Beyond Tests: Benchmarking and Training Long-Horizon Coding Agents with Agentic Judgments

Anonymous ACL submission

Abstract

Coding models are increasingly used in real-world software development, but existing benchmarks still emphasize function-level programming, bug fixing, or loosely connected long-horizon tasks. This leaves two important settings underexplored: building projects from high-level user requests and evolving existing projects through related feature development. We introduce CODEFOUNDRY, a benchmark and training framework with 200 tasks from 200 repositories across 8 programming languages, covering both from-scratch project construction and multi-step project evolution, with UX variants that preserve high-level user requirements. To evaluate such open-ended implementations, we combine executable tests with an agent-as-judge protocol based on behavior-oriented rubrics, and train CODEFOUNDRY-JUDGE-35B, which outperforms GLM-5.1 on repository-level judging. Experiments show that CODEFOUNDRY remains challenging, with frontier coding models achieving below 30% overall pass@1. Extensive analyses show that task formulation shapes agent efficiency, failure modes, and the design of reliable Judge evaluation protocols. We further train CODEFOUNDRY-35B, showing that executable-test supervision and rubric-based judgment can jointly improve long-horizon coding agents. We hope CODEFOUNDRY will support more realistic evaluation and training of coding agents for open-ended software development.

1 Introduction

Recent advances in large language models (LLMs) have enabled coding agents that plan, edit, execute, and validate code inside interactive development environments (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025). This raises a central question: *can they build runnable projects from human requirements and evolve existing projects through coordinated feature development?*

Existing coding benchmarks remain limited in answering this question. Short-horizon benchmarks such as HumanEval (Chen et al., 2021) and SWE-bench (Jimenez et al., 2024) primarily evaluate standalone code generation, bug fixing, or localized modification. Recent repository-level benchmarks, including SWE-bench Pro (Deng et al., 2025), SWE-EVO (Thai et al., 2025), NL2Repo (Ding et al., 2025), and Roadmap-Bench (Xu et al., 2026), increase task scale, but often extend horizons by concatenating weakly related subtasks (Thai et al., 2025; Xu et al., 2026) or make tasks testable by prescribing project structure, APIs, test boundaries, and detailed requirements (Ding et al., 2025). These settings differ from how users often initiate development: through high-level, ambiguous goals that admit multiple valid designs. As a result, test execution alone is insufficient for judging whether an agent has satisfied user intent in open-ended software development.

To address these gaps, we introduce CODEFOUNDRY, a long-horizon coding benchmark with 200 tasks across 200 repositories and 8 programming languages. CODEFOUNDRY contains four complementary task families: GREENFIELD, GREENFIELD-UX, BROWNFIELD, and BROWNFIELD-UX. GREENFIELD tasks require agents to build projects from an empty workspace, while BROWNFIELD tasks require agents to evolve existing real-world projects from one stage to the next. The UX variants preserve high-level and underspecified user requirements, requiring models to choose project architecture, APIs, and implementation strategies rather than following a fully specified target solution. Together, these tasks evaluate both project construction and project evolution under realistic degrees of design freedom.

Because such tasks admit diverse valid implementations, we further propose an *agent-as-judge* evaluation paradigm for repository-level coding tasks. Instead of relying only on fixed tests or a

Benchmark	Language	Code	Task Type		Instruction		Code Gen.	Evaluation		Training
	Scope	LOC	0→1	1→n	Doc.	Human	Free	Tests	ASJ	Trainable
SWE-bench Verified (Chowdhury et al., 2024)	Python	~ 33	✗	✓̸	✗	✓	✗	✓	✗	✗
SWE-bench Pro (Deng et al., 2025)	Multi	~ 107	✗	✓	✓	✗	✗	✓	✗	✗
FeatureBench (Zhou et al., 2026)	Python	~ 790	✗	✓̸	✓	✗	✗	✓	✗	✗
TerminalBench (Merrill et al., 2026)	Multi	~ 280	✓	✓̸	✗	✓̸	✓̸	✓	✗	✗
SWE-EVO (Thai et al., 2025)	Python	~ 611	✗	✓	✓	✗	✗	✓	✗	✗
NL2Repo (Ding et al., 2025)	Python	$\sim 3,000$	✓	✗	✓	✗	✗	✓	✗	✗
ROADMAPBENCH (Xu et al., 2026)	Multi	$\sim 3,700$	✗	✓	✓	✗	✗	✓	✗	✗
CodeFoundry (ours)	**Multi**	$\boldsymbol{\sim 4,500}$	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison between representative coding benchmarks and CodeFoundry (ours). ✓, ✓̸, and ✗ indicate fully addressed, partially addressed, and not addressed, respectively.

single reference solution, agent-as-judge uses structured rubrics to inspect complete repositories and assess functional behavior, requirement satisfaction, and design quality. Based on this paradigm, we train CODEFOUNDRY-JUDGE-35B, which outperforms GLM-5.1 in repository-level judging.

Experiments show that even the strongest current coding agents remain below 30% pass@1 on both GREENFIELD and BROWNFIELD, indicating that long-horizon, design-sensitive, and underspecified software development remains far from solved. We also find that executable tests and agent-as-judge provide complementary supervision: training with both signals yields CODEFOUNDRY-35B, which achieves leading performance under both test-based and agent-judged evaluation and generalizes strongly to external benchmarks.

Overall, our main contributions are as follows:

- We introduce CODEFOUNDRY, a benchmark that advances coding-agent evaluation from local code generation and defect repair to long-horizon, open-ended software development.
- We propose an *agent-as-judge* repository-level evaluation paradigm and train CODEFOUNDRY-JUDGE-35B for assessing open-ended implementations with diverse valid solutions.
- We reveal key capability bottlenecks of frontier coding agents and show that executable tests and agent-as-judge provide complementary supervision for training stronger long-horizon coding agents.

2 Related Work

Coding Benchmarks Coding benchmarks have progressed from short, self-contained generation tasks such as HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2025) to repository-level issue resolution in SWE-bench (Jimenez et al., 2024) and SWE-bench Verified (Chowdhury et al., 2024). Recent benchmarks further expand the horizon to complex repository changes, repository generation, and multi-step software evolution (Deng et al., 2025; Zhou et al., 2026; Ding et al., 2025; Thai et al., 2025; Xu et al., 2026). These efforts make coding evaluation more realistic, but still often emphasize localized modification, fixed interfaces, or highly specified targets, as summarized in Table 1. CODEFOUNDRY complements them by covering both from-scratch construction and brownfield evolution, with instructions ranging from detailed specifications to human-like user requests.

Coding Agent Coding agents are shifting from conversational code completion to situated software-development systems that operate over repositories, terminals, tests, and external tools. Systems such as SWE-agent (Yang et al., 2024), AutoCodeRover (Zhang et al., 2024), and OpenHands (Wang et al., 2025) demonstrate repository search, editing, execution, and iterative repair, while Claude Code and OpenAI Codex bring similar workflows into practical environments (Anthropic, 2026; OpenAI, 2025). As agents are used for user-facing engineering tasks, evaluation must also move beyond local patch correctness toward alignment with user intent: agents need to interpret high-level requests, preserve user-visible behavior, make reasonable design choices, and validate project-level changes. CODEFOUNDRY follows this direction through UX task variants and through training supervision from both executable tests and agent-as-judge feedback.

3 CodeFoundry

CODEFOUNDRY is constructed from real repositories and real project evolution histories. The pipeline first mines candidate sources from GitHub, then turns them into executable task instances, and finally generates either detailed implementation instructions or UX-style user requests, as illustrated in Figure 1.

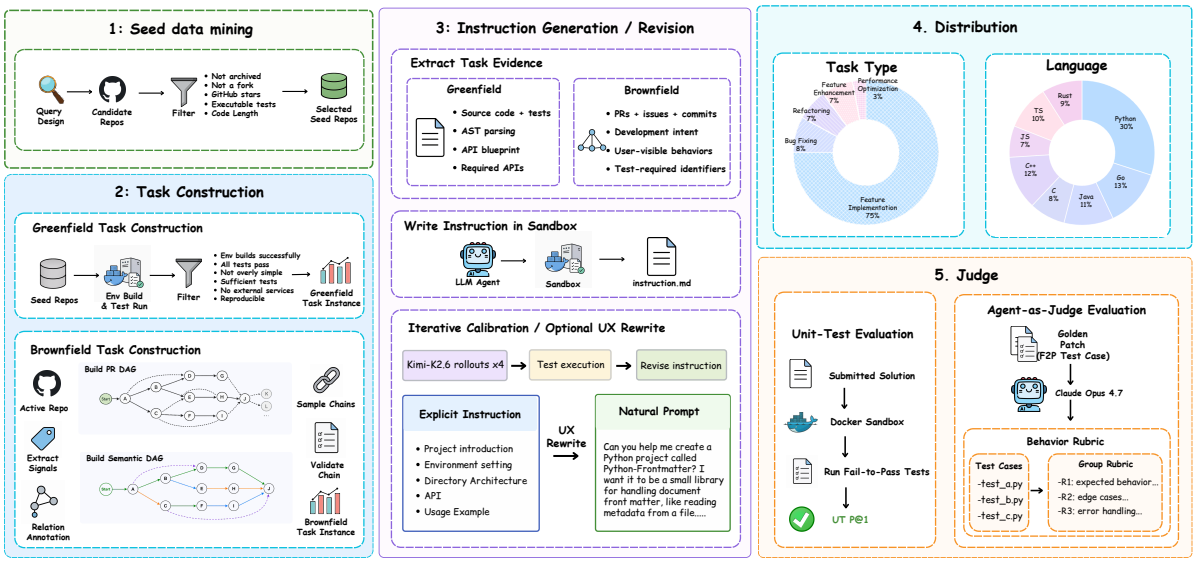

Figure 1: **Overview of CODEFOUNDRY.** The pipeline mines real repositories, constructs Greenfield and Brownfield tasks, calibrates task instructions, and optionally rewrites them into UX-style prompts. Submissions are evaluated with executable tests or agent-as-judge scoring.

3.1 Greenfield tasks

Greenfield tasks evaluate project construction from an empty workspace. We mine repositories that cover diverse development scenarios, project types, and language ecosystems. For each domain, we issue multiple GitHub search queries from different perspectives, including topic tags, description keywords, and repository naming patterns. We then filter repositories by activity, popularity, code size, test availability, and whether their public interfaces can be parsed into an API blueprint.

Each retained repository must also be reproducible. We construct its execution environment and run the original test suite, keeping only projects whose tests pass in the constructed environment. This step removes repositories that are overly simple, insufficiently tested, dependent on external services, or difficult to rebuild reliably. For each validated repository, we derive a from-scratch task by extracting the public API surface and identifying the subset of interfaces exercised by tests. This produces the required API surface for test-based evaluation while still allowing the implementation to be written from scratch. An instruction-generation agent then inspects the repository in a sandbox and produces a detailed `instruction.md` that describes the project goal, dependencies, expected directory structure, required interfaces, and behavioral constraints. We further check the generated instruction with source-file coverage and required-API matching signals before retaining the task.

3.2 Brownfield tasks

Brownfield tasks evaluate multi-step project evolution in an existing codebase. We start from active repositories with rich pull-request and issue histories and retain merged code changes as the source of task construction. From these changes, we build a directed PR graph using rule-based signals from commit topology, textual references, branch information, and code-coupling patterns. Starting from each PR, we traverse up to 10 downstream nodes in the graph and annotate the resulting candidate PR pairs with semantic relations, prioritizing strongly related categories such as `Same_Feature`, `Dependency`, and `Follow_Up`. These labels are used as semantic edge weights so that selected chains capture coherent evolution rather than unrelated issue concatenation. The rule-based PR edge signals are summarized in Table 5.

Candidate PR chains are sampled from the annotated graph under bounded length and changed-code constraints. To ensure that a chain corresponds to a recoverable development task, we create a working tree from the base commit of the first PR and apply each base-to-merge-commit diff in chain order. If any patch fails to apply, the chain is discarded. For chains that pass patch application, we build the Docker environment and run the final test suite, retaining only chains whose environments can be built and whose tests all pass. The surviving chains are converted into single coherent development instructions that preserve the original

symptoms, motivations, and user-visible behaviors from the underlying PRs and issues.

3.3 Instruction calibration and UX rewriting

Detailed Greenfield and Brownfield instructions intentionally expose the interfaces, parameter names, return semantics, constants, and attributes needed for executable test evaluation. To reduce instruction–test mismatch, we run four independent Kimi K2.6 rollouts for each task and execute the associated tests in a sandboxed environment. Opus 4.7 then analyzes failed executions to distinguish model limitations from underspecified or inconsistent task descriptions. When a mismatch is found, only the instruction is revised, and the task is retained only after an additional Opus 4.7 verification pass confirms that the mismatch has been resolved.

We additionally create Greenfield-UX and Brownfield-UX variants by rewriting the detailed instructions into more natural user-facing requests. These UX instructions describe missing capabilities, abnormal behaviors, user pain points, or desired project evolution while omitting test-facing identifiers and implementation details, including public identifiers, parameter names, default values, return semantics, public constants, and public attributes directly used by tests. This yields tasks that better reflect realistic interaction while preserving the same underlying validated repositories and development targets.

4 Evaluation

CodeFoundry evaluates coding agents with two complementary signals: unit-test execution and agentic judgment. Unit tests measure functional correctness for Greenfield and Brownfield tasks with explicit APIs and expected behaviors (§ 4.1), while agent-as-judge assesses UX-oriented tasks whose high-level user requests leave implementation details underspecified (§ 4.2).

4.1 Unit-Test Evaluation

For Greenfield and Brownfield tasks, we evaluate each submitted solution by executing the task-specific test suite in a sandboxed Docker environment. The primary signal is the fail-to-pass test set, which is constructed from the original repository tests or from the validated pull-request chain and is expected to fail before the task and pass on the reference implementation. We report unit-test pass@1, denoted as UT P@1. A task is considered solved if a single agent rollout passes all fail-to-pass tests while preserving the existing behavior covered by the regression tests.

4.2 Agent-as-Judge Framework

Rubric Generation. As shown in the first step of Figure 2, we construct a natural-language rubric for each software engineering task using Claude Opus 4.7 from the original user task, the golden patch, the test patch, and the fail-to-pass test cases. We use an n-to-1 aggregation strategy, where multiple related test cases are compressed into one functional group rather than producing a separate rubric for each test case. Each group captures a high-level requirement and contains fine-grained rubric items. The rubric is behavior-oriented: it describes the expected external behavior without requiring the implementation to reproduce the golden patch or use specific function names, class names, or parameter names.

SFT Training. In the Judge training stage of Figure 2, we first build SFT data by prompting GLM-5.1, Kimi-K2.6, and GPT-OSS-120B to solve 400 software engineering tasks, with 4 rollouts per model for each task. We also include the golden patch as a positive implementation. For each sample, we save the Docker environment and final code state, run both fail-to-pass and regression tests, and use the test outcome as the ground truth. After downsampling incorrect samples, we obtain around 2000 Docker environments for Judge training.

We then feed the final code state and corresponding rubric into GLM-5.1 for Judge annotation, with 4 Judge rollouts per sample. The Judge only uses the final code state and rubric, without access to the golden patch or agent trajectory. A sample is retained for SFT only when the Judge prediction exactly matches the ground-truth test result, yielding around 1,500 high-confidence Judge samples after filtering.

RL Training. After SFT, the Judge training stage in Figure 2 continues with GRPO. For each training sample, the model generates multiple candidate Judge outputs. Each candidate output is evaluated by comparing its group-level predictions against the ground-truth group labels, and the resulting reward is used for GRPO optimization.

Each rubric contains a set of functional groups G. The ground-truth label y_g for each group $g \in G$ is derived from its associated fail-to-pass tests: the group is positive only if all associated tests

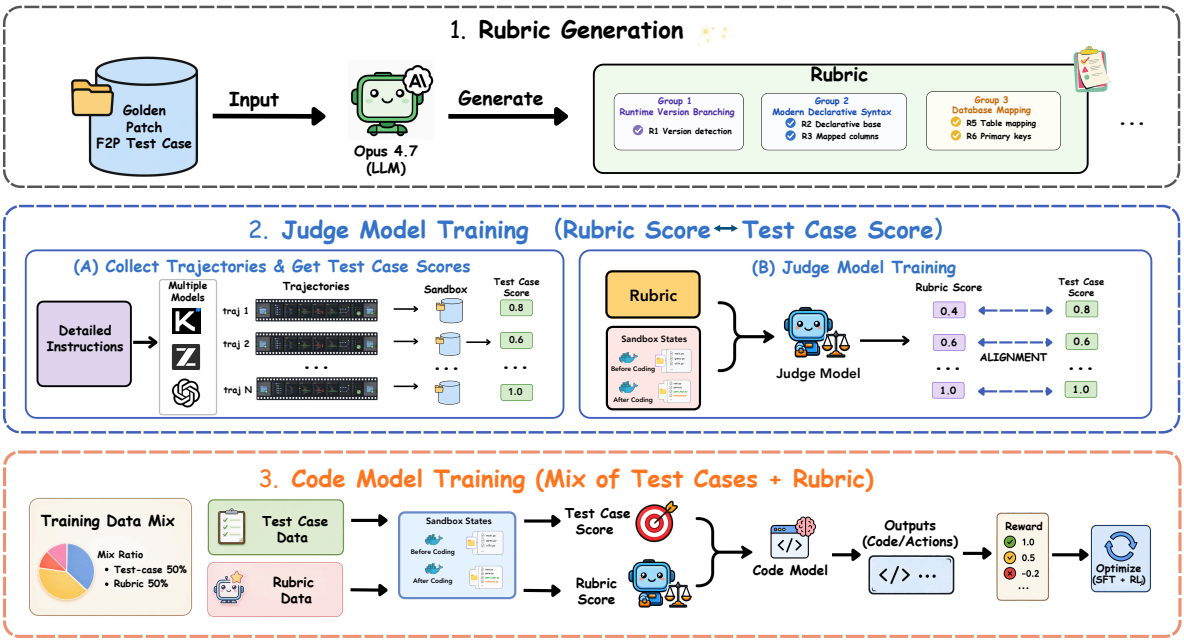

Figure 2: **Overview of CodeFoundry training.** The pipeline builds behavior-oriented rubrics from golden patches and fail-to-pass test cases, trains CodeFoundry-Judge from sandbox-evaluated solutions, and then optimizes the coding model with both test-based and rubric-based signals.

pass. The Judge predicts item-level satisfaction and aggregates these predictions into a group-level label $\hat{y}_g$, where a group is satisfied only if all its rubric items are satisfied. The final reward is task-level and all-or-nothing:

$$R = \mathbf{1}\left[\forall g \in G,\ \hat{y}_g = y_g\right].$$

This reward encourages the Judge model to make globally consistent judgments across all functional groups, rather than optimizing for partial correctness over individual groups.

5 Coding Model Training

We train the coding model in two stages, using executable-test signals for non-UX tasks and rubric-based Judge signals for UX tasks. This corresponds to the third step in Figure 2. The coding model is initialized from Qwen3.5-35B-A3B (Qwen Team, 2026). Additional training implementation details are provided in Appendix D.

SFT Training. In the first stage, we construct a separate training set for supervised fine-tuning. This training set contains 400 newly constructed tasks from 200 repositories: 200 GREENFIELD tasks, each derived from a distinct repository, and 200 BROWNFIELD tasks built from the same set of repositories but using different PR chains. For each task, we also create the corresponding UX variant. We roll out GLM-5.1 on both the detailed and UX formulations. For non-UX tasks, trajectories are selected using task-specific fail-to-pass and regression tests; for UX tasks, trajectories are selected using the trained Judge model under the corresponding rubric. We retain accepted solutions and remove failed trajectories and low-quality final code states. After filtering, we obtain approximately 1,000 supervised fine-tuning examples.

RL Training. In the second stage, we optimize the coding model with reinforcement learning on the same 400 training tasks. Each task starts from its fixed initial Docker image, and the reward is computed by the same mixed evaluator used for SFT filtering: executable tests for non-UX tasks and the trained Judge for UX variants. We use GRPO (Shao et al., 2024) as the RL algorithm. For each task, the model generates candidate solutions from the fixed initial environment, and each candidate receives a binary task-level reward after applying and evaluating the produced code changes. The reward is positive only when the solution satisfies the corresponding test-based or rubric-based evaluator.

6 Experiments

6.1 Settings

Models and agents. We evaluate recent frontier coding models from both closed-source and open-

Model	CF-Greenfield		CF-Greenfield-UX		CF-Brownfield		CF-Brownfield-UX		Overall	
	UT	P@1	RJ	P@1	UT	P@1	RJ	P@1	Score	P@1
OpenHands										
GPT-5.5	**41.6**	**17.0**	**70.8**	21.0	–	**30.0**	**45.8**	**37.0**	**52.7**	**26.3**
Claude-Opus-4.7	39.8	15.0	69.5	**23.0**	–	29.0	42.2	35.0	50.5	25.5
GLM-5.1	35.2	15.0	58.9	20.0	–	27.0	37.2	31.0	43.8	23.3
Gemini-3.1-Pro	34.7	14.0	60.8	18.0	–	28.0	38.5	30.0	44.7	22.5
Qwen-3.7-Max	32.6	12.0	57.4	19.0	–	26.0	39.0	27.0	43.0	21.0
DeepSeek-V4-Pro	31.8	13.0	55.9	17.0	–	25.0	34.6	28.0	40.8	20.8
Kimi-K2.6	29.7	12.0	56.8	17.0	–	24.0	34.0	27.0	40.2	20.0
Claude Code										
Claude-Opus-4.7	**41.5**	17.0	**71.2**	**25.0**	–	**31.0**	**45.6**	37.0	**52.8**	**27.5**
GPT-5.5	41.3	**18.0**	71.0	23.0	–	30.0	45.2	**38.0**	52.5	27.3
GLM-5.1	36.4	16.0	60.2	22.0	–	28.0	38.5	32.0	45.0	24.5
Gemini-3.1-Pro	35.8	15.0	59.8	20.0	–	29.0	38.0	31.0	44.5	23.8
Kimi-K2.6	34.0	14.0	58.8	21.0	–	27.0	37.4	30.0	43.4	23.0
DeepSeek-V4-Pro	33.0	13.0	57.1	19.0	–	26.0	36.0	29.0	42.0	21.8
Qwen-3.7-Max	32.7	14.0	56.8	18.0	–	25.0	35.6	28.0	41.7	21.3

Table 2: **Performance on CodeFoundry and CodeFoundry-UX.** CF denotes CodeFoundry. Greenfield and Brownfield denote creating a repository from scratch and modifying an existing codebase, respectively. UT denotes unit-test score, RJ denotes the **Claude-Opus-4.7 Judge score**, and P@1 denotes pass@1. UX variants are evaluated with RJ because high-level user instructions may admit valid implementations that fixed tests under-estimate.

source families, including GPT-5.5, Claude-Opus-4.7, GLM-5.1, Gemini-3.1-Pro, Qwen-3.7-Max, DeepSeek-V4-Pro, and Kimi-K2.6. Each model is evaluated under two agent scaffolds: OpenHands and Claude Code. This separates the capability of the underlying language model from the effect of the coding-agent interface.

Implementation details. All models use a sampling temperature of 1.0 and the maximum context length supported by each model or service. We report unit-test score (UT) and pass@1 for test-based tasks, and Judge score (RJ) and pass@1 for UX tasks evaluated by the Judge described in § 4, with Claude-Opus-4.7 as the Judge model. Additional training parameters are provided in Appendix D.

6.2 Main Results

We report the main benchmark results on CODEFOUNDRY in Table 2, and the training ablations in Table 3. Key findings are summarized below.

CODEFOUNDRY Remains Challenging, Especially for Greenfield Tasks. CODEFOUNDRY remains challenging for all evaluated agents. Under the OpenHands scaffold, GPT-5.5 achieves the strongest performance, with an Overall Score of 52.7 and an Overall P@1 of 26.3%, followed by Claude-Opus-4.7 with 50.5 and 25.5%, respectively. The remaining models range from 20.0% to 23.3% Overall P@1, indicating a gradual capability spectrum rather than a single dominant outlier. Brownfield tasks are consistently more tractable than Greenfield tasks: averaged over the seven models, CF-Brownfield achieves 27.0% P@1, compared with 14.0% for CF-Greenfield. This suggests that extending an existing PR-chain project is easier for agents than building one from scratch, since Brownfield tasks provide reusable code structure, interfaces, and implementation patterns. The Claude Code scaffold shows the same broad trend, with stable model ordering and minor rank changes among lower-ranked systems.

The Judge Tracks Tests and Captures Open-Ended Quality. UX tasks provide a setting for examining Agent-as-Judge evaluation. Judge-based P@1 is close to the unit-test-based P@1 of the corresponding detailed tasks: CF-Greenfield-UX reaches 19.3% versus 14.0% for CF-Greenfield, and CF-Brownfield-UX reaches 30.7% versus 27.0% for CF-Brownfield. This suggests that the Judge gives a coarse notion of correctness broadly aligned with executable tests, rather than over-rewarding open-ended tasks. Its advantage appears in the Judge score (RJ): UX tasks allow multiple valid implementations with different APIs, function signatures, or implementation decompositions, which fixed unit tests are difficult to encode. The

Model	CF		CF-UX		CF-Overall		SWE-Pro	NL2Repo
	UT	P@1	RJ	P@1	Score	P@1		
Baseline	14.0	5.0	24.0	7.0	19.0	6.0	44.6	20.5
SFT	16.2	6.5	27.6	8.5	21.9	7.5	45.0	21.4
+ Test RL	17.8	7.5	28.4	9.0	23.1	8.3	45.4	22.4
+ Judge RL	16.9	7.0	30.8	10.5	23.9	8.8	45.3	22.1
+ Mixed RL	**19.2**	**8.5**	**32.0**	**11.5**	**25.6**	**10.0**	**46.1**	**24.0**

Table 3: Training ablations for CodeFoundry-35B. CF reports unit-test score and pass@1 on non-UX tasks, CF-UX reports Judge score and pass@1 on UX tasks, and CF Overall averages the two groups.

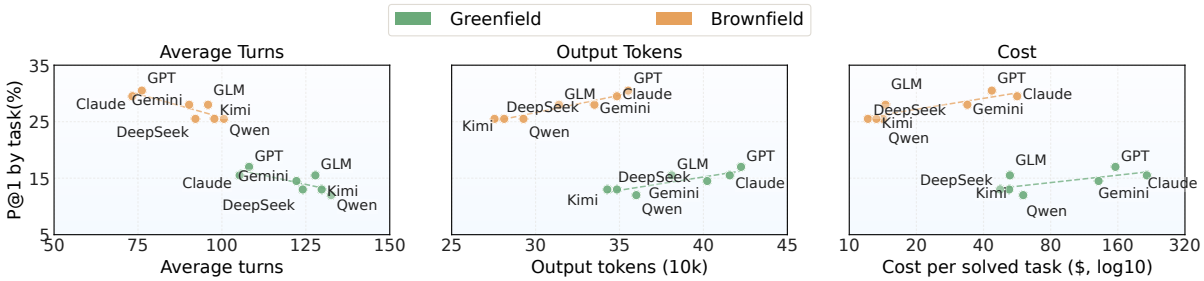

Figure 3: **Operational efficiency by task family.** Relationship between task-family P@1 and average turns, output tokens, and cost per solved task after averaging over scaffolds. Colors denote Greenfield vs. Brownfield, and text labels mark the underlying model. Dashed lines show the fitted trend within each task family.

Judge can therefore capture both binary task completion and more fine-grained solution quality in open-ended scenarios.

Mixed RL Benefits from Complementary Signals. Table 3 reports the training results for CODEFOUNDRY-35B. SFT already improves over the baseline, increasing CF-Overall Score from 19.0 to 21.9 and P@1 from 6.0% to 7.5%. Adding Mixed RL on top of SFT gives the strongest result, reaching 25.6 CF-Overall Score and 10.0% Overall P@1, with the best transfer performance on both SWE-Pro and NL2Repo. Test-only RL and Judge-only RL also improve over SFT, but each is more specialized: Test-only RL reaches 17.8 UT and 7.5% P@1 on CF, while Judge-only RL reaches 30.8 RJ and 10.5% P@1 on CF-UX. Mixed RL outperforms both because it combines the complementary strengths of executable tests and Judge feedback. These results suggest that test execution anchors functional correctness, while Judge feedback supplies supervision for open-ended requirement satisfaction.

7 Analysis

7.1 Behavior Analysis

Resource Usage Reveals Different Agent Strategies As shown in Figure 3, Brownfield tasks are generally more efficient than Greenfield tasks: their points shift upward and leftward, indicating higher success rates with lower resource use. This suggests that existing repository structure provides useful guidance for agents, reducing both task difficulty and interaction overhead. The figure also reveals a performance–cost trade-off. GPT-5.5 and Claude-Opus-4.7 lie on the high-performance frontier, but they reach this frontier with substantially larger token or cost budgets. Models further differ in how they allocate compute: Kimi-K2.6 and Qwen-3.7-Max use more interaction turns with shorter generations, suggesting iterative exploration and feedback-driven repair, whereas GPT-5.5 and Claude-Opus-4.7 use fewer turns but longer generations, consistent with more upfront planning. These differences suggest that agent evaluation should characterize not only outcome quality, but also the interaction-cost profile required.

Failure Modes We further analyze failure modes across task families and model strengths using the taxonomy in Appendix C.1. Figure 4 shows a shift from structural to semantic failures as task context becomes richer: Greenfield tasks more often fail because agents must construct the project structure and public interface from scratch, while Brownfield tasks reduce these failures by providing an executable repository and shift residual errors toward

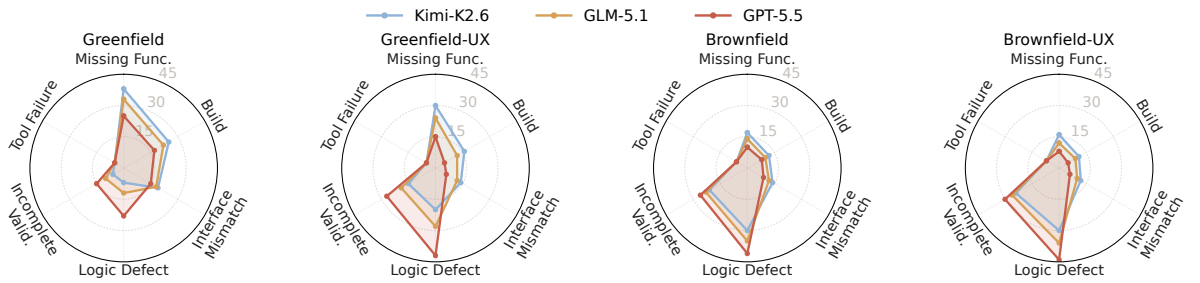

Figure 4: **Failure mode distribution by task family.** Failed-rollout attribution for four CodeFoundry task families. Each radar subplot fixes one task family and compares three models selected from Table 2 (Kimi-K2.6, GLM-5.1, and GPT-5.5) to span weak, middle, and strong performance.

Model	GF-UX		BF-UX	
	Task F1	Group F1	Task F1	Group F1
Final state only				
Claude-Opus-4.7	**91.0**	**85.1**	**92.4**	**86.6**
GPT-5.5	90.3	84.5	91.7	85.8
CF-Judge-35B	82.8	77.6	85.1	79.4
GLM-5.1	82.5	76.9	84.1	78.8
Kimi-K2.6	81.5	76.3	83.7	78.1
CF-Judge-35B-SFT	78.2	72.9	80.4	75.0
Final state + trajectory				
Claude-Opus-4.7	**90.5**	**85.2**	**92.1**	**86.9**
Kimi-K2.6	81.4	75.2	84.0	78.3
Trajectory only				
Claude-Opus-4.7	**77.3**	**71.6**	**78.7**	**73.6**
Kimi-K2.6	71.9	66.9	74.9	69.1

Table 4: **Agent-as-Judge ablations.** TASK F1 and GROUP F1 measure agreement with test-derived task labels and RJ rubric groups. The first block varies the Judge model; the lower blocks vary trajectory evidence for Kimi-K2.6 and Claude-Opus-4.7.

logic and validation. UX variants further amplify this trend, reflecting the increased need for requirement interpretation and behavioral completeness, while stronger models shift the bottleneck from runnable code generation to nuanced behavioral correctness.

7.2 Ablation Study

Judge model and input ablations. We analyze two factors that may affect Agent-as-Judge reliability: the choice of Judge model and the evidence provided to the Judge. Table 4 reports agreement with test-derived labels using TASK F1 for task-level agreement and GROUP F1 for rubric-group predictions.

Judge model strength. Under final-state judging, stronger Judge models consistently achieve higher agreement. Claude-Opus-4.7 performs best, improving over Kimi-K2.6 by 9.5 and 8.7 points in TASK F1 on GF-UX and BF-UX, and by 8.8 and 8.5 points in GROUP F1. GPT-5.5 is close to Claude-Opus-4.7, trailing by 0.7 points in TASK F1 on both task families and by at most 0.8 points in GROUP F1. This suggests that Judge model strength affects both task-level correctness judgments and finer-grained rubric-group attribution.

Trajectory evidence. We next ablate Judge evidence using Kimi-K2.6 and Claude-Opus-4.7 as weaker and stronger Judges. Trajectory-only judging is consistently less reliable than final-state judging, reducing TASK F1 by 8.8–13.7 points and GROUP F1 by 9.0–13.5 points. This indicates that intermediate actions alone are a poor proxy for whether the final implementation satisfies the task. In contrast, adding the trajectory to the final repository state changes performance by at most 0.5 TASK F1 and 1.1 GROUP F1. Thus, the final repository state provides nearly all evidence needed for reliable judging, while the trajectory contributes little once the final state is available. We therefore adopt final-state judging as the default protocol.

8 Conclusion

We introduced CODEFOUNDRY, a benchmark for long-horizon coding agents spanning from-scratch repository construction, multi-PR repository evolution, and UX-style variants. By combining executable tests with rubric-based Agent-as-Judge evaluation, CODEFOUNDRY captures both strict functional correctness and open-ended requirement satisfaction, exposing failures that are missed by isolated bug-fix benchmarks. Our results show that frontier agents still struggle with these tasks, while repository context changes both efficiency and failure patterns; future evaluation should therefore jointly measure task formulation, repository context, evaluation protocol, and deployment cost.

Limitations

CODEFOUNDRY contains 200 tasks from 200 open-source repositories across 8 programming languages. Although this is broader than many existing repository-level benchmarks, it is still a finite sample of public GitHub projects, and may underrepresent proprietary development workflows, large monorepos, domain-specific infrastructure, and tasks that require long-running services, private dependencies, or extensive human coordination. In addition, the benchmark relies on mined repositories, validated test suites, pull-request histories, and LLM-assisted instruction generation. We apply filtering, calibration, and verification to reduce instruction-test mismatch, but the resulting tasks may still reflect biases in repository selection, available tests, and generated descriptions. The UX variants are designed to resemble high-level user requests, but they are not collected from live user-agent interactions, and therefore may not capture the full ambiguity, negotiation, and preference variation of real software development.

References

Anthropic. 2026. Claude code. https://claude.com/product/claude-code.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing swe-bench verified. https://openai.com/index/introducing-swe-bench-verified/. Updated February 24, 2025; accessed May 22, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, and 1 others. 2025. Nl2repobench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*.

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

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

OpenAI. 2025. Introducing codex. *OpenAI Blog Post*. https://openai.com/index/introducing-codex/.

Qwen Team. 2026. Qwen3.5-35B-A3B. https://huggingface.co/Qwen/Qwen3.5-35B-A3B. Accessed: 2026-05-25.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*.

Minh VT Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. 2025. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*.

Xinbo Xu, Ruihan Yang, Haiyang Shen, Wendong Xu, Bofei Gao, Ruoyu Wu, Kean Shi, Weichu Xie, Xuanzhong Chen, Ming Wu, and 1 others. 2026. Roadmapbench: Evaluating long-horizon agentic software development across version upgrades. *arXiv preprint arXiv:2605.15846*.

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

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, and 1 others. 2026. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*.

Appendix

A Task Examples

This section provides representative task examples for the four CodeFoundry task families. Each example text is shown in a framed block to distinguish detailed instructions from UX-style user requests.

A.1 Greenfield Example

```
# Goal
Implement a compact version of ‘netlab‘, a CLI-based virtual
networking lab
abstraction layer, from scratch. The project should read a
human-authored YAML
topology and compile it into normalized topology data, device
facts, provider
metadata, and helper outputs that can later be consumed by
Ansible or container
providers. It is a compiler and orchestration helper, not a
packet simulator.

# Scope
Create a Python package named ‘netsim‘ with the same public
shape expected by
tests. The important areas are:

- ‘netsim.data‘: Box creation helpers, list helpers,
removed-attribute handling,
  and dictionary merge utilities.
- ‘netsim.augment‘: topology transformation, addressing,
nodes, links, devices,
  groups, plugins, tools, and validation helpers.
- ‘netsim.modules‘: module-attribute resolution across
interface, link, node,
  topology, and defaults scopes.
- ‘netsim.cli‘: command parsers and entry points for reading,
inspecting,
  showing, creating, deploying, and validating labs.

# Core behavior
The main pipeline is
‘netsim.augment.main.transform(topology)‘. It should:

1. Load topology data into canonical Box objects.
2. Normalize global defaults, addressing pools, nodes, links,
groups, modules,
   tools, providers, and plugins.
3. Expand included components and inherited group/node
settings.
4. Assign node identifiers, management data, loopbacks, link
interfaces,
   prefixes, bridges, and provider-specific attributes.
5. Apply module lifecycle hooks in order: pre-transform,
transform, link
   transform, node post-transform, and final cleanup.
6. Return a topology object that is stable enough for
inventory, graph, report,
   and validation commands.

# Required APIs
Implement these APIs precisely:

- ‘netsim.data.get_box(init)‘ returns a ‘Box‘ configured with
  ‘default_box=True‘, ‘default_box_none_transform=False‘, and
‘box_dots=True‘.
- ‘netsim.data.append_to_list(BX, list_name, item,
flatten=False)‘ creates the
  list when missing, ignores duplicates, optionally extends
with list items, and
  returns the final list. If ‘BX‘ is ‘None‘, return an empty
list.
- ‘netsim.augment.devices.get_device_features(node,
defaults)‘ merges static
  device features from defaults with ‘node._features‘, with
node-level values
  taking precedence.
- ‘netsim.augment.nodes.ghost_buster(topology)‘ returns a
copy of the topology
  with unmanaged nodes removed, link interfaces filtered,
link counts/types
  recomputed, bridge names refreshed, and empty links dropped.
- ‘netsim.modules.get_effective_module_attribute(path,
intf=None, link=None,
  node=None, topology=None, defaults=None, merge_dict=True)‘
searches from most
  specific to least specific scope. When merging Box values,
combine defaults
  upward with removed-attribute semantics; when an attribute
is explicitly
  removed, stop traversal and return the accumulated value.

# Device feature validation
Provide ‘netsim.augment.devices.FC_MODE‘ with
blacklist/whitelist and
success/error values, plus ‘check_optional_features(...)‘.
The validator should
recursively compare requested feature trees against device
support data. Missing
features are errors in whitelist mode and ignored in
blacklist mode. Boolean and
string feature declarations should map to success, warning,
or hard-error
results while logging meaningful diagnostics.

# CLI behavior
Expose parser helpers and ‘run(...)‘ entry points under
‘netsim.cli‘. The CLI
does not need to start real virtual machines, but it must
parse common command
options, load topology files, call the transform pipeline,
and produce stable
data for commands such as ‘read‘, ‘inspect‘, ‘show‘, ‘graph‘,
‘create‘,
‘validate‘, ‘up‘, and ‘down‘. External commands should be
wrapped so tests can
mock command availability and execution.

# Implementation constraints

- Use the declared dependencies: PyYAML, python-box, Jinja2,
netaddr, requests,
  rich, packaging, filelock, and the Python standard library.
- Keep module imports lightweight enough that individual
submodules can be
  imported in isolation.
- Preserve public names, signatures, constants, and enum
values used by tests.
- Prefer deterministic ordering for generated lists, tables,
and dictionaries.
- Treat invalid topology data as validation errors instead of
uncaught crashes.

# Verification
Provide packaging metadata so ‘pip install -e .‘ works. Add
smoke checks that
import the package, parse a small topology, run the transform
pipeline, remove
unmanaged nodes with ‘ghost_buster‘, resolve a merged module
attribute, and call
representative CLI entry points with mocked external commands.
```

A.2 Greenfield-UX Example

```
I need a command-line tool for building virtual network labs
from readable
topology files. Network engineers should be able to describe
nodes, links,
addressing, device types, groups, modules, and provider
settings in YAML, then
ask the tool to normalize that lab into consistent data that
other automation
systems can use.

The tool should behave like a topology compiler rather than a
simulator. It
```

should validate the input, fill in reasonable defaults, expand inherited group
settings, assign node and interface details, calculate addressing information,
and prepare provider-specific metadata for backends such as container or virtual
machine lab environments.

Users also need inspection and workflow commands around that compiler. They
should be able to read and inspect a topology, show supported devices and
features, create derived outputs such as inventory or graph data, validate lab
readiness, and run lifecycle-style commands for bringing a lab up or down. These
commands should be deterministic and easy to test, with external system calls
kept behind wrappers so they can be mocked.

The implementation should be robust enough for malformed or partial topology
files: report validation problems clearly, avoid crashing on missing optional
sections, and keep imports lightweight so individual parts of the package can be
used independently.

A.3 Brownfield Example

Add Schedule and Chain autoscaling policies to FleetAutoscaler, promote ScheduledAutoscaler to Alpha, and implement validation and controller logic for time-based fleet scaling

This change is a follow-up to earlier work toward scheduled fleet autoscaling and introduces two new FleetAutoscaler policy types--Schedule and Chain--so that users can scale fleets based on calendar time and cascading policy lists. The PRs frame the work as adding API changes and validation for FleetAutoscaler Schedule and Chain policies, adding autoscaling logic for the new Chain and Schedule policies, and adding end-to-end tests for chain and schedule policies while bumping the ScheduledAutoscaler feature gate to Alpha.

The entire capability is guarded by the ‘FeatureScheduledAutoscaler‘ feature gate, which must be promoted from Dev to Alpha, remain disabled by default in the runtime feature defaults, and be added to the project’s alpha feature-gate manifests. A Schedule policy lets users define a time window with RFC3339 start and end datetimes, an optional timezone, a UNIX cron expression (‘StartCron‘), and a duration. When the current time falls within the window and the cron schedule is active, the schedule applies a nested policy such as Buffer, Counter, List, or Webhook. When inactive, the autoscaler should return the fleet’s current replica count and must not record a warning Kubernetes event. A Chain policy is an ordered list of ‘ChainEntry‘ items. Each entry has a unique ‘ID‘ and an inline ‘FleetAutoscalerPolicy‘. The controller walks the chain in order and applies the first entry whose policy succeeds: for schedule entries this means the schedule is active, and for webhook entries this means the webhook call succeeds. If an entry fails because its schedule is inactive or its webhook errors, the controller falls through to the next entry. If no entry applies, the controller falls off the chain and returns the current fleet size.

On the API surface, ‘FleetAutoscalerPolicy‘ must gain a ‘Schedule *SchedulePolicy‘ field and a ‘Chain ChainPolicy‘ field. The policy type enum must include ‘SchedulePolicyType‘ with value ‘"Schedule"‘ and ‘ChainPolicyType‘ with value ‘"Chain"‘. The new types must be named exactly ‘SchedulePolicy‘, ‘ChainPolicy‘, ‘ChainEntry‘, ‘Between‘, and ‘ActivePeriod‘. ‘SchedulePolicy‘ contains ‘Between‘, ‘ActivePeriod‘, and a nested ‘Policy‘ of type ‘FleetAutoscalerPolicy‘. ‘ChainEntry‘ contains an ‘ID‘ string and embeds ‘FleetAutoscalerPolicy‘ inline. ‘Between‘ holds ‘Start‘ and ‘End‘ times. ‘ActivePeriod‘ holds ‘Timezone‘, ‘StartCron‘, and ‘Duration‘ strings.

Validation must enforce that both Schedule and Chain policies are forbidden unless ‘FeatureScheduledAutoscaler‘ is enabled. For a schedule, validation must reject an end time that is before the current time or before the start time, an invalid IANA timezone, an invalid or missing UNIX cron expression in ‘StartCron‘, and any ‘CRON_TZ‘ or ‘TZ‘ prefix in the cron string; duration must be a valid Go duration string. For a chain, validation must reject duplicate ‘ID‘s among entries, entries that lack a valid nested policy, nested chain policies (chains inside chains), and must recursively validate each entry’s nested policy fields. In particular, validation must return field errors at paths like ‘spec.policy.schedule.between.end‘ for an invalid end time, ‘spec.policy.schedule.between‘ when the start time is after the end time, ‘spec.policy.schedule.activePeriod.startCron‘ for malformed cron expressions, ‘spec.policy.chain‘ for duplicate IDs or a disabled feature gate, ‘spec.policy.chain[0]‘ when an entry is missing a policy, and ‘spec.policy.chain[1].policy.buffer.minReplicas‘ for invalid nested policy values.

The ‘FleetAutoscaler‘ validation logic should be restructured so that nested policies can be validated recursively; a ‘FleetAutoscalerPolicy‘ should be able to validate itself and its children. The core fleet-size computation logic should accept a ‘FleetAutoscalerPolicy‘ directly so that nested policies inside a schedule or a chain entry can be evaluated recursively. The autoscaler controller should integrate the new policy types into its sync loop. When the top-level policy is a schedule that is currently inactive, the controller should return the fleet’s current size without emitting an error event. When the top-level policy is a chain, it should evaluate entries sequentially, falling through on inactive schedules or webhook failures, and ultimately falling back to the current fleet size if nothing matches. Finally, the example manifests for chain and schedule autoscalers should reflect the final schema, and generated deepcopy and client code should be updated for the new types.

A.4 Brownfield-UX Example

We run game fleets whose traffic changes predictably around events, evenings, and regional launches, but the current FleetAutoscaler cannot express those calendar-based scaling rules directly.
Please add a way for an autoscaler policy to become active only during a configured time window and recurring schedule, then apply an existing scaling policy while that schedule is active.
Outside the active window, the controller should leave the fleet at its current size instead of treating the inactive schedule as an error.

We also need a fallback model for more complex operations. For example, we may want to try a scheduled policy first, fall back to a webhook-driven policy if it is available, and otherwise keep the current fleet size.
Please support an ordered chain of autoscaling policies where the controller evaluates entries in order and uses the first one that can produce a valid scaling decision.
Inactive schedules and failed webhooks should allow the chain to continue rather than failing the whole autoscaler.

This should be introduced as an alpha scheduled-autoscaling capability behind the existing feature-gate system.
The API should reject invalid schedules, duplicate chain entries, nested chains, malformed cron expressions, invalid timezones, and policies used while the feature gate is disabled.
Please include example manifests and coverage for validation, controller behavior, and end-to-end chain and schedule autoscaling flows.

B Construction and Calibration Details

B.1 Brownfield PR-Chain Selection

Brownfield tasks are built from semantically coherent chains of merged pull requests. The pipeline first constructs a PR graph from rule-based signals, annotates semantic relations, samples candidate chains, and validates each chain through patch

application, build execution, and test execution. Candidate repositories must have stable execution environments and rich evolution histories: at least 500 stars, an update within the past six months, and more than 1,000 pull requests and issues in total. Only merged code changes are retained.

B.2 Iterative Calibration

Each task is checked through model rollouts and a mismatch analysis step that separates model limitations from instruction–test mismatches. When a mismatch is detected, the instruction is revised while preserving the tests, and the task is retained only after verification indicates that the mismatch has been resolved. In the current pipeline, each task receives four independent Kimi K2.6 rollouts, and the associated tests are executed in a sandboxed environment. Opus 4.7 analyzes failed executions to determine whether they reflect an instruction–test mismatch rather than insufficient implementation capability. If a mismatch is detected, one calibration round revises only the task instruction while leaving the test cases unchanged. Opus 4.7 then performs an additional verification pass, and the task is retained only if no instruction–test mismatch remains.

B.3 UX Rewriting

For Greenfield-UX and Brownfield-UX, CodeFoundry rewrites detailed instructions into higher-level user-facing requests. The rewritten requests describe missing capabilities, abnormal behaviors, user pain points, or expected project evolution. They avoid explicitly listing public identifiers, parameter names, default values, return semantics, public constants, or public attributes that are directly used by tests, thereby reducing exposure of test implementation details while preserving the same underlying task targets.

C Experiment Implementation Details

Agent execution. All evaluated agents operate inside the same sandboxed task environments used by the benchmark. For each rollout, the agent receives the task instruction and repository state, edits files through the agent interface, and may run commands for inspection or validation. We use a sampling temperature of 1.0 for all evaluated models and use each model's maximum supported context length by default. After the rollout terminates, we evaluate the final repository state rather than intermediate patches or reasoning traces.

OpenHands and Claude Code. OpenHands and Claude Code are used as two independent coding-agent scaffolds for the same underlying model list. We keep the benchmark task, initial image, and evaluation script fixed across scaffolds. This setup allows the comparison to reflect differences in agent interaction and tool-use behavior while keeping the task distribution unchanged.

Unit-test evaluation. For CF-Greenfield and CF-Brownfield, we run the task-specific fail-to-pass tests in the provided Docker environment. A rollout is counted as pass@1 only if all required tests pass on the final code state. When regression tests are available, they are used to check that the submitted solution does not break existing behavior.

Rubric-judge evaluation. For CF-Greenfield-UX and CF-Brownfield-UX, we evaluate the final repository with the trained Judge model. The Judge receives the task instruction, final code state, and structured rubric, and predicts whether the solution satisfies each requirement group. The reported RJ score is the aggregated rubric satisfaction score, while pass@1 requires the solution to satisfy the full rubric.

C.1 Failure Mode Taxonomy

We assign each failed rollout to one primary failure mode. `Missing functionality` means that a required capability, component, or end-to-end workflow is absent. `Build errors` cover solutions that cannot compile, import, install, or execute the evaluator because of syntax, dependency, or runtime errors. `Interface mismatch` indicates that the public API, function signature, command, schema, or configuration surface does not match the expected contract. `Logic defects` are cases where the solution runs and exposes the intended interface, but implements incorrect behavior. `Incomplete validation` covers missing checks, error handling, edge-case handling, or behavioral constraints. `Tool failure` is reserved for failures caused by agent tooling, sandbox execution, or external execution problems rather than the submitted code itself.

D Training Implementation Details

Our coding model is initialized from Qwen3.5-35B-A3B (Qwen Team, 2026). SFT is trained on approximately 1,000 filtered repository-level trajectories collected from CodeFoundry tasks, and RL further

optimizes the SFT model with GRPO (Shao et al., 2024) under the mixed evaluator described in § 5. For RL, each prompt is paired with the fixed initial Docker image of the corresponding task, the model samples 8 candidate coding trajectories, and the final repository state is scored with executable tests for non-UX tasks or the trained rubric Judge for UX tasks. Training is conducted on 32 GPUs with a 256K-token context length; we use a sampling temperature of 1.0 for rollout generation and a binary task-level reward for optimization.

E Notations

Table 5 summarizes the core notations used in this paper.

Table 5: The 13 types of PR relationships and their description.

Edge Type	**Description**
merged_on_top_of	The base.sha of the source PR exactly matches the merge_commit_sha of the target PR
merge_commit	The merge_commit_sha of the target PR appears in the commits[] of the source PR
commit_parent_chain	A commit in the source PR has a parent that belongs to the target PR's commits
referenced_commit	A referenced commit in the source PR timeline can be traced back to the target PR's commits
closes	Source PR's title/body/commit message/issue/review comment contains fix/-close/resolve keywords referring to a merged PR
commit_msg_pr_ref	A commit message in the source PR contains (N) or "Merge pull request N" referring to a merged PR
revert_of	The source PR reverts a merged target PR
cherry_picked_from	The source commit message contains "cherry picked from commit <sha>"
url_link	The source body/comments contain a URL pointing to the target PR
same_head_branch	Two merged PRs have the same (head.repo.full_name, head.ref)
force_push_lineage	A force-pushed commit in the source PR timeline appears in another merged PR
patch_overlap	Two merged PRs modify the same file and the added line ranges overlap
mentioned	A plain "N" reference in the source PR text referring to a merged target PR