Synthesizing Code Completion Trajectories for Mid-training in Software Engineering

Siwei Xu[1], **Qiyuan Liu**[1], **Ruifan Li**[1,2,3*]
[1]School of Artificial Intelligence, Beijing University of Posts and Telecommunications, China
[2]Engineering Research Center of Information Networks, Ministry of Education, China
[3]Key Laboratory of Interactive Technology and Experience System,
Ministry of Culture and Tourism, China
{xusw, qiyuanliu, rfli}@bupt.edu.cn

Abstract

Large language models (LLMs) have shown remarkable capabilities in interactive real-world software engineering (SWE) scenarios. However, the execution-aware training data are hard to scale and are mostly used for post-training, leaving agentic mid-training underexplored. To this end, we propose COMP-SYNTH, a novel automated pipeline for synthesizing reflective code completion trajectories. Specifically, based on executable repository environments, COMP-SYNTH identifies test-covered target functions, masks their implementations, and prompts an LLM to complete them under execution feedback. Then, each generated completion is verified by running the associated tests. Furthermore, failed attempts are augmented with diagnostic signals to extract useful supervision. Thus, we construct COMP-TRAJ, a mid-training dataset that exposes models to repository context, execution feedback, and iterative code revision. Extensive experiments on repository-level SWE benchmarks show that mid-training on COMP-TRAJ effectively enhances capabilities in SWE. Our code and data are publicly available at https://github.com/uuam5/comp-traj.

1 Introduction

In recent years, Large language models (LLMs) have made rapid progress in code generation (Chen et al., 2021; Wang et al., 2023b). They can generate code from natural-language instructions, assist with program implementation, and modify existing code (Ribeiro, 2023; Gao et al., 2025). These advances have enabled LLMs to solve a wide range of coding tasks with strong fluency and effectiveness. A key reason for this progress is the continuous improvement of training data.

Early synthetic code datasets are primarily constructed as independent, single-turn instruction-solution pairs (Li et al., 2022; Wang et al., 2024).

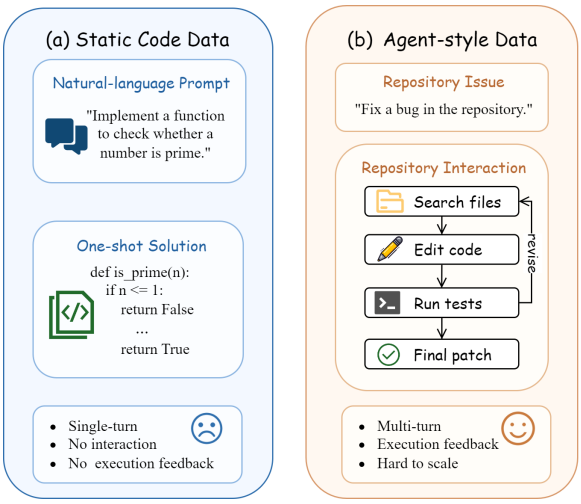


Figure 1: The comparison between traditional static code data and our targeted agent-style data.

Each sample couples a natural-language prompt with a self-contained snippet answer. This format improves instruction-following and local code generation, but it remains far from real-world repository-level software engineering (SWE). In repository-level SWE, code must be consistent with the broader codebase and verifiable through execution. More recent work has therefore paid increasing attention to agent-style SWE settings (Jimenez et al., 2024; Yang et al., 2024, 2025), where models interact with repositories, tools or execution environments. Figure 1 highlights the difference between these two forms of code data.

However, scaling such agentic data remains difficult. High-quality trajectories depend on reliable executable environments and meaningful interaction traces. Manual collection is costly, while automatic collection depends on repository environments that are hard to build, and verify at scale. Moreover, most agentic data are introduced only during post-training (Zhao et al., 2024; Song et al., 2026; Golubev et al., 2025). This creates a gap between pre-training and final task alignment. Pre-

*Corresponding author.

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 4619–4630
November 6-10, 2026 ©2026 Association for Computational Linguistics

training gives models broad code knowledge, but it rarely exposes them to repository interaction or execution feedback. As a result, post-training alone may be insufficient to develop complex agentic coding behaviors.

To this end, we need to consider the following questions. **First**, how can we automatically obtain large amounts of diverse, environment-grounded interaction data for code models? We need a reliable pipeline for synthesizing verifiable tasks paired with fully executable environments. A natural solution is to start from existing public repositories with executable environments, and derive new software engineering tasks and interaction trajectories from them. This requires that the synthesized task remain grounded in realistic repository context while also being verifiable through execution. **Second**, when is the optimal stage to introduce such interaction trajectories? Mid-training emerges as an intermediate stage between large-scale pre-training and final post-training. It has been shown to effectively inject targeted domain knowledge, such as long-context and code (Yang et al., 2026). It can also provide stronger priors for later reinforcement learning (Wang et al., 2025c). We argue that this stage is suitable for environment-interaction trajectories. This helps models build stronger agentic coding priors before final alignment.

To tackle these questions, we propose COMP-SYNTH, a pipeline that synthesizes execution-grounded code completion trajectories as mid-training data from containerized repositories. **First**, we construct code completion tasks directly within executable repository environments, so that each training instance is grounded in realistic project structure and constraints. Specifically, we identify target functions that are covered by tests so that each candidate completion can be automatically verified. Simultaneously, we gather their relevant repository context, including surrounding definitions and key calling relations. We then mask the target function and prompt an LLM to complete it based on the provided context. All generated attempts are immediately validated by running the associated test in the same environment. We repeat this process until all the tests pass or a maximum attempt is reached. In this way, existing executable repositories are automatically expanded into large-scale, diverse, and verifiable interaction data.

Second, we organize the synthesized trajectories into learnable supervision by incorporating test execution feedback into the mid-training data. A naive option is to keep only the final passing solution. But this would discard the failure signals that guide debugging, which are central to real software development. Conversely, retaining raw trial-and-error traces can introduce substantial noise and are difficult to learn from. Therefore, for persistent failures, we further augment trajectories by adding a detailed root-cause analysis. Specifically, we provide an LLM with the history of failed patches and execution errors, together with the ground-truth solution. The LLM then explains why earlier attempts failed and how the correct implementation resolves the failing tests. This design exposes models to the write–run–debug loop. Models learn to understand environment feedback, revise code accordingly, and build agentic coding behaviors for later post-training.

Our contributions are summarized as follows.

- We propose COMP-SYNTH, a fully automatic pipeline that synthesizes high-quality multi-turn execution-grounded code completion trajectories with executable feedback from real-world repositories.
- We build COMP-TRAJ, a novel training dataset for mid-training that teaches models to improve code generation under execution feedback rather than imitating final solutions.
- We conduct extensive experiments on three repository-level SWE benchmarks to show the effectiveness of COMP-TRAJ in improving SWE capabilities through mid-training.

2 Related Work

2.1 Synthetic Code Data

Due to the limited availability of high-quality human-annotated data, researchers have increasingly explored synthetic data generation as a scalable alternative. Early methods (Wang et al., 2023a; Xu et al., 2024; Luo et al., 2024; Wei et al., 2024; Wang et al., 2024) leverage pretrained LLMs to generate instruction–response pairs for fine-tuning. More recent work moves toward repository-level software engineering data, where training instances are grounded in executable environments, issue contexts, tests, or agent interactions. Executable repository environments provide the foundation for such data by enabling agents to run code, execute tests, and receive verifiable feedback (Pan et al., 2025; Badertdinov et al., 2025; Jain et al., 2025;

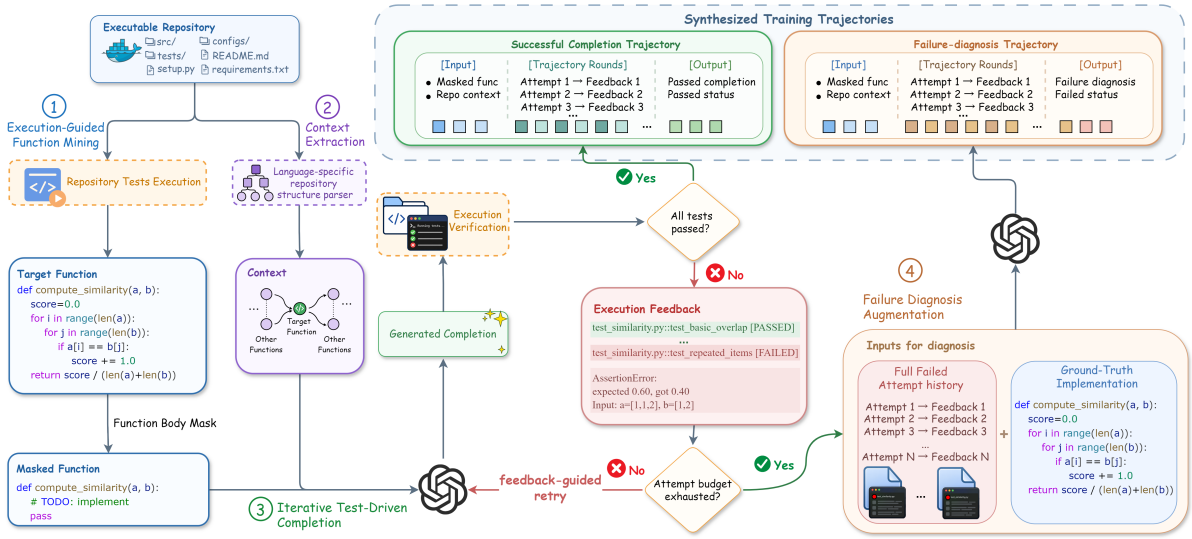


Figure 2: The overview of the proposed COMP-SYNTH pipeline.

Guo et al., 2026). Recent studies further curate such executable environments and synthesize issue-resolution trajectories for improving repository-level code generation (Yang et al., 2025; Wang et al., 2025a; Zeng et al., 2025; Wang et al., 2025b). These studies show that environment-grounded data is crucial for training models to solve realistic software engineering tasks.

2.2 Mid-training for Code Models

Recent code LLMs increasingly treat mid-training as a stage between pre-training and post-training. Existing works have used this stage to expose models to more structured supervision before the final alignment stage. For example, Qwen3-Coder-Next (Cao et al., 2026) uses targeted mid-training to further improve code reasoning and repository-level understanding. In daVinci-Dev (Zeng et al., 2026), mid-training includes agent-style trajectories so that models can observe more complete software engineering processes. CWM (team et al., 2025) introduces execution traces, agent trajectories, and executable environments during mid-training to improve the modeling of dynamic code behavior. Our work is most closely related to this line of research, but introduces a new method for synthesizing execution-aware code completion trajectories from real repositories.

3 Problem Formulation

Formally, a pre-configured software environment $\mathcal{E}$ contains three elements: 1) a repository $\mathcal{R}$ with the source code $\mathcal{C}$, 2) a test suite $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$, and 3) a runtime environment, including dependencies, interpreter, and system configurations.

Completion instance. Given an environment $\mathcal{E}$, we identify a set of test-covered target functions $\mathcal{F} = \{f_1, f_2, \ldots, f_m\} \subseteq \mathcal{C}$, where each function f_i is covered by at least one test case in $\mathcal{T}$. For each target function $f_i \in \mathcal{F}$, we construct a completion instance $I_{f_i} = (f_i^*, \mathcal{X}(f_i), \mathcal{T}(f_i))$, where f_i^* denotes the masked function, $\mathcal{X}(f_i)$ denotes the repository context of f_i, and $\mathcal{T}(f_i) \subseteq \mathcal{T}$ denotes the associated tests used for verification. The context $\mathcal{X}(f_i)$ includes the function signature and surrounding dependency context. Given this instance, an LLM generates a candidate completion g for the masked function body. We reconstruct the repository code as $\mathcal{C}' = \mathcal{C}[f_i \leftarrow g]$, and verify g by running $\mathcal{T}(f_i)$ in the same environment.

Trajectory synthesis. Thus, based on each completion instance I_{f_i}, we synthesize a multi-turn trajectory $\tau = \{(p_0, g_0, r_0), (p_1, g_1, r_1), \ldots, (p_K, g_K, r_K)\}$. Here, p_k is the prompt at round k, g_k is the generated completion, and r_k is the execution feedback obtained after applying g_k and running the associated tests. If the tests fail, the feedback r_k is incorporated into the next prompt, and the model is asked to revise its completion. The process terminates when all associated tests pass or when the maximum attempt budget is reached. The final synthesized dataset is denoted as $\mathcal{D}_{\text{synth}} = \{\tau_i\}_{i=1}^{N}$.

4 Data Synthesis Methodology

COMP-SYNTH synthesizes execution-grounded code completion trajectories for mid-training with four stages: execution-guided function mining, context extraction, iterative test-driven completion, and failure diagnosis augmentation. Figure 2 illustrates the overall COMP-SYNTH pipeline.

4.1 Execution-Guided Function Mining

To identify target functions that support automatic verification, we mine functions that are exercised by the repository test suite. Specifically, we run the test suite $\mathcal{T}$ in the environment $\mathcal{E}$ with coverage instrumentation enabled. This produces a line-level execution signal $\texttt{cov}(l)$ for each source line l. We then map covered lines back to functions using the repository's syntax structure. For a function f, let $\mathrm{Lines}(f)$ denote its source line span. We define a function-level coverage score as follows,

$$\mathrm{cov}_{\mathcal{T}}(f) = \sum_{l \in \mathrm{Lines}(f)} \mathbb{I}[\texttt{cov}(l) > 0]. \quad (1)$$

We include f in the target set if $\mathrm{cov}_{\mathcal{T}}(f) > 0$, i.e., it is executed by at least one test. This could produce the target set $\mathcal{F}$ by Eq. (1). In addition, we record the associated test set $\mathcal{T}(f) \subseteq \mathcal{T}$ for each target function.

4.2 Context Extraction

For each target function $f \in \mathcal{F}$, we extract a compact repository context $\mathcal{X}(f)$. This context has two types of information that are critical in repository-level completion: (i) how f is used by other parts of the codebase and (ii) what code f relies on. Specifically, we construct a directed call graph over functions in the repository. Each node corresponds to a function, and an edge from node u to node v indicates that function u calls function v. We identify these relations by extracting function calls within each function body based on the syntactic names, such as `foo()` or `obj.foo()`.

Furthermore, we define the context of a target function as follows,

$$\mathcal{X}(f) = \big(\mathrm{Def}(f),\ \mathrm{Callers}(f),\ \mathrm{Callees}(f)\big), \quad (2)$$

where $\mathrm{Def}(f)$ contains the function header information, such as the signature and docstring; $\mathrm{Callers}(f) = \{\mathrm{Def}(u) \mid u \to f\}$ summarizes representative caller-side snippets containing call sites; and $\mathrm{Callees}(f) = \{\mathrm{Def}(v) \mid f \to v\}$ summarizes direct dependency snippets invoked by f. We serialize these components into a prompt-friendly format with explicit section boundaries. Thus, the model can ground its completion in specific repository usage and dependencies.

Algorithm 1 Iterative Test-Driven Completion for a target function f

Require: Environment $\mathcal{E}$, code $\mathcal{C}$, target function f, masked function f^*, context $\mathcal{X}(f)$, tests $\mathcal{T}(f)$, budget `maxtry`
Ensure: Trajectory $\tau = \{(p_k, g_k, r_k)\}_{k=0}^{K}$

$\tau \leftarrow \emptyset$
$H_0 \leftarrow \emptyset$
for $k = 0$ to `maxtry`-1 **do**
  $\widetilde{H}_k \leftarrow \mathrm{Recent}(H_k)$
  $p_k \leftarrow \mathrm{BuildPrompt}(f^*, \mathcal{X}(f), \widetilde{H}_k)$
  $g_k \leftarrow \mathrm{LLM}(p_k)$
  $\mathcal{C}'_k \leftarrow \mathcal{C}[f \leftarrow g_k]$
  $r_k \leftarrow \mathrm{RunTests}(\mathcal{E}, \mathcal{C}'_k, \mathcal{T}(f))$
  $\tau \leftarrow \tau \cup \{(p_k, g_k, r_k)\}$
  if r_k indicates all tests pass **then**
    break
  end if
  $H_{k+1} \leftarrow H_k \cup \{(g_k, r_k)\}$
end for
return τ

4.3 Iterative Test-Driven Completion

Given a target function f and its masked version f^*, together with the repository context $\mathcal{X}(f)$ and the associated tests $\mathcal{T}(f)$, we model repository-level function completion as an iterative refinement process driven by environmental feedback. Let L_θ be an LLM parameterized by θ.

At iteration k, the model conditions on the trajectory history, $H_k = \{(g_0, r_0), \ldots, (g_{k-1}, r_{k-1})\}$, where g_j is the generated completion at step j and r_j is the corresponding execution feedback. In our implementation, we use a compact prompt history

$$\widetilde{H}_k = \begin{cases} \emptyset, & k = 0, \\ \{(g_{k-1}, r_{k-1})\}, & k > 0, \end{cases}$$

which retains only the most recent completion-feedback pair to control the context length. The model produces a completion g_k for the masked function body, i.e.,

$$g_k \sim P(g \mid \mathcal{X}(f), \widetilde{H}_k; \theta). \quad (3)$$

We apply g_k to reconstruct code $\mathcal{C}'_k$ and execute the

Initial prompt ($k = 0$): $p_0 = \Pi(f^*, \mathcal{X}(f), \emptyset)$	
Goal	Complete the masked function so that the repository tests in $\mathcal{T}(f)$ pass.
Target	`Path: <file_path>  Function: <signature>` `<masked function code, i.e., f*>`
Repository context	**Local:** `<relevant imports / nearby definitions>` **Callers:** `<representative call sites that invoke the target function>` **Callees:** `<functions invoked by the target function>`
Verification	The completion will be validated by running `pytest` on `<tests covering the target function>` in the same environment.
Output	Return *only* the completion code for the masked body. Do not include explanations.
Revision prompt ($k > 0$): $p_k = \Pi(f^*, \mathcal{X}(f), H_k)$	
History	**Previous completion:** $\langle g_{k-1} \rangle$ **Execution feedback:** $\langle r_{k-1} \rangle$ (e.g., failing tests, traceback, error messages).
Instruction	Revise the completion to fix the failing tests. Keep the function signature unchanged.
Output	Return *only* the updated completion code.

Table 1: Our prompt template for iterative test-driven completion. Here f^* denotes the masked target function, $\mathcal{X}(f)$ denotes its repository context, and H_k denotes the interaction history before round k. Placeholders are shown in <>.

associated tests to obtain feedback r_k, i.e.,

$$r_k = \text{Execute}(\mathcal{C}'_k, \mathcal{T}(f)).$$

The process terminates if $\text{Pass}(\mathcal{C}'_k, \mathcal{T}(f)) = \text{True}$. Otherwise, we incorporate the execution feedback into the next prompt and ask the model to revise its completion. Algorithm 1 summarizes the iterative test-driven completion.

We implement Eq. (3) with a deterministic prompt template. For the initial round, the prompt is constructed as $p_0 = \Pi(f^*, \mathcal{X}(f), \emptyset)$, For revision rounds, we construct the prompt using the compact history $\widetilde{H}_k$, i.e., $p_k = \Pi\left(f^*, \mathcal{X}(f), \widetilde{H}_k\right), \quad k > 0$. Table 1 shows the template. This compact prompt provides the latest execution signal while avoiding unnecessary growth of the interaction history.

4.4 Failure Diagnosis Augmentation

Note that not all target functions can be completed within the attempt budget. However, failed trajectories still contain valuable learning signals. When all attempts fail, we augment the trajectory with a diagnosis step. Specifically, for a failed trajectory, we provide an LLM with the original completion prompt, the full history of attempted completions and execution feedback, and the ground-truth implementation of the masked function. The LLM then produces a concise root-cause diagnosis d_τ, explaining why previous attempts failed and how the ground-truth solution resolves the failing tests. We append d_τ to the failed trajectory.

After iterative completion and failure diagnosis augmentation, we obtain the final synthesized dataset $\mathcal{D}_{\text{synth}} = \mathcal{D}_{\text{succ}} \cup \mathcal{D}_{\text{fail}}$, where $\mathcal{D}_{\text{succ}}$ contains trajectories that pass the associated tests within the attempt budget, and $\mathcal{D}_{\text{fail}}$ contains trajectories that remain failing and are augmented with diagnosis supervision. We use this dataset to mid-train a target code model.

5 Mid-training Objective

Our objective trains the target model to learn from both successful test-driven refinement trajectories and failure-derived diagnostic supervision. Specifically, our mid-training objective is as follows,

$$\mathcal{J}(\theta) = \mathcal{J}_{\text{succ}}(\theta) + \mathcal{J}_{\text{fail}}(\theta). \tag{4}$$

For successful trajectories, the model learns to generate each completion g_k conditioned on the masked function, repository context, and previous interaction history, i.e.,

$$\mathcal{J}_{\text{succ}}(\theta) = \sum_{\tau \in \mathcal{D}_{\text{succ}}} \left(\sum_{k=0}^{K_\tau} \log P(g_k \mid f^*, \mathcal{X}(f), H_k; \theta) \right). \tag{5}$$

For failed trajectories, the model learns from the appended root-cause diagnosis d_τ, i.e.,

$$\mathcal{J}_{\text{fail}}(\theta) = \sum_{\tau \in \mathcal{D}_{\text{fail}}} \log P(d_\tau \mid f^*, \mathcal{X}(f), H_\tau; \theta). \tag{6}$$

6 Experiments

6.1 Experimental Setup

Evaluation benchmarks and metrics. We evaluate all models on three SWE benchmarks: SWE-bench Verified (OpenAI, 2024), SWE-bench Multilingual (Zan et al., 2025), and SWE-bench Pro (Deng et al., 2026). SWE-bench Verified is a human-validated subset of 500 instances from the

Model	Params.	Funcs.	Pass Count		Avg. Rounds
			First	Final	
gpt-oss-120b	117B	1000	255	339	1.47
deepseek-v3.2	685B	1000	233	310	1.41
glm-4.7	358B	1000	118	249	2.03

Table 2: The performance comparison of candidate models for our trajectory synthesis.

original SWE-bench. SWE-bench Multilingual extends the setting beyond Python to multiple programming languages. SWE-bench Pro provides a more challenging setting with harder issue instances than predecessors. Furthermore, we use the OpenHands scaffold with a 150-step budget. We report Pass@1 as the resolved percentage, where a task is considered resolved if the generated patch passes the official tests. For each setting, we report the average result over two runs.

Data sources for synthesis. Our synthetic trajectory data are built from executable repository environments drawn from SWE-rebench (Badertdinov et al., 2025), SWE-Gym (Pan et al., 2025), and DockSmith (Zhang et al., 2026). For each repository instance, we identify test-covered target functions, mask the function body, and ask the synthesis model to iteratively complete the function under execution feedback. We keep only trajectories that can be verified in the corresponding environment.

Base model and training protocol. We use Qwen3-8B-Base (Team, 2025) as the base model for all the experiments. Our baseline applies supervised fine-tuning (SFT) to the base model with the open-source SFT dataset (StepFun AI, 2025). Furthermore, we investigate whether our synthesized repository-level trajectories are more effective when used in mid-training before the final SFT stage. Unless otherwise specified, all proposed variants follow the same two-stage recipe: 1) mid-training on synthetic trajectory data produced by our pipeline, and 2) SFT on the same SFT dataset. This design isolates the effect of our synthetic data from the downstream instruction-tuning stage.

6.2 Experiments on Synthesis Model Selection

To select the best model for trajectory synthesis, we compare gpt-oss-120b (OpenAI et al., 2025), DeepSeek-V3.2 (DeepSeek-AI et al., 2025), and GLM-4.7 (Team et al., 2025). Specifically, we randomly sample 200 repository instances and extract up to five target functions from each instance, yielding 1,000 function completion tasks. For each function, we allow at most five completion attempts and report first-round success, eventual success within the five-round budget, and the average number of attempts required to reach a passing completion.

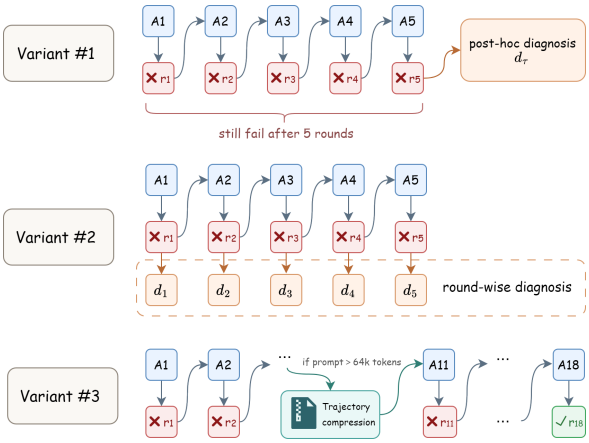


Figure 3: Illustration of three synthetic pipeline variants.

As shown in Table 2, gpt-oss-120b achieves the best overall performance among the compared models. Therefore, we use it as the default model for data synthesis. It offers an optimal trade-off between effectiveness and computational cost, making it practical for large-scale trajectory generation.

6.3 Synthetic Pipeline Variants and Statistics

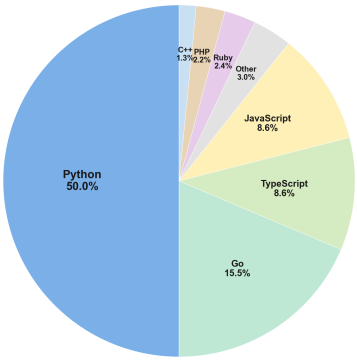
After constructing executable function completion tasks, we synthesize trajectories using three variants of our data-generation pipeline. These variants differ in their attempt budgets and in how failure diagnosis is incorporated into the trajectory. Figure 3 illustrates the differences among the three variants. The details of three variants are given as follows.

Variant #1: post-hoc diagnosis after bounded attempts. This variant follows the default synthesis pipeline described in Section 4, using a 5-round attempt budget. If the function still fails after 5 attempts, we collect the execution feedback from the five rounds together with the final completion attempt and ask the model to produce an overall failure analysis. In practice, execution feedback is often verbose but highly redundant, so it can therefore be distilled into a shorter representation without losing the key error signals, making the pipeline practical for large-scale synthesis.

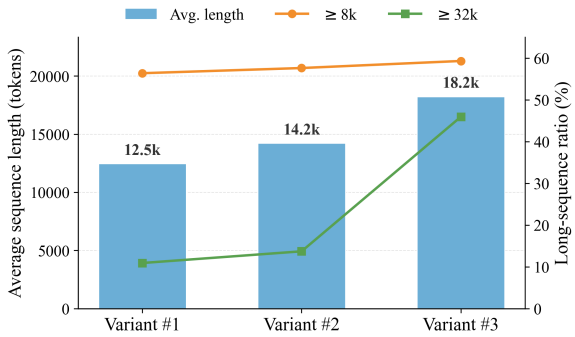
Variant #2: round-wise fine-grained diagnosis. This variant also uses at most 5 completion attempts, but differs in how failed trajectories are analyzed. Instead of producing a single diagnosis after all attempts fail, Variant #2 generates a diagnosis for each failed round. For each round, the

Source	Lang.	Repo Envs.	Trajectory	Max Attempts	Total Funcs.	Success Count		Avg. Rounds
						First	Final	
SWE-rebench	Python	7,512	Variant #1&#2	5	110,885	29,508 (26.61%)	40,591 (36.61%)	1.481
			Variant #3	20		41,757 (37.66%)	66,573 (60.04%)	2.628
SWE-Gym	Python	1,073	Variant #1&#2	5	21,431	9,671 (45.13%)	12,516 (58.40%)	1.369
			Variant #3	20		10,003 (46.68%)	14,906 (69.55%)	2.346
DockSmith	Multi	23,645	Variant #1&#2	5	321,364	129,083 (40.17%)	164,466 (51.18%)	1.352
			Variant #3	20		132,032 (41.08%)	191,070 (59.46%)	2.310

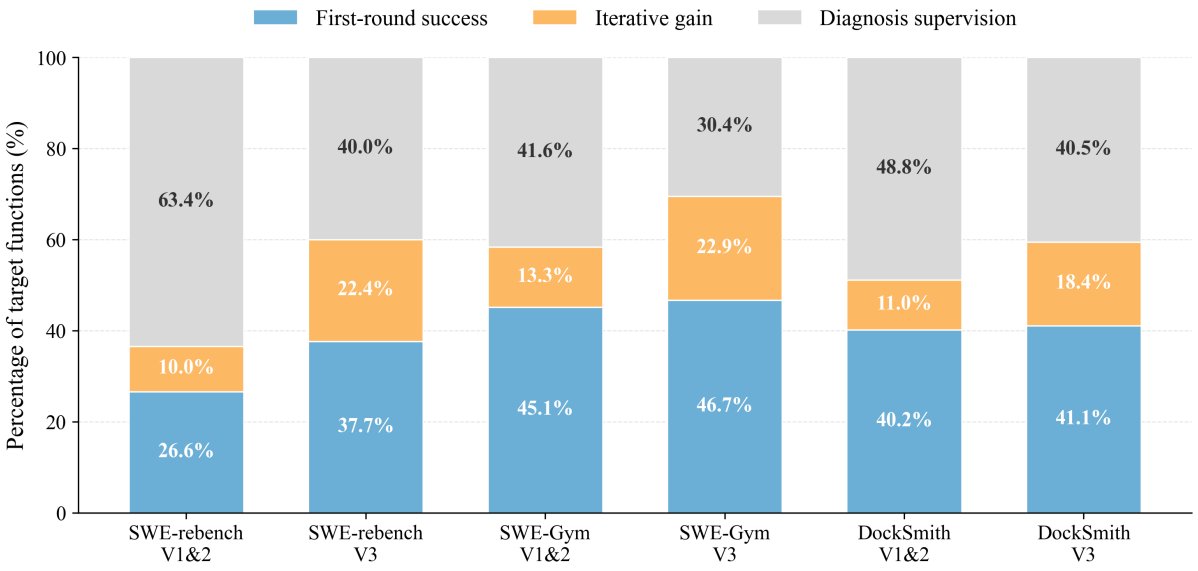
Table 3: The synthetic data statistics by source and trajectory variant. Repo Envs. denotes executable repository environments, and Total Funcs. denotes function-level completion instances. Final success is defined as success within the allowed attempt budget.



(a) Language distribution



(b) Trajectory length profile



(c) Contribution of execution feedback

Figure 4: Visualization of the synthesized trajectory data. (a) Language distribution of executable repository environments. (b) Length profile of synthesized trajectories, including average sequence length and long-sequence ratios. (c) Contribution of iterative feedback to trajectory synthesis.

model analyzes the failed completion and the corresponding test feedback. This design alleviates the context burden of full-history analysis in Variant #1 and provides finer-grained attribution signals.

Variant #3: long-horizon synthesis with trajectory compression. This variant extends the attempt budget to 20 rounds. To support longer interaction histories, we introduce automatic trajectory compression. Whenever the prompt length exceeds 64k tokens, we summarize the accumulated history into a compact reflective memory and insert it back into the prompt. This allows us to preserve long-horizon debugging information while keeping generation within the model's context limit.

Method	Prompt Policy	SWE-bench		
		Verified	Multi.	Pro
Baseline	–	29.2	10.1	6.2
Variant #1	No_Instruction	34.5	12.7	10.9
Variant #2	No_Instruction	36.8	14.8	12.9
Variant #3	No_Instruction	38.4	16.2	18.0
Variant #1	Masked_Instruction	34.7	2.7	8.8
Variant #2	Masked_Instruction	34.5	17.2	12.2
Variant #3	Masked_Instruction	34.0	7.8	9.3

Table 4: Main results on three software engineering benchmarks under a unified training recipe. All proposed variants perform synthetic mid-training before the same downstream SFT stage.

Mid-training Data	SWE-bench		
	Verified	Multi.	Pro
SFT-only baseline	29.2	10.1	6.2
Static function completion	33.5	12.1	9.6
Successful trajectories only	36.9	14.6	14.2
Full trajectories	38.4	16.2	18.0

Table 5: Ablation study of mid-training data under the Variant #3 and No Instruction setting. Static function completion uses the same target functions and repository context without execution feedback. Successful trajectories only keep trajectories that pass within the attempt budget.

Data Statistics and Insights. Table 3 reports detailed synthesis statistics across different data sources and trajectory variants. For each source, we report the number of executable repository environments, target functions, first-round successes, eventual successes within the allowed attempt budget, and average rounds-to-success over successful samples. The variation across sources reflects differences in repository complexity, test coverage, and target function difficulty.

Moreover, Figure 4 presents an intuitive view of our synthesized trajectory data. First, the language distribution in (a) shows that our data involves multiple programming languages beyond Python. Second, the length profile in (b) shows that Variant #3 produces longer trajectories and a higher proportion of samples over 32k tokens, reflecting the effect of long-horizon synthesis. Third, (c) highlights the value of iterative feedback. The orange segments represent functions that fail in the first round but are later solved through feedback-driven revision. The additional successes indicate that execution feedback improves trajectory synthesis rather than merely serving as a final verifier.

6.4 Main Experiments and Results

We compare an SFT Qwen3-8B-Base baseline against the six methods. These methods involve three synthetic pipeline variants and two prompt policies. All three synthetic pipeline variants introduce execution-aware coding behaviors during mid-training. This allows us to test whether repository-grounded synthetic trajectories provide useful supervision before the final post-training stage. The two prompt policies control whether prompt instructions are included during training. Specifically, No_Instruction excludes prompt instruction text during trajectory stitching. In contrast, Masked_Instruction concatenates it into the sequence but masks it out from the loss.

The experimental results are reported in Table 4. Overall, synthetic mid-training consistently improves downstream SWE performance over the baseline. The baseline achieves 29.2% on SWE-bench Verified, 10.1% on SWE-bench Multilingual, and 6.2% on SWE-bench Pro. In contrast, most settings outperform the baseline on all three benchmarks. This shows that repository-grounded synthetic trajectories provide useful supervision beyond standard post-training alone. This improvement is consistent with the nature of repository-level SWE tasks. SWE-bench-style issue resolutions ultimately require localized code generation or modification under the repository context. Therefore, repository-level code completion is a fundamental atomic capability for these tasks. Our trajectory data directly targets this underlying operation through test-verified completion and revision, making the resulting mid-training signal well aligned with downstream issue-resolution benchmarks.

6.5 Further Analysis and Discussion

Effect of context efficiency. We analyze two trajectory organization strategies, No_Instruction and Masked_Instruction. As shown in Table 4, No_Instruction provides a more stable strategy across trajectory variants and benchmarks. This suggests that the main supervision signal in our data comes from the execution-grounded trajectory content itself, including generated completions, test feedback, and subsequent revisions, rather than from retaining the original prompt instructions. This difference is closely related to context efficiency. Although masked instruction tokens are excluded from the loss, they still remain in the input sequence and participate in attention.

Method	Backbone	Mid-training	Post-training	Scaffold	SWE-V
R2EGym-Agent (Jain et al., 2025)	Qwen2.5-Coder-32B-Inst	✗	SFT	R2E-Gym	34.4
SWE-Dev (Wang et al., 2025a)	Qwen2.5-Coder-32B-Inst	✗	SFT+RL	OpenHands	36.6
Skywork-SWE (Zeng et al., 2025)	Qwen2.5-Coder-32B-Inst	✗	SFT	OpenHands	38.0
SWE-Agent-LM (Yang et al., 2025)	Qwen2.5-Coder-32B-Inst	✗	SFT	SWE-Agent	40.2
DeepSWE-Preview (Luo et al., 2025)	Qwen3-32B-Inst	✗	RL	R2E-Gym	42.2
Seed-Coder-Instruct (Seed et al., 2025)	Seed-Coder-Base-8B	✗	SFT+RL	Agentless	19.2
SWE-Mirror-LM (Wang et al., 2025b)	Qwen2.5-Coder-Instruct-7B	✗	SFT	OpenHands	22.8
Our SFT-only baseline	Qwen3-base-8B	✗	SFT	OpenHands	29.2
Our COMP-SYNTH	Qwen3-base-8B	✓	SFT	OpenHands	38.4

Table 6: Performance comparison with representative methods on SWE-bench Verified.

Effect of longer interaction. We discuss Variant #1 and #3 on whether extending the interaction horizon from 5 rounds to 20 rounds produces additional gains. We focus on the No_Instruction setting for this analysis, since it better isolates the effect of trajectory content without introducing additional prompt tokens into the training sequence. Under this setting, Variant #3 consistently outperforms #1 across all three benchmarks. This indicates that longer write–run–debug trajectories provide richer supervision beyond short completion traces. With more rounds of completion and execution feedback, the model observes how revisions change test outcomes.

Effect of diagnosis granularity. We examine Variant #1 and #2 to show the effect of diagnosis granularity under an identical 5-round attempt budget. The results show that Variant #2 generally improves over #1. This suggests that the fine-grained design brings more consistent gains than the post-hoc design.

6.6 Ablation Study

We further conduct ablations under the best-performing setting, Variant #3 with the No Instruction policy, to isolate the effect of different mid-training data designs. The results are shown in Table 5. **First**, static function completion improves over the SFT-only baseline, indicating that repository-level completion data itself is beneficial for downstream SWE performance. However, it remains clearly below full trajectory training across all three benchmarks. This suggests that the improvement is not only due to adding more repository training data. **Second**, successful trajectories only further improve over static function completion, showing the benefit of retaining intermediate completions and execution feedback. Further incorporating diagnosis-augmented failed trajectories brings additional gains.

6.7 Comparison with Representative Methods

Table 6 shows the results of our method with representative methods on SWE-bench Verified. Among 7B/8B-scale models, our COMP-SYNTH achieves the strongest result, reaching 38.4% compared with 19.2% for Seed-Coder-Instruct and 22.8% for SWE-Mirror-LM. Our method also outperforms two larger-backbone methods evaluated with OpenHands, SWE-Dev and Skywork-SWE. These results show that execution-grounded mid-training enables a relatively small 8B model to achieve competitive performance on SWE-bench Verified.

7 Conclusion and Future Work

In this paper, we propose COMP-SYNTH, an automatic pipeline for synthesizing execution-grounded code completion trajectories from executable repositories. The resulting dataset COMP-TRAJ exposes models to repository context, execution feedback, and iterative code revision, making it promising for mid-training before final post-training. Extensive experiments on SWE benchmarks show that such trajectories effectively improve software engineering capabilities. In future work, we plan to extend our synthesis framework to more comprehensive SWE trajectories, covering bug localization, cross-file reasoning, and broader tool-use behaviors.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFC3305902, in part by the Special Project for Industrial Foundation Reconstruction and High-Quality Development of Manufacturing under Grant Agreement No. ZC25T320057/100, in part by the Key Research and Development and Achievement Transformation Program of Inner Mongolia Autonomous Region under Grant No. 2026YFHH0570.

Limitations

Our method relies on executable repository tests for verification. Although test execution provides a practical way to automatically validate synthesized completions, its effectiveness depends on both the availability and the quality of tests. When a repository or target function lacks executable tests, the completion cannot be directly verified and thus cannot be used for synthesis. When test coverage is incomplete, an implementation may pass execution while still missing important corner cases or intended behaviors. A natural extension is to incorporate automatic test generation or test augmentation. This would make more target functions verifiable and help the synthesized trajectories cover a broader range of repository behaviors.

In addition, our current synthesis process focuses on repository-level function completion. This setting captures an important operation underlying many software engineering tasks, but it does not cover the full decision-making process of software engineering agents, such as issue localization, multi-file planning, and tool-use decisions. Future extensions could increase task complexity by moving beyond single-function completion, requiring the model to complete multiple related functions.

References

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. 2026. Qwen3-coder-next technical report. *Preprint*, arXiv:2603.00729.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R. Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2026. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *Proceedings of the 43rd International Conference on Machine Learning*.

Ruofan Gao, Amjed Tahir, Peng Liang, Teo Susnjak, and Foutse Khomh. 2025. A survey of bugs in ai-generated code. *Preprint*, arXiv:2512.05239.

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

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. 2026. SWE data construction, automatically! *Proceedings of the ACM on Software Engineering*, 3(FSE):525–546.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. 2025. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl. `https://pretty-radio-b75.notio`

n.site/DeepSWE-Training-a-Fully-Open-sourced-State-of-the-Art-Coding-Agent-by-Scaling-RL-22281902c1468193aabbe9a8c59bbe33. Notion Blog.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, and 108 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

OpenAI. 2024. Introducing swe-bench verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed: 2026-04-14.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with swe-gym. In *Forty-second International Conference on Machine Learning*.

Francisco Ribeiro. 2023. Large language models for automated program repair. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH 2023, page 7–9, New York, NY, USA. Association for Computing Machinery.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, and 8 others. 2025. Seed-coder: Let the code model curate data for itself. *Preprint*, arXiv:2506.03524.

Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Yiming Jia, and 1 others. 2026. Swe-master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411*.

StepFun AI. 2025. Step-3.5-flash-sft. https://huggingface.co/datasets/stepfun-ai/Step-3.5-Flash-SFT. Accessed: 2026-04-14.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, and 32 others. 2025. Cwm: An open-weights llm for research on code generation with world models. *Preprint*, arXiv:2510.02387.

GLM Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, and 152 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *Preprint*, arXiv:2508.06471.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. 2025a. SWE-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 3742–3761, Vienna, Austria. Association for Computational Linguistics.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. 2025b. Swe-mirror: Scaling issue-resolving datasets by mirroring issues across repositories. *Preprint*, arXiv:2509.08724.

Yejie Wang, Keqing He, Dayuan Fu, Zhuoma GongQue, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, Jingang Wang, Mengdi Zhang, Xunliang Cai, and Weiran Xu. 2024. How do your code LLMs perform? empowering code instruction tuning with really good data. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14027–14043, Miami, Florida, USA. Association for Computational Linguistics.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023a. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023b. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Zengzhi Wang, Fan Zhou, Xuefeng Li, and Pengfei Liu. 2025c. Octothinker: Mid-training incentivizes reinforcement learning scaling. In *2nd AI for Math Workshop at Forty-second International Conference on Machine Learning 2025*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. In *Advances in Neural Information Processing Systems*, volume 37, pages 62787–62874. Curran Associates, Inc.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *International Conference on Learning Representations*, volume 2024, pages 30745–30766.

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

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. Swe-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, Yanhao Li, Yue Liu, Zhenxing Hu, Kaitai Zhang, Shuyi Wang, Huarong Chen, Flood Sung, Yang Liu, Yang Gao, and 2 others. 2026. Kimi-Dev: Agentless Training as Skill Prior for SWE-Agents. In *International Conference on Learning Representations*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc.

Ji Zeng, Dayuan Fu, Tiantian Mi, Yumin Zhuang, Yaxing Huang, Xuefeng Li, Lyumanshan Ye, Muhang Xie, Qishuo Hua, Zhen Huang, Mohan Jiang, Hanning Wang, Jifan Lin, Yang Xiao, Jie Sun, Yunze Wu, and Pengfei Liu. 2026. daVinci-Dev: Agent-native mid-training for software engineering. In *Proceedings of the 43rd International Conference on Machine Learning*.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and Yahui Zhou. 2025. Skywork-swe: Unveiling data scaling laws for software engineering in llms. *Preprint*, arXiv:2506.19290.

Jiaran Zhang, Luck Ma, Yanhao Li, Fanqi Wan, Di Qi, Xu Zhao, Jieyi Hou, Zhe Xie, Mengqiang Ren, Xin Wu, and 1 others. 2026. Docksmith: Scaling reliable coding environments via an agentic docker builder. In *Forty-Third International Conference on Machine Learning*.

Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. 2024. RePair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 16415–16429, Bangkok, Thailand. Association for Computational Linguistics.

A Training Details

We provide the training hyperparameters used in our experiments. All models are initialized from Qwen3-8B-Base, and each training stage is run for one epoch. The SFT-only baseline directly applies the SFT recipe to Qwen3-8B-Base. For settings with synthetic mid-training, we first train the base model on the synthesized trajectory data and then apply the same downstream SFT recipe as the baseline.

Mid-training configuration. For synthetic mid-training, we use a global batch size of 128 and a sequence length of 32,768 tokens. The peak learning rate is set to 2×10^{-5} with 100 warmup steps, and the weight decay is set to 0.1.

SFT configuration. After mid-training, all proposed models are further fine-tuned using the same SFT recipe. We use a global batch size of 128 and a sequence length of 131,072 tokens. The peak learning rate is set to 1×10^{-5} with 200 warmup steps.

B AI Assistance Disclosure

We used generative AI tools to assist with language polishing, grammar checking, and improving the clarity of writing. All research ideas, experimental design, data construction, analysis, and conclusions were developed and verified by the authors.