CHAIN OF PULL REQUESTS: UNLOCKING LONG-HORIZON AGENCY DATA-EFFICIENTLY

Anonymous authors
Paper under double-blind review

ABSTRACT

While Large Language Models (LLMs) excel at short-term tasks, scaling them to long-horizon agentic workflows remains challenging for the scarcity of data that captures authentic long-dependency structures and cross-state evolutionary dynamics. Existing synthesis methods either limit tasks to single-feature scenarios or incur prohibitive human annotation costs, failing to provide scalable, high-quality supervision. We address this by reconceptualizing data synthesis through the lens of real-world software evolution. Our key insight: PR sequences naturally embody the very supervision signals needed for long-horizon learning. They decompose objectives into units, maintain iterative coherence, and encode bug-fix refinements. Building on this, we propose **C**hain **o**f **P**ull **R**equests (CoPR), which mines structured supervision from PR sequences through three interlocking mechanisms: (1) progressive task decomposition via continuous commits, (2) long-term consistency enforcement through unified functional objectives, and (3) verifiable refinement from authentic bug-fix trajectories. CoPR preserves causal dependencies and iterative refinements, naturally aligning with full-cycle task modeling. The resulting trajectories are substantial—averaging 85k tokens and 116 tool calls—yet remarkably data-efficient: fine-tuning GLM-4.6 on 239 CoPR samples yields broad benchmark improvements, notably achieving a 47% relative gain on Toolathlon. Further analysis confirms the effective internalization of long-horizon behaviors and unveils training and inference scaling for extended horizons. Our work establishes CoPR as a scalable paradigm beyond limitations of single-feature synthesis, proving that modeling real-world evolutionary trajectories offers a principled path to unlock the intrinsic long-horizon potential of agents.

1 INTRODUCTION

Recent LLMs have made substantial progress in code generation, tool use, and reasoning (Zeng et al., 2025; Team et al., 2025; Cai et al., 2025), enabling agentic systems to handle many short-horizon tasks. The central challenge is therefore shifting from producing a locally correct patch to coordinating long-horizon software objectives in realistic development environments. Agent harnesses increasingly demand state-aware decomposition: models must break the complex target into executable subtasks while maintaining an explicit awareness of evolving task state, including ordering constraints, inter-stage dependencies, and potential conflicts. This requirement is reflected in multi-agent systems such as agent swarm (Team et al., 2026), which decomposes complex tasks into specialized subagents as a form of proactive context management.

This shift imposes new requirements on training data. Traditional code corpora and single-turn instruction data mainly supervise local code generation, function completion, or isolated bug fixing, and therefore provide limited guidance on how to decompose a large software objective into multiple interdependent stages. SWE-Bench and SWE-Smith (Yang et al., 2025b; Deng et al., 2025; Fu et al., 2026) mark an important step forward by moving beyond single-turn code generation toward repository-level software tasks, where models must modify code in realistic repository contexts and validate their changes through tests. However, from the perspective of supervision structure, such data is still largely organized around single-task or single-PR supervision: each instance typically corresponds to one issue, one repository snapshot, and one final patch. Although this setting is substantially closer to real software development than standard instruction data, it still compresses software development into a one-shot issue-to-patch process, without explicitly modeling how a

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.

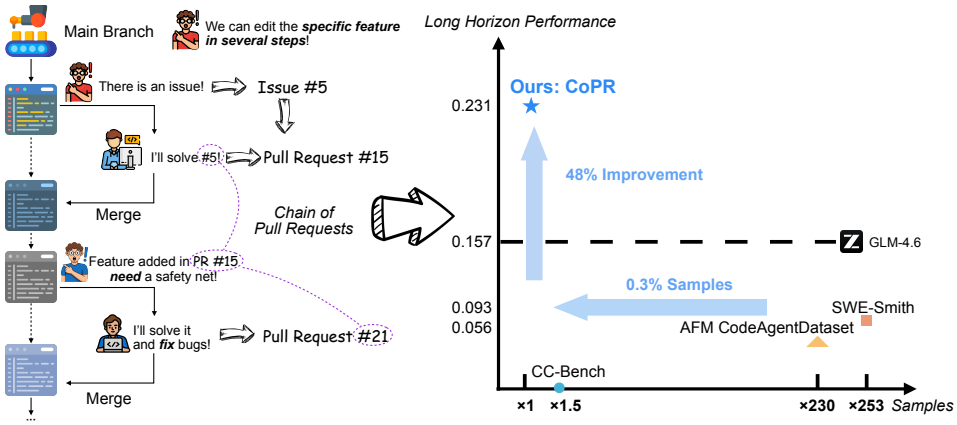

Figure 1: CoPR turns real pull-request evolution into chained and verifiable supervision, achieving stronger long-horizon performance with only 239 trajectories.

feature is progressively decomposed, maintained long-term consistency, and refined over its full evolution cycle (Guo et al., 2025; Xue et al., 2024; Radhakrishnan et al., 2023; Yao et al., 2022).

Current long-horizon harnesses can explicitly decompose complex tasks into subtasks (Team et al., 2026; Anthropic, 2026), but such external scaffolding does not teach LLMs to decompose tasks themselves or maintain cross-subtask state dependencies, as shown in Figure 4. This limitation is particularly important in long-horizon tasks, where subtasks are rarely independent local problems that can be solved in isolation or simply parallelized. Each subtask changes the code state, constraints, and decision space inherited by subsequent subtasks. Therefore, long-horizon development is not merely a matter of taking more steps or using more subagents; it is a state evolution process with causal dependencies across stages.

In this context, GitHub Pull Request (PR) sequences offer a representative candidate data modality. In practice, a feature is rarely completed through a single isolated modification. Instead, it often proceeds through multiple stages, such as initial implementation, interface adaptation, feature extension, test repair, refactoring, and compatibility adjustment shown in Figure 1. These stages are frequently reflected in a sequence of related pull requests. Each PR is a reviewable and often testable repository-level state transition; subsequent PRs then operate on the code state produced by earlier PRs, inheriting their implementations, fixing their defects, or extending their functionality. As a result, a PR chain naturally records how a high-level software objective is decomposed into semantically related subtasks, and how the effects of these subtasks accumulate over an evolving repository state.

Inspired by this, we propose **C**hain **o**f **P**ull **R**equests (CoPR), a novel data synthesis paradigm for long-horizon agent learning. Grounded in continuous PR evolution of real-world software development as Figure 2, CoPR explicitly organizes multiple PRs centered on a common objective into interdependent task chains. This fosters long-horizon interactions and feedback within evolving code states, thereby extending single-patch generation into a multi-turn submission-feedback-refinement trajectory. Under this setting, the agent must not only achieve phase-wise progression but also maintain overall goal consistency and perform effective error correction under constraints of evolving code states and external feedback. In practice, by chaining up to five PRs, we construct task chains yielding trajectories with an average length of 85k tokens and 116 tool calls, providing scalable supervision for training agents' capabilities in long-term planning, consistency, and iterative improvement.

Our experiments demonstrate the effectiveness of CoPR in long-horizon tasks. Fine-tuning GLM-4.6 on merely 239 such samples yields broad improvements on general benchmarks, notably a 47% gain on Toolathlon. These results underscore the immense potential of long-horizon synthetic trajectories in unlocking the complex task-solving capabilities of autonomous agents. Furthermore, our case study reflects from a behavioral level that the fine-tuned model exhibits stronger performance in task decomposition, long-term consistency, and refinement. This enables more stable planning of stage goals, maintenance of global alignment, and effective error correction across multi-round feedback. Simultaneously, by modulating the interaction temporal domain, we explore the scaling properties of

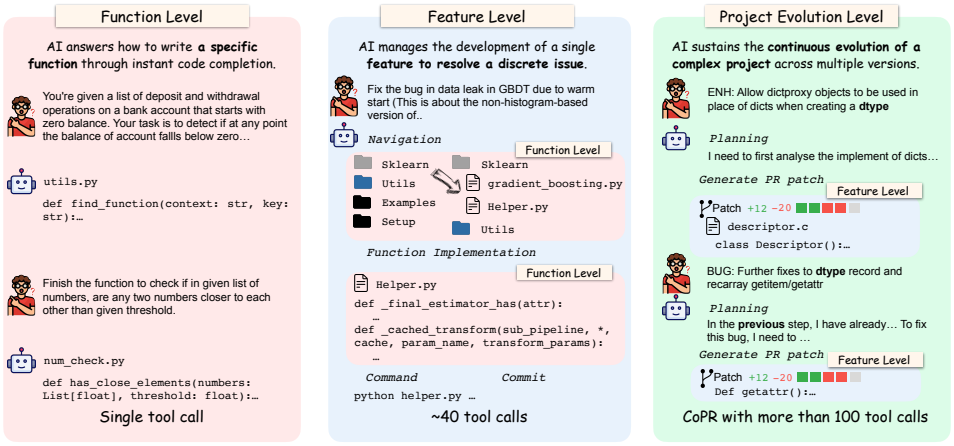

Figure 2: Comparison of software-engineering horizons. CoPR targets project evolution, where agents must preserve objectives across dependent PRs and evolving code states.

CoPR, demonstrating that extending longer long-horizon closed-loop training data is an effective path to continuously breaking through the performance ceiling of agents.

Our contributions:

(1) We propose Chain of Pull Requests (CoPR), a paradigm utilizing real-world PR evolution to construct verifiable long-horizon training data. Fine-tuning GLM-4.6 on just 239 CoPR samples yields broad improvements, notably a 47% relative gain on Toolathlon.

(2) We present significant efficiency improvements through real-world long-horizon task modeling. Moreover, case studies on three meta-skills confirm that CoPR enables robust planning and refinement.

(3) We unveil data scaling laws specific to long-horizon tasks, establishing that expanding both the training horizon and inference-time interaction budgets is a critical mechanism for continuously advancing agentic performance.

2 Related Works

Agentic Language Model The evolution of Agentic Language Models, pioneered by Toolformer (Schick et al., 2023) and ReAct (Yao et al., 2022), signifies a shift toward autonomous decision-making. While recent foundation models like GLM-4.5 (Zeng et al., 2025) and Kimi-K2 (Team et al., 2025) have integrated native tool capabilities, they still struggle with trajectory calibration in complex, multi-turn interactions, often resulting in incoherent tool usage. To address this, CoPR constructs high-density, long-horizon trajectory data that mandates frequent environmental interactions. This paradigm enhances the model's sensitivity to execution feedback, fundamentally improving tool-calling precision and stability in real-world engineering tasks.

Long-Horizon Agency As the capabilities of autonomous agents continue to evolve, both academia and industry have increasingly focused on the demands and challenges associated with solving long-horizon tasks. Following the introduction of SWE-bench (Jimenez et al., 2023), subsequent benchmarks such as Toolathlon (Li et al., 2025a), UltraHorizon (Luo et al., 2025), and SWE-Bench Pro (Deng et al., 2025) have further intensified the requirements for agentic models, necessitating execution across extended time scales and complex decision dimensions. While recent works such as SWE-rebench (Badertdinov et al., 2025) and SWE-Smith (Yang et al., 2025b) generate long-horizon data through GitHub-based interactive environments, they do not explicitly cultivate key long-horizon skills, including task decomposition, long-term consistency, and iterative refinement shown in Figure 4. Constrained by teacher capabilities and outcome-based supervision, these skills emerge only implicitly, often yielding trajectories insufficient for complex engineering tasks. To address this gap, we propose Chain of Pull Requests, which mines the natural cross-stage evolution of PRs to derive explicit supervision signals that are difficult to obtain through manual annotation, enabling scalable, high-quality training for long-horizon agentic capabilities.

3 PRELIMINARY

3.1 AGENT-ENVIRONMENT INTERACTION PARADIGM

We formulate the agentic long-horizon task as a harness-mediated Markov Decision Process (Puterman, 1990), where an agent $\mathcal{A}$ interacts with a software environment $\mathcal{E}$ through an execution scaffold that couples observation, reasoning and tool execution. At step j, the agent observes $o_j \in \mathcal{O}$, produces reasoning m_j such as planning or reflection, and may invoke a tool action t_j that modifies the environment, yielding

$$\tau = \{(o_j, m_j, t_j)\}_{j=0}^{N}, \tag{1}$$

where N is the horizon length. This formulation captures the role of the harness in long-horizon software tasks, where success requires not only a locally correct patch but a sequence of state-aware decisions that decompose the target, respect inter-stage dependencies, and transform the repository from S_{init} to a query-satisfying state S_{target}.

3.2 STATE EVOLUTION AND CHAINS OF PULL REQUESTS

Let S_t denote the repository state before stage t. A feature evolution task is a sequence $S_0 \xrightarrow{\Delta_1} S_1 \xrightarrow{\Delta_2} \cdots \xrightarrow{\Delta_k} S_k$ where each patch Δ_t is selected under a stage query q_t and the state is inherited from all previous patches. The difficulty lies in the conditional structure

$$\Delta_t \sim \pi_\theta(\cdot \mid q_t, S_0, \Delta_{<t}, o_{\leq j}, m_{<j}, t_{<j}). \tag{2}$$

A pull request can be viewed as an atomic state transition. We denote a PR as $pr = (x, \hat{\Delta})$, where x is its natural-language context and $\hat{\Delta}$ is the human-authored patch. A PR chain is an ordered sequence $\mathcal{C} = (pr_1, \ldots, pr_k)$ with dependency relation $pr_i \to pr_{i-1}$, indicating that a later PR fixes, extends, adapts, or reinterprets earlier work. Unlike a chronological set of unrelated changes, a PR chain captures stateful dependencies, goal consistency across decomposed work, and refinement signals from subsequent revisions. CoPR uses these state-aware transition signals as its training target.

3.3 TASK FORMULATION

Let π_θ denote the agent policy parameterized by θ. Given a task query q derived from a PR chain $\mathcal{C}$, the agent interacts with the environment to instantiate a reasoning trajectory τ. At each step j, the agent samples reasoning and tool actions $(m_j, t_j) \sim \pi_\theta(\cdot \mid o_{\leq j}, m_{<j}, t_{<j}, \mathcal{C}, q)$ conditioned on the interaction history, culminating in a final submission patch p. To guarantee the fidelity of synthetic data, we employ an evaluation function $(\text{comment}, s) = E(q, p, \hat{p})$ to score semantic and functional alignment between the generated patch p and the ground-truth patch $\hat{p}$. This rejection-sampling process yields a high-quality training set

$$\mathcal{D}_{\text{train}} = \{(\mathcal{C}, q, \tau) \mid E(q, p, \hat{p}) = (\text{comment}, s),\ s \geq 0.8\}. \tag{3}$$

We then optimize the policy with supervised fine-tuning to extract supervision from cross-stage evolution:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(\mathcal{C},q,\tau)\sim\mathcal{D}_{\text{train}}} \left[\sum_{t=0}^{T} \log \pi_\theta(a_t \mid \mathcal{C}, q, o_{\leq t}, a_{<t}) \right]. \tag{4}$$

4 CHAIN OF PULL REQUESTS: A LONG-HORIZON DATA SYNTHESIS PARADIGM

The effectiveness of Chain of Pull Requests stems from capturing the skills inherent in long-horizon agentic behaviors through continuous feature evolution.

4.1 DATA RESOURCE

We curate nine representative GitHub repositories as primary data sources, as summarized in Table 4, using an expert-led protocol to ensure data integrity and diversity. The selection follows three

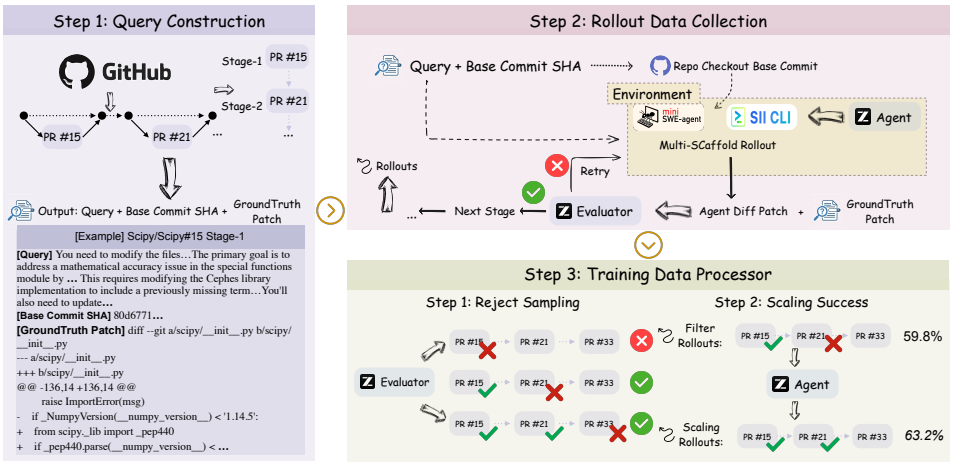

Figure 3: Overview of the CoPR Data Synthesis Paradigm. The pipeline initiates with query construction, mining PRs with dependency structures from GitHub to form topological task chains, providing reliable state evolution signals as supervision.

criteria, namely (1) Scale and Maturity, requiring over 7,000 effective PRs to capture rich inter-PR dependencies; (2) Community Interactivity, favoring active code-review environments with abundant natural-language reasoning signals; and (3) Linguistic Diversity, covering Python, Java, C, and other technology stacks. As shown in Figure 7, this human-in-the-loop curation yields long-horizon data that reflects heterogeneous real-world development settings.

4.2 Data Curation Pipeline

To construct long-horizon scenarios, we first recover PR-chain dependencies and then synthesize stage-wise queries shown in Figure 3.

PR-chain combination For each repository, we recover PR-chain dependencies from GitHub API metadata, including commit messages, review comments, linked issues, explicit PR citations, and textual follow-up references. These signals define an ordered chain $\mathcal{C} = \{pr_1, \ldots, pr_k\}$ with dependencies $pr_i \rightarrow pr_{i-1}$, where later PRs fix, extend, adapt, or reinterpret earlier work. Chains need not be contiguous in wall-clock time; the key criterion is semantic dependence on the repository state and design commitments induced by prior stages.

A sanity check over 10 sampled chains shows that eight contain explicit functional dependencies via tests or shared code paths, while two reflect the same file refactoring or documentation evolution. This suggests that the mined links encode state evolution rather than chronological proximity.

Query construction We formalize query generation as $q = f(x, \hat{p}, \mathcal{R})$, where x is the PR context, $\hat{p}$ the human patch, and $\mathcal{R}$ the repository context. For each PR, an LLM synthesizes a stage query that captures intent and high-level reasoning, omitting exact implementation details. The first stage additionally receives a global chain overview to encourage macroscopic planning and cross-stage consistency. This converts PR chains into supervision for state-evolution behaviors: decomposition into subtasks, preservation of inherited decisions, and refinement through follow-up repairs.

This process produces about 2k candidate chain queries before rollout. The final 239 training samples are successful outputs of the staged rollout and rejection-sampling, rather than manually selected examples. Thus, CoPR shifts the main cost from human annotation to execution and filtering, enabling scalable data synthesis while retaining high-fidelity long-horizon supervision.

4.3 Rollout Data Collection for Training Dataset

Environment for rollout CoPR simulates incremental software evolution by executing each PR stage on the repository state inherited from previous stages. Let $\mathcal{B}_t$ be the base repository snapshot

Table 1: **Performance comparison across long-horizon datasets on GLM-4.6.** Despite being trained on only 239 samples, CoPR outperforms baselines trained on datasets that are orders of magnitude larger. SciCode-MP denotes the SciCode main problem.

Training Data	Samples	SWE-bench	Toolathlon	DS-1000	τ^2-bench retail	τ^2-bench airline	SciCode-MP	AVG
GLM-4.6	–	0.608	0.157	0.522	0.675	**0.620**	0.062	0.441
SWE-Smith	66,000	0.404	0.093	0.470	0.586	0.565	0.123	0.373
CC-Bench	260	0.618	0.000	0.526	0.697	0.605	0.169	0.436
CodeAgent	59,939	0.184	0.056	0.308	0.564	0.560	0.015	0.281
CoPR$_{\text{SinglePR}}$	2,786	0.166$\downarrow$ 73%	0.120$\downarrow$ 24%	0.396$\downarrow$ 24%	0.590$\downarrow$ 13%	0.465$\downarrow$ 25%	0.154$\uparrow$ 148%	0.315$\downarrow$ 29%
CoPR$_{\text{SinglePRSuccess}}$	874	0.612$\uparrow$ 1%	0.185$\uparrow$ 18%	0.533$\uparrow$ 2%	0.651$\downarrow$ 4%	0.610$\downarrow$ 2%	0.154$\uparrow$ 148%	0.457$\uparrow$ 4%
CoPR$_{\text{TemporalChain}}$	600	0.616$\uparrow$ 1%	0.213$\uparrow$ 36%	**0.541**$\uparrow$ 4%	0.472$\downarrow$ 30%	0.525$\downarrow$ 15%	**0.185**$\uparrow$ 198%	0.425$\downarrow$ 4%
CoPR	239	**0.632**$\uparrow$ 4%	**0.231**$\uparrow$ 47%	0.526$\uparrow$ 1%	**0.707**$\uparrow$ 5%	0.597$\downarrow$ 4%	0.154$\uparrow$ 148%	**0.475**$\uparrow$ 8%

for stage t, and let $\widehat{\Delta}_{<t}$ be the composed agent patch from stages $1, \ldots, t-1$. The initial state of stage t is $s_t^0 = \text{Apply}\left(\mathcal{B}_t, \widehat{\Delta}_{<t}\right)$. The agent then produces a rollout trajectory under the distribution induced by its policy π_θ and the environment dynamics $\mathcal{E}$:

$$\tau_t \sim \mathbb{P}_\theta^{\mathcal{E}}\left(\cdot \mid s_t^0, q_t\right), \qquad \widehat{\Delta}_t = \text{Patch}(\tau_t). \tag{5}$$

The cumulative patch is updated as $\widehat{\Delta}_{<t+1} = \widehat{\Delta}_{<t} \circ \widehat{\Delta}_t$, so later stages are conditioned on the repository state produced by earlier agent decisions. CoPR exposes the agent to state carryover, delayed error accumulation, and refinement over an evolving codebase.

Evaluator for rejection sampling The evaluator is a conservative high-precision filter rather than a reward model. Given the query, ground-truth patch, and agent patch, it judges whether the generated patch implements the same functional intent. False positives corrupt the training set, while false negatives mainly reduce data yield, so we use a strict threshold $s \geq 0.8$ and allow up to three refinement attempts. Table 5 illustrates a comparative analysis of data performance metrics when using versus not using rejection sampling.

Rollout scaffold We deploy GLM-4.6 across SII-CLI (Lin et al., 2025) and mini-swe-agent (Yang et al., 2024), logging trajectories $\tau_i = \{(o_{i,j}, m_{i,j}, t_{i,j})\}_{j=0}^{N_i}$. Each PR stage allows up to three attempts; if a stage still fails, the pipeline retries the stage-level rollout up to four times. Failed attempts and retries are discarded and never enter fine-tuning.

5 Experiments

5.1 Experiment Setups

Baselines We evaluate whether CoPR improves long-horizon agents as a data-synthesis paradigm rather than as a single-model artifact. The benchmarked base models include GLM-4.6 (Zeng et al., 2025), Kimi-K2-thinking (Team et al., 2025), DeepSeek-v3.2 (Liu et al., 2025), Qwen3-235B-A22B-Instruct (Yang et al., 2025a), and Qwen3-30B-A3B-Thinking-2507 (Yang et al., 2025a), covering different architectures and sizes. We fine-tune models with the slime framework (Zhu et al., 2025). Each +CoPR row is trained on the same 239 trajectories with the same hyperparameter configuration.

Dataset variants We compare CoPR with three external agentic datasets. SWE-Smith (Yang et al., 2025b) provides SWE-agent trajectories collected with Claude-3.7-Sonnet, CC-Bench (Zeng et al., 2025) provides trajectories collected with Claude Code and strong rollout models, and AFM CodeAgentDataset (Li et al., 2025b) provides code Q&A agentic trajectories. We also construct controlled internal variants. CoPR-SinglePR trains on isolated PR trajectories. CoPR-SinglePR-Success retains only successful isolated PR trajectories under the same filtering principle. CoPR-TemporalChain concatenates chronologically adjacent PRs from the same repository without explicit semantic dependency links. These controls separate the effects of data quantity, local PR success, chronological length, and semantic state evolution.

Table 3: Comprehensive comparison against model baselines and cross-model fine-tuning analysis. Every +CoPR row is fine-tuned on the same 239 CoPR samples.

Model	SWE-bench	Toolathlon	DS-1000	τ^2-bench retail	τ^2-bench airline	SciCode-MP	AVG
DeepSeek-v3.2	0.456	**0.250**	0.320	0.640	0.530	0.000	0.366
Kimi-K2-Thinking	0.318	0.213	0.521	0.702	0.610	0.062	0.404
Qwen3-235B	0.504	0.046	0.258	0.610	0.435	0.000	0.309
Qwen3-4B	0.098	0.009	0.256	0.338	0.285	0.000	0.164
+ CoPR	0.116 ↑ 18%	0.019 ↑ 111%	0.261 ↑ 2%	0.360 ↑ 7%	0.255 ↓ 11%	0.000	0.168 ↑ 2%
Qwen3-32B	0.298	0.019	0.398	0.536	0.415	0.015	0.280
+ CoPR	0.310 ↑ 4%	0.037 ↑ 95%	0.410 ↑ 3%	0.526 ↓ 2%	0.435 ↑ 5%	0.031 ↑ 107%	0.292 ↑ 4%
Qwen3-30B-A3B	0.242	0.037	0.370	0.579	0.510	0.031	0.295
+ CoPR	0.262 ↑ 8%	0.046 ↑ 24%	0.343 ↓ 7%	0.608 ↑ 5%	0.540 ↑ 6%	0.046 ↑ 48%	0.307 ↑ 4%
GLM-4.6	0.608	0.157	0.522	0.675	**0.620**	0.062	0.441
+ CoPR	**0.632** ↑ 4%	0.231 ↑ 47%	**0.526** ↑ 1%	**0.707** ↑ 5%	0.597 ↓ 4%	**0.154** ↑ 148%	**0.475** ↑ 8%

Benchmarks We evaluate long-horizon agentic capability on six benchmark domains. SWE-bench Verified (Jimenez et al., 2023) measures feature-level software engineering proficiency. Toolathlon (Li et al., 2025a) and τ^2-bench (Barres et al., 2025) evaluate tool use and interactive task completion. DS-1000 (Lai et al., 2022) and SciCode (Tian et al., 2024) evaluate file-level and research-level coding. SWE-bench uses SWE-Agent (Yang et al., 2024) as the execution scaffold. Toolathlon, DS-1000, and SciCode follow their original evaluation settings. For τ^2-bench, GLM-4.6 is used as the simulated user agent across evaluated models.

5.2 Main Results

Performance against Agent Datasets The results in Table 1 highlight data efficiency as a central advantage of CoPR, and the results of Qwen family are shown in Table 6. Despite using far fewer trajectories than SWE-Smith and CodeAgent, CoPR achieves stronger performance, suggesting that effective long-horizon supervision depends less on the number of trajectories than on the state-evolution structure of the trajectory itself. Loss comparison in Figure 8 between datasets presents that unlike SWE-Smith, which exhibits extremely low loss from the earliest training steps and CodeAgent, which has high loss at the latest step, CoPR rapidly converges from a high initial loss to a low-loss regime within only a few optimization steps. This suggests that CoPR provides a more efficient and informative supervision signal, which is consistent with its stronger downstream performance.

Impact of Data Curation Results in Table 3 show that $\text{CoPR}_{\text{SinglePR}}$, lacking cross-stage evolution within the PR mechanism, fails to provide supervision for state evolution, resulting in negligible gains in long-horizon agency. Meanwhile, the chronologically ordered $\text{CoPR}_{\text{TemporalChain}}$, despite lacking explicit semantic links, yields limited long-horizon benefits by implicitly reinforcing task decomposition through repository-level context. In contrast, our CoPR achieves superior performance with higher data efficiency. By preserving the precise logical dependencies of authentic code development, CoPR provides comprehensive supervision from real evolution of PRs, confirming that intrinsic evolutionary logic, not just sequence length, is the determinant for mastering complex agentic long-horizon tasks.

Table 2: **Evaluator classification ablation**. Precision remains high across thresholds and models.

Judge	Thr.	Acc.	Prec .	Rec.	F1
GLM-4.6	0.8	0.688	**0.941**	0.523	**0.672**
GLM-4.6	0.7	0.686	0.935	0.523	0.671
GLM-4.6	0.9	0.686	**0.941**	0.520	0.670
Kimi-K2	0.8	0.684	0.935	0.520	0.668

5.3 Comparison Analysis Against Model Baselines

We further examine whether the supervision signals from this state evolution can be transferred across model families. Table 3 compares base models with their CoPR-tuned counterparts. Every +CoPR row uses the same 239 examples, so the table measures whether CoPR is a reusable supervision signal rather than a GLM-specific artifact.

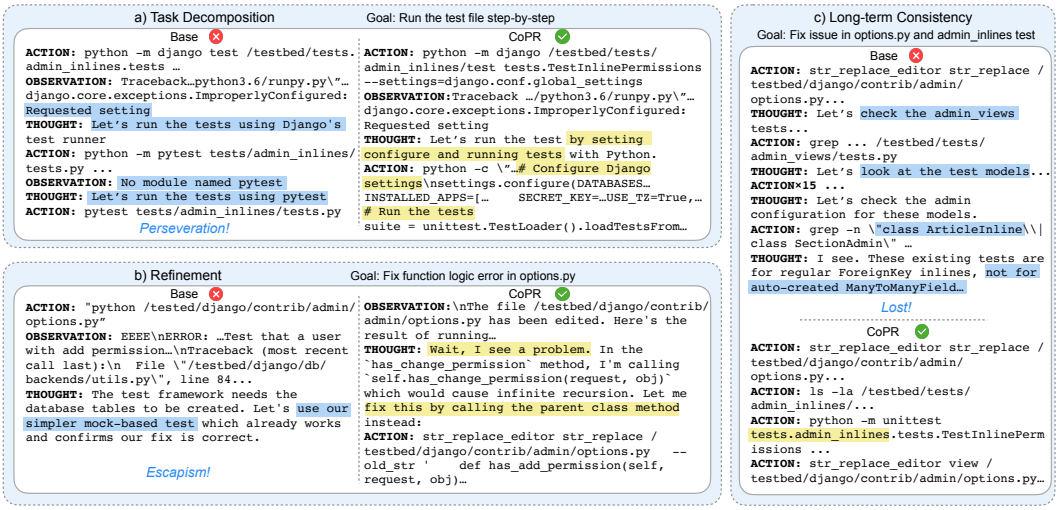

Figure 4: Case study on SWE-bench. CoPR improves decomposition, consistency, and refinement, while the baseline shows goal drift and escapism after errors.

The gains are not identical across benchmarks, which is expected because CoPR is not optimized for any particular test set. The important pattern is that the average score improves for all fine-tuned model families, including GLM-4.6 and multiple Qwen variants. This supports the interpretation that CoPR teaches general execution patterns such as decomposition, state carryover, verification, and refinement, rather than memorizing repository-specific patches.

5.4 Rejection Sampling Analysis

We study whether evaluator-based rejection sampling is reliable for constructing long-horizon rollouts. The evaluator is used only as a conservative admission filter for supervised data, where accepted-sample precision matters more than accuracy because false positives directly inject misleading trajectories into training. We use the model-generated patches from SWE-bench Verified trajectories produced by GLM-4.7 as test cases, and compare the classification accuracy and precision of the patch-based evaluator against real unit-test execution, as shown in Table 2. The evaluator maintains classification precision above 0.93 across thresholds and judge backbones shown in Table 2, supporting its use as a high-precision filter. Table 5 provides a comparison of CoPR performance on GLM-4.6, specifically analyzing the impact of whether or not to include relaxation techniques, indicating that filtering is essential for converting chain-of-prs rollouts into useful supervision.

6 Long-Horizon Analysis

6.1 Meta-Skills Performance Comparison

To visualize the acquisition of long-horizon agency, we analyze the execution trajectories of both GLM-4.6 and GLM-4.6-CoPR on SWE-bench Django Issue #11149 in Figure 4. The baseline exhibits failures that go beyond local coding errors. It drifts from the target by spending multiple turns in irrelevant modules such as `admin_views`, and it bypasses unresolved environment configuration issues with oversimplified scripts. These behaviors suggest a lack of state-aware decomposition, since the model neither maintains the evolving task state nor uses intermediate observations to constrain subsequent actions. GLM-4.6-CoPR shows the complementary behavior. It first stabilizes the configuration, then executes validation from general checks to more specific test cases, reflecting a structured decomposition aligned with the harness state. During code modification, it also performs an explicit refinement step by detecting a potential infinite recursion before execution and replacing the call with `super()`.

This case illustrates the capability gap targeted by CoPR. Baselines can often produce plausible local edits, but they struggle to preserve goals, respect evolving constraints, and repair delayed conse-

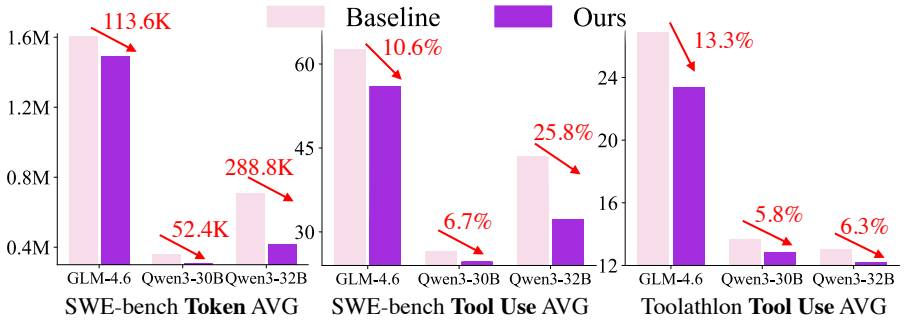

Figure 5: Token and tool-use efficiency on long-horizon tasks. CoPR-trained agents solve tasks with fewer redundant tokens and tool calls, indicating improved state management.

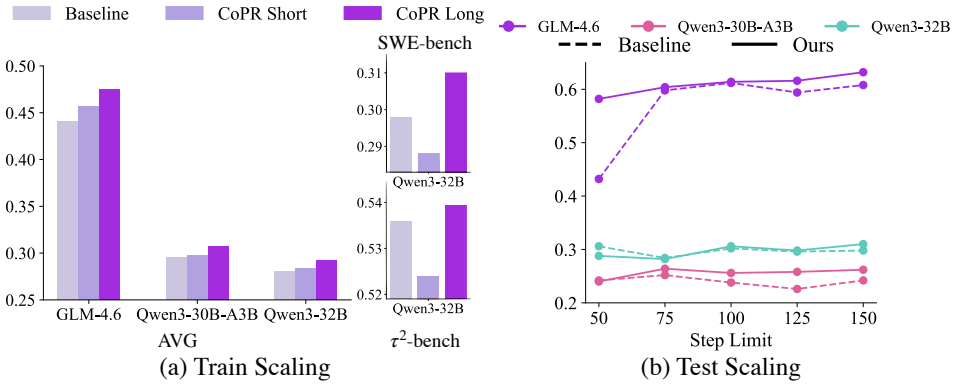

Figure 6: Performance scaling analysis across training horizons and inference budgets. **Left:** Training Horizon Scaling. **Right:** Inference-time Budget Scaling on SWE-Bench.

quences. CoPR helps to enhance these by training on chain-of-PRs supervision, where decomposition, consistency, and refinement are all grounded in realistic feature evolution.

6.2 Token Efficiency and Scaling Across Horizons

As illustrated in Figure 5, models trained on CoPR demonstrate a remarkable improvement in execution efficiency compared to the baseline. CoPR does not merely induce longer trajectories; it reduces redundant tokens and tool calls, indicating a more efficient execution policy under evolving state. As shown in Figure 6, longer training horizons consistently improve long-horizon performance, while CoPR-trained agents make more effective use of additional test-time compute. These results suggest that CoPR mitigates error accumulation over extended interaction sequences, enabling more stable and sustained problem solving under larger inference budgets.

7 Conclusion

To address the critical scarcity of interaction trajectories for long-horizon agentic tasks, we propose Chain of Pull Requests (CoPR), a long-horizon data synthesis paradigm. By mining the functional evolution inherent in real-world software, CoPR transforms discrete PRs into task chains characterized by explicit feature evolution dependencies. This paradigm provides rigorous structural supervision, enabling a model fine-tuned on merely 239 samples to substantially outperform baselines trained on massive datasets, yielding significant gains on rigorous benchmarks like Toolathlon. This finding not only challenges the reliance on sheer data scale but also unveils the dual scaling laws specific to training and inference in long-horizon tasks. Our work reveals the feasibility of overcoming single-feature synthesis by chains of pull requests, thereby charting a scalable evolutionary path for building long-horizon agents.

REFERENCES

Anthropic. Claude code overview. https://code.claude.com/docs, 2026. Official Claude Code documentation. Accessed April 12, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. 2025. URL https://arxiv.org/abs/2506.07982.

Yuxuan Cai, Lu Chen, Qiaoling Chen, Yuyang Ding, Liwen Fan, Wenjie Fu, Yufei Gao, Honglin Guo, Pinxue Guo, Zhenhua Han, et al. Nex-n1: Agentic models trained via a unified ecosystem for large-scale environment construction. *arXiv preprint arXiv:2512.04987*, 2025.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, et al. davinci-env: Open swe environment synthesis at scale. *arXiv preprint arXiv:2603.13023*, 2026.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. 2022. URL https://arxiv.org/abs/2211.11501.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, et al. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. *arXiv preprint arXiv:2510.25726*, 2025a.

Weizhen Li, Jianbo Lin, Zhuosong Jiang, Jingyi Cao, Xinpeng Liu, Jiayu Zhang, Zhenqiang Huang, Qianben Chen, Weichen Sun, Qiexiang Wang, Hongxuan Lu, Tianrui Qin, Chenghao Zhu, Yi Yao, Shuying Fan, Xiaowan Li, Tiannan Wang, Pai Liu, King Zhu, He Zhu, Dingfeng Shi, Piaohong Wang, Yeyi Guan, Xiangru Tang, Minghao Liu, Yuchen Eleanor Jiang, Jian Yang, Jiaheng Liu, Ge Zhang, and Wangchunshu Zhou. Chain-of-agents: End-to-end agent foundation models via multi-agent distillation and agentic rl, 2025b. URL https://arxiv.org/abs/2508.13167.

Jifan Lin, Xiaojie Cai, Yang Xiao, Yumin Zhuang, Qishuo Hua, Junfei Wang, and Pengfei Liu. Sii cli: Building next-generation cognitive agentic intelligence ecosystem. https://github.com/GAIR-NLP/SII-CLI, 2025. Technical Report on Cognitive Agentic Intelligence Framework.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Haotian Luo, Huaisong Zhang, Xuelin Zhang, Haoyu Wang, Zeyu Qin, Wenjie Lu, Guozheng Ma, Haiying He, Yingsha Xie, Qiyang Zhou, et al. Ultrahorizon: Benchmarking agent capabilities in ultra long-horizon scenarios. *arXiv preprint arXiv:2509.21766*, 2025.

Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

Ansh Radhakrishnan, Karina Nguyen, Anna Chen, Carol Chen, Carson Denison, Danny Hernandez, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, et al. Question decomposition improves the faithfulness of model-generated reasoning. *arXiv preprint arXiv:2307.11768*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists. 2024. URL https://arxiv.org/abs/2407.13168.

Shangzi Xue, Zhenya Huang, Jiayu Liu, Xin Lin, Yuting Ning, Binbin Jin, Xin Li, and Qi Liu. Decompose, analyze and rethink: Solving intricate problems with human-like reasoning cycle. *Advances in Neural Information Processing Systems*, 37:357–385, 2024.

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

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

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 LIMITATIONS

Currently, constrained by rollout success rates, our implementation connects a maximum of 5 PRs to construct task chains. Our training-horizon scaling results suggest that extending these chains is a promising path for continuous improvement, but doing so requires stronger stage-wise agents and more reliable evaluators. Another limitation is that the current repository set contains nine high-quality projects; while it spans multiple languages and domains, it is not exhaustive. Future work should scale to more repositories and non-code domains with similarly verifiable state transitions.

B BROADER IMPACT

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

C REPOSITORIES IN COPR

The source repositories form the empirical basis of CoPR and determine the diversity of software-evolution trajectories available for chain construction. We therefore report both the number of PRs processed from each project and the number of intent-based task queries finally produced after filtering, since different repositories vary substantially in PR density, patch structure, and evaluator-verifiable outcomes.

Table 4: List of the source repositories used in CoPR. The table lists the repository names, the total number of Pull Requests processed (PR Num), and the final number of constructed task queries (Query Num)

Repo ID	Repo Name	PR Num	Query Num
313853293	cloudquery/cloudquery	9705	490
62117812	apache/pulsar	9803	228
1460385	scipy/scipy	9511	238
523043277	astral-sh/ruff	8115	180
139199684	PrefectHQ/prefect	8433	215
908607	numpy/numpy	9355	354
680120071	microsoft/autogen	1467	18
330914717	ivy-llc/ivy	4160	9
543276238	tursodatabase/libsql	1261	30
SUM		61810	1762

Table 4 lists the nine source repositories used in CoPR. The selected repositories cover scientific computing, distributed systems, developer tooling, and high-performance systems programming, allowing the synthesized chains to include heterogeneous programming languages, execution environments, and development patterns.

D REJECTION SAMPLING ABLATION

The combined filtering-and-rollout analysis in Table 5 further connects evaluator quality to the practical generation pipeline. Panel A measures the downstream effect of retaining only evaluator-approved trajectories, while Panel B checks whether the rollout success rate is sensitive to the attempt and retry budgets used during data synthesis.

To conduct rollout, a setup using 128 NVIDIA H200 GPUs requires approximately 8 days of inference time. In our study, we utilized 64 H200 GPUs for the training phase. For deployment and experimental testing, we consistently used 32 H200 GPUs.

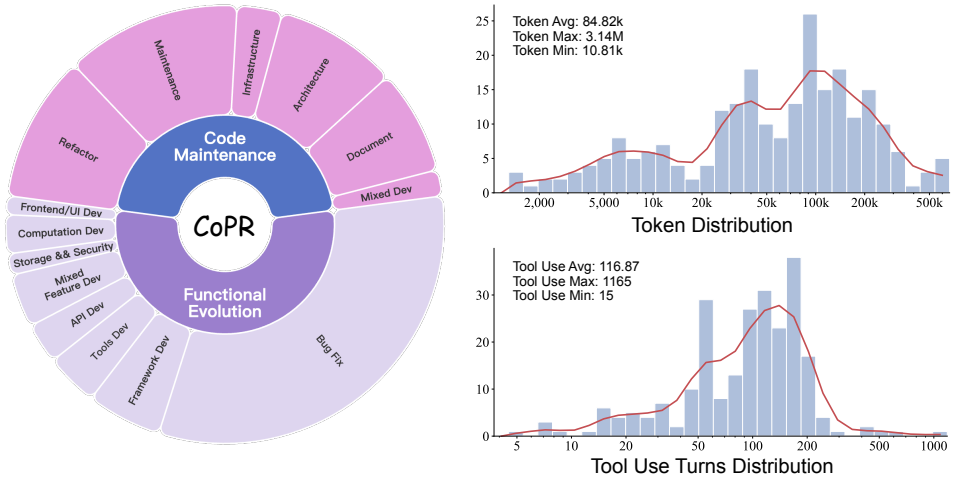

Figure 7: Characteristics of CoPR data. Left: Domain coverage across CoPR. Right: Distributions of trajectory length and tool utilization illustrating the significant complexity inherent in long-horizon agentic tasks.

Table 5: Evaluation of trajectory filtering and rollout attempts. Panel A measures the downstream effect of rejection sampling. Panel B measures the sensitivity of rollout success to attempt and retry budgets on 30 sampled queries containing 210 PR stages.

(a) Rejection-sampling evaluation.

Model	Query	SWE-bench	Toolathlon	DS-1000	τ^2-Retail	SciCode-MP	AVG
GLM-4.6	–	0.608	0.157	0.522	0.675	0.062	0.405
+ No Rejection	1,181	0.090 -0.518	0.000 -0.157	0.322 -0.200	0.568 -0.107	0.046 -0.016	0.205 -0.200
+ Rejection	1,181	0.618 +0.010	0.144 -0.013	0.544 +0.022	0.632 -0.043	0.169 +0.107	0.421 +0.016

(b) Rollout-attempt sensitivity.

Attempt	Retry	Success
2	4	0.067
3	4	**0.076**
4	4	**0.076**
3	3	0.067
3	5	0.071

The rejection-sampling study separates the effect of long-horizon data quantity from the effect of long-horizon data quality. As shown in Table 5, training on unfiltered trajectories can be actively harmful, whereas evaluator-filtered trajectories provide higher-fidelity supervision and lead to more stable downstream gains.

E Training Loss Comparison

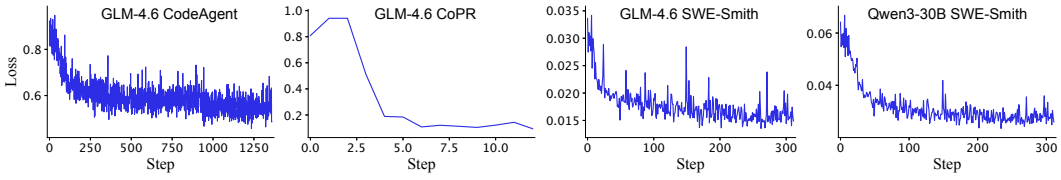

Figure 8: Training loss comparison between different baseline datasets. Model is struggle to converge CodeAgent data, with a loss below 0.5. SWE Smith showed extremely low losses from the outset across different models, while CoPR achieved low losses within only 12 steps.

F QWEN DATASET ABLATION

To test whether the benefits of CoPR are tied to a single backbone model, we repeat the dataset comparison on Qwen3-30B-A3B. Table 6 shows that the same qualitative trend holds beyond GLM-4.6: full PR-chain supervision is more effective than single-PR or purely temporal-chain variants.

Table 6: Qwen3-30B-A3B comparison across external datasets and CoPR variants. The trend mirrors GLM-4.6: full CoPR outperforms single-PR and temporal-chain variants. Lower-right values in the CoPR row are changes over the Qwen3-30B-A3B base model. Tiny bottom-right annotations in the highlighted row report absolute changes over the Qwen3-30B-A3B base model.

Training Data	**SWE-bench**	**Toolathlon**	**DS-1000**	**τ^2-bench retail**	**τ^2-bench airline**	**SciCode-MP**	**AVG**
Qwen3-30B-A3B	0.242	0.037	**0.370**	0.579	0.510	0.031	0.295
SWE-Smith	0.158	0.019	0.195	0.233	0.305	0.015	0.154
CC-Bench	0.200	0.000	0.090	0.567	0.455	0.000	0.219
CodeAgent	0.130	0.009	0.075	0.202	0.275	0.000	0.115
CoPR-SinglePR-Success	0.204	0.037	0.084	0.549	0.521	0.031	0.238
CoPR-TemporalChain	0.182	0.037	0.091	0.531	0.455	0.000	0.216
CoPR	**0.262** +0.020	**0.046** +0.009	0.343 -0.027	**0.608** +0.029	**0.540** +0.030	**0.046** +0.015	**0.307** +0.012

G WITHIN-STAGE ATTEMPT PERFORMANCE

During chain construction, each PR stage may be attempted multiple times to improve the chance of producing a valid patch. Table 7 reports that three attempts provide a strong balance between downstream performance and generation cost, with little additional benefit from increasing the budget further.

Table 7: Effect of the number of within-stage attempts on downstream performance using 105 samples. Performance is stable, and 3 attempts balances quality and rollout efficiency.

Attempt	**Samples**	**DS-1000**	**τ^2-Retail**	**SciCode-MP**
2	105	0.526	0.663	0.092
3	105	**0.529**	**0.680**	**0.105**
4	105	**0.529**	0.676	**0.105**

H PIPELINE ABLATIONS

Repository-diversity scaling Repository diversity affects whether the synthesized trajectories expose the model to a broad range of engineering patterns. Table 8 compares a smaller Python-focused pool with a more diverse multi-language pool, indicating that adding repositories from different domains improves generalization in the exploratory setting.

Table 8: Repository-diversity scaling in exploratory experiments with 105 samples. Expanding from three Python-only repositories to five repositories including Java and Rust improves downstream performance.

Num. Repos	**Samples**	**DS-1000**	**τ^2-Retail**	**SciCode-MP**
3	105	0.491	0.561	0.077
5	105	**0.529**	**0.680**	**0.105**

Run consistency Since agentic benchmarks can exhibit rollout-level variance, we evaluate GLM-4.6 + CoPR with two independent runs. Table 9 reports both runs and the conservative value used in the main tables, ensuring that headline results do not rely on a single favorable rollout.

Table 9: Two independent evaluation rollouts for GLM-4.6 + CoPR. We report the lower result in the main tables when applicable, yielding a conservative estimate.

Rollout	SWE-bench	Toolathlon	DS-1000	τ^2-Retail	τ^2-Airline	SciCode-MP
Rollout 1	0.636	0.231	0.526	0.707	0.597	0.169
Rollout 2	0.632	0.231	0.532	0.717	0.614	0.154
Reported	0.632	0.231	0.526	0.707	0.597	0.154

I AgencyBench Code

AgencyBench Code evaluates multi-turn, long-horizon coding behavior in an external benchmark setting. Table 10 shows that GLM-4.6 fine-tuned with CoPR improves over the backbone and competitive open-source baselines, suggesting that PR-chain supervision transfers beyond the benchmarks used for development.

Table 10: Performance on multi-turn, long-horizon tasks in AgencyBench Code.

Model	AgencyBench
GLM-4.6	11.9
DeepSeek-v3.2	11.6
Kimi-K2-Thinking	11.8
Qwen3-235B	4.6
GLM-4.6-CoPR	**15.9**

J Query Constructor Prompt

The following prompt is used to convert raw PR information into an intent-based coding query for the rollout agent. Its purpose is to preserve the functional motivation of each PR while avoiding leakage of implementation-specific identifiers or step-by-step patch instructions.

Query Constructor Prompt

Role
You are an AI Model Test Case Designer. Your expertise lies in reverse-engineering code changes to create an “intent-based query" used for benchmarking AI code generation models.

Core Task
Your task is to analyze the provided Pull Request (PR) data and generate an intent-based query.

This query serves one purpose: It will be the input prompt for a separate AI coding agent. The test is to benchmark if that agent can reproduce the functional changes described in the original PR data.

Key Rules & Constraints

Core Thought Process
You must follow this 4-step analysis to generate the query:
- Identify Core Intent (The “Why"): Analyze the PR title and description to find the fundamental problem being solved or the strategic goal.
- Pinpoint Conceptual Locus (The “Where"): Examine the patch to identify the modified

system area (files or classes). You must inform the file paths to be executed and conceptually explain the variables involved.
- Abstract Key Logic (The “How"): Analyze the patch and execution plan. Abstract the key logic (e.g., variables, conditional logic) into a “vague plan," noting only the logical associations between functional areas.
- Synthesize Directive (The “Plan"): Combine the "Why," "Where," and "How" into a single, high-level narrative. Start with the goal (Step 1), guide the AI to the correct conceptual area (Step 2), and explain the required functional change using the abstracted logic (Step 3). This plan must focus purely on intent.

Key Constraints
The query you generate MUST be a “conceptual reverse-instruction".
- It MUST capture the core "why" (strategic intent or functional goal) behind the PR, not the "what" (the specific code changes).
- It MUST contain enough conceptual information to guide the AI to a correct implementation.
- It MUST specify file paths for execution and explain the purpose (not the literal name) of variables, but it cannot include detailed execution instructions.

Strictly Prohibited
- DO NOT generate step-by-step instructions, operational lists, or bullet points.
- DO NOT mention literal function names (e.g., calculate_cost()), class names, or variable names from the code.
- DO NOT include specific URLs or code snippets.

Example
Input:
- Repository: microsoft/autogen
- PR Title: Refined the `user_proxy` description.
- PR Description: ## Why are these changes needed?

Refined the user_proxy description in cases where it is fully automated. Too often, other agents assumed it could answer on behalf of a user in human_input=NEVER mode, but it cannot.

- Patch JSON: [“path": "autogen/agentchat/user_proxy_agent.py", “patch": "@@ -20,7 +20,7 @@ class UserProxyAgent(ConversableAgent): DEFAULT_USER_PROXY_AGENT_DESCRIPTIONS = “ALWAYS": “An attentive HUMAN user who can answer questions about the task, and can perform tasks such as running Python code or inputting command line commands at a Linux terminal and reporting back the execution results.", “TERMINATE": “A user that can run Python code or input command line commands at a Linux terminal and report back the execution results.",- “NEVER": “A user that can run Python code or input command line commands at a Linux terminal and report back the execution results.",+ “NEVER": “A user that CANNOT WRITE CODE, or answer other questions, but CAN EXECUTE PYTHON CODE and CAN INPUT COMMAND LINE sh COMMANDS at a Linux terminal, and REPORT BACK THE EXECUTION RESULTS.", def __init__(", “path": “autogen/agentchat/user_proxy_agent.py", “patch": "@@ -20,7 +20,7 @@ class UserProxyAgent(ConversableAgent): DEFAULT_USER_PROXY_AGENT_DESCRIPTIONS = “ALWAYS": “An attentive HUMAN user who can answer questions about the task, and can perform tasks such as running Python code or inputting command line commands at a Linux terminal and reporting back the execution results.", “TERMINATE": “A user that can run Python code or input command line commands at a Linux terminal and report back the execution results.",- “NEVER": “A user that CANNOT WRITE CODE, or answer other questions, but CAN EXECUTE PYTHON CODE and CAN INPUT COMMAND LINE sh COMMANDS at a Linux terminal, and REPORT BACK THE EXECUTION RESULTS.",+ “NEVER": “A computer terminal that performs no other action than running Python scripts (provided to it quoted in “‘python code blocks), or sh shell scripts (provided to it quoted in

"'sh code blocks).", def __init__("]
- Specific Execute Plan: I need to update the user_proxy agent's description to clearly state that it cannot provide user-level responses when human_input is set to NEVER, ensuring other agents don't mistakenly rely on it for input in fully automated workflows. I'll start by modifying the docstring and internal prompt in user_proxy_agent.py to reflect this limitation more explicitly, based on recent team discussions. The main challenge will be phrasing the message clearly without introducing ambiguity, so I'll prioritize precision and consistency with our agent communication patterns.

Output:
In a multi-agent system, it is crucial to prevent other agents from misinterpreting a fully automatic UserProxyAgent as a conversational entity, such as a human or an LLM, especially when its human_input_mode is set to 'NEVER'. In this mode, the agent's sole function is code execution. Your task is to examine the operational logic within autogen/agentchat/user_proxy_agent.py. Focus on the UserProxyAgent's initialization process to identify and adjust the key variable storing its default descriptions. You must modify the default description specifically for the 'NEVER' mode to unambiguously define its role as a "code terminal," making it clear that it only executes code and cannot process invalid natural language requests, such as seeking opinions or confirmations.

Pull Request (PR) data
- Repository: {REPO_NAME}
- PR Title: {PR_TITLE}
- PR Description: {PR_DESCRIPTION}
- Patch JSON: {PATCH_JSON}
- Specific Execute Plan: {REASONING}

Output Format
- Your response MUST ONLY contain the generated 'query' string.
- Your answer MUST be in ENGLISH.
- Do not include any additional explanations, apologies, or preambles (e.g., "Here is the query you requested...").

K EVALUATOR PROMPT

The evaluator prompt defines the semantic acceptance criterion used during rejection sampling. Rather than requiring token-level or syntactic identity with the reference PR, it asks the judge to focus on functional equivalence, critical bugs, and requirement violations so that valid alternative implementations can still be retained.

Evaluator Prompt

 # Code Model Judge Prompt

You are an expert code reviewer tasked with evaluating whether an AI-generated patch achieves the same functional effect as a reference GitHub PR patch. Your role is to identify errors, bugs, and deviations from requirements while recognizing that different implementations can be functionally equivalent.
Input Format

You will receive the following inputs:

Original Ground-Truth GitHub PR patch:

{{ground_truth_patch}}

****AI-generated patch:****

{{diff_patch}}

****User query:****

{{test_query}}

Evaluation Criteria

What to Consider as CORRECT
- **Functional equivalence**: Different implementation approaches that achieve the same end result
- **Code style variations**: Different variable names, code organization, or formatting
- **Enhanced implementations**: AI adds extra features or improvements not explicitly forbidden
- **Alternative logic structures**: Different conditional flows that produce identical outcomes
- **More comprehensive tests**: AI writes additional test cases beyond minimum requirements

What to Flag as ERRORS

1. **Critical Bugs**
- Runtime errors (infinite recursion, null pointer exceptions, etc.)
- Logic errors that would cause incorrect behavior
- Security vulnerabilities
- Performance issues that significantly impact functionality

2. **Requirement Violations**
- Missing required functionality explicitly stated in the user query
- Implementing opposite behavior from what was requested
- Ignoring explicit constraints or specifications
- Breaking existing functionality that should be preserved

3. **API Contract Violations**
- Wrong method signatures when specifically required
- Missing required return values
- Incorrect exception types when specified
- Breaking changes to public interfaces

Analysis Framework

Step 1: Functional Correctness Analysis
1. **Core functionality**: Does the AI implementation achieve the primary objective?
2. **Edge cases**: Are boundary conditions and error scenarios handled correctly?
3. **Integration**: Does it work correctly with existing code?
4. **Side effects**: Are there unintended consequences?

Step 2: Requirement Compliance Check
1. **Explicit requirements**: Check each requirement from the user query
2. **Implicit requirements**: Consider standard practices and conventions
3. **Constraints**: Verify all limitations and restrictions are respected
4. **Test coverage**: Ensure all specified test scenarios are implemented

Step 3: Bug Detection
1. **Syntax errors**: Check for compilation/parsing issues
2. **Runtime errors**: Look for potential crashes or exceptions
3. **Logic errors**: Verify algorithmic correctness
4. **Resource issues**: Check for memory leaks, infinite loops, etc.

Final Response Format

Return a **strict JSON object** with the following shape (no additional keys or prose):

json
{ “score": 0.0,
“comment": "Short summary that references the most critical findings or explicitly states the patch is acceptable."
}

- ‘score’ must be a floating-point number between **0.0** and **1.0**. Use **1.0** only when the patch is acceptable with no critical errors; use values close to **0.0** when you discover blocking issues.
- ‘comment’ must be a concise natural-language explanation describing your overall verdict (e.g., highlight the key blocking issue or state that the patch aligns with the ground truth).

Do **not** wrap the JSON in Markdown fences, and do **not** include any additional narrative outside the JSON object.

L TRAINING PARAMETERS

For reproducibility, we provide the main supervised fine-tuning, parallelism, and optimizer settings used in our experiments. These parameters specify the rollout interface, batch construction, distributed training configuration, and optimizer schedule used to train on CoPR.

Training Parameters

```
SFT_ARGS=(
−rollout-function-path slime.rollout.sft_rollout.generate_rollout
−prompt-data $input_path
−input-key messages
−tool-key tools
−rollout-shuffle
−num-epoch 2
−rollout-batch-size 64
−global-batch-size 64

−loss-type sft_loss
−calculate-per-token-loss
−disable-compute-advantages-and-returns
−debug-train-only
)

PERF_ARGS=(
−tensor-model-parallel-size 8
−sequence-parallel
−pipeline-model-parallel-size 4
−context-parallel-size 2
−expert-model-parallel-size 16
```

```
−expert-tensor-parallel-size 1

−recompute-granularity full
−recompute-method uniform
−recompute-num-layers 1
−use-dynamic-batch-size
−max-tokens-per-gpu 65536
)

OPTIMIZER_ARGS=(
−optimizer adam
−lr 2e-6
−lr-decay-style cosine
−min-lr 1e-7
−lr-warmup-fraction 0.1
−weight-decay 0.1
−adam-beta1 0.9
−adam-beta2 0.98
−optimizer-cpu-offload
−overlap-cpu-optimizer-d2h-h2d
−use-precision-aware-optimizer
)
```