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

Yaxin Du[1]*, **Yuzhu Cai**[2]*, **Yifan Zhou**[3], **Cheng Wang**[1], **Yu Qian**[1], **Xianghe Pang**[1], **Qian Liu**, **Yue Hu**[4], **Siheng Chen**[1]
[1]Shanghai Jiao Tong University, [2]Beijing University of Aeronautics and Astronautics,
[3]Soochow University, [4]University of Michigan

Abstract

Large Language Models (LLMs) have shown strong capability in diverse software engineering tasks, e.g. code completion, bug fixing, and document generation. However, feature-driven development (FDD), 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 14,000 training and 500 test samples) designed to evaluate and train autonomous coding systems on real-world feature 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 (SFT), but also enables Reinforcement Learning (RL) by delivering accurate reward signals from executable unit tests. Our extensive evaluations on SWE-Dev, covering 17 chatbot LLMs, 10 reasoning models, and 10 Multi-Agent Systems (MAS), reveal that FDD is a profoundly challenging frontier for current AI (e.g., Claude-3.7-Sonnet achieves only 22.45% Pass@3 on the hard test split). Crucially, we demonstrate that SWE-Dev serves as an effective platform for model improvement: fine-tuning on training set enabled a 7B model comparable to GPT-4o on *hard* split, underscoring the value of its high-quality training data. Code is available here https://github.com/DorothyDUUU/SWE-Dev.

1 Introduction

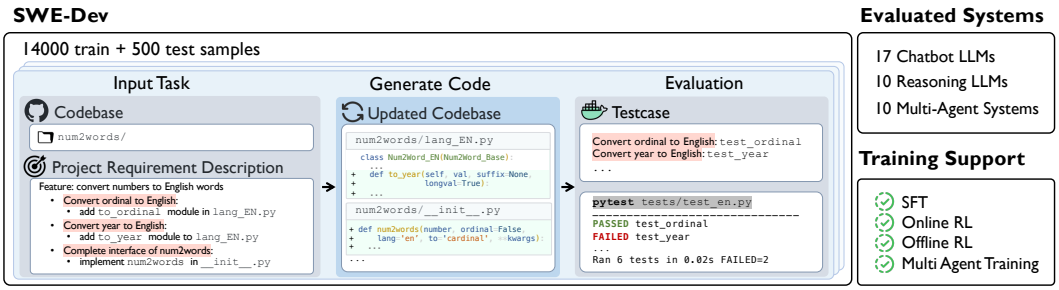


Figure 1: Overview of SWE-Dev, a software development dataset providing feature development tasks with feature description and codebase as input and test cases for evaluation. It is uniquely grounded in real-world repositories and paired with executable test suites, enabling reliable, functionally verifiable supervision. SWE-Dev is evaluated on 37 autonomous coding systems and supports advanced training paradigms like SFT, RL, and multi-agent training.

*Equal contribution. Correspondence to: Siheng Chen <sihengc@sjtu.edu.cn>

Preprint. Under review.

Large Language Models (LLMs) are rapidly transforming autonomous programming, with capabilities extending from generating isolated code snippets to complex interactions within entire repositories [1, 2]. 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 [3] measures only localized bug fixes described by GitHub issues, and RepoBench [4] evaluates the completion of a few unrelated functions within a repository. However, they overlook the core tasks of developing and integrating significant new functionalities, which truly define how real-world codebases evolve.

The task of developing and integrating new functionalities is formally defined as feature-driven development (FDD) [5, 6], which consists of 40% coding tasks of all development efforts [7, 8]. FDD involves the end-to-end creation of new features, from interpreting requirements in large, existing codebases to generating functionally correct and integrated code (see Figure 1). FDD is how most modern software, from large applications to essential libraries, primarily evolves and delivers continuous value [9, 10]. Consequently, mastering FDD is a critical way towards achieving more comprehensive and genuinely autonomous programming capabilities with coding systems.

Recognizing the central role of FDD and the limitations of current evaluation benchmarks, we introduce a feature-driven **S**oft**W**ar**E** **Dev**elopment dataset, **SWE-Dev**, which is the first large-scale dataset designed to evaluate and train autonomous AI systems on real-world FDD 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: (1) **Realistic scale and complexity**: SWE-Dev requires substantial code modifications (avg. 190 LOC across 3 files), challenging models with the cross-file dependencies, large contexts, and significant implementation scope characteristic of real-world feature development. (2) **Robust and grounded evaluation**: Each SWE-Dev sample is grounded in a real open-source repository, guided by a well-defined project requirement description (PRD), and evaluated using executable test cases to ensure the functional correctness of the proposed implementation. This design ensures alignment between task objectives and evaluation, enabling robust assessment and model supervision. (3) **Verifiable training set with executable test suites**: Uniquely, all 14,000 training instances are paired with runnable environments and executable unit tests, providing crucial execution-based feedback that enables effective Supervised Fine-Tuning (SFT) validation, Reinforcement Learning (RL) with accurate rewards, and Multi-Agent System (MAS) training, refer to Table 1.

Our extensive experiments using SWE-Dev reveal several critical insights. Firstly, Repository-level feature development is challenging: our findings show even top-tier models like Claude-3.7-Sonnet[11] and GPT-4o[12] solve only 22.45% *hard* samples and 68.70% *easy* samples with Pass@3. Secondly, MASs generally outperform single-agent baselines in modest margins. Interestingly, simple general-purpose multi-agent methods (e.g., Self-Refine[13], Reflexion[14]) often outperform more complex code-specific agents, while requiring fewer model calls and lower cost. Lastly, task-specific training on this task gets substantial gains on all training methods. After training, a 7B fine-tuned model is comparable to GPT-4o on *hard* subset after task-specific training.

These findings point to several promising directions for future research. First, the difficulty of FDD for LLMs necessitates enhancing LLMs' core reasoning and long-context capabilities for software development. Second, current MAS designs often suffer from unnecessary communication overhead and limited coordination efficiency. Future work should explore lightweight agent architectures and better context-sharing mechanisms for repository-level development. Lastly, our initial experiments with RL and role-based multi-agent training show that training can be beneficial, but headroom remains. Future work could investigate multi-agent training and long-context RL with SWE-Dev.

Our contributions are as follows:

- We introduce **SWE-Dev**, the first real-world dataset for autonomous feature-driven software development. The dataset includes both training and test splits, each with runnable environments and test cases, enabling a wide range of evaluation and training.
- Our evaluations on SWE-Dev offer novel insights into the proficiency and deficiencies of **various coding systems (chatbot LLMs, reasoning LLMs, and MAS)** on complex FDD tasks.
- We demonstrate SWE-Dev **enabling and validating diverse training paradigms** (SFT, RL, and MAS training), establishing its utility for advancing training-based adaptation.

Table 1: Comparison of SWE-Dev with existing repository-level benchmarks. Task (FC: Function Completion, PG: Project Generation, LC: Line Completion, IS: Issue Solving), usage of real-repository, availability of training sets, Number of Samples, and task statistics are compared here. Detailed statistics information is demonstrated e.g., line of code (LOC), task description PRD length.

	Task	Real Repo	Train Existence	w. Testcases Train	w. Testcases Test	# Samples Total	# Samples w. Testcases	Avg. PRD Tokens	Avg. LOC
ComplexCodeEval[15]	FC	✔	✗	✗	✗	7k	0	134.2	38.21
CoderEval[16]	FC	✔	✗	✗	✔	234	234	119.26	20.64
DevEval[17]	FC	✔	✗	✗	✔	2k	2k	91.5	112
rSDE-Bench[18]	PG	✗	✗	✗	✔	53	53	1553	157.88
M2rc-Eval[19]	LC	✔	✔	✗	✗	59k	0	0	1
RepoBench[4]	FC	✔	✗	✗	✔	23k	0	0	89
SWE-Bench[3]	IS	✔	✔	✗	✔	19k	2k	195.1	32.8
SWE-Dev	**FDD**	✔	✔	✔	✔	14.5k	**14.5k**	**1845.4**	**190**

2 Related Work

2.1 Coding benchmarks

LLMs show significant potential in coding tasks, driving the need for robust benchmarks. Early benchmarks such as HumanEval[20], MBPP[21], APPS[22], and CodeContests[23] primarily focus on isolated, function-level tasks. These benchmarks test for correctness in constrained settings: short snippets, well-specified inputs, and short expected outputs. While useful for early-stage capability testing, such tasks fall short of reflecting the complex, multi-file dependency and long contexts nature of real-world software development tasks. To address this, repository-level benchmarks emerged, such as SWE-Bench[3] (issue fixing), RepoBench[4], and M2RC-Eval[19] (code completion/understanding). Despite this progress, they often face two main issues: (1) The scope of required code generation or modification remains limited (e.g., avg. 32.8 LOC in SWE-Bench, 38.21 LOC in ComplexCodeEval[17]), inadequately simulating large-scale feature development or refactoring. (2) Weak or inconsistent evaluation protocols: several benchmarks [15, 19, 4] rely heavily on proxy metrics such as code similarity or static heuristics, which often fail to reflect functional correctness. This compromises both the robustness of evaluation and the comparability of results across models[24, 25]. SWE-Dev directly tackles these limitations by providing large-scale repository-level feature development tasks with executable unit tests. Its tasks involve substantial modifications, addressing shortcomings in both code scope and trainable environments, thereby significantly increasing task complexity and realism.

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[24] and Phi[26] leverage massive code corpora to learn syntax and general programming patterns. To improve instruction following and task completion, many works adopt SFT. Code Alpaca[27] employs self-instruct generation, WizardCoder[28] leverages Evol-Instruct[29] 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[30]. Beyond SFT, RL frameworks such as CodeRL[31] utilize test-case-based feedback to optimize model behavior. While promising, both SFT and RL approaches largely focused on function-level tasks, limiting their applicability to more complex development scenarios. To address this, SWE-Gym[32] explores extending training to repository-scale tasks using multi-agent systems. However, due to the lack of an executable training set in SWE-Bench, SWE-Gym resorts to constructing a separate dataset of 2,438 tasks, ultimately yielding only 500 trajectory samples for training. In contrast, our proposed SWE-Dev provides a large-scale repository-level training set with runnable environments and unit-test-based supervision. It supports SFT, RL, and multi-agent training with executable feedback, enabling realistic and scalable development of code generation systems.

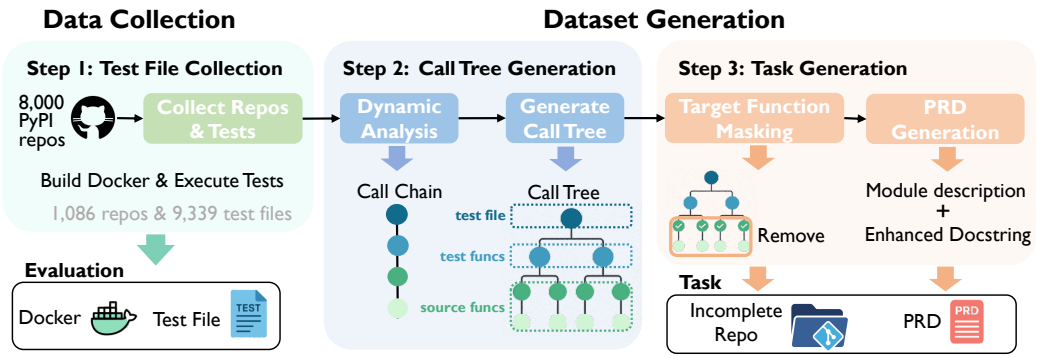


Figure 2: Overview of SWE-Dev dataset construction. **Step 1**: We collect real-world repositories with passing test files in Dockerized environments, **Step 2:** trace test executions to construct function-level call trees linking test cases to invoked source code, and **Step 3**: mask core functions 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.

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), and key statistics (§ 3.3).

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 reflect 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 1,086 validated repositories (as of December 12, 2024) and 9,339 executable test files.

Step 2: Call Tree Generation To locate the specific functions and 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

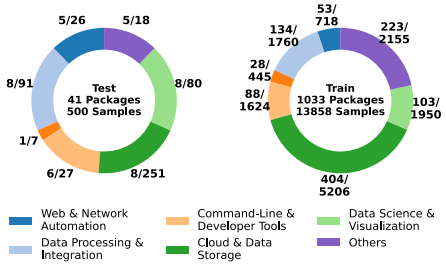


Figure 3: Distribution of SWE-Dev training and test samples across 6 major PyPI application domains.

Table 2: 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

each test file using `pytest` inside a containerized Docker environment and apply Python's built-in `trace` module to record all triggered functions in source code. This results multiple linear call chains that record the sequence of invoked source functions. (2) Call tree ensemble: We aggregate the call chains into 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 convert it into a feature development task by (1) masking the relevant code and (2) generating a natural language requirement for this feature. These two components constitute a typical development scenario in which a feature is functionally defined but not yet implemented. 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 select function nodes that represent the core logic under test. The corresponding implementation code is removed from the repository, leaving a functional gap to fill. (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. 9 and prompt in Appendix H.

3.2 Dataset Features

Controlled Complexity via Call Tree: Leveraging call-tree analysis, SWE-Dev enables systematic adjustment of task difficulty by adjusting dependency depth for task generation. This uniquely supports rigorous assessment of model capabilities against varying complexities, see § 5 discussion.

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.

Executable Training Support: SWE-Dev includes runnable environments and test cases for every sample, enabling training paradigms such as SFT and RL with execution-based feedback.

3.3 Statistics

Table 2 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

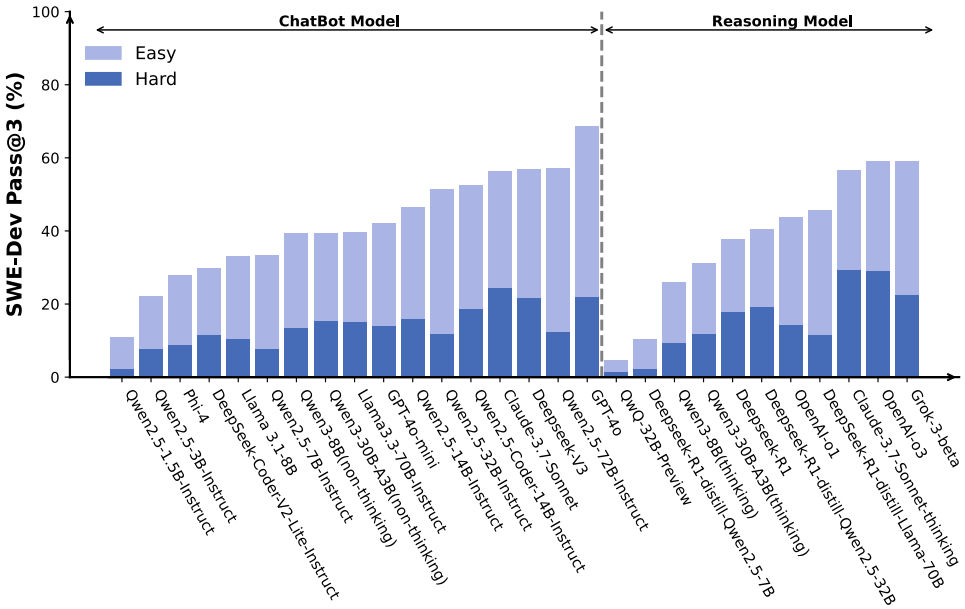


Figure 4: Comparison of Pass@3 scores for 17 chatbot and 10 reasoning LLMs on SWE-Dev across *Easy* and *Hard* splits. SWE-Dev poses substantial challenges and effectively distinguishes model capabilities under both difficulty levels. See Appendix B for full results.

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 3 shows the distribution of training and test samples across six major PyPI application domains, demonstrating the diversity of software categories represented in the dataset. More statistics are in Appendix A.

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 FDD 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 [20]. For inference code context, since SWE-Dev requires both the PRD and codebase as inputs. The codebases consist of many tokens (an average of 202K lines, see Table 2), exceeding typical LLM context window. Thus, in all the experiments below, we provide only the relevant code context—i.e., the specific files involved in the task—rather than the entire codebase.

4.1 Testing Results

This section presents the performance of 17 chatbot LLMs, 10 reasoning LLMs, and 10 multi-agent systems on SWE-Dev, under the single-LLM and multi-agent settings. Full details of the evaluated methods are provided in Appendix F.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. Figure 4 reports Pass@3 performance of chatbot and reasoning LLMs on SWE-Dev. We observe that:

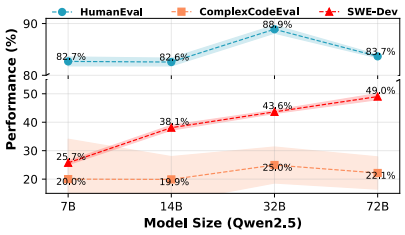


Figure 5: Comparison of benchmarks on various model sizes. SWE-Dev shows clear performance scaling with model size, while HumanEval[20] fails to distinguish between models. ComplexCodeEval[15] using CodeBLEU[33] exhibits high variance, making it less stable for evaluation.

Table 3: 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 [14]	39.77	2.12	0.83	13.32	2.18	1.35
Self Refine [34]	**40.02**	5.00	5.78	**20.03**	5.00	5.8
Self Consistency [35]	37.62	6.00	4.30	18.55	6.00	7.08
LLM Debate [36]	38.48	7.00	5.95	14.56	7.00	9.35
MAD [37]	31.50	7.00	2.48	15.31	7.00	3.40
Agentverse [38]	**38.67**	4.52	1.40	13.42	4.83	2.90
EvoMAC [18]	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 [39]	24.55	21.01	6.05	5.87	23.41	10.55
ChatDev [40]	35.13	26.61	3.53	11.70	30.87	6.10

(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 LLM (Claude-3.7-Sonnet[11]) achieves just over 20% on the *Hard* split. This still fall short of achieving strong performance, indicating that current models are not yet fully capable of handling tasks that approximate the complexity of real-world developing scenarios.

Reasoning models generally underperform their base counterparts, with Claude-3.7-Sonnet being a exception. While Claude with thinking outperforms its base variant, most reasoning models yield worse results. This suggests that current reasoning strategies do not consistently translate into gains for complex, repository-level generation tasks. We further explain this in Appendix C.2.

SWE-Dev provides stable and discriminative evaluation of model capabilities.

Figure 5 compares the performance of Qwen2.5 [41] family on SWE-Dev, HumanEval [20], and ComplexCodeEval [15] across three runs. We use Pass@1 for SWE-Dev and HumanEval and ComplexCodeEval for CodeBLEU [33]. The lines represent the average performance, and the shaded regions show the variance. We observe that: (1) SWE-Dev 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.

4.1.2 Multi-Agent Inference

Table 3 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 AppendixF.1. Key observations are

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[34] and EvoMAC[18] improve performance to 20.03% and 13.60%, respectively. These results highlight the advantage of multi-agent collaboration 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 Map-

Table 4: 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. Fine-tuning yields consistent gains.

	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

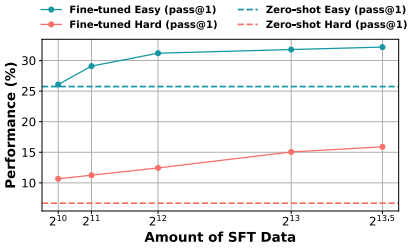


Figure 6: Training data scaling of SFT Qwen2.5-7B-instruct on SWE-Dev. As data size increases, performance improves steadily under SFT.

Coder 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.

Our results highlight MAS's potential for complex tasks on SWE-Dev but reveal a gap between simple and complex MAS, indicating that scalable, efficient MAS remain a challenge. Future work could focus on balancing collaboration benefits with resource costs and mitigating error amplification from LLM hallucinations across agent interactions.

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 F.2.

4.2.1 Single LLM SFT

We conducted experiments on Qwen2.5-Intstruct 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 F.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 4, 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. Figure 6 illustrates the scaling law of training using Qwen2.5-7b-instruct. In this experiment, we measured model performance across varying amounts of fine-tuning data, specifically tracking changes in Pass@1 for both easy and hard task. As shown in the figure, performance improves steadily as the amount of fine-tuning data increases, with larger improvements observed for harder tasks.

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 F.4.

Table 5: 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**

Table 6: Comparison of multi-agent role-wise training, base MAS and single LLM's performance on Qwen2.5-7B-Instruct. Δ indicates the relative improvement over the base MAS system. Partial fine-tuning of either agent also leads to consistent gains, demonstrating the effectiveness of role-specific supervision enabled by SWE-Dev.

	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

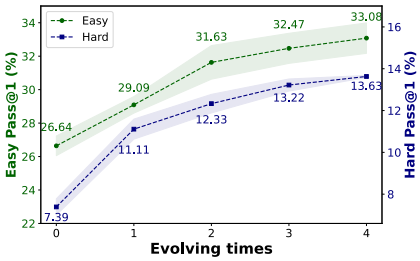


Figure 7: EvoMAC performance trajectory under ground truth test case supervision on SWE-Dev with Qwen2.5-7B-Instruct. EvoMAC iteratively improves across reasoning rounds, guided by executable test feedback.

Both online and offline RL improve performance, especially on hard tasks. Table 5 shows that both PPO [42] and DPO [43] significantly improve Pass@1 performance, especially on the *Hard* split. Furthermore, PPO outperforms SFT 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[44]. Therefore, while RL improves performance, significant headroom remains, and more advanced methods or further training are needed to achieve improvements across tasks.

4.2.3 Multi-Agent Training

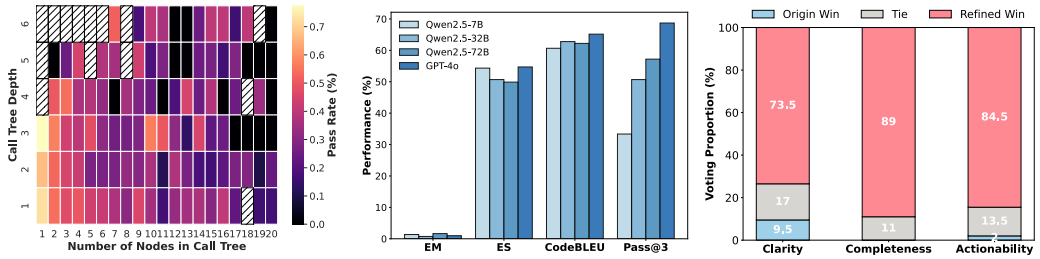
MAS has shown promising results on SWE-Dev, and we further investigate the training process of MAS on this dataset. As depicted in Fig. 7, the ground truth test case supervision in SWE-Dev enables EvoMAC [18] to improve its performance across multiple rounds of reasoning. This iterative refinement process 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 6 presents the 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.

5 Dataset Analysis

We analyze SWE-Dev's task complexity, evaluation setup, and PRD quality to demonstrate its uniqueness and reliability.



(a) Complexity analysis (b) Metric Comparison (c) PRD Quality Analysis

Figure 8: Analysis of SWE-Dev Benchmark Characteristics. (a) Compares GPT-4o's performance across tasks grouped by call tree depth and node count, showing that greater structural complexity correlates with lower accuracy. (b) Compares several evaluation metrics; Pass@3 shows the clearest differentiation across model scales. (c) Compares human ratings of original vs. refined PRD on 100 samples of 3 dimensions, revealing SWE-Dev's high PRD quality.

Analysis of Task Difficulty and Call Tree Characteristics. We analyze how task difficulty in SWE-Dev correlates with call tree complexity. As introduced in § 3.1, a call tree reflects the dynamic function invocation structure for this task, where nodes represent functions and edges denote their call relationships. We use two metrics: depth, indicating the maximum call nesting, and node count, representing the total number of distinct functions involved in the task. Fig. 8a shows that GPT-4o's performance declines as depth and node count increase, revealing a strong correlation between structural complexity and task difficulty. This suggests that deeper and broader call structures introduce more functional requirements and interdependencies, making tasks more challenging.

Evaluation Method Precision. SWE-Dev uses execution-based evaluation with test cases, enabling precise performance signals. We compare metrics: Exact Match (EM) [19], Exact Sequence (ES) [19], CodeBLEU [33], and Pass@3, using Qwen2.5 models and GPT-4o. As Fig 8b shows, Pass@3 best reflects capability scaling, separating models by size and quality. In contrast, EM, ES, and CodeBLEU show minimal variance, failing to distinguish models. This demonstrates that SWE-Dev's test-case-based evaluation provides a more robust and realistic signal of model performance, better reflecting the functional correctness required in real-world software development.

PRD Quality. SWE-Dev includes a PRD for each task to simulate realistic developer-facing requirements, which are primarily derived from the original docstrings found within the repository source code. While many functions in open-source code include docstrings, we found that these are often incomplete—lacking clear descriptions of behavior, parameters, or edge cases. To improve instruction clarity without fabricating content, we lightly refine existing docstrings using GPT-4o, grounded in the related file and surrounding context. To evaluate instruction quality, we conducted a human assessment on 100 sampled tasks. Two experienced engineers rated the original and refined PRDs along Actionability, Completeness, and Clarity (Appendix C.1 includes human instruction). As shown in Fig. 8c, refined PRDs consistently scored higher across all dimensions. This supports SWE-Dev's goal of providing realistic, well-scoped requirements for reliable model evaluation.

6 Conclusion

In this work, we introduced SWE-Dev, the first dataset for evaluating and training autonomous coding systems on feature-driven development task. SWE-Dev consists of 14,000 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 FDD 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 long-context reasoning, agent orchestration, and execution-aware training towards genuinely autonomous software engineering.

References

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

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

[3] 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*.

[4] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

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

[6] Stephen R Palmer and John M Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall PTR, 2002.

[7] Luan Xavier, Marco D'Ambros, André Hora, Romain Robbes, and Marco Tulio Valente. Characterizing and comparing the distribution of software evolution tasks in industrial and open source projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2017.

[8] Harness Team. Getting started with feature driven development: A comprehensive guide. April 2025. Accessed: May 10, 2025. This guide discusses FDD principles including planning by feature and aligning with business requirements, which are conducive to automated assessment via acceptance criteria.

[9] Lily Kazerouni, April Mitchell, John Smith, and Emily Johnson. Large language models for software engineering: A systematic literature review. *Journal of Systems and Software*, 211:111999, 2024.

[10] Chunqiu Steven Xia and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*, pages 1106–1118. IEEE Press, 2023.

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

[12] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[13] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

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

[15] Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 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, 2024.

[16] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 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, 2024.

[17] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *CoRR*, 2024.

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

[19] Jiaheng Liu, Ken Deng, Congnan Liu, Jian Yang, Shukai Liu, He Zhu, Peng Zhao, Linzheng Chai, Yanan Wu, Ke Jin, et al. M2rc-eval: Massively multilingual repository-level code completion evaluation. *arXiv preprint arXiv:2410.21157*, 2024.

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

[21] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

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

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

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

[26] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

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

[28] 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*.

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

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

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

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

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

[34] 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. Self-refine: Iterative refinement with self-feedback, 2023.

[35] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[36] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023.

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

[38] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors, 2023.

[39] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024.

[40] 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. Chatdev: Communicative agents for software development, 2024.

[41] 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, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

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

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

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

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

[46] 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, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[47] Jian Hu, Jason Klein Liu, and Wei Shen. Reinforce++: An efficient rlhf algorithm with robustness to both prompt and reward models, 2025.

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

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

[50] 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, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

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

[52] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, and et al. The llama 3 herd of models, 2024.

[53] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 technical report, 2024.

[54] 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, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, and etal. Deepseek-v3 technical report, 2025.

[55] 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, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024.

[56] 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, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024.

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

[58] 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. Openai o1 system card, 2024.

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

[60] 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. Metagpt: Meta programming for a multi-agent collaborative framework, 2024.

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

[62] 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. Maslab: A unified and comprehensive codebase for llm-based multi-agent systems, 2025.

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

[64] Charlie Snell, Ilya Kostrikov, Yi Su, Mengjiao Yang, and Sergey Levine. Offline rl for natural language generation with implicit language q learning, 2023.