements arrive one after another, each step implements one requirement on top of the artifact produced by the preceding steps, and the agent does not know what will be requested next. In this setting, immediate correctness is not sufficient. Each step must satisfy the current requirement while leaving code future steps can build on.

A change optimized only for the current requirement can make subsequent requirements infeasible or substantially more expensive to satisfy. SlopCodeBench grades three to eight consecutive requirements per problem on the same codebase and requires all tests introduced by earlier steps to continue passing. None of the fifteen evaluated frontier agents solves a complete problem under this protocol (Orlanski et al., 2026). We study the capability required to avoid this failure in software development.

Existing training approaches for coding agents do not explicitly optimize for this capability. The reward is assigned when the current task is evaluated: an edit is scored by its similarity to a reference patch or by the task's own tests, and each episode begins from a repository snapshot created by a human developer and ends with that task's verdict (Wei et al., 2025a;b; Du et al., 2025; Golubev et al., 2025). When training spans multiple tasks, the information carried across task boundaries is typically the agent's context or reusable skills rather than the evolving artifact itself (Wang et al., 2025; Yao et al., 2026; Chen et al., 2026). Existing objectives therefore do not assign credit to an

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.

earlier step for the downstream consequences of the artifact it produces; Section 2 examines these approaches in detail.

General engineering principles alone are insufficient to determine how a particular change will affect a particular codebase. The central challenge is to connect an early implementation decision to its consequences for later requirements. We address this challenge by mining *consequence chains* from the commit histories of open-source repositories: sequences of requirements in which later changes build on structures introduced earlier. These histories supply the requirements and reference implementations. During training, the agent implements each requirement on top of its own preceding changes, allowing the resulting development process to reveal whether later steps can build on those changes or must rework them.

Training over increasingly long requirement sequences incurs substantial rollout costs, motivating a method that learns from short chains and is evaluated on longer sequences. We first improve a model-based judge's ability to assess downstream impact through offline calibration. The judge initially evaluates an edit using only information available at that step, then uses the execution outcomes of subsequent steps as feedback to improve its judgment. During policy training, we use three successive requirements, a practical length for exposing downstream effects. The judge evaluates each completed chain using its calibrated judgment together with observed execution outcomes and returns a single hindsight score. Combined with whole-chain correctness, this score provides a reward designed to support generalization beyond the training horizon. Since the criteria obtained by comparing outcomes in hindsight apply to any step, supervision from three-step chains can transfer to longer sequences.

On ChainSWE, training improves the complete-chain success rate on the agent's own accumulated patches by at least 28% relative to the corresponding base model, across all three scales from 2B to 35B. Our 27B model achieves a sequential bug-resolution rate of 45.7% on ChainSWE, compared with 42.4% for Claude Opus 4.6 under the same harness; among the agents we evaluate, only Claude Opus 5 performs better. On SlopCodeBench, checkpoint-level improvements also extend beyond the three-step training horizon. On sequences of up to eight steps, the trained 35B model improves core solve rate by 8.0 percentage points at steps four through eight, compared with 2.8 points over the first three steps. On SlopCodeBench, where an agent extends its own earlier code, strict solve rates rise from 4.1% to 7.1% for the 35B model and from 8.2% to 12.2% for the 27B model. The 2B model also improves its strict pass rate (Section 4.2).

Our contributions are threefold:

- We formulate the setting and capability required for long-horizon software development: the agent acts without knowing future requirements, implements each requirement on its own accumulated changes, and must satisfy the current requirement while leaving code future steps can build on (Section 2.1).
- We construct *consequence chains* from real commit histories and train agents on the outcomes of implementing successive requirements on their own evolving code. The reward requires whole-chain correctness and uses a single hindsight judgment of the completed trajectory to further distinguish successful implementations (Sections 3.1 and 3.4).
- We show that the method improves the complete-chain success rate on ChainSWE across all three model scales, that the trained 27B model exceeds Claude Opus 4.6 in sequential bug resolution under the same harness, and that the 35B model's checkpoint-level improvement on SlopCodeBench extends beyond the three-step training horizon (Section 4).

2 Background

2.1 Long-horizon work under an unknown future

We study sequential software development in which requirements are revealed one at a time, code carries over, and earlier behavior must remain correct. In our training and evaluation episodes, requirements are fixed in advance but hidden from the agent until their respective steps.

Consider a task manager that must support tagging and filtering tasks (Figure 1). Storing a tag as a task attribute can pass the initial tests. Later requirements introduce multiple tags per task, then tags

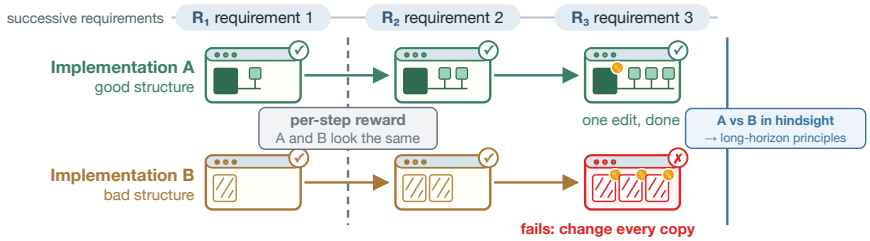


Figure 1: Two implementations of the same three successive requirements: A with a good structure (one shared module that every part uses) and B with a bad one (each requirement copies the same logic into its own block). Both pass the tests at R_1 and R_2; only R_3, which changes that logic, is one edit in A but touches every copy in B. A per-step reward therefore scores A and B alike, while comparing them in hindsight yields judging criteria that can review any step.

with their own attributes, such as a color, and renaming operations. These changes require giving tags their own identity and reworking the original representation. Current tests verify the requested behavior but do not reveal this later rework. The initially shorter implementation with fewer abstractions can therefore prove harder to extend: brevity and simplicity alone do not determine how well a change supports subsequent work. Comparing the two outcomes in hindsight yields judging criteria that apply to any step, not only to the three executed here.

Recent benchmarks evaluate successive requirements on one repository with accumulating tests (Orlanski et al., 2026; Deng et al., 2026; Shen et al., 2026; Jin et al., 2026; Sobal et al., 2026; Huang et al., 2026). Tasks solved at over 80% in isolation drop to 38% when performed in sequence (Deng et al., 2026). Agents score 22 to 40 percentage points higher when each step starts from a reference solution rather than their own accumulated code (Shen et al., 2026). On ChainSWE, resolution rises with position when bugs start from reference fixes to earlier bugs, but falls for every evaluated agent when fixes accumulate on its own code (Jin et al., 2026). These comparisons distinguish the difficulty of individual tasks from that of continuing on agent-produced code. The decline at later positions cannot be explained solely by those tasks being harder; the state inherited from earlier steps also matters.

General engineering advice does not resolve this difficulty. On SlopCodeBench, *plan first* and *avoid constructions that are hard to extend* lower strict solve rates by 3.6 and 2.4 percentage points relative to the neutral prompt (Orlanski et al., 2026). The strongest per-step agent also preserves no more earlier behavior than substantially weaker agents (Appendix A). These findings motivate training the judgment needed for continued development, as advocated by the benchmark's call for training-side intervention (Orlanski et al., 2026).

2.2 What is trained today

Post-training for software agents pays for correctness on the task at hand. SWE-RL scores an edit by its similarity to the reference patch (Wei et al., 2025a). Self-play and feature-driven environments score it by the task's tests, paying only when the new tests pass and the repository's existing tests keep passing (Wei et al., 2025b; Du et al., 2025). Multi-turn training refines one fixed task across turns, and those turns are long — a stateful environment, hundreds of thousands of tokens of interaction (Golubev et al., 2025) — but the episode still begins at a snapshot a human committed and ends when that one task is graded. Asynchronous infrastructure has made rollouts of many hours trainable (Hou et al., 2026; Fu et al., 2025). Since length is not what limits these methods, what limits them is what they pay for.

A second line is not content with tests alone and puts the quality of the code into the objective. Static quality metrics enter directly (Robeyns & Aitchison, 2025), and judge preferences over readability and style enter through preference optimization (Weyssow et al., 2024). Its most thorough member walks a repository's real commit history, attempts each change blind, compares its attempt against the reference diff, and distils what it missed into a skill document covering the project's conventions, its internal APIs and its implicit architectural constraints (Li et al., 2026). What this line optimizes

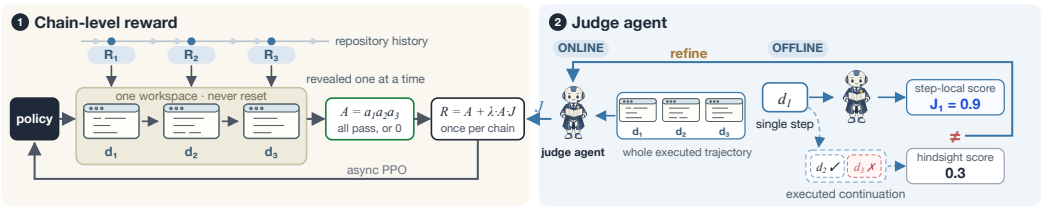


Figure 2: Method overview. **(1)** Consequence chains $R_{1:3}$ are mined from repository history and revealed one requirement at a time. The agent implements them on one never-reset workspace, producing edits $d_{1:3}$ and correctness outcomes $a_{1:3}$; the reward $R = A + \lambda AJ$ with $A = a_1a_2a_3$ is paid once per chain. **(2)** Offline, the judge scores a step from step-local evidence and again in hindsight after the rollout's executed continuation; discrepancies refine the judging criteria (Section 3.3). During policy training the same judge, with fixed criteria, reads the complete trajectory $d_{1:3}$ and returns one hindsight score J.

is conformance to what the codebase already is, scored against the reference patch and judged on scope, redundancy and style. Conformance to existing convention does not entail having left room for a requirement nobody has stated yet.

A third line does let return cross the boundary of the current task. SAGE trains on chains of tasks so that skills generated in an earlier task are paid for by their use in a later one (Wang et al., 2025), and SkillRise sequences related instances and supervises curation with discounted downstream outcomes (Yao et al., 2026). CoD assigns credit across interleaved solve-task and update-context episodes in a single environment (Chen et al., 2026). The problem-setter in SSR is paid by the solve rate of later attempts at the bug it injected, and its authors list interdependent development across sessions as future work (Wei et al., 2025b). In each of these the sequence exists so that the agent can find and reuse something that is already there: a solution pattern shared by similar tasks, or a fixed hidden fact about one environment, where a good note is one that records the ground-truth mapping the agent has worked out for itself. What is carried from task to task is a description of a world that does not change, and later tasks get easier for having it.

Taken together, these training approaches improve current-task correctness, conformance to existing code conventions, or the reuse of knowledge across tasks. What remains under-supervised is how an agent's implementation shapes subsequent development on the same codebase.

3 METHOD

Training this capability requires both data that exposes the consequences of architectural decisions and a reward designed to support generalization beyond the training horizon. First, the data must contain dependencies through which early architectural choices affect the implementation of later requirements. Executing those requirements on the agent's own code reveals whether later steps can build on the earlier structure or must rework it. Second, the cost of long rollouts limits the requirement sequences we can afford to train on, while our goal is to support development beyond those sequences. Whole-chain correctness establishes success on the observed requirements, but does not distinguish successful implementations by how well they support further development. This motivates supplementing correctness with an assessment of whether each step leaves code that supports subsequent requirements, informed by evidence from actual later execution.

We mine consequence chains from repository histories, selecting sequences in which later changes build on structures introduced earlier. During training, the agent implements three successive requirements in a persistent workspace, receiving each requirement only when its step begins. We calibrate judging criteria offline by comparing assessments made at individual steps with hindsight assessments informed by subsequent execution. During policy training, the judge reviews each completed trajectory and its execution results and returns a single hindsight score. The reward requires whole-chain correctness, then uses this score to distinguish successful trajectories. We evaluate whether this short-chain supervision improves performance on longer requirement sequences. Figure 2 gives an overview.

3.1 CONSEQUENCE CHAINS

A training sample is a chain: one repository, one starting commit, and three consecutive requirements fed to the agent in order,

$$\mathcal{C} = \big(b,\ R_{1:3},\ T_{1:3},\ O_{1:3}\big), \tag{1}$$

where b is a real commit whose tests pass, R_k is a natural-language requirement, T_k is the acceptance test for step k, and O_k is the reference patch, taken from the repository's own later history by `git cherry-pick`. We mine 142 chains from 17 Python repositories, selecting histories that were extended repeatedly along one *axis*, a family of related requirements the project kept growing (Table 13, Appendix F). None of the 17 repositories appears in either evaluation benchmark (Appendix F). Reference patches establish a historical implementation path; the consequences of the agent's own implementation are observed through the rollout described below.

Every chain must satisfy four conditions. The requirements are **coupled**: in the repository's own history the patches $O_{1:3}$ modify the same code, since meeting one requirement well laid down structure that the next then built on. The coupling is **invisible to the specification**: R_k describes user-visible behavior and never names the design that later steps rely on. It is **invisible to the grading of the current step**: T_k asserts on behavior alone, so a correct implementation built on a different architecture passes equally. Finally, the coupling must be **supported by the continuation**: the audit identifies how later reference patches use the earlier structure and examines whether bypassing it would require redoing earlier work. These checks select histories with structural dependencies while leaving the agent free to choose an implementation. They do not by themselves measure the cost of alternative implementations.

Chains are built by a four-stage pipeline: a script sweeps the full commit history and recalls candidate structural decisions; an agent adjudicates the candidates; a second agent works backwards from the reference patches to the specifications and builds each survivor into a chain; a third audits the finished instance and repairs or discards it.

3.2 CHAIN EXECUTION AND CORRECTNESS

Each step is a complete engineering task: the agent reads the requirement, makes its own architectural choices, and implements them. The workspace is never reset, so each requirement is implemented on top of whatever the agent wrote before. At step k, the task provides a one-sentence repository description and the current requirement alongside the agent's working tree; acceptance tests, reference patches, and later requirements are withheld, and history after b is stripped from the workspace.

A rollout produces edits d_1, d_2, d_3 on the never-reset workspace. After step k the harness restores the test tree, applies T_k, and records the binary outcome a_k, which requires every fail-to-pass test of step k to pass and every pass-to-pass test established by steps 1 through k to keep passing. The chain outcome is the product $A = a_1 a_2 a_3 \in \{0, 1\}$: all three steps correct, or nothing. Earlier actions therefore receive supervision from later outcomes: if any step fails its acceptance tests, the correctness reward for the entire trajectory is zero. This directly rewards completing the whole chain successfully.

3.3 OFFLINE CALIBRATION OF JUDGING CRITERIA

Before policy training, a one-time offline study of executed agent rollouts forms the criteria used for judging. The judge first assesses a step using only the evidence available at that step. That assessment is then compared with a hindsight assessment informed by the rollout's executed continuation. Discrepancies inform the criteria subsequently used during policy training.

The continuation supplies evidence about what later steps accomplished and whether they reworked earlier code. Assessing how those outcomes relate to an earlier implementation remains a judgment; the execution result does not itself provide a ground-truth architectural score. Calibration concerns this relation between a step and its observed continuation. It does not observe requirements beyond the rollout. The resulting criteria are held fixed during policy training.

3.4 REWARD AND POLICY OPTIMIZATION

Two trajectories can pass every test while differing in how well their early changes support later implementation, for example in whether later steps must rework earlier code. A gives these successful trajectories the same reward. We refine it with a single hindsight score $J \in [0, 1]$, obtained after the chain finishes using the criteria of Section 3.3 and evidence from the executed chain.

For decisions before the final step, later execution within the chain provides evidence for the assessment. For the final step, the same criteria are applied without an observed continuation. This part of the assessment estimates support for further development; it is not a measurement of an executed future. Whether the resulting reward improves performance beyond the training horizon is tested in Sections 4.2 and 4.3.

The reward is paid once, at the end of the episode:

$$R \;=\; A \;+\; \lambda\, A\, J, \qquad \lambda = 0.4. \tag{2}$$

Equation 2 is a gate followed by a refinement, not a weighted sum. When $A = 0$ a hindsight score buys nothing, so the achievable reward is $\{0\} \cup [1, 1{+}\lambda]$ with nothing in between. The reward first requires whole-chain correctness, then distinguishes successful trajectories by how well their early changes support later implementation. A higher judge score cannot compensate for a failed chain. The gate also bounds what a miscalibrated judge can cost, since its score distorts the reward by at most λ and can never make an incorrect chain outperform a correct one. Since chain rollouts take hours, we optimize with asynchronous PPO and a learned critic (Schulman et al., 2017; Hou et al., 2026).

4 EXPERIMENTS

We test whether supervision from downstream execution improves agents' ability to implement successive requirements on their own accumulated code. We evaluate agents trained on three-step consequence chains on ChainSWE and SlopCodeBench, examining complete-chain success, continued implementation, and gains beyond the training horizon. Reference-reset comparisons and continuation after successful prefixes further characterize these gains. Finally, we ablate the judge term to assess its contribution beyond whole-chain correctness.

4.1 SETUP AND METRICS

Models. We train MINICPM5-2B, QWEN3.6-35B-A3B, and QWEN3.8-27B with the recipe of Section 3.4 (hyperparameters in Appendix C). The reward judge is a frozen copy of the base model for QWEN3.8-27B and a frontier model for the other two models; swapping between the two judges leaves the ordering of trajectory scores nearly unchanged (Appendix D), so this difference does not alter the reward signal. Each base model and its trained counterpart use the same harness, prompts, and turn budget (Appendix B). Unless stated as relative changes, differences are in percentage points and are computed before rounding.

ChainSWE (Jin et al., 2026) requires an agent to fix successive bugs in one repository, retaining its earlier patches without seeing future bugs in advance. We denote this sequential protocol with memory by SEQ+MEM. Our primary metric is *complete-chain success rate*, which requires every bug in a chain to be resolved; per-bug resolution is reported alongside it. We retain the fixed denominators of 100 chains and 304 bugs; 97 of the 100 chains contain exactly three bugs, and all per-position analyses use these 97 three-step chains. In ORACLE mode, each bug starts from the reference fixes to preceding bugs, providing a comparison between reference code and the agent's own accumulated code.

SlopCodeBench (Orlanski et al., 2026) comprises 36 programs with three to eight successive requirements, or *checkpoints*, totaling 196. The agent extends its own implementation at each checkpoint. *Core* covers the current requirement's core tests, *isolated* its full test set, and *strict* additionally includes tests from earlier checkpoints. A *solve rate* counts checkpoints passing every test in scope; a *pass rate* aggregates the fraction of tests passed with equal weight per checkpoint. These metrics capture complete implementation and partial test success, respectively.

Table 1: ChainSWE: (a) Seq+Mem resolution rates, with blue gains over Base in percentage points; (b) Oracle minus Seq+Mem gaps (lower is better). Bold marks Base/Ours pair bests, including ties. Oracle rates are in Table 12.

(a) Seq+Mem resolution rates

	Chain (%) ↑		Per-bug (%) ↑	
Model	Base	Ours	Base	Ours
MINICPM5-2B	3.0	**9.0** +6.0	27.3	**31.6** +4.3
QWEN3.6-35B-A3B	13.0	**17.0** +4.0	40.5	**41.8** +1.3
QWEN3.8-27B	14.0	**18.0** +4.0	43.4	**45.7** +2.3
Claude Opus 5		28.0		56.2
Claude Opus 4.6		14.0		42.4

(b) Oracle − Seq+Mem gap ↓

	Chain gap (pp) ↓		Per-bug gap (pp) ↓	
Model	Base	Ours	Base	Ours
MINICPM5-2B	4.0	**1.0**	10.2	**8.2**
QWEN3.6-35B-A3B	5.0	**0.0**	10.2	**7.9**
QWEN3.8-27B	12.0	**8.0**	12.2	**11.2**
Claude Opus 5		3.0		5.6
Claude Opus 4.6		7.0		10.5

Table 2: SlopCodeBench results; bold marks base/ours pair bests, including ties. Frontier results are from the just-solve runs in Table 8 of Orlanski et al. (2026), which omit pass rates; the full comparison is in Appendix A.

		Solved (%)			Pass rate		
Model		Core	Isolated	Strict	Core	Isolated	Strict
MINICPM5-2B	base	3.1	**1.0**	**1.0**	.176	.194	.239
	ours	**5.6**	**1.0**	0.5	**.182**	**.219**	**.252**
QWEN3.6-35B-A3B	base	32.7	6.6	4.1	.543	.525	.575
	ours	**37.8**	**11.7**	**7.1**	**.601**	**.584**	**.628**
QWEN3.8-27B	base	57.1	22.4	8.2	.803	**.771**	.790
	ours	**60.2**	**24.5**	**12.2**	**.807**	.766	**.800**
Claude Opus 4.6		67.3	20.9	9.7	—	—	—
GPT-5.5		66.8	28.1	14.8	—	—	—

4.2 PERFORMANCE ON SUCCESSIVE REQUIREMENTS

Training improves complete-chain success. Table 1 shows increases from 3%, 13%, and 14% to 9%, 17%, and 18% for the 2B, 35B, and 27B models, respectively: relative gains of 200.0%, 30.8%, and 28.6%. Per-bug resolution rises by 4.3, 1.3, and 2.3 points. Complete-chain success requires successive repairs to work together on the same evolving codebase, directly measuring the agent's ability to complete the entire sequence. Under reference resets, complete-chain success moves from 7% to 10% for MINICPM5-2B, from 18% to 17% for QWEN3.6-35B-A3B, and stays unchanged for QWEN3.8-27B. The per-bug ORACLE–SEQ+MEM gaps narrow from 10.2, 10.2, and 12.2 to 8.2, 7.9, and 11.2 points. These comparisons locate the complete-chain gains in the setting where agents build on their own changes.

Training helps agents extend their own earlier code. On SlopCodeBench (Table 2), core solve rates rise from 3.1%, 32.7%, and 57.1% to 5.6%, 37.8%, and 60.2%, adding 5, 10, and 6 fully solved checkpoints. Strict pass rates also rise, from .239, .575, and .790 to .252, .628, and .800. For QWEN3.6-35B-A3B, isolated solve rate increases from 6.6% to 11.7%, and strict solve rate from 4.1% to 7.1%: 14 checkpoints pass all strict tests, up from 8. For QWEN3.8-27B, isolated solve rate increases from 22.4% to 24.5%, and strict solve rate from 8.2% to 12.2%: 24 checkpoints pass all

Table 3: Next-step success gaps (percentage points) after earlier tests pass on three-step chains. For each model variant and position, both modes are evaluated on the same set of chains whose earlier SEQ+MEM steps pass their tests. Gap is ORACLE minus SEQ+MEM success rate; lower is better, and a negative gap means SEQ+MEM outperforms ORACLE on this subset. Bold marks the lower gap between Base and Ours for each model and step, including ties.

	Step 2 gap ↓		Step 3 gap ↓	
Model	Base	Ours	Base	Ours
MINICPM5-2B	+7.5	**+2.3**	+5.0	**+4.3**
QWEN3.6-35B-A3B	+11.5	**+3.6**	−3.6	**−6.1**
QWEN3.8-27B	+11.9	**+11.7**	+12.5	**+2.9**

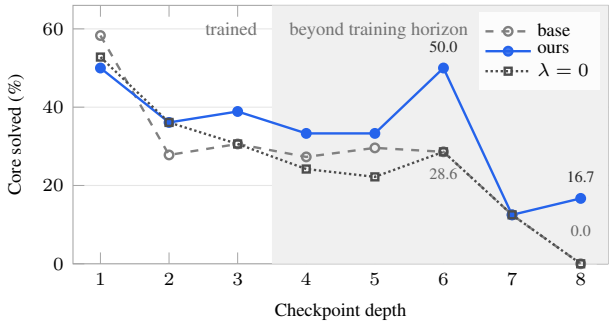


Figure 3: SlopCodeBench core solve rate by depth for QWEN3.6-35B-A3B. Shading marks depths beyond the three-step training horizon; $\lambda = 0$ uses correctness only. Counts and pass rates are in Table 10.

strict tests, up from 16. Core solve rate captures complete implementation of the new requirement; strict pass rate captures joint test performance on new and previously established functionality.

Training improves next-step success after earlier tests pass. Passing the tests for earlier requirements does not ensure success on the next requirement. We therefore compare continuation from the agent's own test-passing prefix with continuation from reference fixes, using the same model in both modes. For each model variant and position in a three-step chain, we select chains whose preceding SEQ+MEM steps all pass, and compare next-step success under both modes on that same set. Step 2 requires step 1 to pass; step 3 requires both preceding steps to pass. The conditional gap is the ORACLE success rate minus the SEQ+MEM success rate. Definitions and counts are in Appendix E.1.

After a successful first step, QWEN3.6-35B-A3B's next-step success rate rises from 53.8% to 58.9%, while its conditional gap falls from 11.5 to 3.6 points (Table 3). After two successful steps, QWEN3.8-27B's next-step success rate rises from 43.8% to 52.9%, and its gap falls from 12.5 to 2.9 points. MINICPM5-2B's step-3 conditional success rate also increases, from 15.0% to 39.1%, and its step-2 gap falls from 7.5 to 2.3 points. These results show improved next-step success even when earlier requirements already pass their tests. The eligible sets are selected separately for Base and Ours; Appendix E.1 also reports the comparison on their intersection.

Training gains extend beyond the three-step horizon. We then examine whether the gains persist beyond the three successive requirements used in training. SlopCodeBench sequences extend to eight checkpoints; Figure 3 reports core solve rates by depth for QWEN3.6-35B-A3B. Training increases core solve rate from 38.9% to 41.7% at depths 1–3 and from 25.0% to 33.0% at depths 4–8, gains of 2.8 and 8.0 points, respectively. Beyond the training horizon, 29 of 88 checkpoints pass every core test, compared with 22 for the base model, and core pass rate rises from .487 to .559.

The trained model matches or exceeds the base model's core solve rate at each of depths 4–8: depth 6 rises from 28.6% to 50.0%, and depth 8 from 0.0% to 16.7%. Full counts and pass rates are in Appendix E.3. Thus, three-step training improves success on later requirements implemented on the agent's own earlier code, with gains extending beyond the training horizon.

4.3 JUDGE REWARD AND SINGLE-STEP TRAINING COMPARISON

Table 4: Judge ablation and single-step SWE training on QWEN3.6-35B-A3B. Both chain-trained arms use the same training iteration; the single-step arm uses different data and reward. ChainSWE uses SEQ+MEM. Bold marks column bests, including ties.

	ChainSWE (%)		SlopCodeBench core solved (%)			Core pass
Setup	Per-bug	Chain	All	1–3	4–8	
Base model, untrained	40.5	13.0	32.7	38.9	25.0	.543
$R = A \quad (\lambda = 0)$	40.5	12.0	31.6	39.8	21.6	.533
$R = A + \lambda AJ$ (ours)	**41.8**	**17.0**	**37.8**	**41.7**	**33.0**	**.601**
Single-step + correctness	38.5	10.0	35.2	**41.7**	27.3	.599

We ablate the judge term on QWEN3.6-35B-A3B by setting $\lambda = 0$ in Equation 2, leaving whole-chain correctness $R = A$. Both arms use the same chains, training budget, and other settings, and are evaluated at the same training iteration. Adding the judge term raises SlopCodeBench core solve rate from 31.6% to 37.8% and ChainSWE complete-chain success from 12.0% to 17.0% (Table 4).

The judge's contribution is larger beyond the training horizon (Figure 3). At depths 4–8, correctness-only training reaches 21.6%, below the base model's 25.0%, while the full reward reaches 33.0%: an 11.4-point difference, versus 1.9 points at depths 1–3. Whole-chain correctness rewards all test-passing trajectories equally; the judge further assesses their implementations using the executed continuation. With chains and budget fixed, these results support the judge term's contribution to performance at greater depths. Appendix D examines sensitivity to the judge model.

To examine transfer from single-step SWE training to successive requirements, we also evaluate a model trained from the same base with the same training hyperparameters, but different single-step SWE data and a correctness-only reward. Relative to the base model, this training raises SlopCodeBench core solve rate from 32.7% to 35.2% overall, including an increase from 25.0% to 27.3% at depths 4–8. It matches our SlopCodeBench core solve rate at depths 1–3 (41.7%) and nearly matches core pass rate (.599 versus .601), but reaches 27.3% versus our 33.0% at depths 4–8 and 10.0% versus 17.0% complete-chain success on ChainSWE. In this experiment, single-step SWE training achieves similar shallow solve rates and average test pass rates to our method, but lower success rates at deeper checkpoints and on complete chains.

5 DISCUSSION AND CONCLUSION

We study how agents can implement successive requirements on their own accumulated code without knowing future requirements in advance. We construct consequence chains from real development histories and train agents to implement successive requirements on their own accumulated changes. The reward requires whole-chain correctness, then uses hindsight judgment grounded in downstream execution to distinguish successful implementations. This improves complete-chain success on ChainSWE across three models. On SlopCodeBench, the 35B model's checkpoint-level gains extend beyond the three-step training horizon; correctness-only training reproduces neither these gains nor the improvement in complete-chain success. These results support using downstream execution as supervision for continued software development.

The broader implication is to make a change's effects on subsequent work part of the training objective. Tests for the current requirement establish immediate correctness; subsequent execution provides evidence of whether later requirements can be completed on the resulting code. This perspective also suggests a direction for agents that continually modify their own software systems: changes to their code and tools should be evaluated by how they affect subsequent tasks. Whether training on consequence chains can support such sustained self-improvement remains an open research question.

ETHICS STATEMENT

This work trains and evaluates coding agents on code from publicly available open-source repositories (Appendix F) and on published benchmarks, in accordance with the repositories' open-source licenses. It involves no human subjects, user data, or personally identifiable information. The method aims to make software agents more reliable over long-horizon development; we do not foresee risks beyond those common to research on code-generation models.

REPRODUCIBILITY STATEMENT

We will release the training and evaluation code, the mined consequence chains, and the judging rubric upon acceptance. The paper records the chain construction pipeline (Section 3), the reward definition (Section 3.4), training hyperparameters and the judge rubric (Appendix C), the evaluation harness, prompts, and sampling settings (Appendix B), and the source repositories of every training chain (Appendix F).

AI USE STATEMENT

Generating synthetic datasets. The training data of this work is constructed by language-model agents as part of the method itself: agents selected development histories, derived requirement specifications, and built and audited the consequence chains (Section 3). Language models also drafted the judging rubrics and scored completed training trajectories. The authors audited the resulting chains for overlap with the evaluation benchmarks (Appendix F).

Drafting and polishing. We used Claude (Anthropic) to draft initial versions of some sections and to edit the authors' text for clarity throughout. All numbers, claims, and citations in AI-drafted text were verified by the authors against experiment logs and original sources.

Retrieval and discovery. We used language models to search for and screen related work; every entry in the bibliography was checked by the authors against its published version.

Implementation and analysis. AI coding assistants helped implement the experiment and evaluation code and assisted in analyzing experimental results. The authors reviewed all AI-written code and verified the reported numbers against the underlying experiment logs.

We take responsibility for the final content of this work, including all text, claims, and artifacts produced with AI assistance.

REFERENCES

Yanxi Chen, Weijie Shi, Boyi Hu, Zeyue Zhang, Zhiwei Wang, Yuexiang Xie, Yaliang Li, Bolin Ding, and Jingren Zhou. Connect the dots: Training LLMs for long-lifecycle agents with cross-domain generalization via Reinforcement Learning. *arXiv preprint arXiv:2606.20002*, 2026.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. SWE-Milestone: Evaluating AI agents on continuous software evolution. *arXiv preprint arXiv:2603.13428*, 2026.

Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. SWE-Dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975*, 2025.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

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.

Zhenyu Hou, Yujiang Li, Jie Tang, and Yuxiao Dong. Single-rollout asynchronous optimization for agentic reinforcement learning. *arXiv preprint arXiv:2607.07508*, 2026.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Qirui Jin, Lingching Tung, Kenan Li, Qiyang Shi, Yushi She, Huanzhong Jia, Harrison Zhao, Kejing Xia, Zhenbang Du, Jiaxin Pei, Zhenyu Zhang, Zhen Qi, Yuyan Duan, Wenke Lee, and Zijian Jin. ChainSWE: Benchmarking coding agents on multi-bug software maintenance. *arXiv preprint arXiv:2607.02606*, 2026.

Mo Li, L. H. Xu, Qitai Tan, Ting Cao, and Yunxin Liu. Learning to commit: Generating organic pull requests via online repository memory. *arXiv preprint arXiv:2603.26664*, 2026.

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

Maxime Robeyns and Laurence Aitchison. Improving LLM-generated code quality with GRPO. *arXiv preprint arXiv:2506.02211*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. EvoCode-Bench: Evaluating coding agents in multi-turn iterative interactions. *arXiv preprint arXiv:2605.24110*, 2026.

Vlad Sobal, Shuo Yang, Yuting Zhang, Wei Xia, and Stefano Soatto. StaminaBench: Stress-testing coding agents over 100 interaction turns. *arXiv preprint arXiv:2606.19613*, 2026.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement Learning for self-improving agent with skill library. *arXiv preprint arXiv:2512.17102*, 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. *arXiv preprint arXiv:2502.18449*, 2025a.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play SWE-RL. *arXiv preprint arXiv:2512.18552*, 2025b.

Martin Weyssow, Aton Kamanda, Xin Zhou, and Houari Sahraoui. CodeUltraFeedback: An LLM-as-a-judge dataset for aligning large language models to coding preferences. *arXiv preprint arXiv:2403.09032*, 2024.

Zhiyuan Yao, Yuxin Chen, Zhengxi Lu, Zishan Xu, Yueqing Sun, Yifu Guo, Yuquan Lu, Zhengzhou Cai, Kangning Zhang, Zhuowen Han, Zi-Han Wang, Ziang Ye, Qi Gu, Xunliang Cai, Weiwen Liu, and Yongliang Shen. SkillRise: Agentic reinforcement learning for cross-task skill evolution. *arXiv preprint arXiv:2607.26784*, 2026.

A THE FIFTEEN FRONTIER AGENTS ON SLOPCODEBENCH

B EVALUATION CONFIGURATIONS

Harness. All benchmarks run on the same agent harness: the agent is CLAUDE CODE in non-interactive mode, working inside a sandboxed container that holds the task environment, with the open models served by SGLANG and the Claude Opus models by their provider's API. On the two multi-step benchmarks a single session spans the whole chain: the first step opens the session and every later step resumes it, issuing the new requirement as the next user message, so the full

Table 5: Core, isolated, and strict solve rates for the 15 frontier agents of Orlanski et al. (2026), from the just-solve runs of their Table 8, sorted by core solve rate. Opus 4.6 leads the core column yet keeps only 9.7% of steps once every test from its earlier checkpoints must keep passing, a rate GLM 5.1 matches from a core rate 27 points lower. Bold marks the best value in each column.

Agent	Core	Iso.	Strict	Agent	Core	Iso.	Strict
Opus 4.6	**67.3**	20.9	9.7	GPT 5.4 Mini	53.1	13.8	5.1
GPT 5.5	66.8	**28.1**	**14.8**	Composer 2	52.6	16.3	6.1
Opus 4.7	65.8	20.9	8.2	Kimi K2.6	51.0	18.9	10.7
GPT 5.4	62.8	23.5	10.7	GLM 5.1	40.3	13.8	9.7
GPT 5.3 Codex	60.7	26.0	11.2	Kimi K2.5	39.8	9.7	4.6
Opus 4.5	57.7	17.3	9.2	GPT 5.3 Spark	29.1	8.2	3.1
Sonnet 4.6	57.7	16.8	7.1	MiniMax M2.7	28.1	4.1	2.0
GPT 5.2 Codex	56.1	21.9	9.7				

conversation history carries over (the SEQ+MEM protocol of Table 1). The context window is the model's native length, 262,144 tokens for the two QWEN models and 131,072 for MINICPM5-2B; when a session reaches 80% of the window (70% for MINICPM5-2B) the harness compacts the older history.

Sampling. The open models sample at temperature 1.0, top-p 0.95, and top-k 20; each model call generates at most 32,768 tokens.

ChainSWE. Each bug allows up to 300 agent turns. The agent sandbox can reach only the model-serving gateway; the scoring sandbox additionally reaches the Python package index. The first user message of every bug is the fixed header below followed by the bug's problem statement, where `<repo>` stands for the absolute path of the repository inside the container:

```
You are working in a repository located at <repo>.

The task below describes a bug to fix or a change to make. Modify the
repository in place to resolve it. A hidden test suite will grade your
work after this step; do not delete or weaken existing tests.

---
```

SlopCodeBench. Each checkpoint allows up to 100 agent turns. We use the neutral prompt, which achieved the highest strict solve rate among the three variants evaluated by Orlanski et al. (2026). The first user message of a problem prepends the preamble below to the first checkpoint's specification; every later checkpoint sends only its new specification:

```
Implement a program that 100% solves the specification. That is all you need to do.
Your task is:
```

C TRAINING DETAILS

Table 6 lists the configuration of the QWEN3.8-27B training run. Training uses asynchronous PPO with a learned critic (Section 3.4): rollouts stream in continuously, and a trajectory may lag the policy that consumes it by at most two optimizer iterations. Each iteration draws eight chains from the training set, sampling each chain three, two, or one time according to its difficulty tier; the critic trains alone for the first ten iterations. Trajectories that hit the generation-length cap or degenerate into repetition are excluded from the loss. The judge runs as a separately served frozen replica and reads the full per-step record of a trajectory, workspace snapshots and test results included, scoring each successful episode once. Checkpoints are saved every 30 iterations.

Judge rubric. The judge scores each successful episode against the following checklist of seven properties of the solution, reproduced verbatim from the training configuration.

- implements the requirement's stated semantics, not just the visible test contour: cases the instruction names must work even where no test pins them

Table 6: Hyperparameters of the QWEN3.8-27B training run.

Rollout iterations	300
Chains per iteration	8
Rollouts per chain	3 / 2 / 1 by difficulty tier
Maximum trajectory staleness	2 iterations
Actor learning rate	6×10^{-7}, 10-iteration warmup
Critic learning rate	3×10^{-6}, 20-iteration warmup
Critic-only warmup	10 iterations
Critic epochs per iteration	2
GAE λ	1.0, normalized advantages
Policy-ratio clip range	$[0.8, 3.0]$
Judge weight λ (Eq. 2)	0.4
Context window / per-call generation cap	262,144 / 32,768 tokens
Agent turns per step during training	≤ 100
History compaction	at 80% of the window
Hardware	96 H100 (4 actor, 4 critic, 4 rollout nodes)
Training parallelism	TP 4 / PP 2 / CP 4; rollout TP 4
Random seed	42

- fixes the root cause rather than suppressing symptoms
- builds only what the current request needs: no options, hooks, or generality justified only by guessed future requirements
- later requirements land as small, local, mostly-additive diffs attached to earlier abstractions; scattering one requirement's edits across many files signals entropy
- each piece of knowledge has one authoritative home: no copy-paste variants or parallel re-implementations of existing repo facilities
- when a requirement no longer fits, the structure is reshaped to absorb it rather than patched over—and reshaping stays proportionate to what requirements actually showed
- no dead code, unused options, or leftover scaffolding; the diff is minimal and targeted

D SENSITIVITY OF THE REWARD TO THE JUDGE MODEL

The reward judge is a frozen copy of the base model for QWEN3.8-27B and a frontier model for the other two runs (Section 4.1). This appendix tests whether that choice matters by re-scoring a fixed set of training trajectories under both judges.

Protocol. We draw 32 trajectories with a passing first step from a QWEN3.8-27B training run over the 142 training chains: 16 whose remaining steps all pass their acceptance tests and 16 with a later failure, selected deterministically with repositories rotated. Both judges receive identical evidence and the rubric of Appendix C, and score each trajectory under two evidence scopes: the full per-step chain record used during training, and per-step views scored one step at a time and averaged over the trajectory.

Table 7: Agreement between the frozen QWEN3.8-27B judge and the frontier judge on 32 training trajectories, with identical rubric and evidence. Direction agreement is the fraction of trajectory pairs, among those the frontier judge separates by at least 0.10, that both judges order the same way.

Evidence scope	Pearson	Spearman	MAE	Direction agreement
Whole chain	.92	.89	.08	93% (309/331)
Per step, averaged	.95	.94	.04	99% (294/296)

Swapping the judge does not change the reward signal. Table 7 compares the two judges under identical rubric and evidence. Their scores correlate at 0.92 (whole chain) and 0.95 (per step), and over trajectory pairs that the frontier judge separates by at least 0.10, the two judges order 93%

Table 8: Conditional success on three-step chains. Success rates are percentages, with successful-chain counts in parentheses; n is the eligible-chain count. The last column gives the counts succeeding only under ORACLE and only under SEQ+MEM, respectively. Gaps are in percentage points.

Model	Variant	Step	n	SEQ+MEM	ORACLE	Gap	O-only/S-only
MINICPM5-2B	base	2	40	50.0 (20)	57.5 (23)	+7.5	3/0
	base	3	20	15.0 (3)	20.0 (4)	+5.0	3/2
	ours	2	44	52.3 (23)	54.5 (24)	+2.3	2/1
	ours	3	23	39.1 (9)	43.5 (10)	+4.3	3/2
QWEN3.6-35B-A3B	base	2	52	53.8 (28)	65.4 (34)	+11.5	6/0
	base	3	28	46.4 (13)	42.9 (12)	−3.6	2/3
	ours	2	56	58.9 (33)	62.5 (35)	+3.6	6/4
	ours	3	33	51.5 (17)	45.5 (15)	−6.1	1/3
QWEN3.8-27B	base	2	59	54.2 (32)	66.1 (39)	+11.9	8/1
	base	3	32	43.8 (14)	56.2 (18)	+12.5	4/0
	ours	2	60	56.7 (34)	68.3 (41)	+11.7	9/2
	ours	3	34	52.9 (18)	55.9 (19)	+2.9	3/2

and 99% of pairs identically. The residual difference is a level shift: the frozen judge scores $+0.06$ higher on average (Wilcoxon $p = 0.002$). Since advantages are normalized (Table 6), a uniform shift leaves the update unchanged; what reaches the policy is the ordering, and the ordering is preserved. The reward of the two frontier-judged runs is therefore not a channel for frontier-model knowledge: replacing the frontier judge with the frozen base copy reproduces it almost exactly.

With 32 trajectories scored once, these comparisons bound the judge's role rather than measure it precisely.

E CONDITIONAL CONTINUATION AND DEPTH ANALYSIS

E.1 SUCCESS CONDITIONAL ON EARLIER STEPS PASSING

We restrict this analysis to three-step ChainSWE chains. Let $S_{c,k}$ and $O_{c,k}$ indicate whether step k of chain c is resolved under SEQ+MEM and ORACLE, respectively. Let $V_{c,k} = 1$ when both modes have a recorded result for that step. For $k \in \{2, 3\}$, the eligible set is

$$\mathcal{E}_k = \left\{ c \in \mathcal{C}_3 : \prod_{j=1}^{k-1} S_{c,j} = 1,\ V_{c,k} = 1 \right\}, \tag{3}$$

where $\mathcal{C}_3$ denotes the three-step chains. We compare both modes on this same set, defining their conditional success rates and gap as

$$p_k^{\text{seq}} = \frac{\sum_{c \in \mathcal{E}_k} S_{c,k}}{|\mathcal{E}_k|}, \qquad p_k^{\text{orc}} = \frac{\sum_{c \in \mathcal{E}_k} O_{c,k}}{|\mathcal{E}_k|}, \qquad G_k = 100\,(p_k^{\text{orc}} - p_k^{\text{seq}})\,. \tag{4}$$

Step 2 requires the first SEQ+MEM step to pass its tests; step 3 requires both preceding steps to pass. The ORACLE outcome concerns the same target bug, starting from reference fixes to preceding bugs. A positive gap indicates higher conditional success under ORACLE; a negative gap indicates higher conditional success under SEQ+MEM. Comparisons are paired by chain and position.

Table 8 reports each position separately, preserving its prefix condition and denominator. Pooling the two positions would yield $(n_2 G_2 + n_3 G_3)/(n_2 + n_3)$, where $n_k = |\mathcal{E}_k|$: the weights would depend on how many chains each model successfully carries to the next position. Reporting G_2 and G_3 separately keeps the target position and conditioning event explicit. The conditional gap characterizes next-step performance under the specified prefix condition.

Table 9: Conditional gaps on the intersection of base and trained eligible sets. S/O gives the successful-chain counts under SEQ+MEM and ORACLE. Gaps and their reductions are in percentage points, computed from counts before rounding. Gap is ORACLE minus SEQ+MEM success rate; lower is better. Bold marks the lower gap between Base and Ours for each model and step, including ties. A positive reduction means the gap narrows after training.

			Base		Ours		
Model	Step	n	S/O	Gap ↓	S/O	Gap ↓	Reduction ↑
MINICPM5-2B	2	37	19/22	+8.1	20/20	**0.0**	+8.1
	3	18	3/4	**+5.6**	4/6	+11.1	−5.6
QWEN3.6-35B-A3B	2	47	27/32	+10.6	28/31	**+6.4**	+4.3
	3	26	12/11	−3.8	13/11	**−7.7**	+3.8
QWEN3.8-27B	2	57	30/37	+12.3	32/38	**+10.5**	+1.8
	3	28	12/15	+10.7	15/16	**+3.6**	+7.1

Table 10: SlopCodeBench results for QWEN3.6-35B-A3B by checkpoint depth. Core solve rates are percentages, with solved-checkpoint counts in parentheses; core pass rates give test fractions on a 0–1 scale. The correctness-only arm ($\lambda = 0$) uses the same training chains and budget as the full reward. Depths 1–3 match the training horizon; depths 4–8 lie beyond it.

		Core solved (%)			Core pass rate		
Depth	Checkpoints	Base	$\lambda = 0$	Ours	Base	$\lambda = 0$	Ours
1	36	58.3 (21)	52.8 (19)	50.0 (18)	0.738	0.714	0.710
2	36	27.8 (10)	36.1 (13)	36.1 (13)	0.567	0.563	0.615
3	36	30.6 (11)	30.6 (11)	38.9 (14)	0.462	0.476	0.582
4	33	27.3 (9)	24.2 (8)	33.3 (11)	0.475	0.477	0.554
5	27	29.6 (8)	22.2 (6)	33.3 (9)	0.486	0.482	0.570
6	14	28.6 (4)	28.6 (4)	50.0 (7)	0.577	0.583	0.658
7	8	12.5 (1)	12.5 (1)	12.5 (1)	0.401	0.308	0.420
8	6	0.0 (0)	0.0 (0)	16.7 (1)	0.457	0.342	0.483
1–3	108	38.9 (42)	39.8 (43)	41.7 (45)	0.589	0.584	0.636
4–8	88	25.0 (22)	21.6 (19)	33.0 (29)	0.487	0.471	0.559
5–8	55	23.6 (13)	20.0 (11)	32.7 (18)	0.494	0.467	0.561
All	196	32.7 (64)	31.6 (62)	37.8 (74)	0.543	0.533	0.601

E.2 COMPARISON ON COMMON ELIGIBLE CHAINS

To hold chain identity fixed across base and trained variants, we also use $\mathcal{E}_k^* = \mathcal{E}_k^{\text{base}} \cap \mathcal{E}_k^{\text{ours}}$. On this intersection, the reduction in the conditional gap is

$$D_k = \frac{100}{|\mathcal{E}_k^*|} \sum_{c \in \mathcal{E}_k^*} \left[(O_{c,k}^{\text{base}} - S_{c,k}^{\text{base}}) - (O_{c,k}^{\text{ours}} - S_{c,k}^{\text{ours}}) \right]. \tag{5}$$

A positive D_k indicates a smaller ORACLE advantage after training on the same chains at the same position. Table 9 complements the variant-specific eligible sets. The step-2 gap for QWEN3.6-35B-A3B decreases from 10.6 to 6.4 points, and the step-3 gap for QWEN3.8-27B from 10.7 to 3.6 points, retaining the direction of both comparisons in Table 3.

E.3 SLOPCODEBENCH RESULTS BY DEPTH

Table 10 gives the depth decomposition underlying Figure 3. The first three checkpoints match the training horizon; checkpoints 4–8 lie beyond it. Each depth uses the number of programs containing that checkpoint as its denominator.

Table 11: Per-bug resolution (%) by position on the 97 three-step ChainSWE chains. Parentheses give successful-bug counts; every column uses the same denominator of 97.

		SEQ+MEM			ORACLE		
Model	Variant	Step 1	Step 2	Step 3	Step 1	Step 2	Step 3
MINICPM5-2B	base	42.3 (41)	29.9 (29)	11.3 (11)	39.2 (38)	43.3 (42)	33.0 (32)
	ours	45.4 (44)	32.0 (31)	19.6 (19)	45.4 (44)	42.3 (41)	35.1 (34)
QWEN3.6-35B-A3B	base	53.6 (52)	39.2 (38)	30.9 (30)	57.7 (56)	51.5 (50)	46.4 (45)
	ours	57.7 (56)	40.2 (39)	29.9 (29)	57.7 (56)	50.5 (49)	45.4 (44)
QWEN3.8-27B	base	60.8 (59)	42.3 (41)	29.9 (29)	62.9 (61)	56.7 (55)	51.5 (50)
	ours	61.9 (60)	42.3 (41)	36.1 (35)	62.9 (61)	59.8 (58)	52.6 (51)
Claude Opus 4.6		59.8 (58)	41.2 (40)	28.9 (28)	60.8 (59)	57.7 (56)	45.4 (44)
Claude Opus 5		70.1 (68)	54.6 (53)	48.5 (47)	66.0 (64)	63.9 (62)	60.8 (59)

Table 12: Oracle resolution rates (%) on ChainSWE, using 100 chains and 304 bugs. Each bug starts from reference fixes to preceding bugs. Reference agents use the same harness.

	Chain (%)		Per-bug (%)	
Model	Base	Ours	Base	Ours
MINICPM5-2B	7.0	10.0	37.5	39.8
QWEN3.6-35B-A3B	18.0	17.0	50.7	49.7
QWEN3.8-27B	26.0	26.0	55.6	56.9
Reference agents (same harness)				
Claude Opus 5	31.0		61.8	
Claude Opus 4.6	21.0		53.0	

E.4 CHAINSWE RESULTS BY POSITION

Table 11 reports per-bug resolution at each position over the 97 three-step chains in the benchmark, without conditioning on prefix success. The corresponding conditional results appear in Tables 3 and 8.

E.5 CHAINSWE REFERENCE-RESET RESULTS

Table 12 reports the Oracle rates underlying the gaps in Table 1.

F TRAINING REPOSITORIES AND BENCHMARK INDEPENDENCE

Repositories. Table 13 lists the 17 repositories the 142 chains come from. Repositories were chosen for histories in which one axis is extended several times, not for domain; the set spans web frameworks, an ORM, a task queue, a data-analysis library, a graph library, a crawler, and an HTTP client.

Independence from the evaluation benchmarks. No repository of Table 13 appears in any benchmark we evaluate on. The 100 ChainSWE chains are drawn from 53 projects: sqlglot (12 chains), dataclasses-jsonschema (7), brutils-python, MONAI, pyccel, sqllineage (4 each), amaranth, docformatter, flake8-simplify, hypothesis, RDT, xarray (3 each), modin, moto, polyfactory, pydantic, seaborn, trimesh (2 each), and one chain each from aiohttp, briefcase, click, cloudpathlib, compliance-trestle, copier, datamodel-code-generator, dspy, faker, falcon, fonttools, freezegun, haystack, hdmf, hikari, instructlab, ipython, jsonpickle, keras, linkding, markdown, markdownify, meltano-sdk, pydatastructs, pydicom, pyfakefs, pynwb, pytask, python-semantic-release, reana-client, rich, sanic-ext, tablib, weasyprint, and woodwork. The closest name, `sanic-ext`, is a separate repository from the `sanic` we train on. SlopCodeBench consists of tasks written for that benchmark rather than mined from public repositories, so it shares no repository with training by construction.

Table 13: The 142 training chains by repository. All repositories are Python projects on GitHub.

Repository	GitHub	Chains
tortoise	tortoise/tortoise-orm	16
werkzeug	pallets/werkzeug	16
celery	celery/celery	14
datasette	simonw/datasette	14
networkx	networkx/networkx	12
pandas	pandas-dev/pandas	9
paramiko	paramiko/paramiko	8
sanic	sanic-org/sanic	8
scrapy	scrapy/scrapy	8
urllib3	urllib3/urllib3	8
coveragepy	nedbat/coveragepy	6
pint	hgrecco/pint	6
tornado	tornadoweb/tornado	6
marshmallow	marshmallow-code/marshmallow	4
trio	python-trio/trio	4
jinja	pallets/jinja	2
datalad	datalad/datalad	1
Total (17 repositories)		142

Pretraining exposure. ChainSWE is mined from public repositories, so its reference fixes likely appear in the pretraining corpora of every model we evaluate, the bases we train from as well as the Claude Opus models. This exposure cannot explain our comparisons. Every claim in Section 4 is a change relative to the same base model: a trained checkpoint shares its pretraining with its base exactly, and training touches only the repositories of Table 13, so whatever the base memorized is held fixed by construction. The SEQ+MEM states are further out of reach of any corpus: from the second bug onward the repository contains the agent's own accumulated patches rather than the human history the reference fixes were written against, so a memorized fix stops applying verbatim precisely where our primary metric measures. Verbatim recall would also predict near-ceiling ORACLE scores, which start from the states the reference fixes were written for; the strongest model we evaluate resolves 61.8% of bugs in that mode. Absolute numbers remain scores on public code and should be read as such, while SlopCodeBench carries no such exposure.