SWE-Dev: Evaluating and Training Autonomous End-to-End Feature-Driven Software Development

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown strong capability in diverse software engineering tasks. However, feature-driven development, a highly prevalent real-world task that involves developing new functionalities for large, existing codebases, remains underexplored. We therefore introduce SWE-Dev, the first large-scale dataset (with 14k training and 500 test samples) designed to evaluate and train autonomous coding systems on real-world end-to-end feature-driven software development tasks. To ensure verifiable and diverse training, SWE-Dev uniquely provides all instances with a runnable environment and its developer-authored executable unit tests. This collection not only provides high-quality data for Supervised Fine-Tuning, but also enables Reinforcement Learning by delivering accurate reward signals from executable unit tests. We evaluated SWE-Dev across 17 base LLMs, 10 reasoning-focused LLMs, 10 multi-agent systems, and 8 tool-augmented LLM agents. Results show substantial headroom: the best single-turn model reaches only 22.51% Pass@1 on the hard split, while OpenHands agents improve to 56.44% but still leave many tasks unsolved. Code is available here.

1 Introduction

Large Language Models (LLMs) are rapidly transforming autonomous programming, with capabilities extending from generating isolated code snippets to complex interactions within entire repositories (GitHub, 2021; Cursor, 2023). As LLMs increasingly engage at this repository scale, rigorously evaluating their proficiency in handling complex coding systems becomes paramount for guiding their advancement. Current prominent benchmarks, while valuable, still struggle to judge how well LLMs perform in realistic, end-to-end development settings (Table 1). For example, SWE-Bench (Jimenez et al.) measures only localized bug fixes described by GitHub issues, and RepoBench (Liu et al.) evaluates the completion of a few unrelated functions within a repository.

Table 1: Comparison of SWE-Dev with existing repository-level benchmarks. We compare task type, the average number of golden lines of code, whether using the real-world executable tests are available, and the number of executable environments.

	Task type	Lines of Code	Real Tests	#Exec Env
ComplexCodeEval	Func Comp	38.2	✗	0
CoderEval	Func Comp	20.6	✔	43
M2rc-Eval	Line Comp	1.0	✗	0
CrossCodeEval	Line Comp	1.0	✗	0
SWE-Bench(Train)	Issue Solve	32.8	✗	0
R2E-Gym	Issue Solve	5.7	✗	13
SWE-Smith	Bug Fix	9.6	✔	28
SWE-Synth	Bug Fix	5.7	✔	7
SWE-Dev	Feature Dev	190.0	✔	1061

In contrast, real-world software development is dominated by feature-driven development, where developers design, implement, and integrate new functionality into large, existing codebases. Prior empirical studies show that feature development accounts for a substantial fraction of daily engineering work (Harness Team, 2025; Thüm et al., 2014). Traditional feature development (Coad et al., 1999; Palmer and Felsing, 2001) typically involves intermediate processes such as developing an overall model or building feature lists. For coding systems, the critical way towards achieving more comprehensive and genuinely autonomous programming capabilities is on the final implementation of features. This end-to-end process, from interpreting requirements within large, existing codebases to generating functionally correct and integrated code, is what we refer to as end-to-end feature-driven software development.

Recognizing the central role of feature development and the limitations of current evaluation benchmarks, we introduce an feature-driven **S**oft**W**ar**E** **Dev**elopment dataset, **SWE-Dev**. which

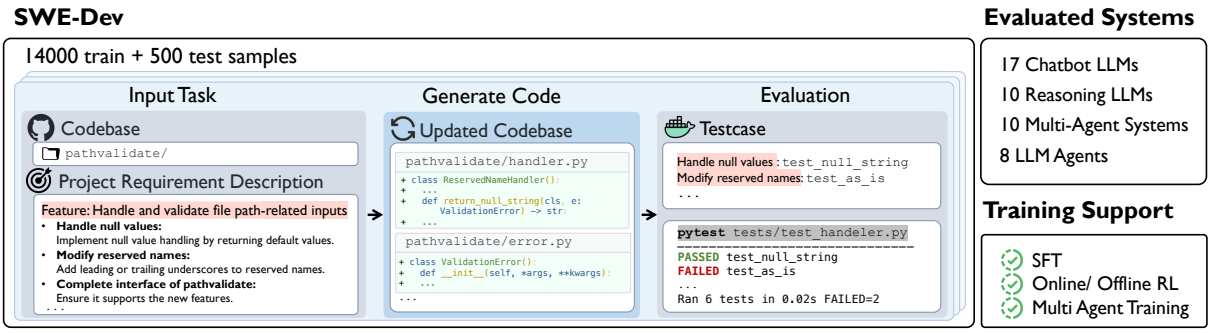


Figure 1: Overview of SWE-Dev, a software development dataset providing feature development tasks with feature description and codebase as input and test cases for evaluation. It is uniquely grounded in real-world repositories and paired with executable test suites, enabling reliable, functionally verifiable supervision. SWE-Dev is evaluated on 45 autonomous coding systems and supports advanced training paradigms like SFT, RL, and multi-agent training.

is the first large-scale dataset designed to evaluate and train autonomous AI systems on real-world end-to-end feature-driven software development tasks. It comprises 14,000 training and 500 test instances derived from over 1,000 open-source projects, and is distinguished by three key characteristics: ❶ **Realistic scale and complexity**: SWE-Dev requires substantial code modifications (avg. 190 lines of code across 3 files), challenging models with cross-file dependencies, long contexts, and feature-level implementation scope. ❷ **Robust and grounded evaluation**: Each sample is grounded in a real open-source repository, specified by a project requirement description (PRD), and evaluated using executable test cases to verify satisfaction of the intended implementation, ensuring reliable assessment and supervision. ❸ **Verifiable training with executable test suites**: All 14,000 training instances include runnable environments and executable unit tests, providing execution-based feedback that supports supervised fine-tuning (SFT), reinforcement learning (RL) with accurate rewards, and multi-agent system (MAS) training.

We validate SWE-Dev from both evaluation and training perspectives. **For evaluation**, we test (i) base LLMs (instruction-tuned and reasoning-focused), (ii) tool-augmented LLM agents implemented via OpenHands, and (iii) multi-agent LLM systems (MAS). The results show that feature development remains challenging: under direct single-turn generation, the best base model Claude-3.7-Sonnet achieves only 49.47% and 22.51% Pass@1 on the *easy* and *hard* split, with most models far below this level, indicating substantial headroom. In contrast, OpenHands agents substantially improve performance, reaching 56.44% on the hard split, but still leaving nearly half of tasks unsolved. Within our MAS suite, gains over a single baseline are modest (11.09% $\rightarrow$ 20.03% on hard), and simple refinement-based strategies can be more efficient than workflow-heavy designs. **For training**, SWE-Dev provides runnable tests and executable environments for every instance, enabling SFT and RL with execution feedback; we observe clear gains from test-supervised training, with a fine-tuned 7B model reaching performance comparable to GPT-4o on the hard split.

Our contributions are as follows:

- We introduce **SWE-Dev**, a real-world benchmark and training corpus for autonomous feature-driven software development, with runnable environments and tests for both train and test splits.
- We comprehensively evaluate **LLMs**, tool-augmented **LLM agents**, and **multi-agent systems** on SWE-Dev, revealing their capabilities and failure modes on feature implementation.
- We show that SWE-Dev **supports diverse training paradigms**, including SFT, RL, and MAS training, establishing its utility for training adaptation with executable feedback.

2 Related Work

2.1 Coding benchmarks

Early benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al.), and CodeContests (Li et al., 2022) mainly target isolated, function-level problems with short contexts and well-specified I/O, which is useful for measuring basic correctness but far from real software development. To better reflect repository-scale settings, a range of repository-level benchmarks have been proposed (Table 1), spanning function completion (e.g., ComplexCodeEval (Feng et al., 2024), CoderEval (Yu

et al., 2024)), line-level completion/understanding (e.g., M2RC-Eval (Liu et al., 2024), CrossCodeEval (Ding et al., 2023)), and issue solving or bug fixing (e.g., SWE-Bench (Jimenez et al.), R2E-Gym (Jain et al., 2025), SWE-Smith (Yang et al., 2025), SWE-Synth (Pham et al., 2025)). Despite this progress, two limitations remain common: (1) the scope of required code changes is still small (often <40 LOC, e.g., 32.8 LOC in SWE-Bench and 38.2 LOC in ComplexCodeEval), which inadequately captures feature-scale development; and (2) evaluation is often weak or inconsistent, with many benchmarks lacking real executable tests or runnable environments, forcing reliance on proxy metrics that can miss functional correctness (Lozhkov et al., 2024; Huang et al., 2022). SWE-Dev addresses these gaps by providing large-scale, repository-level feature development tasks with substantial code modifications and **real-world executable tests** in **runnable environments** for every instance, improving realism and enabling training with execution feedback.

2.2 Code LLMs Training

Training LLMs for coding tasks typically involves three stages: pre-training, supervised fine-tuning (SFT), and reinforcement learning (RL). Pre-trained models such as StarCoder (Lozhkov et al., 2024) and Phi (Abdin et al., 2024) leverage massive code corpora to learn syntax and general programming patterns. To improve instruction following and task completion, many works adopt SFT. Code Alpaca (Wang et al., 2023b) employs self-instruct generation, WizardCoder (Luo et al.) leverages Evol-Instruct (Xu et al., 2024) to synthesize more complex instructions. However, SFT fundamentally lacks exploration: it teaches models to imitate ground-truth outputs rather than to reason or build autonomously (Ouyang et al., 2022). Beyond SFT, RL frameworks such as CodeRL (Le et al., 2022) utilize test-case-based feedback to optimize model behavior. Early RL (Dai et al., 2023) for code mostly targeted short, function-level problems with immediate execution rewards, but recent work has scaled RL to more realistic SWE settings: SWE-RL (Wei et al., 2025) applies policy-gradient updates in an agentless scaffold on SWE-Bench, while DeepSWE (Luo et al., 2025) and SkyRL-v0 (Cao et al., 2025) (see also (Golubev et al., 2025)) train interactive, multi-turn SWE agents over full repositories and tools. In parallel, SWE-Gym (Pan et al., 2024) explores repository-scale training via multi-agent systems; however, due to the lack of an executable training split in SWE-Bench, it constructs a separate dataset of 2,438 tasks and yields only 500 trajectory samples for training. In contrast, SWE-Dev provides a large-scale repository-level training set with runnable environments and unit-test-based supervision, enabling scalable SFT, RL, and MAS training with direct execution feedback.

3 SWE-Dev

SWE-Dev is the first dataset designed to train and evaluate autonomous coding systems on feature-driven software development tasks. Each instance requires the model to implement a new capability within an existing codebase, based on a natural language requirement and evaluated through real-world unit tests. This section describes the construction of the dataset (§ 3.1), its core features (§ 3.2).

3.1 Dataset Construction

Our dataset construction leverages a straightforward principle: test files in real-world repositories can serve both as a source of feature requirements and as a means of verifying correct implementation. In PyPI packages, developers create high-quality test files to ensure that specific modules or features function reliably across updates. For example, in `numpy`, `test_random.py` validates random array generation, aligning closely with the feature it tests. These test files provide executable, feature-specific validation, making them ideal for defining and evaluating development tasks.

Using these developer-authored tests as ground truth, we gain two advantages. First, they provide executable, functionality-level feedback for model evaluation. Second, by tracing the test cases back to their associated implementation code, we can identify and mask the relevant source code, forming the basis of an incomplete development task. These traced functions also guide the generation of precise task descriptions. Based on this process, we divide our construction into three stages: (1) collecting repositories, test files and building Docker environments, (2) generating test-to-code call trees via dynamic tracing, and (3) creating the final task by masking the relevant source code and producing the feature specification.

Step 1: Test File Collection To support realistic feature-level tasks and test-based evaluation, we begin by collecting real-world repositories that re-

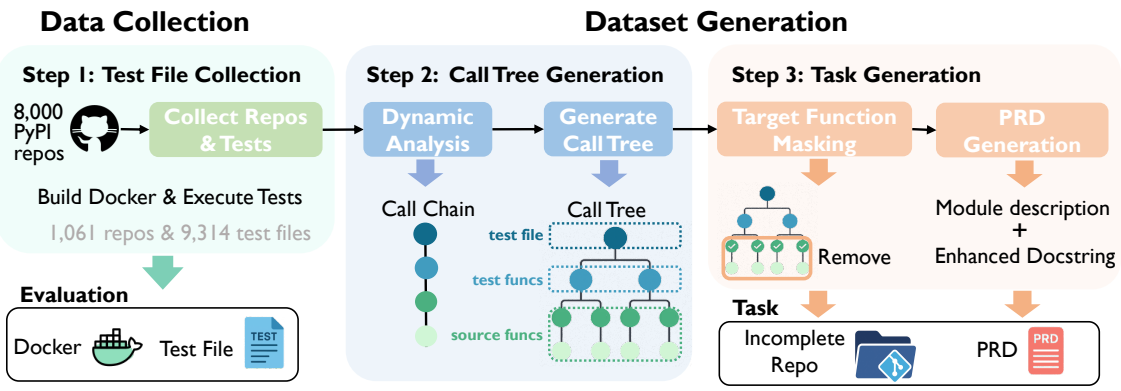


Figure 2: Overview of SWE-Dev dataset construction. **Step 1**: We collect real-world repositories with passing test files in Dockerized environments, **Step 2:** trace test executions to construct function-level call trees linking test cases to invoked source code, and **Step 3**: mask core functions while generating refined PRDs to create tasks. Each sample includes an incomplete repository, a natural language requirement, and executable test cases-enabling realistic, verifiable feature development.

flect common development scenarios. Specifically, we select 8,000 popular PyPI packages based on star counts. However, not all repositories are suitable: many lack usable tests or need sophisticated installation. Therefore, we applied a strict filtering process focused on test suite executability. Repositories were retained only if they met two criteria: (1) they include an identifiable test suite (e.g., using `pytest` or `unittest`), and (2) their test files could be executed successfully within the package Docker environment, with all tests passing. This ensures the resulting tasks are grounded in verifiable, runnable functionality. After filtering, we obtain 1,072 validated repositories (as of December 12, 2024) and 9,314 executable test files.

Step 2: Call Tree Generation To locate the specific methods involved in implementing a feature, we capture the runtime interactions between test cases and their corresponding source code through dynamic analysis. This process has two main parts: (1) Dynamic analysis: We execute each test file using `pytest` inside a Docker environment and apply Python's built-in `trace` module to record all triggered functions in source code. This results in multiple linear call chains that record the sequence of invoked source functions. (2) Call tree ensemble: We aggregate the call chains into a hierarchical call tree, where the nodes of call tree represent functions, and edges capture dependency relationships. The call tree is rooted from test functions and followed by triggered source functions. The depth and width of the tree reflect the complexity of the feature logic, including nested structures and cross-file dependencies. These trees provide a precise mapping from test behavior to implementation code, enabling us to localize relevant functions and systematically control task difficulty later.

Step 3: Task Generation Once we have localized the implementation logic using call trees, we transform it into a feature development task by (1) masking the existing implementations of feature components and (2) generating a natural language requirement for this feature. These components represent a typical development scenario where a new feature needs to be added to an existing system, but its implementation is yet to be realized. To achieve this, we perform the following: (1) Target function masking: We use structural properties of the call tree (e.g., depth and node count) to identify key function nodes that represent the core logic for the feature. The corresponding implementation code is removed from the repository, creating a space for the new feature to be integrated. (2) Project Requirement Document (PRD) generation: We construct the feature description in PRD by using GPT-4o to synthesize a high-level module description from the test file and augmenting the masked function's docstring with implementation-level details. These two elements are combined into PRD, which serves as the task prompt. See example in Fig. 6 and prompts in Appendix J.

3.2 Dataset Features

Controlled Complexity via Call Tree: Leveraging call-tree analysis, SWE-Dev enables systematic adjustment of task difficulty by varying the masked dependency depth during task generation, supporting rigorous assessment under controlled complexity levels (see § 5). We define test split *Easy* tasks as those with call-tree depth 1–2, and *Hard* tasks

as those with depth ≥ 3, which typically require implementing 5+ interdependent functions.

Reliable Test-Based Evaluation: Assessment uses the original, developer-authored unit tests, validated for correct execution in a controlled environment. This execution-based pass/fail verification provides an objective, reproducible, and functionally accurate measure of code, directly reflecting real-world correctness criteria.

Repository-Level Split for Clean Evaluation: SWE-Dev enforces a repository-level split between training and test sets to reduce contamination from shared repository context. The training split contains **1,033** repositories, while the test split contains **39** repositories, with no repository overlap between splits. More statistics are shown in § C.

4 Experiment

In this section, we empirically evaluate the effectiveness of various coding systems and training paradigms on SWE-Dev. We first compare the performance of single-LLM(§ 4.1) and MAS(§ 4.1.2) on the feature-driven software development tasks. Then, the effectiveness of different training approaches, including SFT (§ 4.2.1), RL (§ 4.2.2), and multi-agent training (§ 4.2.3) is discussed.

Setup. We employed the Pass@k as an evaluation metric in SWE-Dev (Chen et al., 2021). For non-tool-use-agent inference code context, since SWE-Dev requires both the PRD and codebase as inputs. The codebases consist of many tokens (an average of $\tilde{2}$0K lines, see Table 6), exceeding typical LLM context window. Thus, in all the experiments below, we provide only the relevant code context in the call tree rather than the entire codebase, since the call tree track exactly the code used for completion the task.

4.1 Testing Results

This section presents the performance of 17 chatbot LLMs, 10 reasoning LLMs, 10 multi-agent systems, and 8 tool-augmented LLM agents (via OpenHands) on SWE-Dev, under the single-LLM, multi-agent, and agentic settings. Full details of the evaluated methods are provided in Appendix H.1.

4.1.1 Single LLM Inference

SWE-Dev presents substantial challenges for current LLMs, revealing a clear gap between existing coding capabilities and real-world software engineering demands.

The left two part of Figure 3 reports Pass@1 performance of chatbot and reasoning LLMs on SWE-Dev (see full results in Table 8). We observe that: (1) LLMs perform better on the *easy* split than the *hard* split. (2) Performance generally scales with model size, especially for LLMs within the same family, aligning with our understanding of LLM capabilities. (3) Even the best-performing chatbot LLM: Claude-3.7-Sonnet (Anthropic, 2025)) achieves 19.74% on the *Hard* split and 53.09% on the *Easy* split. The reasoning LLM: Grok-3-beta achieves 53.63% and 18.97% on *Easy* and *Hard* split, respectively. This still falls short of achieving strong performance, indicating that current models are not yet fully capable of handling tasks that approximate the complexity of real-world development scenarios.

4.1.2 Multi-Agent Inference

Table 10 compares the performance, call times and total costs of various MAS against the single-agent baseline driven by GPT-4o-mini. Details of MAS are given in AppendixH.1.

MAS generally outperforms single-agent baselines on complex tasks. While the single-agent approach achieves only 11.09% Pass@1 on hard tasks, Self Refine (Madaan et al., 2023) and EvoMAC (Hu et al., 2025b) improve performance to 20.03% and 13.60%, respectively. These results highlight the advantage of MAS in solving complex, reasoning-intensive problems.

Simpler multi-agent strategies offer strong performance–efficiency trade-offs. Methods such as Self Refine strike an effective balance between performance and cost. On the easy subset, Self Refine achieves the highest Pass@1 of 40.02% using only 5 calls. In contrast, more complex systems like ChatDev, despite making over 26 calls, fall behind in performance (35.13%), indicating that additional agent complexity does not necessarily lead to better results.

Human-designed, workflow-heavy MAS often introduce unnecessary overhead. Systems with manually defined roles and interaction protocols, such as ChatDev and MapCoder, tend to be less effective. On hard tasks, ChatDev requires over 30 calls yet only achieves 11.7%, while MapCoder performs even worse, with 5.87% despite 23.41 calls. These results suggest that handcrafted workflows may introduce redundant operations without improving code generation quality.

4.1.3 LLM Agent Inference

The right part in Figure 3 reports the performance of tool-augmented **LLM agents** evaluated on SWE-

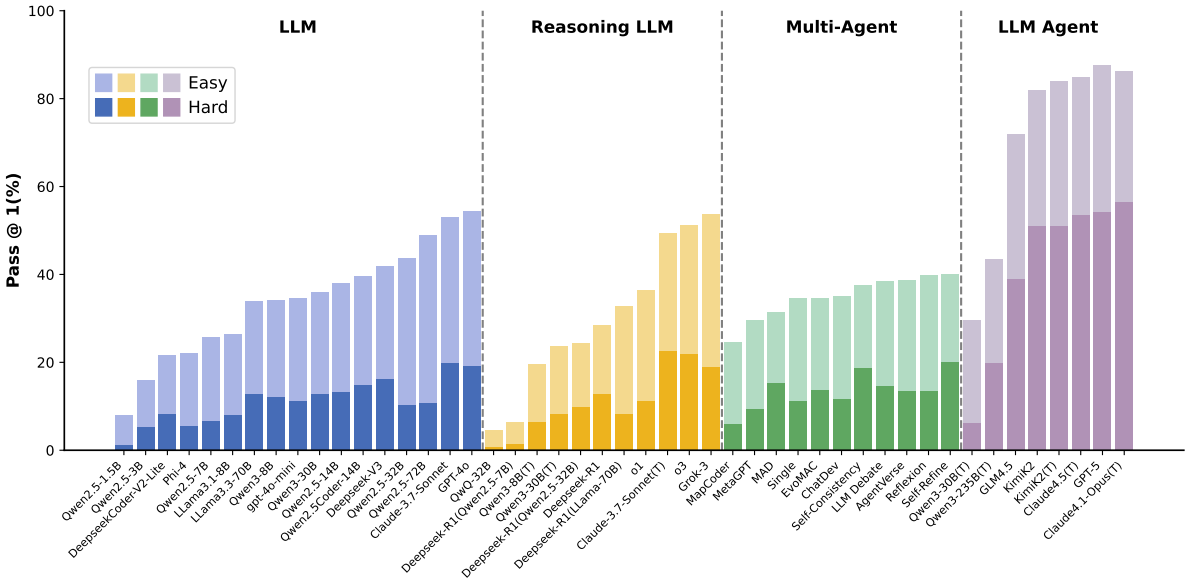


Figure 3: Pass@1 comparison of single LLMs, reasoning-focused LLMs, multi-agent systems, and tool-augmented LLM agents on SWE-Dev. Paired bars report *Easy* and *Hard* splits, highlighting the difficulty of hard feature-development tasks and the gains from agentic execution-feedback workflows.

Dev using the OpenHands (Wang et al., 2024) framework (see Table 11 in Appendix D.3 for detailed number). Compared to direct single-LLM inference, agentic execution with tools and test feedback leads to substantial performance improvements.

Agentic execution significantly improves feature development performance. Across all evaluated models, LLM agents consistently outperform their single-turn counterparts. Strong agents achieve over 50% Pass@1 on the *Hard* split (e.g., Claude4.1-Opus-thinking at 56.44% and GPT-5 at 54.21%), representing more than a 2× improvement over the best single-LLM results. This demonstrates the importance of iterative execution, error diagnosis, and repair when tackling repository-level feature development.

Reasoning-enabled agents are strong under execution feedback. With OpenHands, reasoning-enabled (Thinking) agents achieve high performance on the *Hard* split (e.g. Kimi-K2-Thinking is comparable to Kimi-K2-Instruct (50.86% vs. 51.05% on *Hard* split), suggesting that the benefits of explicit reasoning are context-dependent and become most meaningful when coupled with execution signals.

Despite strong gains, feature development remains unsolved. Even with agentic scaffolding, a large fraction of tasks on the *Hard* split remain unsolved, highlighting the inherent difficulty of end-to-end feature development and leaving substantial headroom for future advances in agent design, planning, and long-horizon reasoning.

4.2 Training Results

In this section, we evaluate SWE-Dev's support for different training methods, including SFT, RL. Additionally, we present preliminary results from our exploration of multi-agent training, offering an initial assessment of MAS-based learning. For detailed training setups, refer to the Appendix H.2.

4.2.1 Single LLM SFT

We conducted experiments on Qwen2.5-Instruct models of various sizes (0.5B, 1.5B, 3B, and 7B) to assess the impact of SFT on performance in SWE-Dev. Experimental setting is in Appendix H.3.

Training significantly improves performance across model sizes. SFT leads to substantial performance improvements across all model sizes, especially for harder tasks. As shown in Table 2, the 7B model achieves a Pass@1 of 36.90% on the easy task set after fine-tuning, up from 25.74% in the zero-shot setting. On the hard task set, the Pass@1 increases from 6.68% to 18.89%, demonstrating the clear benefits of training in enhancing model performance.

SWE-Dev effectively supports the scaling law of training. We also measure model performance across varying amounts of fine-tuning data using Qwen2.5-7B-Instruct, specifically tracking changes in Pass@1 for both easy and hard tasks. Performance improves steadily as the

Table 2: **SFT Training** Comparison of zero-shot and SFT performance (Pass@1) on SWE-Dev using Qwen2.5 models. Results are reported on both Easy and Hard test splits across model sizes from 0.5B to 7B. The Δ columns indicate relative improvement after SFT.

	Zero-shot		SFT			
	Easy	Hard	Easy	Δ(%)	Hard	Δ(%)
0.5B	6.39	1.00	12.12	+90	4.37	+337
1.5B	8.05	1.23	18.20	+126	7.64	+521
3B	15.93	5.27	27.53	+73	12.46	+136
7B	25.74	6.68	36.90	+43	18.89	+183

Table 3: **RL Training** Performance comparison of Qwen2.5-7B-Instruct as base model, SFT-Tuned and RL-Tuned models on SWE-Dev.

Method	Pass@1		Pass@3	
	Easy	Hard	Easy	Hard
vanilla	25.74	6.68	33.35	7.73
SFT	27.09	9.77	**34.49**	13.63
PPO (online RL)	**28.30**	**12.25**	32.69	14.33
DPO (offline RL)	25.89	10.36	31.32	**14.66**

amount of fine-tuning data increases, with larger improvements observed for harder tasks (see Appendix H.3).

In summary, our results underscore the importance of fine-tuning in improving performance on SWE-Dev. The scaling law observed here further supports the idea that SWE-Dev is a valuable dataset for studying the effects of model size and training data on task performance.

4.2.2 Single LLM RL

SWE-Dev provides precise test cases enabling accurate rewards for coding tasks, supporting both online and offline RL. In this section, we explore the impact of RL on the Qwen2.5-7B-instruct using SWE-Dev. Considering the computational cost of RL, we limit our experiments in this section to 2k training samples. For full training setup, refer to the Appendix H.4.

Both online and offline RL improve performance, especially on hard tasks. Table 3 shows that both PPO (Schulman et al., 2017) and DPO (Rafailov et al., 2023) significantly improve Pass@1 performance, especially on the *Hard* split. Furthermore, PPO outperforms SFT (2k, aligned with RL) on the same training data. These findings highlight the advantages of RL training.

RL boosts one-shot success but not multi-sample gains. While RL fine-tuning yields improvements in Pass@1, it has minimal impact on Pass@3. Specifically, PPO achieves a Pass@1 of 28.30% on easy tasks, a noticeable increase from the base model's 25.74%, but the Pass@3 remains lower than the SFT-training, even the original model's performance. These results suggest that RL can be beneficial in refining Pass@1, particularly for more complex tasks, by increasing the model's efficiency in generating correct answers in fewer attempts. However, this efficiency comes at the cost of reduced exploration. This aligns with findings from prior work (Yue et al., 2025). Therefore, while RL improves performance, significant headroom remains, and more advanced methods or further training are needed to achieve improvements across tasks.

Table 4: **MAS Training:** Role-wise training of EvoMAC with Qwen2.5-7B-Instruct, where Org and Coder denote the organizer and coding agents. Δ is relative to the base MAS; *FT* denotes fine-tuning.

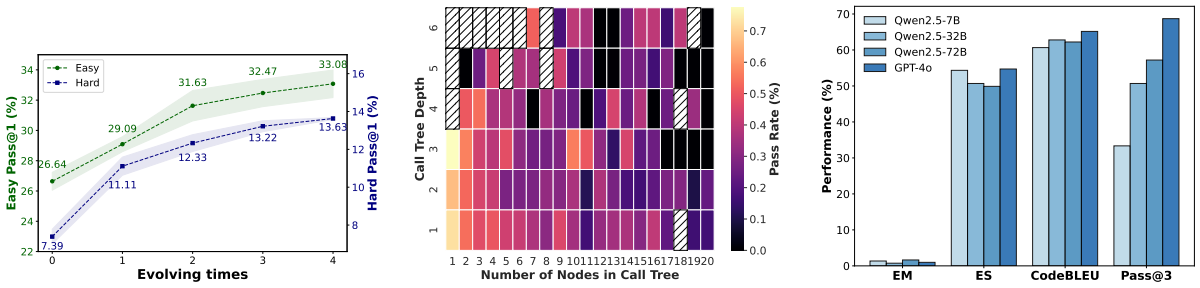
	Org	Coder	Easy	Δ(%)	Hard	Δ(%)
Single	-	-	25.74	-	6.68	-
MAS	base	base	26.64	-	7.39	-
	FT	base	30.04	+12.76	12.36	+67.25
	base	FT	31.42	+17.94	11.49	+55.48
	FT	FT	**31.65**	+18.80	**12.70**	+71.85

4.2.3 Multi-Agent Training

MAS has shown promising results on SWE-Dev, and we further investigate the training process of MAS on this dataset. As depicted in Fig. 4a, the ground truth test case supervision in SWE-Dev enables EvoMAC (Hu et al., 2025b) to improve its performance across multiple rounds of reasoning. This iterative refinement process motivates us to explore EvoMAC as the MAS for training in SWE-Dev. We apply rejection sampling to enhance agent performance via role-wise training.

Trajectory Collection. We use Qwen2.5-7B-Instruct to collect training data for the MAS, following these steps: (1) **EvoMAC iterative reasoning**: EvoMAC performs multiple reasoning rounds, benefiting from ground truth test case supervision to progressively improve its performance. (2) **Rejection sampling**: At each iteration, we apply rejection sampling based on training sample testcases to select high-quality trajectories that show improvement over the previous round, ensuring the retention of beneficial data. (3) **Role-wise training**: The selected trajectories are used to role-wise train two agents (organizer and coder) in EvoMAC, allowing each agent to specialize in its task for better overall performance.

Training Effectiveness. Table 4 presents the



(a) EvoMAC trajectory (b) Complexity analysis (c) Metric comparison

Figure 4: Analysis of SWE-Dev benchmark and training characteristics. (a) EvoMAC improves across test-supervised reasoning rounds. (b) Performance decreases with call-tree complexity. (c) Pass@3 better separates model capabilities than static metrics.

performance of different training configurations in terms of Pass@1. We see that: i) Fine-tuning both the organizer and coder agents results in the highest performance, with Pass@1 of 31.65% on easy tasks and 12.70% on hard tasks, outperforming all other configurations; ii) When only one agent is fine-tuned, we also see improvements over the baseline. These findings highlight the effectiveness of role-wise training for MAS training.

Table 5: Head-to-head comparison of training data sources under matched training settings. We report Pass@1 (%); small colored numbers denote absolute gains or losses over the Qwen3-32B base model.

	SWE-Bench-Verified	SWE-Dev	
Training Data	**Pass@1**	**Easy**	**Hard**
Qwen3-32B	45.6	32.5	9.3
+ SWE-Dev	46.6 (+1.0)	**38.6** (+6.1)	**12.9** (+3.6)
+ SWE-Smith	50.2 (+4.6)	33.1 (+0.6)	9.4 (+0.1)
+ SWE-ReBench	**50.8** (+5.2)	32.9 (+0.4)	9.2 (-0.1)

5 Dataset Analysis

We analyze SWE-Dev's dataset validity from three perspectives: task complexity, evaluation metric reliability, and training-data complementarity.

Complexity Analysis. We first analyze whether SWE-Dev difficulty reflects structural complexity. As introduced in § 3.1, call trees reflect dynamic function invocation structures, where nodes represent functions and edges denote call relationships. Fig. 4b shows that GPT-4o's performance declines as call-tree depth and node count increase, suggesting that deeper and broader functional dependencies make feature implementation more challenging.

Metric Analysis. We further compare Exact Match (EM) (Liu et al., 2024), Exact Sequence (ES) (Liu et al., 2024), CodeBLEU (Ren et al., 2020), and Pass@3 using Qwen2.5 models and GPT-4o. As Fig. 4c shows, Pass@3 best reflects capability scaling, while EM, ES, and CodeBLEU show limited variance. These results support SWE-Dev's use of execution-based tests as a more informative signal for repository-level functional correctness.

Training Data Complementarity. Table 5 compares training on SWE-Dev, SWE-Smith, and SWE-ReBench under matched settings. SWE-Smith and SWE-ReBench transfer better to SWE-Bench-Verified, consistent with their issue-repair focus. In contrast, SWE-Dev gives the largest gains on SWE-Dev Easy and Hard, improving the base model by 6.1 and 3.6 points. This indicates that SWE-Dev provides complementary supervision for feature implementation rather than duplicating existing issue-repair datasets.

6 Conclusion

In this work, we introduced SWE-Dev, the first dataset for evaluating and training autonomous coding systems on end-to-end feature-driven development task. SWE-Dev consists of 14k training and 500 test instances, each uniquely equipped with runnable environments and developer-authored executable unit tests, which provides essential execution-based feedback for advanced training paradigms like SFT, RL, and multi-agent learning. Our experiments show feature-driven software development is profoundly challenging for current autonomous coding systems. We also validate that training on SWE-Dev can yield encouraging performance gains. These findings validate SWE-Dev as a critical platform for identifying current limitations and fostering breakthroughs in AI-driven software development. We hope the release of SWE-Dev spurs innovation in agent orchestration, and execution-aware training towards genuinely autonomous software engineering.

7 Limitations

SWE-Dev currently focuses on Python repositories with executable unit tests. While this setting covers a large portion of open-source development, it does not fully represent projects in other programming languages or workflows evaluated through integration tests, UI tests, or product-level acceptance criteria. SWE-Dev is derived from selected open-source repositories and developer-authored tests. It may therefore inherit biases from these projects, and cannot capture some widely-used but propietary applications.

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, and 1 others. 2024. Phi-4 technical report. arXiv preprint arXiv:2412.08905.

Anthropic. 2025. Claude 3.7 Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: May 10, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. arXiv preprint arXiv:2108.07732.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In Proceedings of the 26th annual international conference on machine learning, pages 41–48.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, and 1 others. 2025. Skyrl-agent: Efficient rl training for multi-turn llm agent. arXiv preprint arXiv:2511.16108.

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.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, and 1 others. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In The Twelfth International Conference on Learning Representations.

Peter Coad, Eric Lefebvre, and Jeff De Luca. 1999. Java Modeling in Color with UML: Enterprise Components and Process. Prentice Hall PTR.

Cursor. 2023. Cursor - The AI-first Code Editor. https://www.cursor.com/. Accessed: May 10, 2025.

Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2023. Safe rlhf: Safe reinforcement learning from human feedback. arXiv preprint arXiv:2310.12773.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 9 others. 2025. Deepseek-v3 technical report. Preprint, arXiv:2412.19437.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, and 21 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. Preprint, arXiv:2406.11931.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. Advances in Neural Information Processing Systems, 36:46701–46723.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. In Forty-first International Conference on Machine Learning.

Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 2024. Complexcodeeval: A benchmark for evaluating large code models on more complex code. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pages 1895–1906.

GitHub. 2021. GitHub Copilot. https://github.com/features/copilot. Accessed: May 10, 2025.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, and 1 others. 2025. Training long-context, multi-turn software engineering agents with reinforcement learning. arXiv preprint arXiv:2508.03501.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948.

Harness Team. 2025. Getting started with feature driven development: A comprehensive guide. https://www.harness.io/harness-devops-academy/getting-started-with-feature-driven-development-a-comprehensive-guide. Accessed: May 10, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. Measuring coding challenge competence with apps. In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2).

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. Preprint, arXiv:2308.00352.

Jian Hu, Jason Klein Liu, and Wei Shen. 2025a. Reinforce++: An efficient rlhf algorithm with robustness to both prompt and reward models. arXiv preprint arXiv:2501.03262.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2025b. Self-evolving multi-agent collaboration networks for software development.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin B Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. CoRR.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. Preprint, arXiv:2409.12186.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. arXiv preprint arXiv:2410.21276.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. Preprint, arXiv:2405.11403.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. arXiv preprint arXiv:2504.07164.

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

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. Advances in Neural Information Processing Systems, 35:21314–21328.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. Science, 378(6624):1092–1097.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging divergent thinking in large language models through multi-agent debate. Preprint, arXiv:2305.19118.

Jiaheng Liu, Ken Deng, Congnan Liu, Jian Yang, Shukai Liu, He Zhu, Peng Zhao, Linzheng Chai, Yanan Wu, Ke Jin, and 1 others. 2024. M2rc-eval: Massively multilingual repository-level code completion evaluation. arXiv preprint arXiv:2410.21157.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code autocompletion systems. In The Twelfth International Conference on Learning Representations.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. 2025. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl. https://pretty-radio-b75.notion.site/DeepSWE-Training-a-Fully-Open-sourced-State-of-the-Art-Coding-Agent-by-Scaling-RL-22281902c1468193aabbe9a8c59bbe33. Notion Blog.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In The Twelfth International Conference on Learning Representations.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. Preprint, arXiv:2303.17651.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, and etal. 2024. Openai o1 system card. Preprint, arXiv:2412.16720.

OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence. Accessed: May 10, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. Advances in neural information processing systems, 35:27730–27744.

Steve R Palmer and Mac Felsing. 2001. A practical guide to feature-driven development. Pearson Education.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. arXiv preprint arXiv:2412.21139.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. 2025. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. arXiv preprint arXiv:2504.14757.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Chatdev: Communicative agents for software development. Preprint, arXiv:2307.07924.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025a. Qwen2.5 technical report. Preprint, arXiv:2412.15115.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025b. Qwen2.5 technical report. Preprint, arXiv:2412.15115.

Qwen. 2025. Qwen3: Think deeper, act faster. https://qwenlm.github.io/blog/qwen3/. Accessed: May 10, 2025.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. Advances in Neural Information Processing Systems, 36:53728–53741.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. Preprint, arXiv:2009.10297.

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

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36:8634–8652.

Charlie Snell, Ilya Kostrikov, Yi Su, Mengjiao Yang, and Sergey Levine. 2023. Offline rl for natural language generation with implicit language q learning. Preprint, arXiv:2206.11871.

Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. Featureide: An extensible framework for feature-oriented software development. Science of Computer Programming, 79:70–85.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. arXiv preprint arXiv:2407.16741.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023a. Self-consistency improves chain of thought reasoning in language models. Preprint, arXiv:2203.11171.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, and 1 others. 2023b. How far can camels go? exploring the state of instruction tuning on open resources. Advances in Neural Information Processing Systems, 36:74764–74786.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. arXiv preprint arXiv:2502.18449.

xAI. 2025. Grok 3 Beta — The Age of Reasoning Agents. https://x.ai/news/grok-3. Accessed: May 10, 2025.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In The Twelfth International Conference on Learning Representations.

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

Rui Ye, Keduan Huang, Qimin Wu, Yuzhu Cai, Tian Jin, Xianghe Pang, Xiangrui Liu, Jiaqi Su, Chen Qian, Bohan Tang, Kaiqu Liang, Jiaao Chen, Yue Hu, Zhenfei Yin, Rongye Shi, Bo An, Yang Gao, Wenjun Wu, Lei Bai, and Siheng Chen. 2025a. Maslab: A unified and comprehensive codebase for llm-based multi-agent systems. Preprint, arXiv:2505.16988.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. 2025b. Mas-gpt: Training llms to build llm-based multi-agent systems. Preprint, arXiv:2503.03686.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pages 1–12.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. Preprint, arXiv:2503.14476.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? Preprint, arXiv:2504.13837.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. 2024. Archer: Training language model agents via hierarchical multi-turn rl. Preprint, arXiv:2402.19446.

A LLM Usage Disclosure

In our work, we utilized GPT-4o exclusively to enhance readability and improve language fluency during the writing process. We take full responsibility for the content of this publication, including any content generated by LLM.

B Other Related Work

RL for Software Engineering Earlier RL and RLHF work on code has mostly targeted short, function-level problems (e.g., HumanEval-style benchmarks), where the model produces a single solution in one shot with an immediate test-based reward. More recently, several works have applied RL to more complex SWE settings (Golubev et al., 2025). SWE-RL (Wei et al., 2025) uses policy-gradient RL in an agentless scaffold on SWE-Bench, but still frames the task as single-turn patch generation, which sidesteps the challenges of stateful, multi-step interaction and long-horizon credit assignment. In contrast, DeepSWE (Luo et al., 2025) and SkyRL-v0 (Cao et al., 2025) scale RL to interactive, multi-turn SWE agents operating over full repositories, tools, and long contexts. Our work is complementary: rather than proposing a new agent or RL algorithm, SWE-Dev provides a large, PRD-driven, feature-level benchmark with executable rewards and examines how standard SFT and PPO behave on multi-file feature implementation. This makes SWE-Dev a natural substrate for future RL-based SWE agents in the spirit of these recent works.

C Dataset

C.1 Dataset Statistics

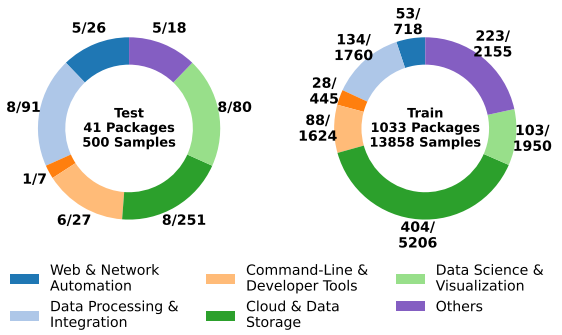


Figure 5: Distribution of SWE-Dev training and test samples across 6 major PyPI application domains.

Table 6 summarizes the key statistics of SWE-Dev, which consists of 14,000 training and 500 test samples. The test set is manually curated and split into two difficulty levels: *easy* and *hard* (250 instances each). Each dataset instance comprises four components: (1) the task, specified by a PRD, with its token count reflecting instruction length; (2) the codebase, referring to the non-test portion of the repository, where we report the number of files and lines of code (LOC); (3) the ground truth (GT) code to be implemented, measured by its LOC and number of target functions; and (4) the test suite, evaluated via the number of test cases and total test LOC per sample. Figure 5 shows the distribution of training and test samples across six major PyPI application domains, demonstrating the diversity of software categories represented in the dataset. SWE-Dev is built from publicly available open-source repositories filtered by declared licenses that permit research use and redistribution of derived benchmark artifacts.

Table 6: Basic statistics of SWE-Dev, including task specification length, repository scale, ground truth implementation size, and evaluation test coverage for both train and test splits.

Category	Metric	Test		Train
		Easy	Hard	
Size	# Samples	250	250	14000
	# Total repos	39	39	1033
Task	# Avg. tokens	1499	2148	1833
Codebase	# Avg. Files	71.31		64.40
	# Avg. LOC	21308		20206
GT Code	# Avg. LOC	109.1	172.4	199.92
	# Avg. funcs	4.75	6.972	6.03
Tests	# Avg. test lines	134.8	123.9	90.9
	# Avg. testcases	6.62	4.29	5.92

C.2 Dataset Information

Figure 6 illustrates a typical task instance in the SWE-Dev, detailing the entire development workflow. The process begins with the Project Requirement Description (PRD), which provides instructions and specifies features to be implemented. Methods to be evaluated then generate code to complete the features mentioned in the PRD, which is subsequently verified against the test suite to produce pass/fail results to calculate pass rate. Additionally, the ground truth implementation for each PRD is included for reference. The tasks in SWE-Dev simulate real-world software development cycles within a repository context. For detailed information about each data field included in SWE-Dev

tasks, please refer to Table 7.

C.3 Dataset Distribution

We present the distribution statistics of the training and test sets in SWE-Dev. Each sample includes a Project Requirement Document (PRD), which describes the feature to be implemented. The average PRD length is 1,845.4 tokens. On average, each sample includes at least 5 unit tests for functional evaluation, spans 3 source files, and requires the implementation of approximately 6 functions.

C.4 Dataset Diversity

We assess the diversity of SWE-Dev from two perspectives: sample-level diversity (Figure 8 and Figure 9), and package-level diversity (Figure 26).

Sample Diversity via t-SNE. To visualize the diversity of feature requirements, we perform t-SNE on PRD embeddings generated using OpenAI's text-embedding-ada-002 model.[1] We use 500 test samples and randomly sample 2,000 training samples. Each point represents a PRD, and the color denotes its corresponding package. The resulting distribution reveals rich semantic variation across tasks, even within the same package, highlighting the dataset's diversity in both content and functionality.

Package Category Diversity. To analyze the functional diversity of the dataset, we classify packages into high-level categories based on their primary domain (e.g., web development, data science, utilities). The classification is performed using GPT-4o with the prompt provided below (see Figure 26). The resulting distribution confirms that SWE-Dev spans a broad spectrum of software domains.

D Inference Results

To assess the capabilities and limitations of current LLMs on realistic feature-driven software development tasks, we conduct comprehensive inference-time evaluations on SWE-Dev. We study both single-agent and multi-agent systems, measuring their performance under consistent execution-based evaluation.

D.1 Single-Agent LLM Performance

We evaluate 27 state-of-the-art LLMs, including general-purpose chatbot models (e.g., GPT-4o, Claude 3.7) and reasoning models, shown in Table 9. Models are assessed using Pass@1 and Pass@3 on SWE-Dev's test set. To contextualize benchmark difficulty, we also compare results on HumanEval (Chen et al., 2021) and ComplexCodeEval (Feng et al., 2024), using Pass@3 and CodeBLEU respectively. Our findings show that SWE-Dev poses significantly greater challenges than existing benchmarks, with leading models achieving under 30% Pass@3 on hard tasks (Table 8).

D.2 Multi-Agent System Performance

We evaluate 10 multi-agent systems (MAS), including both general-purpose MAS (e.g., AgentVerse, LLM Debate) and code-specific designs (e.g., EvoMAC, MapCoder). As detailed in Table 10, we compare each MAS against a single-agent baseline on execution success (Pass@1), total API call count, and cost-efficiency. Results show that while MAS can outperform single agents on complex tasks, simple strategies (e.g., Self-Refine) often strike a better balance between performance and resource usage than workflow-heavy systems like ChatDev. We visualize the MAS in the following Figure 10.

D.3 Agent LLM Results

Agent Evaluation Setup. All agent-based experiments are conducted using the **OpenHands** (Wang et al., 2024) framework. We use a unified configuration for all agent LLMs: temperature $= 0.6$, maximum agent steps $= 100$, execution timeout $= 300$ seconds per task, and maximum context length $= 64$K tokens. Performance is measured by executing the original SWE-Dev unit tests in the provided environments and reporting Pass@1 on the Easy and Hard splits.

E Analysis

E.1 Evaluation Metric Validness

SWE-Dev provides stable and discriminative evaluation of model capabilities. Figure 11 compares the performance of Qwen2.5 (Qwen et al., 2025a) family on SWE-Dev, HumanEval (Chen et al., 2021), and ComplexCodeEval (Feng et al., 2024) across three runs. We use Pass@1 for SWE-Dev and HumanEval and ComplexCodeEval for CodeBLEU (Ren et al., 2020). The lines represent the average performance, and the shaded regions show the variance. We observe that: (1) SWE-Dev

[1] https://platform.openai.com/docs/guides/embeddings

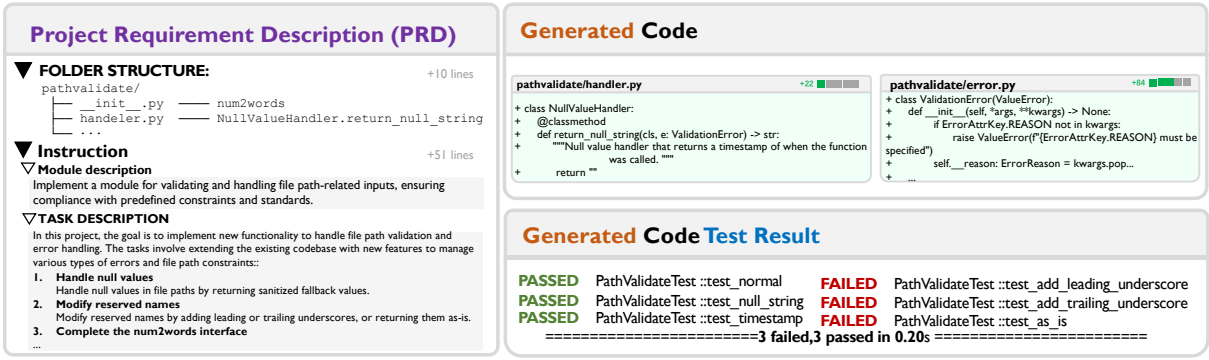


Figure 6: Example of a SWE-Dev Sample. Each sample includes a Project Requirement Description (PRD) with folder structure, module-level task description, and masked docstrings; the corresponding ground truth implementation. Generated code is evaluated by the test function execution results. This structure supports realistic, testable feature development in a repository context.

yields low variance performance and consistent scaling with model size, demonstrating SWE-Dev's stability and reliability in evaluating model capabilities. (2) In contrast, HumanEval—despite being stable—is too simple to differentiate models meaningfully. (3) Meanwhile, ComplexCodeEval shows high variance due to its reliance on similarity-based metrics, CodeBLEU, which limits its reliability for evaluating complex generation tasks.

E.2 Analysis of PRD Quality

Our Project Requirement Descriptions (PRDs) are primarily derived from the original docstrings found within the repository source code. To enhance the quality and utility of these PRDs, we employed GPT-4o to refine and improve the original docstrings. To objectively assess this improvement, we recruited two domain experts to evaluate both the original and GPT-4o-enhanced docstrings across 100 randomly selected samples from SWE-Dev.

The experts rated each docstring on three critical dimensions—Clarity, Completeness, and Actionability—using a 0 to 5 scale, where higher scores indicate superior quality. The human evaluation guideline is shown in Figure 27.

Participants were fully informed about the evaluation process and the nature of the task. The assessment involved only reviewing documentation and posed no ethical or privacy risks, adhering strictly to ethical standards for research involving human subjects. This evaluation provides a rigorous measure of how GPT-4o-refined docstrings enhance PRD quality in SWE-Dev.

E.3 Analysis on the Underperformance of Reasoning Models

Instruction Following Rate (IFR). Previous experiments have shown that reasoning models perform poorly on SWE-Dev. To investigate the reasons behind this, we analyzed the instruction-following ability of these models. We measured the percentage of code files that meet the PRD requirements for each model's generated code as a metric of instruction following rate (IFR). The metric is formally defined as:

$$\text{IFR} = \frac{1}{n} \sum_{i=1}^{n} \frac{|\mathcal{G}_i \cap \mathcal{T}_i|}{|\mathcal{T}_i|}$$

where n denotes the total number of tasks, $\mathcal{G}_i$ represents the set of files generated by the model for task i, and $\mathcal{T}_i$ denotes the set of ground truth files required by the PRD for task i.

To further explore this, we compared reasoning models with their chatbot counterparts by evaluating their instruction following rate. Specifically, in Figure 13, the x-axis represents the instruction following rate, and the y-axis shows the performance of both reasoning models and their chatbot counterparts on tasks where their instruction following rate is 100%.

As shown in the figure, we see that: (i) Reasoning models generally have a lower instruction-following rate compared to their chatbot version, which explains why they underperform when handling multiple tasks simultaneously. Reasoning models tend to struggle with tasks on SWE-Dev that involve performing several steps in a single call, resulting in poorer performance overall. (ii)

Table 7: Description of each field of an SWE-Dev task instance object.

Field	Description
PRD	A natural language document describing the project requirements, including the specific features to be implemented.
file_code	Incomplete code contents of the core files involved in the task.
test_code	Content of the test code used to verify the task's functionality.
dir_path	Root directory path of the project corresponding to this task instance.
package_name	Name of the software package or module to which this task instance belongs.
sample_name	Unique identifier or name for this task instance or sample within the benchmark.
src_dir	Relative directory path where the source code files for the project or task are located.
test_dir	Relative directory path where the test code files for the project or task are located.
test_file	Relative path of the unit test file used for executing tests.
GT_file_code	Ground Truth source code for the file to complete.
GT_src_dict	Ground Truth source dictionary, mapping file names/paths to their expected correct code content.
dependency_dict	Dictionary describing the dependencies required by the current task (e.g., internal modules) and their relationships.
call_tree	Function call tree or call graph of the code, representing the relationships between function calls.

However, on tasks where both reasoning models and their chatbot versions have an instruction-following rate of 100%, reasoning models typically outperform the chatbots. This indicates the potential of reasoning models when they can fully adhere to instructions. (iii) Claude 3.7-Sonnet is an exception to this trend, as both its reasoning and chatbot versions exhibit similar instruction-following rates and performance, which contributes to Claude's superior results.

E.4 Analysis of MAS

While multi-agent systems (MAS) demonstrate performance gains over single-agent baselines on SWE-Dev, their behaviors vary significantly across designs. To better understand these differences, we analyze MAS from a communication and coordination perspective. Specifically, we examine how agent structure and interaction overhead relate to execution success on repository-level feature development tasks.

Table 12 reveals two consistent trends. First, simple and low-overhead MAS achieve the best performance–efficiency trade-off. General-purpose methods such as Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023) use only two to three roles and require relatively few model calls. Despite their simplicity, they reach the highest Pass@1 scores. For example, Reflexion achieves 39.77% Pass@1 with only 2.12 LLM calls and 10.77 agent-to-agent messages on average. These systems rely on iterative self-refinement rather than explicit division of labor. As a result, the global task context remains concentrated within a small number of agents, reducing coordination cost.

Second, workflow-heavy and code-specific MAS suffer from coordination overhead and context fragmentation. Systems such as MetaGPT (Hong et al., 2024) and MapCoder (Islam et al., 2024) introduce more roles, more calls, and substantially more inter-agent communication. However, this increased interaction does not translate into better execution success. MapCoder, for instance, uses over 21 LLM calls and exchanges more than 60 agent-to-agent messages on average, yet achieves the lowest Pass@1 among all compared methods. Each additional handoff between agents introduces opportunities for partial context loss and error propagation. This effect is particularly harmful on SWE-Dev

Table 8: Evaluation of model performance across benchmarks. This table compares 37 general-purpose and reasoning-focused LLMs on SWE-Dev (Pass@1 and Pass@3 for Easy and Hard splits), ComplexCodeEval (CodeBLEU), and HumanEval (Pass@1 and Pass@3).

	Abbreviated Name	**Model**	**SWE-Dev pass@1**		**SWE-Dev pass@3**		**ComplexCodeEval**	**HumanEval**	**HumanEval**
			Easy	Hard	Easy	Hard	**CodeBLEU**	**pass@1**	**pass@3**
Chatbot	Qwen2.5-1.5B	Qwen2.5-1.5B-Instruct	8.05%	1.23%	10.76%	2.22%	29.72%	57.20%	69.76%
	Qwen2.5-3B	Qwen2.5-3B-Instruct	15.93%	5.27%	21.99%	7.47%	12.27%	62.68%	75.00%
	Qwen2.5-7B	Qwen2.5-7B-Instruct	25.74%	6.68%	33.35%	7.73%	20.00%	82.68%	87.13%
	Llama3.1-8B	Llama-3.1-8B-Instruct	26.43%	7.94%	33.01%	10.24%	20.18%	68.20%	77.87%
	Qwen3-8B	Qwen3-8B	34.04%	12.09%	39.26%	13.33%	17.47%	86.34%	89.09%
	Qwen2.5Coder-14B	Qwen2.5-Coder-14B-Instruct	39.51%	14.82%	52.49%	18.44%	35.52%	90.48%	92.93%
	Qwen2.5-14B	Qwen2.5-14B-Instruct	38.08%	13.16%	46.32%	15.89%	19.90%	82.56%	87.87%
	DeepseekCoder-V2-Lite	DeepSeek-Coder-V2-Lite-Instruct	21.53%	8.19%	29.68%	11.33%	26.63%	80.98%	85.18%
	Qwen3-30B	Qwen3-30B-A3B	35.84%	12.76%	39.45%	15.20%	14.84%	89.27%	90.30%
	Phi-4	Phi-4	21.99%	5.57%	27.89%	8.56%	33.85%	86.46%	90.01%
	Qwen2.5-32B	Qwen2.5-32B-Instruct	43.64%	10.15%	51.24%	11.69%	19.76%	88.90%	92.44%
	Qwen2.5-72B	Qwen2.5-72B-Instruct	49.01%	10.62%	57.20%	12.33%	22.15%	83.66%	86.46%
	Llama3.3-70B	Llama3.3-70B-Instruct	33.84%	12.85%	39.57%	14.95%	21.29%	84.51%	88.54%
	Deepseek-V3	Deepseek-V3	41.95%	16.22%	56.79%	21.62%	28.32%	90.36%	92.92%
	GPT-4o	GPT-4o	**54.37%**	19.13%	**68.70%**	21.91%	33.38%	88.41%	92.93%
	gpt-4o-mini	GPT-4o-mini	34.47%	11.09%	41.94%	13.84%	25.00%	85.97%	89.00%
	Claude-3.7-Sonnet	Claude-3.7-Sonnet	53.09%	19.74%	56.35%	24.25%	29.63%	93.66%	95.36%
Reasoning	Claude-3.7-Sonnet(T)	Claude-3.7-Sonnet-thinking	49.47%	**22.51%**	56.58%	**29.28%**	**29.80%**	91.22%	97.62%
	Deepseek-R1(Qwen2.5-7B)	Deepseek-R1-distill-Qwen2.5-7B	6.30%	1.29%	10.30%	1.95%	21.05%	86.10%	93.29%
	Qwen3-8B(T)	Qwen3-8B-thinking	19.47%	6.36%	25.91%	9.22%	20.98%	89.63%	91.89%
	Qwen3-30B(T)	Qwen3-30B-A3B-thinking	23.63%	8.30%	31.00%	11.60%	25.00%	93.04%	99.57%
	Deepseek-R1(Qwen2.5-32B)	Deepseek-R1-distill-Qwen2.5-32B	24.25%	9.79%	40.53%	19.04%	27.98%	95.17%	97.87%
	Deepseek-R1(Llama-70B)	DeepSeek-R1-distill-Llama-70B	32.73%	8.19%	45.72%	11.33%	25.95%	96.95%	98.53%
	Deepseek-R1	Deepseek-R1-671B	28.55%	12.84%	37.62%	17.72%	34.47%	**98.65%**	**100%**
	QwQ-32B	QwQ-32B-Preview	4.50%	0.70%	8.90%	1.22%	24.78%	82.31%	97.01%
	Grok-3	grok-3-beta	**53.63%**	18.97%	59.08%	22.26%	27.96%	87.15%	89.99%
	o1	o1	36.36%	11.09%	43.77%	14.27%	33.63%	97.43%	98.78%
	o3	o3	51.21%	21.86%	59.05%	28.98%	26.53%	98.04%	98.78%

tasks, which already involve long code contexts and multi-file dependencies.

Overall, these results suggest that, for repository-level feature development, maintaining a stable and unified reasoning context is more important than fine-grained role specialization. Simple self-refinement style MAS exhibit an emergent robustness by minimizing communication overhead while preserving global task coherence. In contrast, heavily engineered multi-role workflows tend to accumulate coordination cost without corresponding performance gains.

E.5 Error Analysis

Figure 14 presents the distribution of failure types for both single-agent and multi-agent systems on SWE-Dev. We sample 500 samples for error analysis and categorize errors into five types: Incomplete, Logic, Syntax, Parameter, and Others, see error classification prompt in Figure 24. Across both agent types, the most prevalent error is the Incomplete Error, where models fail to implement PRD-required functions—indicating persistent challenges in task decomposition and execution coverage.

For single-agent models, Logic Errors are the second most common, followed by Parameter Errors and Syntax Errors. Interestingly, GPT-4o and Claude-3.7 show relatively fewer Syntax Errors, suggesting better adherence to Python syntax, while smaller models like GPT-4o-mini show higher incidence of both Syntax and Parameter issues, reflecting their limited reasoning capacity and weaker control over function signatures.

In contrast, multi-agent systems exhibit a different pattern. While they reduce Incomplete Errors to some extent, they often incur higher Logic or Syntax Errors—especially in methods like MAD and Self-consistency—suggesting that while agents may cover more PRD content, coordination breakdown or hallucinated reasoning steps can introduce new failure modes.

Overall, the analysis highlights the need for improved function selection, robust reasoning alignment, and stronger control over generation structure—especially in collaborative multi-agent settings.

E.6 Limitation and Future Work

Language Scope. SWE-Dev currently targets Python repositories with executable unit tests, which, while widely used, does not reflect the full diversity of real-world programming languages or projects whose expected behavior is specified through integration tests, manual evaluation, or product-level acceptance criteria. A natural extension is to support other major languages such as Java, JavaScript, and C++, enabling broader evalu-

Table 9: Information of evaluated LLMs.

Model	Size	Release Date	Open	Link
Qwen/Qwen2.5-Coder-14B-Instruct	14B	2024-11-12	✓	Qwen2.5-Coder-14B-Instruct
deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct	16B	2024-06-17	✓	DeepSeek-Coder-V2-Lite-Instruct
microsoft/phi-4	14B	2024-12-12	✓	phi-4
Qwen/Qwen2.5-1.5B-Instruct	1.5B	2024-09-19	✓	Qwen2.5-1.5B-Instruct
Qwen/Qwen2.5-3B-Instruct	3B	2024-09-19	✓	Qwen2.5-3B-Instruct
Qwen/Qwen2.5-7B-Instruct	7B	2024-09-19	✓	Qwen2.5-7B-Instruct
meta-llama/Llama-3.1-8B-Instruct	8B	2024-07-23	✓	Llama-3.1-8B-Instruct
Qwen/Qwen3-8B	8B	2025-04-29	✓	Qwen3-8B
Qwen/Qwen2.5-14B-Instruct	14B	2024-09-19	✓	Qwen2.5-14B-Instruct
Qwen/Qwen3-30B-A3B	30B	2025-04-29	✓	Qwen3-30B-A3B
Qwen/Qwen2.5-32B-Instruct	32B	2024-09-19	✓	Qwen2.5-32B-Instruct
Qwen/Qwen2.5-72B-Instruct	72B	2024-09-19	✓	Qwen2.5-72B-Instruct
meta-llama/Llama-3.3-70B-Instruct	70B	2024-12-06	✓	Llama-3.3-70B-Instruct
deepseek-ai/DeepSeek-V3	-	2024-12-26	✓	DeepSeek-V3
gpt-4o (OpenAI)	-	2024-05-13	×	gpt-4o/
gpt-4o-mini (OpenAI)	-	2024-07-18	×	gpt-4o-mini
claude-3.7-sonnet (Anthropic)	-	2025-02-25	×	claude-3.7-sonnet
grok-3-beta (xAI)	-	2025-02-19	×	grok-3-beta
o1 (OpenAI)	-	2024-12-05	×	o1
o3 (OpenAI)	-	2025-04-16	×	o3
deepseek-ai/DeepSeek-R1-Distill-Qwen-7B	7B	2025-01-20	✓	DeepSeek-R1-Distill-Qwen-7B
deepseek-ai/DeepSeek-R1-Distill-Qwen-32B	32B	2025-01-20	✓	DeepSeek-R1-Distill-Qwen-32B
Qwen/QwQ-32B-Preview	32B	2025-03-06	✓	QwQ-32B-Preview
deepseek-ai/DeepSeek-R1-Distill-Llama-70B	70B	2025-01-20	✓	DeepSeek-R1-Distill-Llama-70B
deepseek-ai/DeepSeek-R1	-	2025-01-20	✓	DeepSeek-R1

ation and enhancing generality.

Repository Coverage. SWE-Dev relies on repository histories and developer-authored tests from selected open-source projects. As a result, it can inherit biases from these projects and may underrepresent proprietary, UI-heavy, infrastructure-heavy, or weakly tested development workflows.

Training Exploration. Our training experiments focus on standard techniques—SFT, RL, and role-wise MAS training—which yield modest gains. Future work could explore stronger RL (Shao et al., 2024; Yu et al., 2025; Hu et al., 2025a), dynamic agent coordination (Ye et al., 2025b), and curriculum learning (Bengio et al., 2009). Notably, SWE-Dev offers fine-grained complexity signals via call trees that can guide complexity-aware training.

E.7 Broader impacts

SWE-Dev is the first dataset tailored for autonomous feature-driven software development, addressing the gap between current automated coding and real-world software engineering demands. By providing large-scale, realistic tasks based on real repositories with executable tests, it enables rigorous and reliable evaluation of automated AI coding systems. SWE-Dev promotes the creation of more capable methods for complex software, driving innovation that can lower development costs and enhance software quality industry-wide.

F Detailed Benchmark Construction

F.1 Call tree generation

To accurately localize the implementation logic associated with each test case, we construct a call tree that captures the dynamic execution path from the test to the relevant source functions. This tree serves as the foundation for identifying the core feature logic and determining task complexity.

Figure 15 shows a generated call tree for the file `test_az.py`, which contains multiple test functions such as `test_cardinal`, `test_year`, and `test_ordinal_num`. Each test function serves as a root for its own call path, triggering downstream functions like `Num2Word_AZ.to_c ardinal` and `Num2Word_AZ.int_to_word`. This tree structure reveals the multi-level and cross-functional logic activated during test execution, illustrating how test files connect to multiple feature implementations across the codebase.

We use the call tree in two key ways:

1. To select target functions for masking during task generation, enabling controllable task complexity.

2. To trace which source files and logic a model

Table 10: Comparison of general and code-specific MAS on SWE-Dev driven by GPT-4o-mini. **Bold** highlights the best performance; underlined indicates results worse than the single-agent baseline. Most MAS methods outperform the single agent, and simpler general MASs are more effective and efficient than complex coding-specific MASs.

Method	Easy			Hard		
	Pass@1	Calls	Price($)	Pass@1	Calls	Price($)
Single	34.47	1.00	0.75	11.09	1.00	0.97
Reflexion	39.77	2.12	0.83	13.32	2.18	1.35
Self Refine	**40.02**	5.00	5.78	**20.03**	5.00	5.8
Self Consistency	37.62	6.00	4.30	18.55	6.00	7.08
LLM Debate	38.48	7.00	5.95	14.56	7.00	9.35
MAD	31.50	7.00	2.48	15.31	7.00	3.40
Agentverse	**38.67**	4.52	1.40	13.42	4.83	2.90
EvoMAC	34.59	7.98	3.20	**13.60**	8.30	4.65
MetaGPT	29.56	9.69	2.20	9.25	10.37	4.95
MapCoder	24.55	21.01	6.05	5.87	23.41	10.55
ChatDev	35.13	26.61	3.53	11.70	30.87	6.10

Table 11: Performance of **tool-augmented LLM agents** on SWE-Dev using the OpenHands framework. We report Pass@1 (%) on the *Easy* and *Hard* splits. **Abbreviated names** match those used in the main figures.

Abbreviated Name	Model Name	Easy	Hard
Claude4.1-Opus(T)	Claude-4.1-Opus-Thinking	86.30	56.44
GPT-5	GPT-5	87.56	54.21
Claude4.5(T)	Claude 4.5 Thinking	84.83	53.55
KimiK2(T)	Kimi-K2-Thinking	83.89	50.86
KimiK2	Kimi-K2-Instruct	81.82	51.05
GLM4.5	GLM-4.5	71.83	39.00
Qwen3-235B(T)	qwen3-235b-a22b-thinking-2507	43.37	19.78
Qwen3-30B(T)	qwen3-30b-a3b-thinking-2507	29.68	6.11

must understand to solve the task, supporting fine-grained evaluation and curriculum learning.

F.2 Docstring Augmentation Prompt

To ensure high-quality task specifications, we augment original function-level docstrings using GPT-4o. Figure 25 shows the prompt we use to generate concise, informative docstrings conditioned on the full code context.

F.3 Masking Procedure and Reproducibility

Our dataset construction does not rely on any task-complexity thresholds such as call-tree depth or node count. Instead, we apply a fixed set of masking rules uniformly across all repositories. Specifically, for each task we derive the dynamic call chain exercised by the upstream test and construct a masked input using one of three modes: (i) masking depth-1 target functions, (ii) masking depth-2 dependent functions, or (iii) masking the entire call chain for call-tree depth ≥ 2. These masking modes are predetermined and do not involve any further filtering or parameter tuning.

G Extended Related Work

G.1 Multi-agent system

For complex SE tasks that strain the context handling of single agents, Multi-Agent Systems (MAS) utilizing collaborating LLMs are an emerging research avenue. Existing frameworks like MetaGPT, ChatDev, and AgentVerse often rely on predefined agent roles and fixed interaction protocols. While effective on specific tasks, their hand-crafted structure limits generalization. Recent research explores trainable MAS, aiming for agents that dynamically adapt their organization or communication strategies. However, empirical studies of such adaptive MAS are largely constrained by benchmark complexity; evaluations are often confined to small-scale or synthetic tasks due to the lack of benchmarks providing complex interaction scenarios and reliable execution feedback. SWE-Dev's scale, complexity, and provision of executable feedback (via unit tests) establish it as the first testbed capable of supporting the training and evaluation of dynamic MAS on realistic, multi-file feature development scenarios.

H Experiment Settings

H.1 Inference

LLMs. We evaluate 17 chatbot LLMs with different model size, including Qwen2.5-Instruct models

MAS System	Pass@1 (%)	Avg. LLM Calls	# Roles	Avg. Agent-to-Agent Msgs
Reflexion	39.77	2.12	3	10.77
Self-Refine	**40.02**	5.00	2	26.79
Self-Consistency	37.62	6.00	2	33.01
LLM Debate	38.48	7.00	2	76.35
MAD	31.50	7.00	4	2.36
AgentVerse	38.67	4.52	4	8.83
MetaGPT	29.56	9.69	6	24.26
MapCoder	24.55	21.01	5	63.03

Table 12: **Communication and coordination statistics of MAS on SWE-Dev.** All systems are driven by the base model (GPT-4o-mini). We report Pass@1, average number of LLM calls per task, number of predefined agent roles, and the average number of agent-to-agent messages.

1.5B/3B/7B/14B/32B/72B (Qwen et al., 2025b), Qwen3 models 8B/30B-A3B (Qwen, 2025), Llama 3.1-8B/3.3-70B-Instruct (Grattafiori et al., 2024), Phi 4 (Abdin et al., 2024), Claude-3.7-Sonnet (Anthropic, 2025), Deepseek-V3 (DeepSeek-AI et al., 2025), GPT-4o (Hurst et al., 2024), Deepseek-Coder-V2-Lite-Instruct (DeepSeek-AI et al., 2024), Qwen2.5-Coder-14B-Instruct (Hui et al., 2024). Additionally, We extend the evaluation to reasoning models, including Deepseek-R1-distill models (Qwen2.5 7B/32B, Llama-70B) (Guo et al., 2025), Qwen3 8B/30B-A3B (thinking) (Qwen, 2025), QwQ-32B-Preview, Deepseek-R1 (Guo et al., 2025), OpenAI-o1 (OpenAI et al., 2024), Claude-3.7-Sonnet-thinking (Anthropic, 2025), and Grok-3-Beta (xAI, 2025).

Multi-Agent Systems. To provide a more comprehensive evaluation of SWE-Dev, we expand our study to include multi-agent systems (MAS) built on LLMs. Prior research has demonstrated that MAS can enhance performance on tasks requiring multi-step reasoning and coordination (Ye et al., 2025b; Hong et al., 2024; Qian et al., 2024). In our experiments, all MAS are implemented using GPT-4o-mini (OpenAI, 2024) as the underlying model to ensure consistency across methods. And for fair comparison, we utilize MASLab (Ye et al., 2025a), a unified framework integrating multiple MAS implementations. We evaluate coordination-based MAS such as LLM Debate (Du et al., 2023), Self Refine (Madaan et al., 2023), Multi-Agent Debate (MAD) (Liang et al., 2024), and Self Consistency (Wang et al., 2023a) that feature relatively simple agent interaction strategies. We further include structured, workflow-oriented MAS designed for code generation, including Agentverse (Chen et al., 2023), MetaGPT (Hong et al., 2024), ChatDev (Qian et al., 2024), MapCoder (Islam et al., 2024), and EvoMAC (Hu et al., 2025b).

H.2 Training

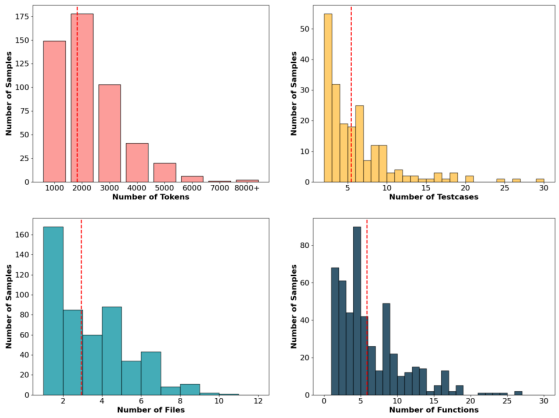
H.3 Single-Agent Supervised Fine-tuning

We fine-tune the model using LoRA, applying low-rank adaptations (rank $r = 16$, scaling $\alpha = 16$, $dropout = 0.05$) to the query, key, value, and output projection matrices of each attention sub-layer. Training is performed with a learning rate of 6×10^{-4} and a batch size of 32 sequences per gradient step, for up to 4 epochs. Checkpoints are saved every 50 steps, and the best model is selected based on validation loss over a held-out set of 100 examples. Fine-tuning is initialized from Qwen2.5-7B-Instruct and completed within 20 hours using 8 NVIDIA A100 GPUs. We leverage DeepSpeed Ulysses and Flash Attention to support efficient training with long input contexts.

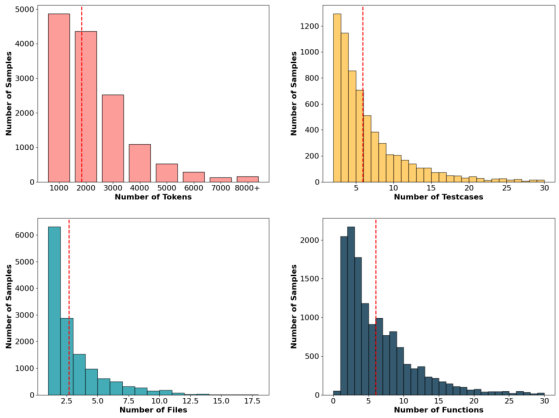
H.4 Single-Agent Reinforcement Learning

For reinforcement learning (RL) training, we sampled 2k instances from SWE-Dev to balance computational feasibility and the ability to capture RL benefits. Specifically, we used Proximal Policy Optimization (PPO) (Schulman et al., 2017) and Direct Preference Optimization (DPO) (Rafailov et al., 2023) for training the Qwen2.5-7B-Instruct using 8 NVIDIA A100 GPUs. All RL experiments are run on a single node with 8×A100 GPUs, using VERL PPO with a vLLM-based rollout backend.

PPO: The training was conducted using a batch size of 256 and trained for 5 epochs, with a learning rate of 1×10^{-6} for the actor and 1×10^{-5} for the critic. We set the KL coefficient to 0.001. The training was set to save checkpoints every 10 steps. We used a maximum prompt length of 8192 tokens and set a micro-batch size of 32. The maximum response length is 2048 tokens. The reward for PPO is calculated based on the pass rate of the test cases. Concretely, for each rollout solution, the reward is defined as the pass rate, i.e., the number of test



(a) Test Set Distribution



(b) Train Set Distribution

Figure 7: Dataset distribution of PRD tokens, number of testcases, number of files to complete, and number of functions per sample.

cases successfully passed by the solution divided by the total number of test cases available for that task. During RL training, we do not employ any external agent framework: the language model directly receives the problem requirement description (PRD) together with the relevant code context as input and generates a single rollout solution.

DPO: For DPO training, we applied LoRA with a rank of 64, scaling factor $\alpha = 128$, and dropout set to 0. The preference loss function (`pref_loss`) was set to sigmoid, which is commonly used in DPO for preference-based optimization. Training was performed for 5 epochs, using a batch size of 8 and a learning rate of 1×10^{-5}.

For a fair comparison with SFT in §4.2.2, we used the same 2k training samples for both SFT and RL. The details for SFT training are outlined in Appendix H.3.

These methods allow us to assess the impact of RL on model performance using the SWE-Dev

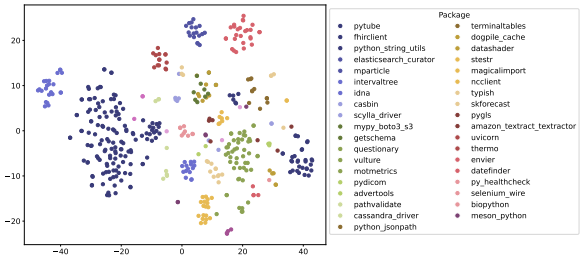


Figure 8: t-SNE visualization of PRD in test set

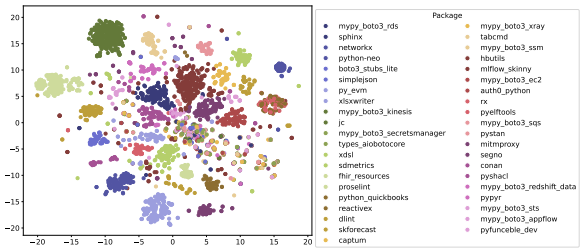


Figure 9: t-SNE visualization of PRD in train set

dataset while maintaining efficient training..

H.5 Multi-Agent Supervised Fine-tuning

In our multi-agent fine-tuning experiments, we utilize a simplified version of EvoMAC (Hu et al., 2025b), retaining only two core roles: **Organizer** and **Coder**, see Figure 20. The fine-tuning process follows an iterative workflow. Initially, the **Organizer** processes the Project Requirement Description (PRD) and breaks it down into clearly defined subtasks or instructions. Subsequently, the **Coder** generates corresponding code implementations for these subtasks. The generated code is then evaluated using the provided ground truth (GT) test cases. Feedback from these evaluations informs subsequent iterations, enabling iterative refinement of both the task decomposition by the Organizer and the code generation by the Coder. Fig. 22 and Fig. 23 respectively show the prompts used for the Coder and Organizer.

Rejection Sampling Procedure.

To effectively leverage the feedback from GT test cases, we employ rejection sampling—a method widely adopted in reinforcement learning and language model fine-tuning (Zhou et al., 2024; Snell et al., 2023). The detailed procedure is as follows:

1. **Iterative Reasoning with EvoMAC**: For each training instance, EvoMAC executes multiple rounds of reasoning. In each iteration, the generated code from the Coder is

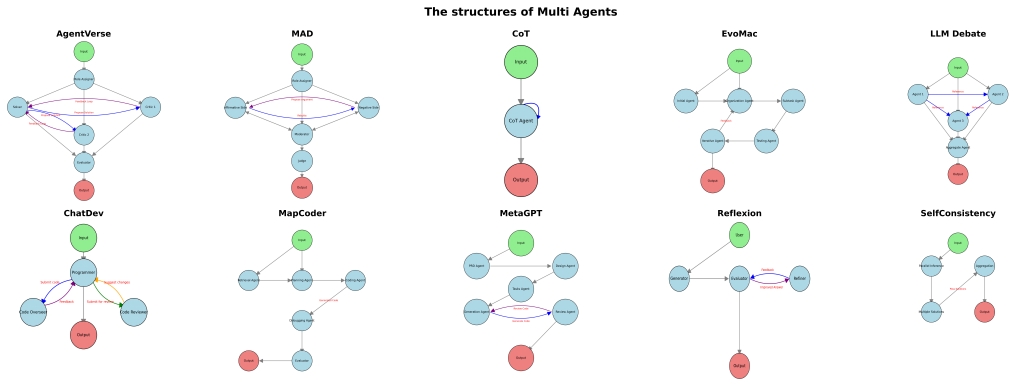


Figure 10: Multiagent system workflow visualization.

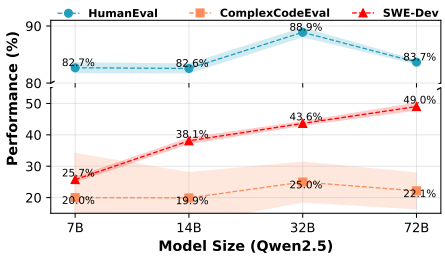


Figure 11: Comparison of benchmarks on various model sizes. SWE-Dev shows clear performance scaling with model size, while HumanEval (Chen et al., 2021) fails to distinguish between models. ComplexCodeEval (Feng et al., 2024) using CodeBLEU (Ren et al., 2020) exhibits high variance, making it less stable for evaluation.

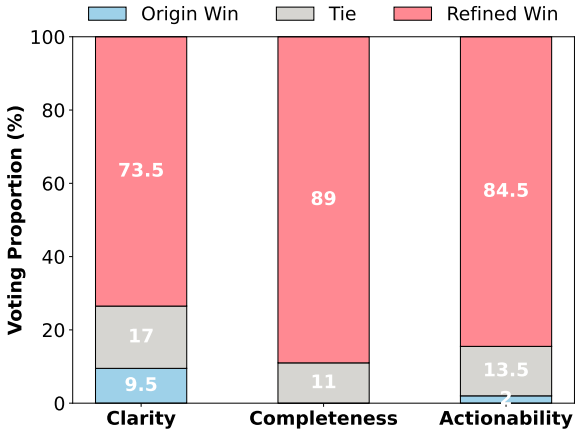


Figure 12: Human ratings of original vs. refined PRDs on 100 sampled tasks across Clarity, Completeness, and Actionability.

tested against the GT test cases to compute its performance.

2. **Selection of High-quality Trajectories**: Trajectories that show improved performance over previous iterations (as indicated by an increased pass rate on GT test cases) are selectively retained. Conversely, trajectories that do not demonstrate progress or degrade in performance are discarded. This ensures that only beneficial and constructive data is used for fine-tuning.

3. **Role-wise Fine-tuning**: The retained high-quality trajectories are utilized to separately fine-tune the Organizer and the Coder. Specifically, the Organizer is trained to better structure and decompose tasks from PRDs, while the Coder is refined to enhance code generation capabilities for defined subtasks. This role-specific fine-tuning promotes specialization and improves overall performance.

As shown in §4.2.3, through this simplified EvoMAC and structured rejection sampling approach, our multi-agent fine-tuning effectively enhances the capabilities of each agent, contributing to significant performance gains on SWE-Dev.

I Licensing

All codebases and data used in this work are sourced from publicly available GitHub repositories. We have ensured compliance with the corresponding licenses of these repositories, respecting all terms of use and attribution requirements.

J Prompts

This section includes all prompts used in the generation, evaluation and analysis process.

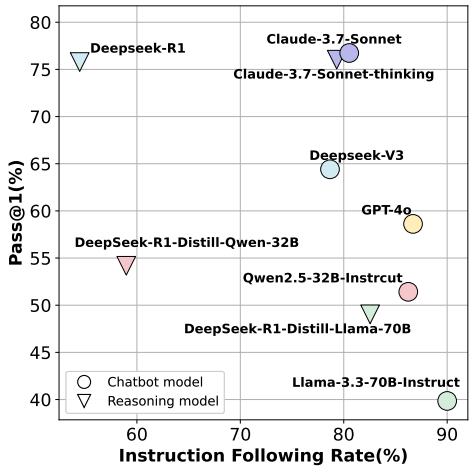


Figure 13: Comparison of reasoning and chatbot LLMs' IFR and performance on SWE-Dev. Reasoning models tend to outperform chatbots when they fully follow instructions, though their overall IFR is lower.

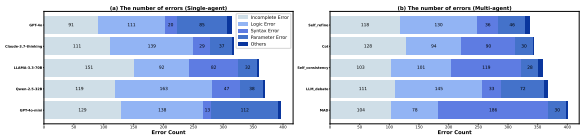


Figure 14: Failure case distribution of Single and Multi-agent.

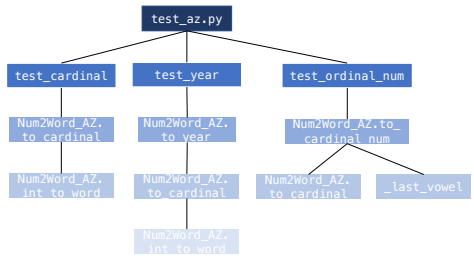


Figure 15: Example of a generated call tree for `test_az.py`.

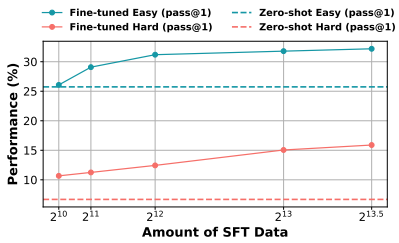


Figure 16: Training data scaling of SFT Qwen2.5-7B-Instruct on SWE-Dev. As data size increases, performance improves steadily under SFT.

Figure 17: The PPO training progress of Qwen2.5-7B-Instrut on the SWE-Dev with 40 steps and 2048 max token length. The light curve shows the raw reward, while the dark curve applies EMA smoothing.

Figure 18: The PPO training progress of Qwen2.5-7B-Instrut on the SWE-Dev with 160 steps and 8192 max token length. The light curve shows the raw reward, while the dark curve applies EMA smoothing.

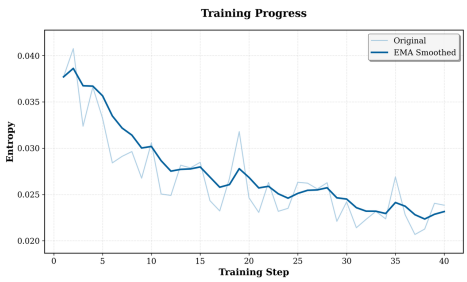


Figure 19: Entropy trajectory during PPO training (40 updates). The light curve shows the raw policy entropy, and the dark curve shows the EMA-smoothed trend. Entropy decreases steadily over training, indicating reduced exploration and a progressively sharper output distribution—consistent with the observed contraction in diversity under RL.

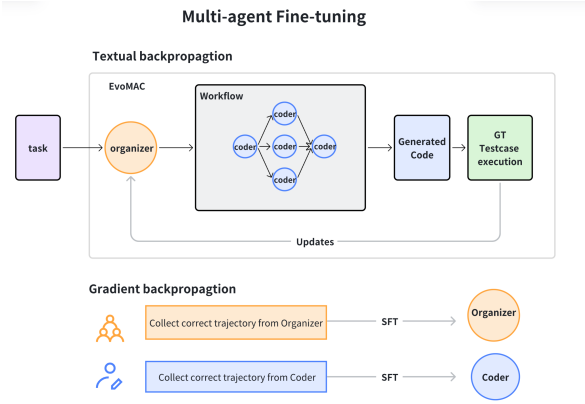


Figure 20: Overview of Multi-agent Fine-tuning in EvoMAC. This framework leverages both textual and gradient-based supervision to improve multi-agent collaboration. During inference, an organizer assigns roles and coordinates a team of coders to generate code, which is then validated using ground truth test cases. Successful execution trajectories are collected and used to fine-tune both the organizer and coders individually via supervised learning, enabling role-specific optimization for complex software development tasks.

Single LLM inference prompt

```
# AIM: You need to assist me with a Python package feature development task. I will provide a
     Product Requirements Document (PRD) that details the functionality and lists the empty
     functions that need to be implemented across different file paths. I will also provide
     the complete "Code Context" of all files mentioned in the PRD.

Your task is to implement ONLY the empty functions described in the PRD while preserving ALL
     OTHER CODE in the provided files exactly as is. This is absolutely critical - you must
     keep all imports, class definitions, functions, comments, and other code that is not
     explicitly mentioned in the PRD for implementation.

When implementing the functions:
1. Carefully identify which functions from the PRD need implementation.Implement them based on
      the docstrings and specifications in the PRD
2. Do not add any new "import" statements unless absolutely necessary
3. Do not modify ANY existing code structure, only implement the empty functions

For each file mentioned in the PRD, you MUST output the COMPLETE file code with your
     implementations inserted. Your output format must follow this exact pattern:

# OUTPUT FORMAT FOR EACH FILE:
@ [relative path/filename]
```python
[COMPLETE file code including ALL original code plus your implementations]
```
@ [relative path/filename]
```python
[COMPLETE file code including ALL original code plus your implementations]
```

IMPORTANT: Make sure your output includes EVERY function, class, import statement, and comment
      from the original code context. The only difference should be that the empty functions
     specified in the PRD are now implemented.

PRD:
{PRD}

Code Context:
{code_snippet}
```

Figure 21: Single LLM Inference Prompt.

EvoMAC Coding Agent Prompt

```
You are Programmer. we are both working at ChatDev. We share a common interest in
    collaborating to successfully complete a task assigned by a new customer.

You can write/create computer software or applications by providing a specific programming
    language to the computer. You have extensive computing and coding experience in many
    varieties of programming languages and platforms, such as Python, Java, C, C++, HTML, CSS,
     JavaScript, XML, SQL, PHP, etc,.

Here is a new customer's task: {task}.

To complete the task, you must write a response that appropriately solves the requested
    instruction based on your expertise and customer's needs.

According to the new user's task and you should concentrate on accomplishing the following
    subtask and pay no heed to any other requirements within the task.

Subtask:
{subtask}.

Programming Language: python,

Codes:
{codes}
```

Figure 22: Role Prompt for EvoMAC Coding Agent.

EvoMAC Organizing Agent Prompt

```
As the Leader of a coding team, you should analyze and break down the problem into several
     specific subtasks and assign a clear goal to each subtask. Ensure each subtask is
     extremely detailed, with a clear description and goal, including the corresponding PRD
     statement where necessary.

The workflow should be divided into minimal, executable subtasks, each involving one method
     implementation. The target_code should only contain the relative paths and function names
      for the specific code that is required for that subtask.

Each subtask should be assigned a unique task_id, and the description should reflect the exact
      requirements of the PRD corresponding to that method or task. The target_code should be
     precise, containing only the specific Python code (relative path and method/function name
     ) that corresponds to the subtask's scope.

The format should strictly follow the JSON structure below:

```json
[
    {
        "task_id":"1",
        "description":"Task Description",
        "target_code": [
            "relative_python_path:function_name",
            "relative_python_path:class_name.method_name"
        ]
    }
]
```

Use the backticks for your workflow only.

Note:

(1) Each subtask should be self-contained and represent one method's implementation.

(2) The `description` should be based on specific statements from the PRD, and it must explain
      what the subtask is aiming to achieve.

(3) The `target_code` should only reference the code paths and function names for the methods
     to be implemented for the subtask.

(4) The number of subtasks should not exceed 5. Some tasks might combine multiple smaller
     functions if needed to fit within the limit.

(5) Each subtask is handled independently by different agents, so the description should be
     thorough, ensuring clarity even without the full context of the PRD.
```

Figure 23: Role Prompt for EvoMAC Organizing Agent.

Error Classification Prompt

```
You are an error classification expert. Based on the provided PRD, LLM-generated code, and
    error message, your task is to analyze and categorize the primary issue.

1. Analyze the root cause of the problem using the PRD, the code, and the error message.
2. If multiple issues exist, return only the most severe and primary one (return exactly one
    ProblemType).
3. Return the result in strict JSON format with the following structure:
{
    "ProblemType": {
        "MainCategory": "Main error category",
        "SubCategory": "Specific sub-category of the issue",
        "Reasoning": {
            "SymptomAnalysis": "Observed abnormal behavior (in Chinese)",
            "RootCause": "Attribution analysis combining PRD and code (in Chinese)",
            "ErrorMechanism": "Technical explanation of how the error occurs (in Chinese)"
        }
    }
}

Below is the data provided to you:
PRD:
{prd}

Generated Code:
{results}

Error Message:
{input_text}

Please ensure your response strictly follows the JSON format above.

The allowed values for MainCategory are limited to the following five options - read them
    carefully and choose the most appropriate one:
1. Logic Error: Logical errors such as assertion failures or failure to meet PRD requirements.
2. Syntax Error: Syntax issues such as unexpected tokens, indentation problems, etc.
3. Parameter Error: The function required by the PRD is present, but input/output parameters
    are incorrect or missing.
4. Incomplete Error: Some required functions are entirely missing as per the PRD. Make sure to
     distinguish between a truly missing function and one that exists but contains logic or
    syntax errors.
5. Others: Any other issues that do not fit the above categories.

You must carefully select the MainCategory to ensure accuracy.
Do not return any MainCategory that is not listed above.
Do not return an empty MainCategory.
```

Figure 24: Prompt Template for Error Type Classification.

Docstring Augmentation Prompt

```
# Context: The following Python code is provided for reference. It includes functions, classes
     , and other elements that provide context for the function or class below. Additionally,
     any constants or variables defined outside functions/classes are considered as part of
     the context and should be explained if used.

# Full Code:
```python
{full_code}
```

Code for {name}:
```python
{code_snippet}
```

Docstring:
Please generate a concise and clear docstring for the above {name} based on the full code
     context. Ensure the docstring briefly explains the {name}'s purpose, parameters, return
     values, and any relevant dependencies or interactions with other parts of the code. If
     there are any constants or variables used within the {name}, explain their role and
     significance, including where they are defined and how they interact with the function or
      class.

For functions: describe the input parameters, expected output, and any important side effects
     in a few sentences. Also, explain any constants used inside the function (if applicable).

For class.methods: describe the input parameters, expected output, and any important side
     effects in a few sentences. Also, explain any constants used inside the function (if
     applicable).

For classes: describe the main attributes and methods, along with the general purpose of the
     class in a brief summary. Mention any constants used in the class and explain their
     purpose and how they interact with class methods and attributes. Keep the docstring
     focused, avoiding unnecessary details or repetition.

Output format
Your response should strictly follow the format below, without any other text or comments.
\"\"\"
docstring
\"\"\"
```

Figure 25: Docstring Augmentation Prompt in Task Generation.

Categories for Classifying Packages

```
You are a Python expert. Given the name of a PyPI package, classify it into ONE category from
    the list below based on its MOST central and primary purpose.

Categories:

1. Web & Network Automation
Packages that support automation of web browsing, API communication, and network protocols.
Criteria: Enables browser control, HTTP requests, network operations, or web server handling.

2. Data Processing & Integration
Packages that extract, parse, or convert structured/unstructured data formats.
Criteria: Handles parsing or converting text, JSON, YAML, XML, or dates.

3. Security & Access Control
Packages that focus on authentication, authorization, or access control mechanisms.
Criteria: Implements rules, policies, or authentication methods.

4. Command-Line & Developer Tools
Packages that assist in building CLI tools, test frameworks, or code quality analysis.
Criteria: Aimed at improving the development experience, command-line interfaces, or code
    quality.

5. Cloud & Data Storage
Packages interacting with cloud services, databases, or data storage solutions.
Criteria: Provides interfaces or tools to access, manage, or validate remote data or cloud
    resources.

6. Data Science & Visualization
Packages used for scientific computing, visualization, or statistical evaluation.
Criteria: Supports data analysis, visualization, or scientific research.

7. Others
Packages that do not clearly belong in the other categories or are too general/specialized.
Criteria: Doesn't strongly align with the definitions above or serves a unique/niche purpose.

Please output only the category number (only one category), no explanation unless asked.
    Choose the single best fit.

Package name: {package_name}

You must strictly follow the format below, only a number no other text:
1
```

Figure 26: Prompt Template for Classifying Packages.

Human Evaluation Guideline for PRD Quality

```
Each docstring is evaluated independently along the following three dimensions, using a 0-5
    scale (0 = very poor, 5 = excellent):

Clarity - How easy the docstring is to understand for a competent software engineer. Consider
    language clarity, readability, and absence of ambiguity.
Completeness - Whether the docstring provides all necessary information to understand the
    function's behavior. Consider whether inputs, outputs, parameters, and important logic
    are described.
Actionability - How effectively the docstring guides actual implementation. Consider whether a
     developer could use the docstring alone to reasonably implement the function.

Rating Scale:
5: Excellent - No issues; highly clear, complete, and actionable.
4: Good - Minor improvements possible.
3: Fair - Understandable but lacking in one area.
2: Poor - Vague or missing key information.
1: Very Poor - Hard to follow or largely unhelpful.
0: Unusable - Cannot inform implementation at all.

If the original docstring is missing or boilerplate-only, please rate accordingly. Docstrings
    are to be rated individually without direct comparison.
```

Figure 27: Human Evaluation Guideline for PRD Quality.