COPE: RECOVERING LONG-HORIZON CODING TASKS THROUGH COMPRESSION OF PULL-REQUEST EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Repository-level coding tasks are commonly derived from historical development artifacts such as issues and pull-requests (PRs). However, such artifact boundaries reflect development workflow rather than the semantic boundaries of functionality, such that the resulting tasks often capture only fragments of a feature's complete behavioral contract. We introduce **COPE** (**C**ompression **O**f **P**ull-request **E**volution), a framework that treats PR evolution history as evidence for task construction rather than as a sequence to reproduce. COPE first identifies semantically related, possibly non-contiguous PRs that shape the same functionality. It then recovers the behavioral specification revealed across their evolution and poses it as a single executable task at an earlier repository state. Each task thus compresses requirements that emerged over multiple development stages into a longer semantic horizon, aligning with semantic boundaries of true software functionality. Using COPE, we construct 24,000 long-horizon coding tasks spanning 14,868 repositories and obtain 4,478 execution-verified trajectories as a training pool. Under the same training-token budget, models trained with COPE supervision outperform those trained with single-PR and other long-horizon supervision on representative software engineering benchmarks. In particular, with only 384 trajectories, COPE PR-Tree yields relative improvements of 21.3% on SWE-Bench Verified and 33.3% on SWE-Bench Multilingual over the base model. The COPE tasks and code will be made publicly available.

1 INTRODUCTION

Coding agents have rapidly evolved from generating isolated code snippets to interacting with complete real-world repositories. This shift has enabled them to solve complex repository-level software-engineering tasks by inspecting large codebases, modifying multiple files, executing repository-specific tests, and iteratively revising implementations (Yang et al., 2024; Wang et al., 2025b). To evaluate and promote this progress, recent repository-level benchmarks and data-construction efforts have increasingly relied on executable tasks drawn from real software projects, where candidate solutions can be validated through repository-specific tests (Jimenez et al., 2024; Guo et al., 2026). However, most existing tasks are defined around software-development artifacts, such as issues, commits, pull-requests (PRs) and releases (Jimenez et al., 2024; Badertdinov et al., 2026; Le et al., 2026). These artifacts offer convenient task boundaries, along with natural-language descriptions, code changes, tests, and reproducible repository states. Nevertheless, they are shaped by development history and workflow, rather than by the semantic structure of the underlying engineering problem.

In real software development, the complete contract of a feature is often refined incrementally rather than specified when the feature is first implemented. For example, an initial PR may cover only the main execution path, while subsequent PRs address previously unspecified edge cases, clarify API semantics, repair error handling, restore compatibility, extend the feature to new contexts, or add tests that codify previously implicit behavior (Jiang et al., 2026; Deng et al., 2026). Consequently, changes that are historically separated may jointly define a single coherent functionality, while temporally adjacent changes may be semantically unrelated. Historical artifact boundaries therefore do not necessarily coincide with the semantic boundaries of a coding task.

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.

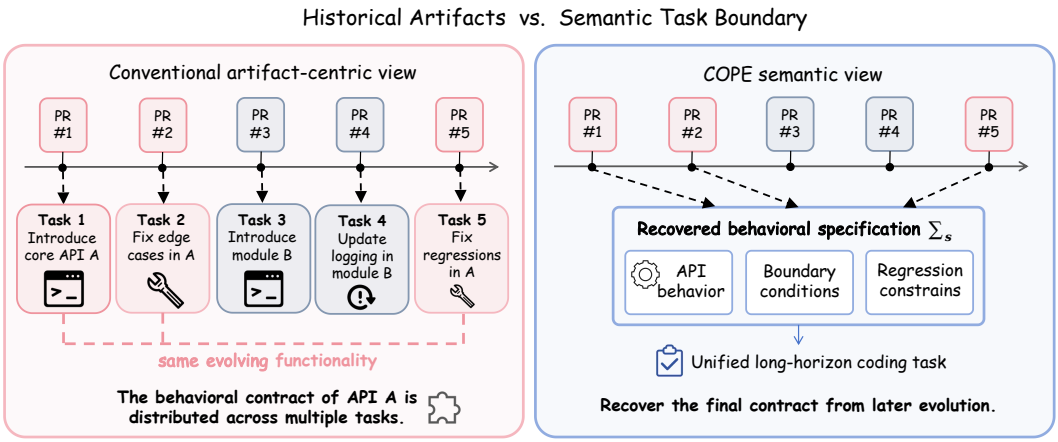


Figure 1: **Historical artifacts versus semantic task boundary.** Conventional task construction treats individual PRs as separate tasks, even when they jointly contribute to the same evolving functionality. COPE instead identifies semantically related, potentially non-contiguous PRs, consolidates the behavioral specification revealed across their evolution into a unified long-horizon coding task.

In response to this mismatch, recent work has moved beyond isolated development artifacts by grouping related changes across PR sequences, milestones, releases, or broader evolution histories (Jiang et al., 2026; Shastry et al., 2026; Deng et al., 2026; Le et al., 2026; Xu et al., 2026). These approaches broaden the temporal or functional scope of a task by defining it around a sequence of related historical changes and asking the agent to execute, aggregate, or reproduce them. However, reproducing or aggregating these changes does not necessarily recover the underlying functionality as a coherent coding task. To illustrate, historical sequences may include incidental changes such as refactoring, dependency updates, formatting edits, or unrelated bug fixes rather than the semantic structure of the functionality. As a result, such sequence can include changes that are irrelevant to the final behavior while leaving behavioral requirements implicit.

We therefore take a different perspective: instead of treating the evolution history as the target of the task, we use it as evidence for reconstructing the behavioral requirements of the underlying functionality. Particularly, later fixes, refinements, extensions, and compatibility changes often reveal requirements that were implicit or unavailable when the functionality was first introduced. This retrospective view allows us to construct a unified coding task from the requirements revealed across the evolution history, rather than reproduce the historical sequence itself. Figure 1 illustrates this distinction.

To this end, we propose Compression Of Pull-request Evolution (COPE), a novel framework for synthesizing long-horizon coding tasks from software evolution in a retrospective view. Given the PR history of a real repository, COPE first identifies *semantic evolution structures*: semantically related, potentially non-contiguous PRs that jointly shape the same functionality. These structures may form *chains*, capturing successive refinements along a single line of evolution, or *trees*, capturing a shared functional foundation that later evolves along multiple related branches. It then performs *retrospective specification recovery* to consolidate the behavioral constraints revealed across PR evolution, and *counterfactual synthesis* to realize the recovered specification as a unified executable task. In this way, COPE compresses requirements that historically emerged across multiple development stages into a single task with a longer semantic horizon. This formulation captures the behavioral requirements revealed across the evolution history while abstracting away from incidental implementation details and the original chronological order.

We instantiate COPE on real-world repositories, generate coding-agent rollout trajectories from the resulting tasks, and use them to train coding models. Under matched training-token budgets, models trained on 576 PR-chain or 384 PR-tree trajectories consistently outperform those trained on 1,063 single-PR trajectories across repository-level benchmarks; for instance, PR-tree supervision improves Qwen3-30B-A3B-Thinking-2507 by 3.2 points on SWE-Bench Verified (Jimenez et al., 2024) and 3.33 points on SWE-Bench Multilingual (Yang et al., 2026) over single-PR supervision.

Our contributions are threefold:

- Driven by the mismatch between historical artifact boundaries and the semantic boundaries of software functionality, we propose to formulate software evolution as retrospective evidence for recovering latent feature specifications.
- We introduce **Compression of Pull-Request Evolution (COPE)**, a framework that compresses PR evolution history by discovering semantically coherent structure to recover their accumulated behavioral specification.
- We evaluate rollout supervision derived from COPE-synthesized tasks for fine-tuning coding models and show that it consistently outperforms supervision derived from isolated PR tasks on repository-level software-engineering benchmarks.

2 Related Work

2.1 Coding Agents for Repository-Level Software Engineering

Coding agents have evolved from localized code generation into interactive software-engineering systems that operate directly over real-world repositories. Representative systems such as SWE-agent and OpenHands enable agents to navigate repositories, modify code, execute tests, and iteratively refine their solutions (Yang et al., 2024; Wang et al., 2025b). Recent work has improved repository-level agents through better repository understanding, search, localization, and solution exploration (Ouyang et al., 2025; Antoniades et al., 2025; Sohrabizadeh et al., 2025). Complementary efforts improve repository-level coding agents through dedicated model training, using supervision derived from issue resolution, agent trajectories, and synthetic interactions (Wang et al., 2025a; Xie et al., 2025; 2026; Shen et al., 2026). Together, these directions highlight the importance of both effective agent scaffolds and high-quality training supervision.

2.2 Software Engineering Benchmarks and Data Construction

Existing coding-agent benchmarks typically define tasks around units drawn from real software development. SWE-Bench and its variants focus on issue-level repository changes (Jimenez et al., 2024), while multilingual benchmarks such as Multi-SWE-bench, SWE-bench Multilingual, and SWE-PolyBench broaden evaluation across diverse languages and task types (Zan et al., 2025; Yang et al., 2026; Rashid et al., 2025). FeatureBench extends this setting to feature-level development (Zhou et al., 2026), whereas SWE-Bench Pro and DeepSWE target more challenging long-horizon repository-level tasks (Deng et al., 2025; Huang et al., 2026). SWE-EVO and Roadmap-Bench further consider broader version or roadmap level software evolution (Le et al., 2026; Xu et al., 2026).

Building on these benchmark formulations, recent work has increasingly automated the construction of executable repository-level supervision. SetUpAgent and SWE-rebench recover historically grounded tasks and evaluation pipelines from real repository histories (Vergopoulos et al., 2025; Badertdinov et al., 2026). SWE-Gym, R2E-Gym, SWE-smith, and SWE-Factory further scale executable supervision toward agent training and automated task generation (Pan et al., 2025; Jain et al., 2025; Yang et al., 2026; Guo et al., 2026). Complementary work such as Repo2Run and SWE-Universe focuses on scalable construction of reproducible and verifiable repository environments (Hu et al., 2025; Chen et al., 2026). Together, these efforts establish scalable foundations for constructing executable supervision from real repository histories.

2.3 Long-Horizon Software Evolution

Existing approaches to long-horizon software evolution can be broadly grouped into two paradigms. One line of work preserves intermediate development stages, repository states, and cross-task dependencies. daVinci-Agency collects sequential development and refinement trajectories along related PR chains, while the conversational setting of SWE-STEPS asks agents to process PRs sequentially in a persistent repository state (Jiang et al., 2026; Shastry et al., 2026). DeepCommit reorganizes fine-grained commits into functionally coherent milestone DAGs, which SWE-Milestone uses to evaluate agents over persistent development states (Deng et al., 2026). These settings expose challenges such as state inheritance, regression propagation, and error accumulation across stages.

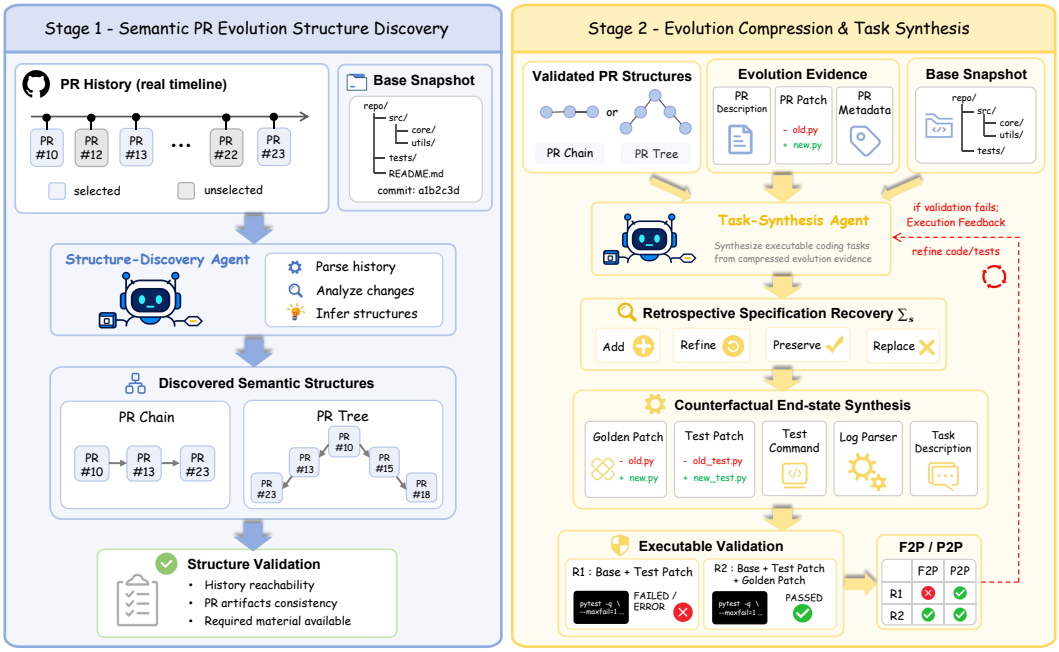


Figure 2: **Overview of COPE.** COPE first discovers semantically coherent PR evolution structures from repository history. It then compresses the validated evolution through retrospective specification recovery and counterfactual task synthesis into a unified executable task at an earlier repository state. Execution-based validation iteratively refines and verifies the synthesized task.

Another line of work organizes a broader evolution history into a single coding objective. The PRD setting of SWE-STEPS merges requirements from a PR sequence into a unified specification, while SWE-EVO and RoadmapBench formulate release or version level evolution as larger source-to-target tasks (Shastry et al., 2026; Le et al., 2026; Xu et al., 2026). These approaches typically begin from a predefined evolution scope, such as a development sequence, release interval, or version transition. COPE instead identifies semantically necessary, potentially non-contiguous PRs and uses their evolution as evidence to recover a coherent behavioral specification, which is then compressed into a unified executable task.

3 Compression of Pull-Request Evolution

We introduce **Compression Of Pull-request Evolution (COPE)**, a task-synthesis framework that converts distributed software evolution into a unified and executable coding task. Given a real repository history, COPE (1) identifies pull-requests (PRs) that jointly describe the evolution of a coherent functionality, (2) retrospectively recovers the behavioral specification revealed across these PRs, and (3) re-instantiates this specification at an earlier repository state with executable supervision. The workflow of COPE can be conceptually formulated as

$$S \longrightarrow \Sigma_S \longrightarrow \tau_S, \tag{1}$$

where S is a semantic PR evolution structure, Σ_S is its recovered specification, and τ_S is the resulting coding task. Figure 2 summarizes the pipeline of COPE.

3.1 Task Formulation

Let $\mathcal{H} = (p_1, \ldots, p_n)$ denote the pull-request history of a repository. Rather than defining a task using one PR or a predefined contiguous history window, COPE selects a *semantic evolution structure* $S \subseteq \mathcal{H}$, whose PRs jointly contribute to the same eventual behavioral contract. The selected PRs need not be temporally contiguous; unrelated changes may intervene, provided that each retained PR contributes a behaviorally relevant change to the same functionality.

We consider two forms of S. A PR chain $\mathcal{C} = (p_1, \ldots, p_k)$ captures a single line of semantic evolution, while a rooted PR tree $\mathcal{T} = (V, E, r)$ captures a shared behavioral foundation followed by multiple related branches. Thus, $S \in \{\mathcal{C}, \mathcal{T}\}$.

Each structure is associated with a code anchor B_S: the base commit of the first PR for a chain, or of the root PR for a tree. B_S represents the repository state before the selected evolution begins.

From the evidence associated with the selected PRs, COPE recovers an accumulated behavioral specification

$$\Sigma_S = G_{\text{spec}}(\mathcal{D}_S), \tag{2}$$

where $\mathcal{D}_S$ contains their textual descriptions, code changes, tests, and evolution relations.

The final COPE instance is $\tau_S = (x_S, \mathcal{O}_S)$ with public agent input $x_S = (B_S, I_S, q_S)$ and private supervision $\mathcal{O}_S = (\Delta_S^\star, \Delta_S^{\text{test}}, c_S, \pi_S, \mathcal{F}_S, \mathcal{P}_S)$.

Here I_S is the executable repository environment, q_S is the unified task description, $\Delta_S^\star$ and Δ_S^{test} are the golden code patch and test patch, c_S is the test command, π_S parses per-test outcomes, and $\mathcal{F}_S, \mathcal{P}_S$ are fail-to-pass (F2P) and pass-to-pass (P2P) test sets.

3.2 Semantic PR Evolution Structure Discovery

The first stage identifies PRs that belong to the same functional evolution. A structure-discovery agent analyzes PR text, code and test changes, repository ancestry, public symbols, and observable behavior to infer semantic dependencies between historical changes. A later PR is considered semantically related to an earlier one when it reuses, extends, refines, fixes, or constrains behavior established by the earlier PR.

Semantic relatedness alone is insufficient. Each retained PR must also make a non-redundant contribution to the eventual behavioral contract: removing it should discard a relevant behavior, edge case, interface decision, or compatibility constraint. This criterion filters out changes that are historically nearby but semantically incidental, including unrelated refactors, formatting changes, dependency updates, and intermediate implementations that are fully superseded.

For a chain, COPE orders the selected PRs according to historical precedence and semantic dependency, $p_1 \rightarrow_s p_2 \rightarrow_s \cdots \rightarrow_s p_k$, while allowing unrelated PRs to occur between consecutive selected nodes in the original history. For a tree, edges represent semantic predecessor relations rather than Git branching structure. Each branching point must introduce a shared semantic contribution—for example, a public API, data abstraction, execution path, or observable behavior—that is subsequently reused, extended, or refined along multiple branches. Shared files or temporal proximity alone are insufficient to establish such a branching relation.

Candidate structures proposed by the structure-discovery agent are further validated against the underlying PR boundaries, code and tests, and repository ancestry to reject unsupported semantic edges or historically incompatible states. In the dataset constructed in this work, we additionally use lightweight size heuristics for tractability: chains typically contain 3–5 PRs, while tree candidates contain at least four PRs and one branching point.

3.3 Retrospective Specification Recovery

Given a verified structure S, the task-synthesis agent reconstructs the behavioral contract revealed across the selected PR evolution. Later fixes, refinements, extensions, and compatibility changes are treated as evidence for requirements that may have been implicit or unknown when the functionality was first introduced. The agent consolidates this evidence into a recovered specification Σ_S, retaining the final observable behavior while discarding intermediate implementation choices that are later superseded.

For a chain, the agent integrates the surviving constraints revealed along the evolution sequence. For a tree, it recovers the shared behavior rooted at r and merges compatible requirements contributed by different branches. Σ_S therefore focuses on externally relevant behavior such as public interfaces, boundary conditions, error semantics, and compatibility requirements, while excluding implementation details that do not affect the final contract.

3.4 COUNTERFACTUAL TASK SYNTHESIS

COPE next re-instantiates the recovered specification at the earlier code anchor:

$$\Delta_S^\star = G_{\text{impl}}(B_S, \Sigma_S, \mathcal{D}_S). \tag{3}$$

This stage is counterfactual: instead of replaying or cherry-picking the original PR patches, COPE directly implements the accumulated specification on B_S. This distinction is necessary because selected PRs may be separated by unrelated development, and later implementations may rely on intermediate design choices that were subsequently revised.

The synthesis agent therefore uses Σ_S as the behavioral target and adapts its implementation to the architecture available at B_S. Supporting abstractions absent from the anchor may be reconstructed when they are necessary for a retained requirement, but unrelated refactors, dependency changes, and other intervening history are excluded. For tree structures, shared and branch-specific requirements are implemented jointly according to their behavioral compatibility rather than by merging the original patches.

The same recovered specification is used to construct the public task description

$$q_S = G_{\text{desc}}(\Sigma_S, \mathcal{D}_S, B_S). \tag{4}$$

The resulting description presents the recovered behavior as a self-contained implementation request, while withholding private supervision such as the golden implementation, hidden tests, original PR identifiers, and other evaluation artifacts.

3.5 EXECUTABLE SUPERVISION AND VALIDATION

COPE constructs a test patch Δ_S^{test} aligned with the recovered specification. It preferentially reuses tests introduced or modified by selected PRs, adapting them when necessary to execute at B_S. Additional tests are synthesized only when the historical tests do not sufficiently cover a retained requirement. Existing tests that pass stably at the anchor are additionally retained as regression supervision.

For each task, COPE builds a test command c_S and an instance-compatible parser π_S that maps test output to $\{\text{PASSED}, \text{FAILED}, \text{ERROR}\}$.

We evaluate two states:

$$R_1 = \text{Run}\left(B_S \oplus \Delta_S^{\text{test}}; c_S\right), \tag{5}$$

$$R_2 = \text{Run}\left(B_S \oplus \Delta_S^{\text{test}} \oplus \Delta_S^\star; c_S\right). \tag{6}$$

Let $s_i(t) = \pi_S(R_i)[t]$ denote the parsed outcome of test t under state R_i.

COPE defines

$$\mathcal{F}_S = \{t \mid s_1(t) \in \{\text{FAILED}, \text{ERROR}\},\ s_2(t) = \text{PASSED}\}. \tag{7}$$

$$\mathcal{P}_S = \{t \mid s_1(t) = s_2(t) = \text{PASSED}\}. \tag{8}$$

Thus, $\mathcal{F}_S$ captures behaviors introduced or repaired by the synthesized implementation, while $\mathcal{P}_S$ provides regression constraints.

A candidate is retained only when its target behavior is represented by valid F2P transitions and the golden implementation introduces no regressions among the retained P2P tests. When validation fails, the execution feedback is returned to the task-synthesis agent, which revises the synthesized code patch, test patch, or execution configuration as needed. The revised artifacts are then re-evaluated by rerunning R_1 and R_2, forming an iterative synthesis-validation loop. Candidates that remain inconsistent after repeated refinement are discarded.

Finally, COPE performs a *golden precheck* in a fresh instance of I_S, independently reapplying the code and test patches and rerunning c_S. Only tasks whose expected F2P and P2P outcomes reproduce successfully enter the final dataset. The rollout agent receives only $x_S = (B_S, I_S, q_S)$, while the remaining artifacts are kept private for evaluation.

4 EXPERIMENTS

We evaluate COPE from three perspectives: task characteristics, downstream training effectiveness, and controlled ablations of semantic PR selection and trajectory filtering.

4.1 EXPERIMENTAL SETUP

Task construction and rollout. We construct COPE tasks from 21,979 GitHub repositories across 75 programming-language labels, using GPT-5.5 (OpenAI, 2026) for both semantic structure discovery and task synthesis. We sample 24,000 tasks for rollout, covering both PR-chain and PR-tree structures. Using MiniSWE-agent (Yang et al., 2024) with DeepSeek V4 Flash (DeepSeek-AI, 2026), we obtain 4,478 execution-verified trajectories. For controlled training, we use 576 PR-chain, 384 PR-tree, and 1,063 single-PR trajectories, with all settings matched to the total training-token budget of the released daVinci-Agency data (Jiang et al., 2026).

Model training. We fine-tune Qwen3 models at three scales, including Qwen3-4B, Qwen3-8B, and Qwen3-30B-A3B-Thinking-2507 (Qwen Team, 2025), using full-parameter training with SLIME (Zhu et al., 2025). The main supervision comparison is performed on Qwen3-30B-A3B-Thinking-2507, with PR-chain and PR-tree supervision further evaluated on Qwen3-4B and Qwen3-8B to assess consistency across model scales.

Evaluation. We evaluate the trained models on SWE-Bench Verified (Jimenez et al., 2024) and SWE-Bench Multilingual (Yang et al., 2026) using task resolution rate as the primary metric.

4.2 TASK CHARACTERIZATION

Before evaluating downstream performance, we characterize COPE-generated tasks and compare their scope with conventional single-PR tasks.

Task complexity. We compare single-PR, PR-chain, and PR-tree tasks in terms of evolution scope, repository modifications, and agent steps in the final training subsets. Here, *intervening PRs* denotes the number of unselected PRs occurring between the earliest and latest selected PRs in repository history. Table 1 summarizes the average statistics.

Table 1: Average task complexity across different supervision sources.

	Single-PR	PR-Chain	PR-Tree
Source PRs	1.00	3.93	5.74
Intervening PRs	0.00	164.97	194.00
Changed files	4.00	9.44	15.53
Patch size (LOC)	219.52	897.71	1,582.59
Agent steps	39.88	50.82	64.15

Compared with single-PR tasks, COPE tasks span multiple stages of software evolution and involve broader repository modifications and longer agent interactions, with PR-tree tasks exhibiting the largest overall scope.

Case Study Figure 3 presents a representative example from the `smallrye-mutiny` repository. The selected PRs are semantically related but non-contiguous in repository history: they progressively introduce and refine `Uni` memoization through API stabilization, concurrency and protocol fixes, and lifecycle cleanup, while an intervening dependency update (PR #500) is excluded as unrelated. COPE consolidates these distributed evolution signals into a recovered behavioral specification and a unified executable task.

The corresponding rollout requires 129 agent steps and modifies four source files, while successfully satisfying all 8 F2P tests and the retained P2P tests. This example illustrates how COPE recovers an executable behavioral contract from distributed software evolution rather than simply aggregating historical PRs.

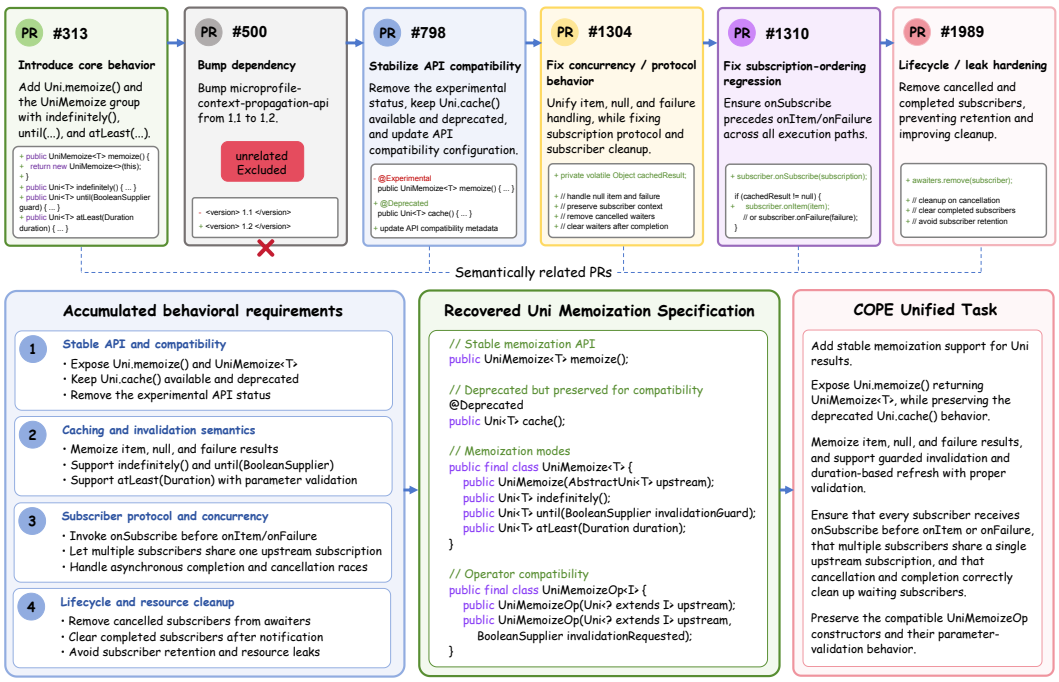


Figure 3: **Case study of `Uni` memoization evolution in `smallrye/smallrye-mutiny`.** Five semantically related but non-contiguous PRs reveal the memoization contract, while an unrelated dependency update (PR #500) is excluded. COPE consolidates the selected evolution into a recovered specification and a unified implementation task. Code snippets are abbreviated for visualization.

4.3 Main Results

We evaluate the effectiveness of COPE supervision from two perspectives. First, we compare COPE supervision with alternative repository-level training data under a matched training-token budget. Specifically, we compare PR-chain and PR-tree trajectories against single-PR and daVinci-Agency (Jiang et al., 2026) supervision. Second, we examine whether the effectiveness of COPE remains consistent across different model scales.

Comparison of training supervision. Table 2 compares PR-chain and PR-tree supervision with single-PR and daVinci-Agency baselines on Qwen3-30B-A3B-Thinking-2507.

Table 2: Comparison of different supervision sources on Qwen3-30B-A3B-Thinking-2507.

Training supervision	# Traj.	SWE-Bench Verified	SWE-Bench Multilingual
Base model	–	24.40%	13.00%
Single-PR	1,063	26.40%	14.00%
daVinci-Agency	239	27.40%	13.67%
PR-Chain	576	28.20%	14.67%
PR-Tree	384	**29.60%**	**17.33%**

Under a matched training-token budget (see Appendix D for details), both PR-Chain and PR-Tree supervision outperform Single-PR and daVinci-Agency supervision. Notably, PR-Tree achieves the best results using 384 trajectories, demonstrating the effectiveness of compressing distributed PR evolution into unified, high-quality training supervision.

Scaling across model sizes. To assess whether the benefits of COPE supervision generalize across model scales, we evaluate PR-chain and PR-tree training on Qwen3-4B, Qwen3-8B, and Qwen3-30B-A3B-Thinking-2507. Table 3 summarizes the results on both benchmarks.

Table 3: Effectiveness of COPE supervision across different model scales.

	SWE-Bench Verified			SWE-Bench Multilingual		
Model	Base	PR-Chain	PR-Tree	Base	PR-Chain	PR-Tree
Qwen3-4B	9.60%	13.00%	**13.20%**	3.33%	**4.67%**	4.33%
Qwen3-8B	17.20%	19.00%	**20.00%**	6.33%	**8.00%**	7.33%
Qwen3-30B-A3B	24.40%	28.20%	**29.60%**	13.00%	14.67%	**17.33%**

Across model scales, both PR-Chain and PR-Tree improve over the base models on both benchmarks, with PR-Tree achieving the best SWE-Bench Verified results at all three scales. On SWE-Bench Multilingual, PR-Chain is slightly stronger at smaller scales, while PR-Tree performs best at 30B-A3B. These results demonstrate the robustness of COPE supervision across model scales.

4.4 Ablation Studies

All ablations use Qwen3-4B on SWE-Bench Verified, with trajectories sampled to match the same training-token budget.

Table 4: Ablation results on SWE-Bench Verified using Qwen3-4B.

Training supervision	# Traj.	SWE-Bench Verified
Base model	–	9.60%
Temporal PRs	714	10.60%
PR-Chain	576	**13.00%**
LLM-FN	376	11.00%

Effect of semantic PR selection. To isolate the effect of semantic selection, we construct a temporal-selection baseline for each PR chain. Starting from the first PR of the original COPE chain, we select the same number of chronologically consecutive PRs, regardless of semantic relatedness, and apply the same synthesis and rollout pipeline. As shown in Table 4, PR-Chain supervision improves over the temporal baseline under the same training-token budget.

Effect of trajectory filtering. LLM-based trajectory filtering has been adopted in prior long-horizon agent data construction, including daVinci-Agency (Jiang et al., 2026). We compare LLM-based filtering with execution-based verification on 7,557 PR-chain rollout candidates using DeepSeek V4 Flash (DeepSeek-AI, 2026) as the judge.

Table 5: Comparison between LLM-based filtering and execution-based evaluation.

	Execution Resolved	Execution Failed
Judge Accepted	966	540
Judge Rejected	1,235	4,816

The LLM judge rejects 1,235 execution-resolved trajectories. We denote these judge-rejected but execution-resolved trajectories as *LLM-FN*. As shown in Table 4, training on LLM-FN trajectories improves over the base model, suggesting that LLM-based filtering can discard useful supervision.

5 Conclusion

We introduced **Compression of Pull-Request Evolution (COPE)**, a framework that constructs long-horizon coding tasks by identifying semantically coherent PR evolution, recovering the behavioral specification revealed across it, and synthesizing unified executable tasks. Experiments show that COPE provides effective training supervision for repository-level coding agents, highlighting software evolution as a valuable source for recovering semantically complete coding tasks.

AI USE STATEMENT

We used generative AI tools in several stages of this work. First, AI-assisted tools were used during paper preparation for language editing, improving readability, organizing sections, and refining the presentation of figures and technical descriptions. All such edits were manually reviewed and verified by the authors. They were also used to support literature retrieval and discovery, including identifying potentially relevant prior work. All cited references were subsequently checked by the authors against the corresponding papers or official publication pages.

Generative AI was also used during research ideation and methodology development, including discussion and refinement of the task formulation, experimental design, and presentation of the COPE framework. All resulting research decisions, methodological claims, and experimental conclusions were reviewed and determined by the authors.

In addition, large language models are explicit components of the COPE data construction pipeline described in this paper. They are used for semantic analysis of software evolution, task synthesis, and coding agent rollout generation. In particular, model-generated tasks and trajectories form part of the synthetic supervision used for training. These artifacts are not accepted solely on the basis of model judgments: synthesized tasks are subjected to the execution-based validation and golden precheck described in Section 3, and rollout trajectories used for training are filtered using executable evaluation.

We did not use generative AI to formulate or prove mathematical claims in this work. All AI-assisted text, references, methodological suggestions, generated artifacts, and experimental outputs were reviewed by the authors, who take full responsibility for the final content, claims, and results presented in this paper.

ETHICS STATEMENT

This work uses publicly available software repositories and their associated development artifacts, including pull requests, code changes, and tests. Our study does not involve human subjects or private repositories. Derived tasks and trajectories are used for research on software-engineering agents and may inherit biases or limitations from the repositories and language models used in their construction. Any released artifacts will respect the applicable licenses and terms of the source repositories. We additionally use executable validation to reduce the risk of releasing unsupported or behaviorally inconsistent tasks.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the COPE task-construction pipeline, including semantic PR structure discovery, retrospective specification recovery, counterfactual task synthesis, and executable validation, in Sections 3 and 4. The appendix further reports the prompts, task and trajectory statistics, training configurations, and additional experimental details. We also specify the rollout, filtering, training-budget matching, and evaluation procedures used in our experiments to facilitate reproduction of the reported results.

REFERENCES

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *International Conference on Learning Representations*, volume 2025, pp. 64485–64515, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a1e6783e4d739196cad3336f12d402bf-Paper-Conference.pdf.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems*

Datasets and Benchmarks Track, 2026. URL https://openreview.net/forum?id=nMpJoVmRy1.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. Swe-universe: Scale real-world verifiable environments to millions, 2026. URL https://arxiv.org/abs/2602.02361.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

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, 2026. URL https://arxiv.org/abs/2603.13428.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks, 2026. URL https://arxiv.org/abs/2506.10954.

Ruida Hu, Chao Peng, XinchenWang, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=fZsd3KLMje.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026. URL https://arxiv.org/abs/2607.07946.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=7evvwwdo3z.

Mohan Jiang, Dayuan Fu, Junhao Shi, Ji Zeng, Weiye Si, Keyu Li, Xuefeng Li, Yang Xiao, Wenjie Li, Dequan Wang, and Pengfei Liu. davinci-agency: Unlocking long-horizon agency data-efficiently, 2026. URL https://arxiv.org/abs/2602.02619.

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*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios, 2026. URL https://arxiv.org/abs/2512.18470.

OpenAI. Introducing gpt-5.5, April 2026. URL https://openai.com/index/introducing-gpt-5-5/.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In *International Conference on Learning Representations*, volume 2025, pp. 30098–30121, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4a4a3c197deac042461c677219efd36c-Paper-Conference.pdf.

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*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

KN Ajay Shastry, Ganesh Senrayan, Shrey Satapara, Pranoy Panda, and Chaitanya Devaguptapu. Beyond isolated tasks: A framework for evaluating coding agents on sequential software evolution, 2026. URL https://arxiv.org/abs/2604.03035.

Ethan Shen, Daniel Tormoen, Saurabh Shah, Ali Farhadi, and Tim Dettmers. SERA: Soft-verified efficient repository agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=GHKj1XcSlt.

Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro. Nemotron-CORTEXA: Enhancing LLM agents for software engineering tasks via improved localization and solution diversity. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 56085–56100. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/sohrabizadeh25a.html.

Konstantinos Vergopoulos, Mark Niklas Mueller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=qnE2m3pIAb.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3742–3761. Association for Computational Linguistics, July 2025a. URL https://aclanthology.org/2025.findings-acl.193/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=OJd3ayDDoF.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. SWE-fixer: Training open-source LLMs for effective and efficient GitHub issue resolution. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1123–1139. Association for Computational Linguistics, July 2025. URL https://aclanthology.org/2025.findings-acl.62/.

Yiqing Xie, Emmy Liu, Gaokai Zhang, Nachiket Kotalwar, Shubham Gandhi, Sathwik Acharya, Xingyao Wang, Carolyn Rose, Graham Neubig, and Daniel Fried. Hybrid-gym: Training coding agents to generalize across tasks. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=2TVSJxxz9u.

Xinbo Xu, Ruihan Yang, Haiyang Shen, Wendong Xu, Bofei Gao, Ruoyu Wu, Kean Shi, Weichu Xie, Xuanzhong Chen, Ming Wu, Jason Zeng, Michael Heinrich, Elvis Zhang, Liang Chen, Kuan Li, and Baobao Chang. Roadmapbench: Evaluating long-horizon agentic software development across version upgrades, 2026. URL https://arxiv.org/abs/2605.15846.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=63iVrXc8cC.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5afa9cb1e917b898ad418216dc726fbd-Paper-Datasets_and_Benchmarks_Track.pdf.

Qixing Zhou, JiaCheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. Featurebench: Benchmarking agentic coding for complex feature development. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=41xrZ3uGuI.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

A Task Statistics

Task construction and rollout. We construct tasks from an initial pool of 21,979 GitHub repositories. The resulting task pool contains 89,627 PR-chain tasks and 36,574 PR-tree tasks, totaling 126,201 tasks. From this pool, we sample 8,000 PR-chain and 16,000 PR-tree tasks for agent rollout. Table 6 reports the rollout and executable-evaluation statistics, together with the Single-PR and Temporal-PR baselines used in our experiments.

Table 6: Executable evaluation statistics for different task sources.

Task source	Resolved / Evaluated	Pass rate
PR-Chain	2,201 / 7,557	29.13%
PR-Tree	2,277 / 15,085	15.09%
Single-PR	2,309 / 4,665	49.50%
Temporal-PR	813 / 2,238	36.33%

Overall, PR-chain and PR-tree tasks yield 4,478 execution-verified trajectories. For downstream training, we further sample trajectories from each supervision source to match the training-token budget of the released daVinci-Agency data (Jiang et al., 2026). This sampling is performed solely for controlled comparison rather than additional quality filtering. The resulting trajectory counts are reported in the corresponding main-text tables.

Trajectory length. We further characterize the interaction length of the execution-verified COPE trajectories. An agent turn corresponds to one assistant action in the recorded trajectory. As shown in Table 7, PR-chain trajectories contain 60.67 turns on average, with a median of 51, while PR-tree trajectories contain 67.03 turns on average, with a median of 58. The upper tail is substantially longer, reaching 112 and 120 turns at the 90th percentile for PR-chain and PR-tree tasks, respectively.

Table 7: Agent-turn statistics of execution-verified COPE trajectories.

Task source	Mean	Median	P75	P90	P95
PR-Chain	60.67	51	79	112	137
PR-Tree	67.03	58	85	120	143.2

Figure 4 further shows the distribution of agent turns. Both PR-chain and PR-tree trajectories exhibit broad interaction-length distributions, with PR-tree trajectories generally shifted toward longer interactions.

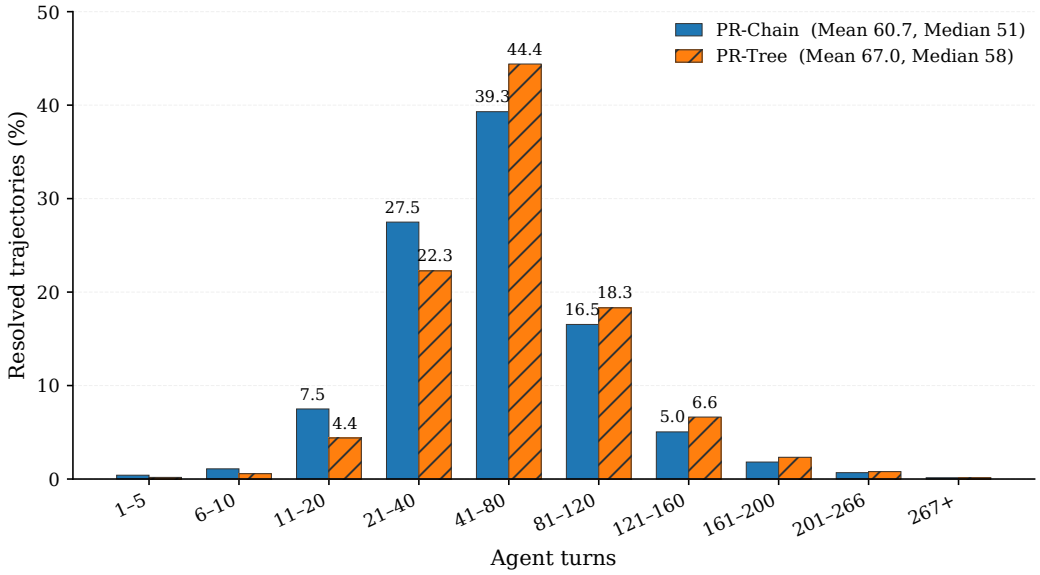


Figure 4: Distribution of agent turns for execution-verified PR-chain and PR-tree trajectories.

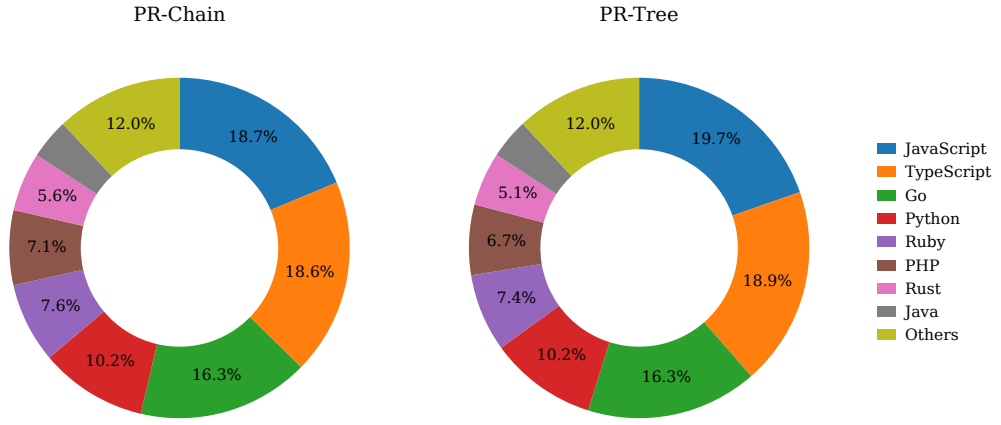


Figure 5: Programming-language distribution of the sampled PR-chain and PR-tree rollout tasks. Languages outside the eight most frequent categories are grouped as *Others*.

Language coverage. The sampled rollout tasks cover a broad range of programming languages, spanning 54 language labels for PR-chain tasks and 60 for PR-tree tasks. Figure 5 shows the language distribution before rollout-based filtering and training subset selection. Both structures exhibit

highly similar language coverage, with JavaScript, TypeScript, Go, and Python accounting for the largest fractions, followed by Ruby, PHP, Rust, and Java. The remaining tasks cover a diverse set of less frequent languages. This result suggests that the COPE pipeline is broadly applicable across different programming language ecosystems.

We additionally report the language distribution of the final training subsets in Figure 6. After execution-based verification and token-budget-matched sampling, the PR-chain and PR-tree training subsets retain 24 and 20 language labels, respectively. Although JavaScript remains the largest category in both subsets, the relative proportions of several languages shift compared with the rollout pool. Overall, the final supervision still covers a diverse set of programming environments while preserving broadly similar language composition across PR-chain and PR-tree training data.

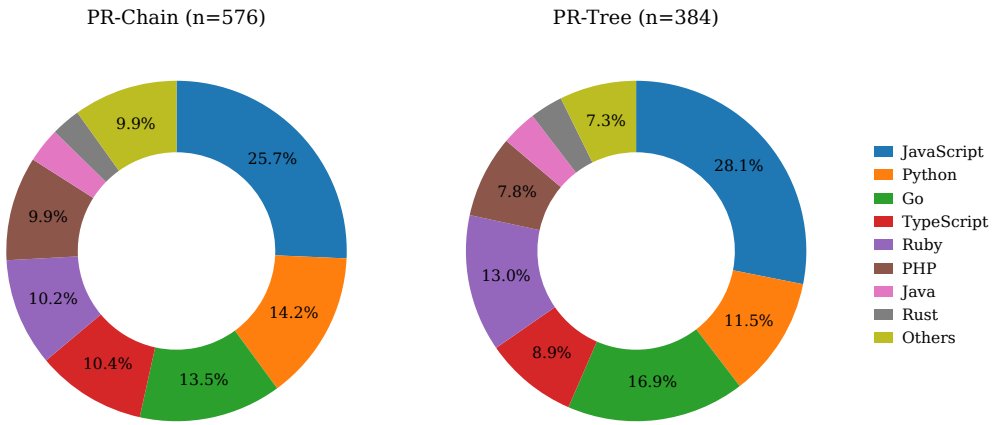


Figure 6: Programming-language distribution of the final PR-chain and PR-tree training subsets, containing 576 and 384 trajectories, respectively. Languages outside the eight most frequent categories are grouped as *Others*.

B Prompt Overview

COPE uses four structured prompts for PR-chain/tree discovery and task synthesis. Although the prompts differ in their structure-specific constraints, they follow a common workflow that transforms repository evolution evidence into a unified executable coding task. The discovery prompts identify semantically coherent evolution structures from PR metadata, descriptions, and code changes, whereas the synthesis prompts recover the accumulated behavioral contract and reconstruct the corresponding task at an earlier repository state. Across both PR chains and PR trees, the prompts emphasize semantic coherence, explicit task boundaries, and execution-based validation rather than temporal adjacency or direct replay of historical patches. Table 8 summarizes the major stages and the differences between PR-chain and PR-tree processing.

C Effect of Task Description Synthesis

To examine the role of task synthesis, we compare COPE's recovered task description with a simple aggregation baseline constructed from the same selected PRs. Specifically, we preserve the original PR order and insert each PR's title and description into the following fixed template:

Table 8: Overview of the structured prompts used in COPE. The discovery prompts cover evidence inspection and semantic structure discovery, while the synthesis prompts perform retrospective synthesis, executable validation, and public task construction.

Stage	Operation	PR Chain	PR Tree
1	**Evidence inspection**	Inspect repository metadata, PR descriptions, and PR-level patches and diffs as evidence of software evolution. Previously discovered chains may additionally serve as retrieval hints for tree discovery.	
2	**Structure discovery**	Select a concise, semantically complete sequence describing one evolving feature line. Each PR must contribute a non-trivial semantic change, while selected PRs need not be temporally adjacent.	Identify a shared functional foundation with multiple coherent branches. Each branching point must provide concrete shared semantics, and every branch must demonstrably reuse, refine, or extend this contribution.
3	**Retrospective synthesis**	Recover the accumulated behavioral contract across the selected PRs and synthesize their final end-state at the chain's base repository state. Later PRs are treated as refinements or extensions of the same evolving behavior.	Recover the shared foundation and branch-specific behavioral contracts, then integrate all retained branches into one end-state at the tree's root repository state. Overlapping changes are reconciled by semantic intent rather than by directly stacking historical patches.
4	**Executable validation**	Validate the synthesized task through a test-only run and a test-plus-code run. The two executions derive fail-to-pass and pass-to-pass tests and detect regressions. For PR trees, the retained behavior of every branch must be jointly covered by executable validation.	
5	**Task construction**	Derive an implementation-actionable task description from the validated behavioral contract. The description preserves externally observable requirements, including relevant APIs, compatibility constraints, boundary conditions, and error behavior, while excluding PR history, patches, tests, and evaluation details.	

Fixed PR-description aggregation template

Implement the cumulative repository change described by the following pull requests. Treat them as one evolving feature in the order shown. Later requirements override earlier ones when they conflict, and the implementation should represent the final cumulative behavior rather than an intermediate state.

Pull Request 1 of N (#PR)
Title: [PR title]
Description: [PR description]
...

For each rollout instance, only the task description is replaced; the repository, base commit, selected PRs, executable tests, and evaluation setup remain unchanged. The aggregation baseline achieves an execution-resolved rate of only 4.76%, compared with 29.13% for the COPE PR-chain tasks reported in Table 6. This substantial gap suggests that simply aggregating historical PR descriptions does not provide a sufficiently coherent implementation objective, highlighting the importance of retrospective specification recovery in consolidating distributed evolutionary evidence into a unified coding task.

D Training Details

Training-budget matching. To ensure a controlled comparison across supervision sources, we match the total number of training loss tokens to that of the released daVinci-Agency data (Jiang et al., 2026). For each source, we sample execution-verified trajectories until the target training-token budget is reached, with the final token count differing from the target by fewer than 100

tokens, which is negligible relative to the overall training budget. Consequently, different supervision sources contain different numbers of trajectories, reflecting differences in trajectory length rather than differences in training budget.

Models and optimization. We perform full-parameter fine-tuning with SLIME on Qwen3 models. The training configuration is summarized in Table 9.

Table 9: Training configuration used in our experiments.

Setting	Configuration
Model and training	
Training	Full-parameter supervised fine-tuning
Framework	SLIME with Megatron-LM
Precision	BF16
Optimization	
Optimizer	AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.98$)
Learning rate	2×10^{-6}
LR schedule	Cosine decay to 1×10^{-7}
Warmup ratio	0.1
Weight decay	0.1
Sequence and batch	
Max sequence length	262,144 tokens
Global batch size	64 trajectories
Gradient accumulation	Dynamic token-budgeted microbatches
Training epochs	2

E DISCUSSION

COPE provides a complementary view of long-horizon repository-level coding by emphasizing the semantic horizon of a task rather than the duration of interaction. Instead of replaying multiple development stages, COPE uses software evolution as evidence to recover behavioral constraints distributed across related PRs and consolidates them into a unified implementation objective. This formulation complements sequential evolution settings, which emphasize persistent state, intermediate decisions, and error propagation, whereas COPE focuses on integrating distributed requirements under a single coding goal.

COPE also has several limitations. It relies on observable repository history and LLM-based synthesis, and some evolved behaviors may not be cleanly reconstructed at an earlier repository state. In addition, the quality of the constructed tasks depends on whether the available PR history sufficiently reveals the underlying behavioral contract.