A Dataset

A.1 Dataset Information

Figure 9 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.

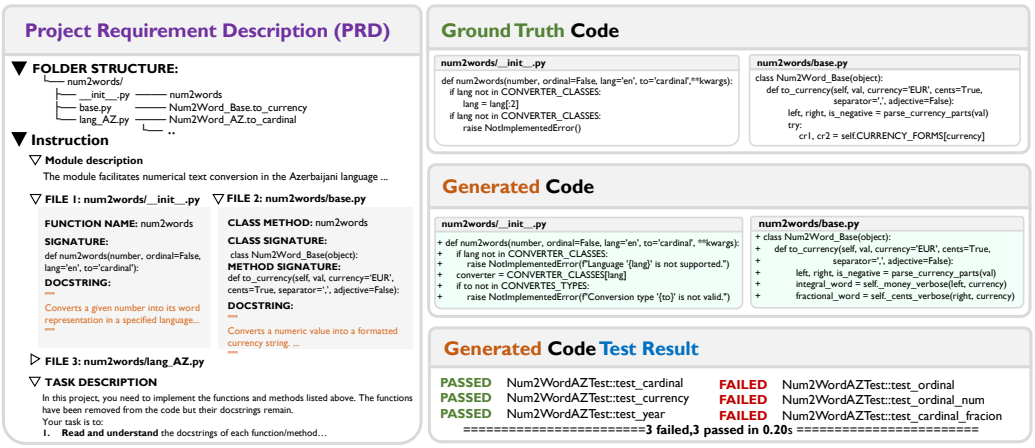


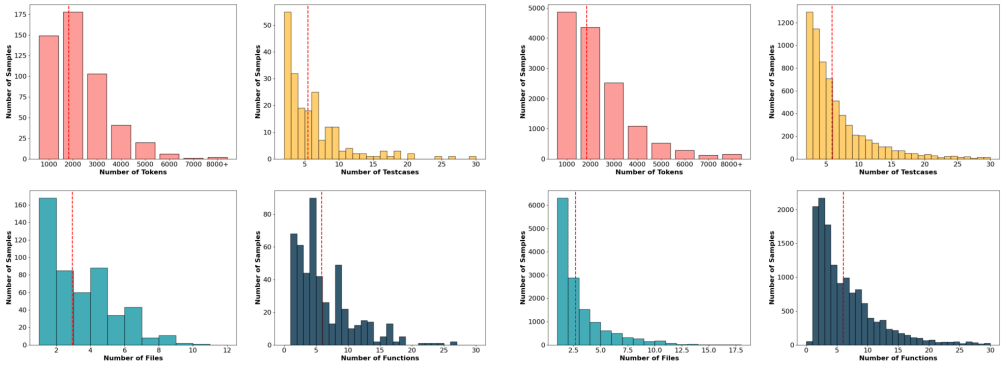
Figure 9: Example of a SWE-Dev Sample. Each sample includes a Project Requirement Description (PRD) with folder structure, module-level task description, and masked docstrings; the corresponding ground truth implementation. Generated code is evaluated by the test function execution results. This structure supports realistic, testable feature development in a repository context.

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

Field	Description
`PRD`	Description of the project development requirements for this task.
`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.

A.2 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.



(a) Test Set Distribution (b) Train Set Distribution

Figure 10: Dataset distribution of PRD tokens, number of testcases, number of files to complete, and number of functions per sample.

A.3 Dataset Diversity

We assess the diversity of SWE-Dev from two perspectives: sample-level diversity (Figure 11 and Figure 12), and package-level diversity (Figure 23).

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.[2] 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-mini with the prompt provided below (see Figure 23). The resulting distribution confirms that SWE-Dev spans a broad spectrum of software domains.

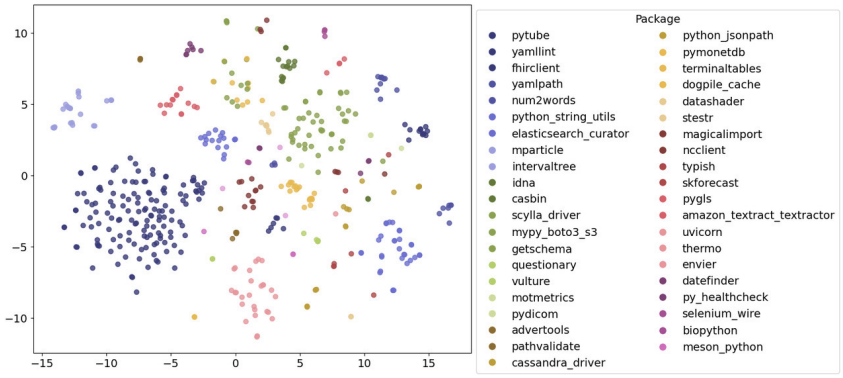


Figure 11: t-SNE visualization of PRD in test set

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

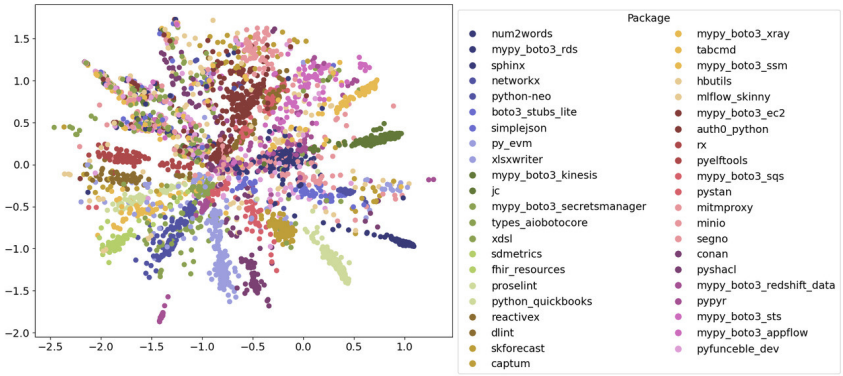


Figure 12: t-SNE visualization of PRD in train set

B Inference Results

To assess the capabilities and limitations of current LLMs on realistic feature-driven development (FDD) 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.

B.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 [20] and ComplexCodeEval [15], using Pass@3 and CodeBLEU respectively. Our findings show that SWE-Dev poses significantly greater challenges than existing benchmarks, with leading models achieving under 25% Pass@3 on hard tasks (Table 8).

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).

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

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

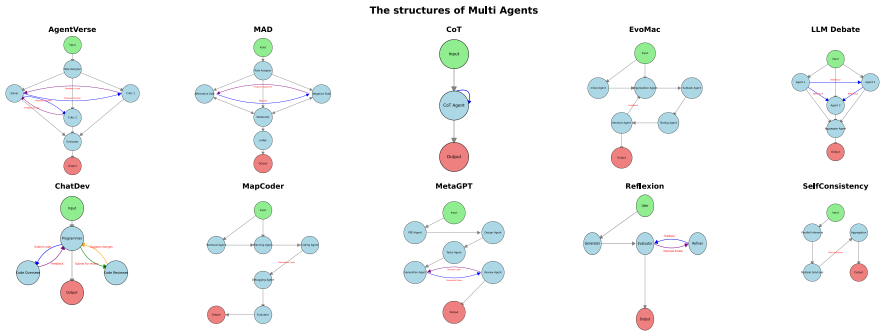


Figure 13: Multiagent system workflow visualization.

B.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 3, 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 13.

C Analysis

C.1 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.

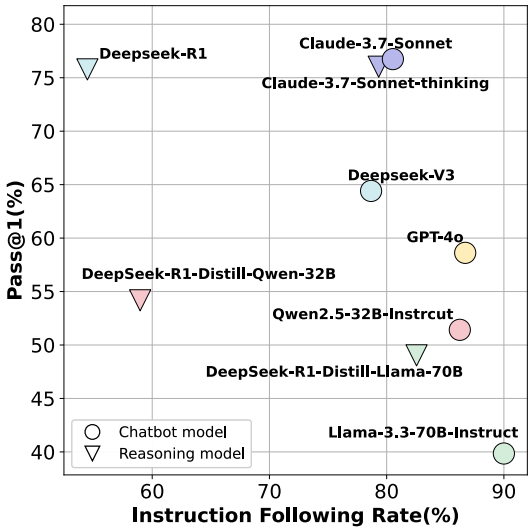


Figure 14: Comparison of reasoning and chatbot LLMs' IFR and performance on SWE-Dev. Reasoning models tend to outperform chatbots when they fully follow instructions, though their overall IFR is lower.

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 24.

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.

C.2 Explanation 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 14, 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) 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.

C.3 Error Analysis

Figure 15 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 21. 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.

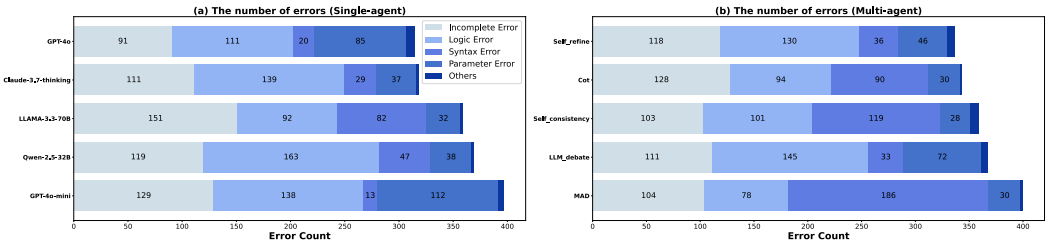


Figure 15: Failure case distribution of Single and Multi-agent.

C.4 Limitation and Future Work

Language Scope. SWE-Dev currently targets Python, which, while widely used, does not reflect the full diversity of real-world programming languages. A natural extension is to support other major languages such as Java, JavaScript, and C++, enabling broader evaluation and enhancing generality.

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 [45–47], dynamic agent coordination [48], and curriculum learning [49]. Notably, SWE-Dev offers fine-grained complexity signals via call trees that can guide complexity-aware training.

C.5 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.

D Detailed Benchmark Construction

D.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.

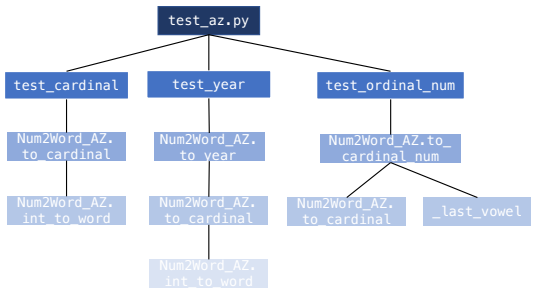


Figure 16: Example of a generated call tree for `test_az.py`.

Figure 16 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_cardinal` 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 must understand to solve the task, supporting fine-grained evaluation and curriculum learning.

D.2 Docstring Augmentation Prompt

To ensure high-quality task specifications, we augment original function-level docstrings using GPT-4o. Figure 22 shows the prompt we use to generate concise, informative docstrings conditioned on the full code context.

E Extended Related Work

E.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.

F Experiment Settings

F.1 Inference

LLMs. We evaluate 17 chatbot LLMs with different model size, including Qwen2.5-Instruct models 1.5B/3B/7B/14B/32B/72B [50], Qwen3 models 8B/30B-A3B [51], Llama 3.1-8B/3.3-70B-Instruct [52], Phi 4 [53], Claude-3.7-Sonnet [11], Deepseek-V3 [54], GPT-4o [12], Deepseek-Coder-V2-Lite-Instruct [55], Qwen2.5-Coder-14B-Instruct [56]. Additionally, We extend the evaluation to reasoning models, including Deepseek-R1-distill models (Qwen2.5 7B/32B, Llama-70B) [57], Qwen3

8B/30B-A3B (thinking) [51], QwQ-32B-Preview, Deepseek-R1 [57], OpenAI-o1 [58], Claude-3.7-Sonnet-thinking [11], and Grok-3-Beta [59].

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 [48, 60, 40]. In our experiments, all MAS are implemented using GPT-4o-mini [61] as the underlying model to ensure consistency across methods. And for fair comparison, we utilize MASLab [62], a unified framework integrating multiple MAS implementations. We evaluate coordination-based MAS such as LLM Debate [36], Self Refine [34], Multi-Agent Debate (MAD) [37], and Self Consistency [35] that feature relatively simple agent interaction strategies. We further include structured, workflow-oriented MAS designed for code generation, including Agentverse [38], MetaGPT [60], ChatDev [40], MapCoder [39], and EvoMAC [18].

F.2 Training

F.3 Single-Agent Supervied 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 sublayer. 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.

F.4 Single-Agent Reinforcement Learning

For reinforcement learning (RL) training, we sampled 2k instances from the SWE-Dev to balance computational feasibility and the ability to capture RL benefits. Specifically, we used Proximal Policy Optimization (PPO) [42] and Direct Preference Optimization (DPO) [43] for training the Qwen2.5-7B-Instruct using 8 NVIDIA A100 GPUs.

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. 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 reward for PPO is calculated based on the pass rate of the test cases.

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 F.3.

These methods allow us to assess the impact of RL on model performance using the SWE-Dev dataset while maintaining efficient training..

F.5 Mingle-Agent Supervied Fine-tuning

In our multi-agent fine-tuning experiments, we utilize a simplified version of EvoMAC [18], retaining only two core roles: **Organizer** and **Coder**, see Figure 17. 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. 19 and Fig. 20 respectively show the prompts used for the Coder and Organizer.

Rejection Sampling Procedure.

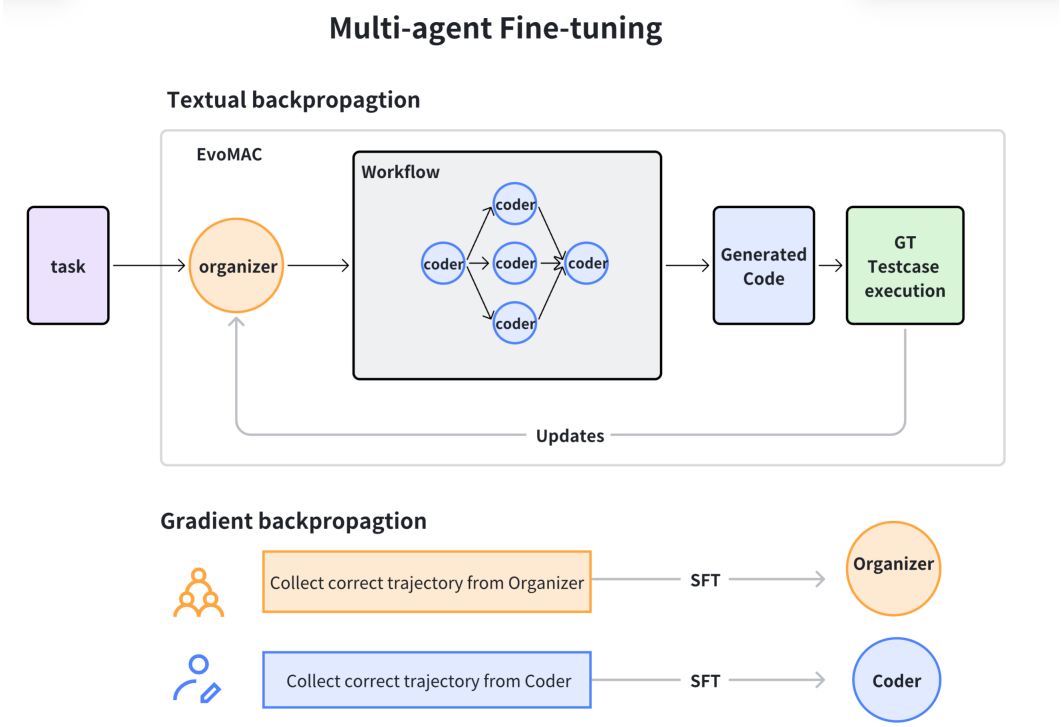


Figure 17: Overview of Multi-agent Fine-tuning in EvoMAC. This framework leverages both textual and gradient-based supervision to improve multi-agent collaboration. During inference, an organizer assigns roles and coordinates a team of coders to generate code, which is then validated using ground truth 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.

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 [63, 64]. 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 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.

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 18: Single LLM Inference Prompt.

G 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.

H Prompts

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

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 19: 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 20: 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 21: 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 22: 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 23: 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 24: Human Evaluation Guideline for PRD Quality.