

# SEAlign: Alignment Training for Software Engineering Agent

Kechi Zhang<sup>†</sup>, Huangzhao Zhang, Ge Li<sup>† \*</sup>, Jinliang You<sup>†</sup>, Jia Li<sup>†</sup>, Yunfei Zhao<sup>†</sup>, Zhi Jin<sup>† \*</sup>

<sup>†</sup>Key Lab of High Confidence Software Technology (PKU), Ministry of Education,

School of Computer Science, Peking University

China

{zhangkechi, lige, zhijin}@pku.edu.cn

## Abstract

Recent advances in code generation models have demonstrated impressive capabilities in automating software development tasks, yet these models still struggle in real-world software engineering scenarios. Although current training methods, particularly post-training, excel at solving competitive programming problems, they fail to adequately prepare models for the complexities of practical software development. This misalignment raises the critical question: Are existing alignment training methods well suited for real-world software engineering tasks? In this study, we identify this issue and propose SEAlign, a novel alignment framework designed to bridge the gap between code generation models and real-world software development tasks. SEAlign leverages the unique characteristics of software engineering processes, including high-quality workflow steps, to enhance model capabilities. Our framework further employs Monte Carlo Tree Search for fine-grained alignment in multi-step decision processes, followed by preference optimization on critical actions to ensure models meet real-world requirements. We evaluate SEAlign on three standard agentic benchmarks for real-world software engineering, including HumanEvalFix, SWE-Bench-Lite, and SWE-Bench-Verified. Experimental results demonstrate state-of-the-art performance with minimal training overhead. In addition, we develop an agent-based software development platform using SEAlign, which successfully automates the creation of several small applications. Human evaluations of these applications highlight significant improvements in both task performance and user experience. Our findings underscore the potential of SEAlign to accelerate the adoption of large code models in real-world software development. We believe that this research makes a meaningful step towards fully automated software engineering.<sup>1</sup>

## 1 Introduction

Code generation has become a pivotal area in artificial intelligence, with models automating essential software development tasks and significantly boosting developer productivity. Recent advances in code models (such as GPT-4 [6], LLaMA [19], and DeepSeekCoder [9]), have shown impressive abilities to generate functional and effective code. These models have greatly improved automation in various code-related tasks, making software development more efficient.

Current code generation models, *i.e.*, Large Language Models (LLMs), typically follow a two-stage training process: **①** pre-training, where the model learns basics of programming languages and foundational skills, and **②** post-training, where the model learns to follow specific instructions and is aligned with human behaviors

and preferences. Recent alignment training techniques for code models, such as CodeDPO [43] and LIMO [40], have been proven highly effective, achieving significant improvements with minimal data and computational resources in certain scenarios. However, the post-training phase of existing code generation models focuses mainly on solving competitive programming contest problems from platforms like LeetCode [15] and Codeforces [4] – there is a significant gap between algorithmic problems and real-world software development scenarios. Practical software engineering demands a composite of various skills, including the retrieval of information from complex repositories and the flexible application of existing algorithms to current code contexts, *etc.*, but existing post-training solutions often perform poorly. Bridging this gap is crucial for accelerating the adoption of code models in software development automation.

Despite the rapid advancements in model capabilities which have surpassed the majority of human programmers on competitive coding benchmarks, these models often struggle in real-world software development scenarios as aforementioned. Benchmarks such as SWE-bench [12], which simulates real-world scenarios of issue fixing, reveal that many open-source code models perform less satisfactorily. This discrepancy raises an important question: Are current code model training schemes (specifically post-training techniques) capable to align the models with the skills required for real-world software development well? To answer the question in a brief and straightforward manner, we conduct a preliminary investigation, evaluating Qwen2.5-Coder-Instruct-14B [10] (which is a powerful code model extensively post-trained) and Openhands [33] (which is a popular agentic framework) against SWE-Bench-Lite [12] (which is a benchmark simulating real-world software engineering tasks). The experiment shows that the model could only solve 3.7% of problems within the benchmark. Diving into the failure cases, we find that the incapability of the model is mainly caused by **①** the poor instruction following ability, **②** the incorrect tool selection and usage, and **③** infinite agentic loops (refer to Section 3.2 for more analysis). Our preliminary investigation results directly reveal that existing post-training methods do not adequately prepare models for the complexities of software engineering tasks, especially within agentic framework.

To address this issue, we propose **SEAlign**, a novel framework designed to align code models with real-world software development agentic workflows. SEAlign focuses on the unique characteristics of software engineering, enhancing models' ability to handle complex tasks. In general, SEAlign enables code models to follow instructions and use developing tools correctly by tuning them under real-world tasks. Specifically, we collect high-quality samples of software development agentic trajectories (*i.e.*, decision processes), identify critical action steps in the workflow, and force the model to

<sup>\*</sup>Corresponding authors.

<sup>1</sup>Code and additional details are available at [2].

produce “good” behaviors by alignment. SEAlign uses Monte Carlo Tree Search (MCTS) to provide detailed scoring and alignment for these multi-step decision processes, followed by preference optimization to ensure the model meets real-world requirements.

We evaluate SEAlign on three standard benchmarks for real-world software engineering tasks: HumanEvalFix [24], SWE-Bench-Lite, and SWE-Bench-Verified [25]. Experimental results show that SEAlign achieves state-of-the-art performance with minimal training overhead. With 14B-parameter model, we achieve resolved rates of **17.7%** and **21.8%** on **SWE-Bench-Lite** and **SWE-Bench-Verified**. This performance represents the best results among models with comparable parameters within open-source methods. Furthermore, SEAlign achieves performance comparable to several commercial products with only hundreds of optimization training samples, highlighting its efficiency and practicality. Inspired by projects like OpenHands [33], we also build an agent-based software development platform using SEAlign, which has already automated the creation of multiple trivial applications. Human evaluations of these constructed applications demonstrate significant improvements in both effectiveness and user experience, highlighting the potential of SEAlign to accelerate software automated development.

The key contributions of our work are listed below:

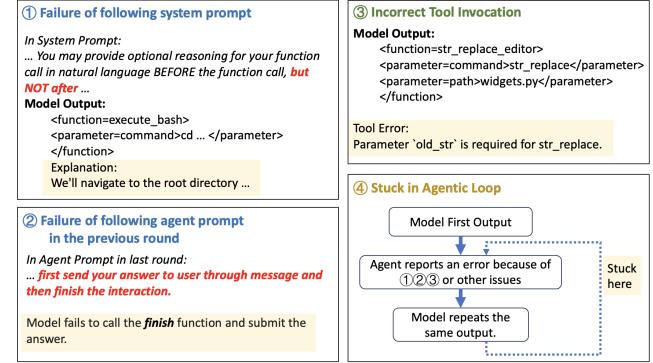
- We conduct a comprehensive analysis of existing code generation models, revealing a significant misalignment issue between existing training methods and the requirements from real-world software development agentic tasks.
- We propose a novel alignment framework specifically designed for real-world software engineering agentic workflows, namely SEAlign. It employs a fine-grained alignment method utilizing MCTS for multi-step processes, and subsequently aligns with real-world requirements through preference optimization.
- We carry out experiments on three standard benchmarks for real-world software engineering agents, *i.e.*, HumanEvalFix, SWE-Bench-Lite, and SWE-Bench-Verified, demonstrating SEAlign’s state-of-the-art performance with minimal training overhead.

SEAlign enables code models to understand and use existing software engineering agentic workflows, making the ultimate sweet fruit of fully automated software development much more feasible. Hopefully, this research could inspire subsequent researchers and contribute to the ultimate realization of the goal.

## 2 Related Work

### 2.1 Large Language Models for Code

Code generation, where models produce source code snippets from natural language (NL) descriptions, has been a hot-spot research area in recent years [42]. LLMs have shown remarkable capabilities in this domain, which can be attributed to their extensive training on diverse datasets [20]. These models are often further fine-tuned through supervised fine-tuning (SFT) to maximize their coding potential and other abilities [45]. Given the challenges associated with collecting high-quality training data for SFT, researchers have increasingly turned to self-instruct methods. In these techniques, the most powerful LLMs are employed to synthesize instruction



**Figure 1: Observed cases of misalignment in existing code models and agentic frameworks.**

data [21, 34, 35]. For instance, Evol-Instruct [21] makes use of complex prompts to improve the quality of the synthesized data. OSS-instruct [35], on the other hand, leverages real-world code snippets to enhance the relevance and practicality of the generated data.

The pre-training stage endows LLMs with excellent language and programming abilities, while the SFT stage enables them to follow instructions and complete a series of downstream tasks. Researchers have thoroughly demonstrated that LLMs have achieved human-level performance on competitive programming benchmarks and beyond [16, 26], such as HumanEval [3] and LiveCodeBench [11]. To master the more complicated tasks that originate from real-world scenarios, post-training techniques are then introduced.

### 2.2 Alignment Post-Training for Code Models

While pre-training equips code models with foundational knowledge of programming and SFT improves their instruction following capabilities, post-training techniques adapt models to perform well in diverse and complex applications [13]. Recent research has increasingly highlighted the critical role of alignment training in shaping model performance and user experience, as evidenced by models such as ChatGPT and DeepSeek-R1 [8]. In the realm of code, alignment training remains in its nascent stage. Code-Optimise [5], CodeDPO [43] and PLUM [41] construct post-training datasets from trivial programming problems, and align code models to perform greatly in such competitive programming tasks. However, these methods concentrate on solving algorithmic problems from platforms such as LeetCode and Codeforces [4, 15]. While valuable, they do not fully capture the complexities inherent in real-world software development. Beyond writing algorithmic code, practical software engineering demands a broader range of skills, *e.g.*, navigating complex codebases, adapting and rewriting code snippets to specific contexts, and restructuring software architecture [12, 44]. This gap between competitive programming and real-world software development underscores the need for more comprehensive alignment training methods tailored to practical software engineering agentic workflows.

### 3 A Motivating Example

We present a motivating example to elaborate on the misalignment issue within current LLMs for code. Despite their great performance under algorithmic code generation, LLMs still struggle in benchmarks simulating real-world scenarios.

#### 3.1 Code Agents & Agentic Tasks

LLMs have demonstrated impressive performance in various code-related tasks, such as code completion, code summarization, and code generation [1]. However, they can mostly be considered as standalone tasks, since LLMs are evaluated to solve trivial problems without requiring external knowledge or interacting with the environment. Taking competitive programming as an example, the desired algorithmic code snippets are typically short and independent of other libraries or packages.

On the contrary, in the real-world, developers usually work with specific code environments to tackle different software engineering tasks [44]. They are supposed to extend functionalities based on existing contexts, locate and fix run-time errors, as well as review code commits from others, *etc.* All these complex tasks require the ability to navigate large codebases, understand software architecture, and generate contextually appropriate code snippets. It is clearly impractical and impossible to feed the entire codebase into LLMs, as this would exceed their maximum context and computational limitations.

To address real-world requirements, researchers have begun to explore agentic frameworks for handling complex tasks [12, 44]. Agentic systems such as OpenHands [33], SWE-Agent [39], CodeAgent [44], and AutoCodeRover [46] have demonstrated significant potential in addressing complex software engineering tasks. SWE-Bench [12] further provides a real-world software engineering benchmark by collecting GitHub issues and evaluating LLMs to address them (*e.g.*, fixing bugs or adding new features for specific repositories). These agentic solutions enable LLMs to autonomously explore and interact with the environment, with multiple pre-defined programming tools and workflow constraints. For example, in the OpenHands framework [33], agents often need numerous interactive rounds to resolve a task. Upon initialization, the LLM reads the task prompt ( $p$ ) and generates its first action ( $a_1$ ). Subsequently, in each round ( $i$ ), the LLM interacts with the environment by invoking a tool using its output ( $a_i$ ), and the environment responds with an observation ( $o_i$ ) that reflects the changes resulting from the tool invocation. The complete process of the LLM resolving a task can be characterized by the following trajectory:

$$\text{Trajectory} = \langle p, a_1, o_1, a_2, o_2, \dots \rangle \quad (1)$$

where the entire trajectory is typically very long, reflecting the complexity of real-world software engineering tasks. Furthermore, real-world code projects exhibit a high degree of diversity and context dependency. This complexity presents a significant challenge in constructing datasets and training LLM that can generalize across different complex scenarios.

#### 3.2 Failure Caused by Misalignment

After introducing agentic frameworks and defining trajectories, we conduct a preliminary experiment to show the misalignment issue

of existing LLMs for code, and further analyze their failure cases. Specifically, we adopt Qwen2.5-Coder-Instruct-14B [10] as the code model and Openhands [33] as the agentic framework. We evaluate the constructed agent against the SWE-Bench-Lite benchmark [12]. To our shock, the agent can only resolve 3.7% of problems within the benchmark. After diving into the failure cases (as shown in Figure 1), we find that one of the major causes of failures is that the base LLM can hardly follow the complicated instructions from the agentic framework. Besides, the LLM often fails during tool invocation (*e.g.*, select incorrect tools or use tool erroneously), and gets stuck in action loops (*e.g.*, infinitely repeat one action). We analyze these three main failure causes as below.

**Failure of Instruction Following.** The prompts in agentic frameworks are usually rather complicated. Due to the difficulty of the real-world tasks, the prompts often include additional information such as output constraints (the first issue in Fig. 1), and workflow stages (the second issue in Fig. 1), *etc.* Although existing already-aligned code models perform well on standalone tasks, they frequently misinterpret such complex prompts or fail to execute multi-step instructions that require understanding context, navigating codebases, or invoking external tools. This limitation highlights a critical gap between their objectives during training and the demands of practical software engineering tasks.

**Incorrect Tool Invocation.** Another common failure mode is that LLMs invoke tools erroneously. *E.g.*, an LLM may select an incorrect tool from the available options or invoke a tool in an improper manner (such as using the wrong parameter in the third issue in Fig. 1). Nevertheless, real-world software engineering tasks often require LLMs to invoke and interact with appropriate tools, such as text editors, linters, or search engines. Existing LLMs often make suboptimal choices during tool invocation, resulting in incomplete or incorrect solutions. This issue highlights the critical need for better alignment between LLMs and the tools they are intended to use.

**Stuck in Agentic Loop.** In addition to the above two reasons, the agentic system has the tendency to get stuck in repetitive or unproductive loops. *E.g.*, as shown in Fig. 1, an LLM might repeatedly attempt the same action without making progress, or generate a sequence of actions irrelevant to the task. This behavior is particularly problematic in long trajectories, where the model’s inability to jump out of such loops often leads to task failure or excessive computational costs.

### 4 SEAlign

We introduce SEAlign as a novel alignment training framework designed to enhance code models for modern software engineering agentic systems. Unlike traditional alignment solutions that focus on standalone algorithmic code generation tasks, SEAlign aligns LLMs’ core capabilities with specific agentic frameworks to better address real-world software development challenges. SEAlign is not only robust but also training-efficient, making it a feasible and practical option for applications.

The framework of SEAlign uses partial trajectory pairs to post-train LLMs. To collect these pairs with preference, we thoughtfully design a trajectory tree pipeline. Figure 2 elaborates the overall framework of SEAlign. **1 Trajectory Dataset Collection.**

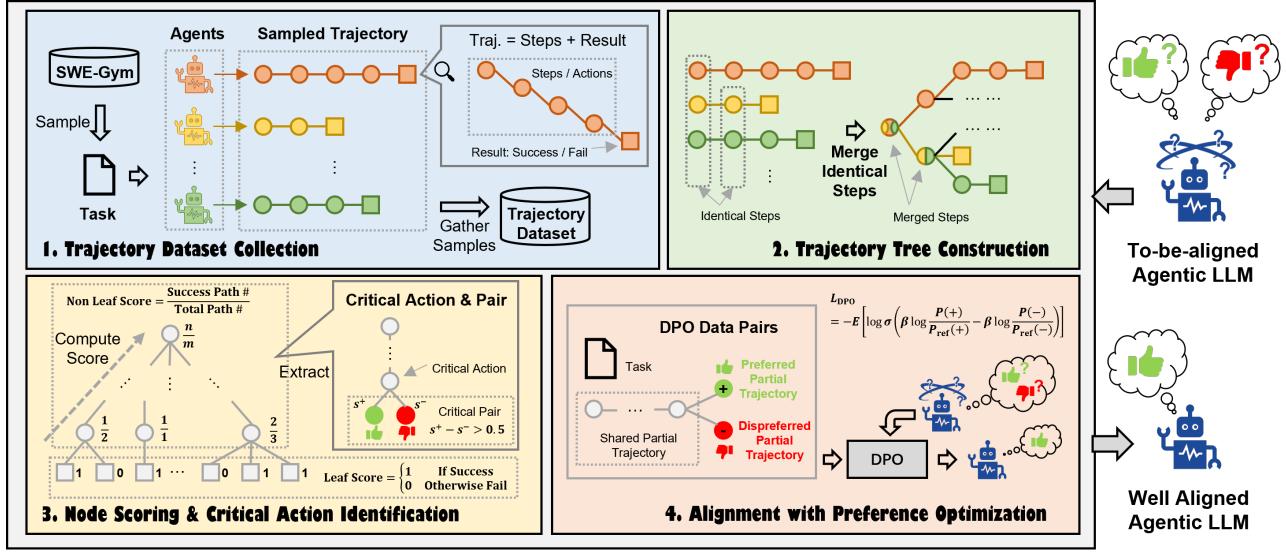


Figure 2: An overall pipeline of SEAlign (best viewed in color). Our framework involves four steps: ① collecting agentic trajectory dataset from real-world software engineering environments, namely SWE-Gym [28], ② aggregating all sampled trajectories and constructing trajectory trees, ③ scoring nodes within trajectory trees and extracting partial trajectory pairs with significant impact, and ④ optimizing model preference with critical nodes. After these steps, we can finally obtain a well-aligned agentic LLM. Note that in step ④, it is usual to SFT the LLM before DPO as a warmup for the instruction following. To facilitate viewing, we do not include the SFT part in the pipeline figure.

We sample trajectories produced by agents constructed with top-performing LLMs and state-of-the-art agentic frameworks. Each trajectory consists of a sequence of actions and the final result. The trajectories are scored based on their corresponding resolved results (1 for success and 0 for failure). **② Trajectory Tree Construction.** For each task, we gather all sampled trajectories to construct a trajectory tree. In this tree, identical action nodes (*e.g.*, tool calls and model outputs) are merged, transforming individual agent decision chains into a unified tree structure. Such a trajectory tree can be considered an MCTS tree, where each path from the root to a leaf node refers to a plausible sample of a complete agent execution process. **③ Node Scoring and Critical Action Identification.** We design a scoring mechanism for nodes in the MCTS tree. Each intermediate node is scored based on the success rate of all paths in its subtree. After scoring every node, we identify actions that significantly impact the final outcome, which we define as critical agent actions. **④ Alignment Training with Preference Optimization.** we propose a specialized alignment training approach for agents. Focusing on the identified critical agent actions, we perform fine-grained preference optimization. We expect the aligned LLM to produce more “good” actions and less “bad” ones. More specifically, “good” actions lead to a higher success rate while “bad” ones tend to fail. We align the model with correct agent trajectories, thereby ensuring that it learns to prioritize effective decision-making paths. This approach achieves strong performance with minimal training data and computational cost.

#### 4.1 Trajectory Dataset Collection

To initiate model alignment, we follow existing work [28], where a training dataset with human-validated execution environments tailored for SWE-Bench-style tasks is provided. Based on this, we construct an initial trajectory dataset derived from high-performing models such as GPT-4o [7] and DeepSeek-v3 [17]. These models’ interactions within various software engineering agentic frameworks are collected to represent a series of decision-making steps taken by agents (*i.e.*, invocations of tools).

Each trajectory is paired with its corresponding final result, determined by whether the task succeeds (*i.e.*, passes all test cases). To avoid data leakage, we carefully exclude all repositories included in our evaluation benchmarks (SWE-Bench [12] and HumanEvalFix [24]). Every collected trajectory can then be formulated as below:

$$\text{Trajectory} = \{\langle p, a_1, o_1, a_2, o_2, \dots \rangle, \text{res} \}, \text{ where } \text{res} \in \{1, 0\} \quad (2)$$

where res is a binary value, referring to the resolved result. To ensure data integrity, we perform necessary de-duplication on the constructed trajectories.

#### 4.2 Trajectory Tree Construction

SEAlign builds trajectory trees for the purpose of scoring and gathering nodes with impacts. We observe frequent overlaps in the decision steps across different trajectories. This leads to the idea of merging identical steps to form a hierarchical trajectory tree

**Algorithm 1** Node Scoring and Critical Action Identification.

---

```

1: procedure COMPUTENODESCORES(traj_tree)
2:   scores, critical_actions, critical_pairs  $\leftarrow \{\}, \{\}, \{\}$   $\triangleright$  Init
3:   for each node  $\in$  PostOrderTraverse(traj_tree) do
4:     if node is a leaf then
5:       scores[node]  $\leftarrow 0$  or 1  $\triangleright$  Determined by the result
6:     else
7:       scores[node]  $\leftarrow \frac{\text{Successful path # from node}}{\text{Total path # from node}}$ 
8:     end if
9:     if node has two children with score_diff > 0.5 then
10:      critical_actions  $\leftarrow$  critical_actions  $\cup \{node\}$ 
11:      critical_pair  $\leftarrow$  Paired partial traj with preference
12:      critical_pairs  $\leftarrow$  critical_pairs  $\cup \{\text{critical\_pair}\}$ 
13:    end if
14:  end for
15:  return scores, critical_actions, critical_pairs
16: end procedure

```

---

structure, where all trajectories for a particular task are integrated into a unified structural representation.

As shown in Figure 2, for each task, we traverse the various trajectories and identify common prefixes across them. These shared prefixes serve as decision pathways that are merged into a single branch of the tree. By aligning identical steps across trajectories, we create a tree structure where ① each non-leaf node refers to an individual decision step (e.g., a tool invocation or an action taken by the agent) and ② each leaf node represents the final outcomes of the trajectory (*i.e.*, success or failure of the task).

With the trajectory tree, SEAlign organizes agent action chains into a hierarchical structure, allowing effective and efficient node scoring and critical action analysis across trajectories. A filtration procedure is further introduced to eliminate low-quality trajectories that contribute little to the model’s performance. There are two main filtering rules:

- Trajectories repeating the same actions for multiple steps without progress are eliminated, which often leads to the stuck-in-loop issue;
- Trajectories that do not overlap in any meaningful way with others are removed, which usually suggests irrelevant outliers.

This filtering procedure ensures that the dataset maintains a high standard of quality and relevance, allowing the trajectory tree to focus on representative pathways.

### 4.3 Node Scoring & Critical Action Extraction

After constructing the trajectory trees, SEAlign is allowed to identify critical pathways and actions, which significantly influence the final results. Please refer to Algorithm 1. To achieve so, we develop a **node scoring mechanism** based on the success rates of subtrees emanating from each node. The scoring process is outlined as follows. ① For each leaf node, its score is assigned to 0 or 1 according to the final result. ② The score for a non-leaf node is computed by propagation in a bottom-up manner – it is the fraction of success path numbers over all paths in the node’s subtree. In general, the scores reflect the plausible success rate of all possible paths extending from this node.

**Table 1: Statistics of Trajectories and Trajectory Trees**

	<b>Metric</b>	<b>Value</b>
<b>Collected Trajectory</b>	Instance #	186
	Trajectory # (After Filtration)	3,168
	– Successful Trajectory #	677
	– Wrong Trajectory #	2,491
<b>Trajectory Tree</b>	Avg. Token Len.	~19,868
	Avg. Path Len.	~41
	Critical Pair #	<b>983</b>

With the assistance of node scores, SEAlign identifies critical actions in the trajectory tree. If there is a significant difference in the scores of two non-leaf nodes under the same parent node, it indicates that these two non-leaf nodes correspond to a set of critical actions, meaning they lead to vastly different outcomes (one for a high success rate and the other for a high failure rate). In our experimental setup, the threshold of score differences is set to 0.5. During alignment training, we aim to train code models using these critical action pairs to ensure that the models consistently produce higher score actions – that is, actions associated with higher success rates. Please refer to Table 1 for statistics of the collected trajectory dataset.

### 4.4 Alignment with Preference Optimization

With the collected trajectory dataset and the critical actions, SEAlign aligns open-source LLMs to high-score actions. Based on the trajectory dataset constructed above, and the critical action nodes identified through the trajectory tree scoring mechanism, we perform subsequent **alignment training** on open-source models. The entire post-training process consists of two phases: ① SFT with correct trajectories, and ② Direct Preference Optimization (DPO) with critical action pairs.

**SFT with Correct Trajectories.** Before actual alignment, it is crucial to ensure that the target LLM can generate valid and appropriate trajectories, and therefore SFT is needed. Given a collected correct trajectory data sample ( $(p, a_1, o_1, a_2, o_2, \dots)$ ), the target LLM should learn the actions ( $a_1, a_2, \dots$ ). Thus, during SFT, we mask out the loss for the environment observations ( $o_1, o_2, \dots$ ), and the loss is formulated as:

$$\mathcal{L}_{\text{SFT}} = -\mathbb{E}_{\text{traj} \in \mathcal{D}} \left[ \sum_{t=1}^T \log P(a_t | p, a_1, o_1, \dots, a_{t-1}, o_{t-1}) \right], \quad (3)$$

where  $T$  is the total number of actions in the trajectory, and  $\mathcal{D}$  is the dataset of all correct trajectories.

**Fine-grained DPO with Critical Action Pairs.** A major challenge of SFT is that most trajectories appear similar, with only a few critical actions making a significant difference. After SFT, the target LLM can generate valid actions in trajectories but requires further alignment to produce “good” actions (*i.e.*, those with higher success rates). The target model is fundamentally limited in distinguishing and optimizing for these critical actions. To tackle this challenge, we adopt DPO [30], which introduces a preference-based mechanism

that extends beyond maximizing the likelihood of ground truth data. Previous studies have shown the promise of DPO in reasoning tasks such as mathematics [14, 36, 43]. By utilizing pairs of preferred responses ( $y^+$ ) and dispreferred responses ( $y^-$ ), DPO aligns the model with human preferences, thereby enhancing its effectiveness for tasks that require nuanced decision-making. In SEAlign, we focus on the critical agentic actions. As introduced before, the decision points which may significantly influence the final outcome are collected. Each critical pair contains a correct partial trajectory and a wrong partial trajectory, which share the same action prefix, formulated as below:

$$\text{Critical Pair} = \langle a_i^+, a_i^- \mid p, a_1, o_1, \dots, a_{i-1}, o_{i-1} \rangle \quad (4)$$

where  $\langle p, a_1, o_1, \dots, a_i^+ \rangle$  refers to the preferred “good” partial trajectory and  $\langle p, a_1, o_1, \dots, a_i^- \rangle$  refers to the dispreferred “bad” partial trajectory. Remember that the score difference threshold is 0.5. Under the same partial trajectory condition ( $\langle p, a_1, o_1, \dots, a_{i-1}, o_{i-1} \rangle$ ), the preferred action ( $a_i^+$ ) leads to much higher success rate, making it critical. DPO can effectively help the code model to learn such preference, and the loss is formally defined as below:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{\text{traj} \in \mathcal{D}} \left[ \log \sigma \left( \beta \log \frac{P(a_i^+)}{P_{\text{ref}}(a_i^+)} - \beta \log \frac{P(a_i^-)}{P_{\text{ref}}(a_i^-)} \right) \right] \quad (5)$$

where  $P(\cdot)$  is the policy model’s likelihood for making decisions, and  $P_{\text{ref}}(\cdot)$  is the reference model’s likelihood. Note that all conditions in the formulation are neglected for a better view (e.g.,  $P(a_i^+)$  in Eq. 5 should have been  $P(a_i^+ | p, a_1, o_1, \dots, a_{i-1}, o_{i-1})$ ).

## 5 Experimental Setups

To evaluate the effectiveness of SEAlign, we conduct a comprehensive large-scale study. We aim to answer the following research questions (RQs):

**RQ1: Does SEAlign improve the correctness of the generated code on non-trivial real-world benchmarks?** We apply SEAlign to variants of SWE-Bench, *i.e.*, SWE-Bench-Lite [12] and SWE-Bench-Verified [25]. Both benchmarks assess LLM’s capability to handle non-trivial real-world software engineering tasks, such as bug fixing and new feature implementation.

**RQ2: Does SEAlign improve the agent capability on more general code-related tasks?** According to the motivating example in Section 3.2, LLMs often struggle to follow complex instructions or have trouble in correctly utilizing tools in software engineering agent workflows. To demonstrate that SEAlign enhances agent capability with minimal training overhead, we evaluate its performance on a general code-related benchmark, namely HumanEvalFix [24].

**RQ3: What is the impact of the fine-grained preference optimization method on agent capability in SEAlign?** The data construction and the loss formulation may have an influence upon SEAlign. To demonstrate the effective design of SEAlign, we conduct ablation studies by comparing different training setups, including variations in dataset construction and key components of our training loss formulation.

**RQ4: How does data scaling influence the performance of SEAlign?** Data scaling is a crucial factor for modern LLMs, as it reflects the efficiency of the training schema. We explore the impact

of data scaling on SEAlign by varying the amount of training data. This analysis shows how data size affects the ability of LLMs to handle real-world software engineering tasks.

**RQ5: How does SEAlign improve the user experience in real-world software development scenarios?** In addition to quantitative analysis, we also conduct human evaluations to assess the effectiveness of SEAlign in enhancing user experience. We construct multiple applications using SEAlign, and study improvements in user experience, in order to demonstrate the SEAlign framework’s practical utility.

## 5.1 Benchmarks

We conduct our main experiments upon three benchmarks, *i.e.* SWE-Bench-Lite [12], SWE-Bench-Verified [25], and HumanEvalFix [24].

❶ SWE-Bench-Lite [12] is a subset of SWE-Bench. The original SWE-Bench is a large-scale benchmark assessing the LLMs with thousands real-world GitHub issues, spanning 12 popular Python repositories. SWE-Bench closely simulates real-world software engineering scenarios, where models are tasked with fixing bugs or implementing new features. To better accommodate functional bug-fixing objectives, SWE-Bench-Lite is curated, containing 300 test instances and 23 development instances sampled from SWE-Bench. The instances in SWE-Bench-Lite are more self-contained, focusing on functional bug-fixing tasks. ❷ SWE-Bench-Verified [25] is another variation of SWE-Bench. OpenAI employs human beings to thoroughly validate instances in SWE-Bench, creating a human-validated dataset. SWE-Bench-Verified contains 500 verified test instances in total. ❸ In addition to variants of SWE-Bench, we also evaluate SEAlign on HumanEvalFix [24], which is a general debugging benchmark. The benchmark introduces bugs into each of the 164 HumanEval [3] solutions, and requires the model to correct functions with subtle bugs, accompanied by relevant unit tests.

SWE-Bench-Lite and SWE-Bench-Verified are widely adopted benchmarks for LLM-based software engineering agents. Following prior works, we evaluate SEAlign on these two benchmarks for a consistent and fair comparison. In contrast to the SWE-Bench family, HumanEvalFix is a bit simpler but much more generic, making it a complementary testbed to assess SEAlign’s ability to improve instruction-following capabilities within agentic frameworks. Note that during data processing in the SEAlign pipeline, we exclude all the repositories and code snippets used by these three benchmarks to prevent data contamination.

## 5.2 Metrics

For SWE-Bench family, we employ the resolved rate as the primary evaluation metric, which measures the percentage of successfully resolved problems. In addition to the primary metric, we also include the following metrics for more detailed analysis [33]: ❶ empty patch rate, which estimates the percentage of trajectories where no code in the repository is edited, and ❷ stuck-in-loop rate, which assesses the percentage of trajectories where the agent gets stuck in an agentic loop by repeating the exact same action for 3 rounds. The evaluations of the subject LLMs are limited by either a maximum of 30 interaction turns or the model’s 32k context window length, whichever is reached first.

For HumanEvalFix, we use the pass rate on the provided unit tests as the primary evaluation metric. As a complementary measurement, we also record the invalid patch rate, which indicates the percentage of trajectories in which the generated patch is incomplete or empty. Both metrics evaluate the agent's instruction-following capabilities under SEAlign.

### 5.3 Baselines

We evaluate widely adopted LLMs in the realm of code generation, focusing on Qwen-2.5-Coder-Instruct-7B and Qwen-2.5-Coder-Instruct-14B [10]. These two models are post-trained from Qwen-2.5-Coder using large-scale, million-level code-related datasets with SFT and DPO. They represent the state-of-the-art LLMs, specialized for coding, within their respective parameter size categories.

We adopt the OpenHands agentic framework [33] to construct a training dataset and conduct alignment training for SEAlign. OpenHands is a general-purpose agentic framework, which utilizes the CodeActAgent strategy [32] for decision-making regarding both tool usage and reasoning steps.

### 5.4 Training & Inference Setups

We leverage powerful general LLMs like DeepSeek-v3 to build agents for data construction in SEAlign. The constructed dataset focuses exclusively on Python, given its widespread usage and relevance for software engineering tasks.

The maximum training epoch number is set to 10, and the best-performing checkpoint is selected based on the lowest validation loss. Learning rate is initialized to  $3 \times 10^{-6}$ , equipped with a linear scheduler and a warm-up phase at the start of training. Greedy search is adopted during inference, as it ensures deterministic and consistent results for evaluation.

## 6 Experimental Results

### 6.1 SWE-Bench Family (RQ1)

To answer **RQ1**, we first compare the results of SEAlign with the baseline LLMs. The comparison evaluates the improvements SEAlign achieves over widely used models (*i.e.*, Qwen-2.5-Coder-Instruct-7B and Qwen-2.5-Coder-Instruct-14B) in terms of resolved rates on SWE-Bench-Lite and SWE-Bench-Verified. Detailed results are presented in Table 2.

Results show that SEAlign achieves significant relative improvements over baseline LLMs. The baseline LLMs, including the 14B parameter variant, struggle to effectively utilize the agentic framework to handle SWE-Bench tasks due to the misalignment issue, as discussed in Section 3.2. With alignment achieved through a relatively small training cost across less than a thousand samples, SEAlign successfully resolves the misalignment between existing code models and agentic frameworks. SEAlign significantly enhances the target LLMs' instruction following and tool utilization capabilities within the agentic framework. Notably, SEAlign demonstrates remarkable improvements in addressing issues such as empty patches and stuck-in-loop, highlighting the efficiency of our alignment method in tackling real-world software engineering challenges.

We further compare the performance of SEAlign with other state-of-the-art agent-based methods, including both commercial applications and open-source approaches. As shown in Table 3,

**Table 2: Performance Comparisons on SWE-Bench-Lite and SWE-Bench-Verified.** “Empty”, “Stuck”, “Resolve” denote empty patch, stuck-in-loop, and resolved rates, respectively.  $\Delta$  denotes the relative improvement between the baseline and SEAlign. “Qwen-7B” is an abbreviation for Qwen2.5-Coder-Instruct-7B, and “Qwen-14” follows the similar abbreviation.

Model	Empty(%) ↓	Stuck(%) ↓	Resolve(%) ↑	$\Delta$ (%) ↑
<b>SWE-Bench-Lite    Instance # = 300</b>				
Qwen-7B	53.0	45.3	1.0	-
SEAlign-7B	22.0	24.0	15.0	+1400.0
Qwen-14B	50.3	35.0	3.7	-
SEAlign-14B	17.0	22.0	17.7	+378.3
<b>SWE-Bench-Verified    Instance # = 500</b>				
Qwen-7B	49.4	39.0	0.8	-
SEAlign-7B	25.2	19.2	18.6	+2225.0
Qwen-14B	52.0	27.8	2.8	-
SEAlign-14B	22.8	15.6	21.8	+678.5

SEAlign consistently outperforms many of these state-of-the-art methods in terms of task resolved rates on both SWE-Bench-Lite and SWE-Bench-Verified.

Notably, SEAlign, using the same OpenHands agentic framework and a 14B LLM, achieves the comparable performance of GPT-4o, which is a commercial closed-source application. Considering the intrinsic complexity of SWE-Bench tasks, the results are particularly surprising, highlighting the effectiveness of SEAlign. On the other hand, among listed open-source methods, SEAlign achieves the best results among LLMs with comparable parameter size, only being outperformed by a few existing 72B models. Through this comparison, we highlight the remarkable performance of the proposed method, which is achieved with a small training cost in this highly competitive landscape.

### 6.2 HumanEvalFix (RQ2)

To evaluate whether SEAlign enhances an LLM's ability to follow complex instructions or correctly utilize tools in software engineering agentic workflows, we conduct experiments on HumanEvalFix, which is a relatively simpler but broader and more popular benchmark. The results are listed in Table 4. It is important to note that the dataset constructed by SEAlign does not include any data related to HumanEvalFix. Additionally, the data styles and formats are significantly different between the two. This ensures that there is no risk of data leakage. We would like to elaborate that SEAlign exhibits generalizability and adaptability in **RQ2**.

Results highlight that SEAlign significantly improves task success rates and reduces the invalid patch rate compared to baseline LLMs within the agentic framework. We also evaluate the performance of directly using the model to solve the task without any agentic framework. For the baseline Qwen models, there is a clear performance gap between using and not using the agentic framework. In contrast, our proposed SEAlign demonstrates robustness across both forms of workflow, achieving high performance

**Table 3: Comparisons with State-of-the-Art Agent Methods.**

Method	Model / Params	SWE-Bench	
		Verified (%)	Lite (%)
<b>Closed-source Methods</b>			
SWE-agent [39]	Claude-3 -Opus	7.0	4.3
RAG [12]	GPT-4	2.8	2.7
RepoGraph [27]		-	19.0
OpenHands [33]		-	21.0
AutoCodeRover [46]	GPT-4o	-	30.7
SWE-SynInfer [22]		-	22.0
Agentless [37]		31.8	20.7
Moatless Tools [23]	Claude-3.5	33.6	23.0
OpenHands [33]	-Sonnet	50.8	40.7
<b>Open-source Methods</b>			
SWE-Llama RAG [12]	13B	1.2	1.0
AutoCodeRover [18, 46]	72B	-	9.3
SWE-Gym [28]	7B	10.6	10.0
SWE-Gym [28]	14B	16.4	12.7
SWE-Gym [28]	32B	20.6	15.3
Lingma SWE-GPT [22]	72B	30.2	22.0
SWE-Fixer [38]	72B	32.8	24.7
SEAlign (Ours)	7B	18.6	15.0
SEAlign (Ours)	14B	21.8	17.7

**Table 4: Performance Comparisons on HumanEvalFix. “Invalid” denotes the invalid patch rate within the agentic mode. “Qwen-7B” is an abbreviation for Qwen2.5-Coder-Instruct-7B, and “Qwen-14” follows the similar abbreviation.**

Model	Agent	Pass@1 (%) ↑	Invalid (%) ↓
Qwen-7B	w/o	46.9	-
	w/	18.9	60.4
Qwen-14B	w/o	54.3	-
	w/	31.1	43.3
SEAlign-7B	w/o	46.3	-
	w/	55.5	13.4
SEAlign-14B	w/o	52.4	-
	w/	62.8	10.4

either with or without the agentic framework. Interestingly, we observe that incorporating the agentic framework further enhances accuracy for SEAlign. This improvement can be attributed to the agentic framework’s provision of appropriate tools (such as executor), which facilitate code-related task resolution. These findings demonstrate that a well-aligned LLM for code obtained by SEAlign, in combination with a compatible agentic framework, can achieve improved generalization across a broader range of code-related tasks.

**Table 5: Ablation Studies on SWE-Bench-Lite.**

Method	Empty(%) ↓	Resolve(%) ↑
Qwen-14B (w/o all)	50.3	3.7
SEAlign-14B (w/ all)	17.0	17.7
<b>Ablations on Training Stage</b>		
w/ SFT, w/o Fine-grained DPO	22.0	13.0
w/o SFT, w/ Fine-grained DPO	24.3	10.7
<b>Ablations on Fine-grained Score</b>		
w/o Fine-grained Score	41.3	5.3

**Table 6: Data Scaling for SEAlign on SWE-Bench-Lite. The percentage values indicates the data ratio utilized for alignment training.**

Method	Empty(%) ↓	Resolve(%) ↑
<b>Data Scaling</b>		
Qwen-14B (0%)	50.3	3.7
25%	35.0	9.0
50%	28.7	13.7
75%	22.0	16.0
SEAlign-14B (100%)	17.0	17.7
<b>Iterative DPO in Sec. 7.3</b>		
+ 50% on-policy dataset	17.0	16.3

### 6.3 Ablation Studies (RQ3)

To answer **RQ3**, we evaluate the improvement at each training stage introduced in Section 4.4. Specifically, we analyze the effects of SFT and fine-grained DPO on final performance. Additionally, we assess the impact of SEAlign’s carefully designed components, including the computation of fine-grained scores in our DPO setting, as shown in Table 5.

Results show that the DPO stage can further enhance the model’s instruction-following capabilities within agentic frameworks, building upon the foundation provided by the SFT stage. Both stages are crucial for achieving the final improvements in performance. Besides the training stages, the fine-grained scoring mechanism in our DPO setting plays a critical role. When critical action identification is omitted and DPO is applied directly to positive and negative samples (the entire trajectories that succeed or fail) of the same task, the results degrade significantly. This fine-grained preference optimization contributes substantially to task success rates by aligning LLMs’ critical decision-making at key steps. These alignments on critical actions have a profound impact on the model’s final performance.

### 6.4 Data Scaling Law for SEAlign (RQ4)

To address **RQ4**, we investigate the impact of scaling training data on the performance of SEAlign. We evaluate SEAlign using training data proportions of 25%, 50%, and 75%, and analyze their respective impacts on model performance. As listed in Table 6, training with

more data examples consistently enhances the performance of the target LLM.

However, further scaling is constrained due to limitations such as extended execution time and the resource costs associated with constructing larger-scale datasets. For detailed discussions on time and efficiency, please refer to Section 7.2. In our experiments, we balance performance gains with training costs, ensuring that improvements are achieved cost-effectively. Future work will explore efficient data scaling methods to investigate SEAlign’s performance upper bounds under extreme training budgets.

## 6.5 User Experience (RQ5)

To answer **RQ5**, we follow the application demo settings of OpenHands [33] and manually design five real-world development tasks: ❶ a to-do list web application, ❷ a snake game, ❸ a weather forecasting application, ❹ a Hacker News query application, and ❺ a personalized homepage. For each task, we employ both baseline models (*i.e.*, Qwen-based models) and SEAlign within the OpenHands framework to automate development. Five independent volunteer evaluators, each with one year or more software developing experience, were invited to assess the results from three distinct perspectives: functionality completeness, code quality (*e.g.*, readability and adherence to coding standards), and aesthetic appeal. Additionally, we invited all evaluators to rank the five tasks by difficulty. The ranking results, along with corresponding human evaluation metrics, are summarized in Table 7.

Results in Table 7 show that SEAlign significantly enhances user experience in real-world development scenarios and demonstrates its practical utility. A to-do list case is presented in Figure 3, where Qwen2.5-Coder-Instruct-14B struggles to produce a fully functional application, while SEAlign successfully completes all required features perfectly<sup>2</sup>. Upon further analyzing the agent execution chains, we find that the Qwen2.5 model frequently encounters issues such as incorrect tool invocation and failure to follow instructions. These errors cause the agentic framework to spend most of its time handling exceptions and recovering from erroneous steps, significantly hindering task completion. In contrast, SEAlign addresses these shortcomings effectively. With minimal training overhead, SEAlign achieves strong alignment with real-world software engineering scenarios, greatly improving the usability of open-source models in complex software development tasks and agentic frameworks.

## 7 Discussions & Future Work

### 7.1 Generalization across Agentic Frameworks

Just as experienced human programmers skillfully utilize their preferred Integrated Development Environments (IDEs), LLMs optimized for specific agents can effectively leverage tools to address complex problems. As demonstrated in our experiments in **RQ1**, even relatively small-scale LLMs, when subjected to framework-specific alignment, can achieve a high degree of proficiency in agentic frameworks, thereby unlocking new possibilities for automation and broader applications.

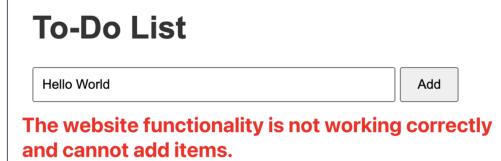
However, a key question remains – can LLMs trained on one specific agentic framework generalize to others? To address this,

<sup>2</sup>Please refer to our open-source reproduction package [2] for other cases.

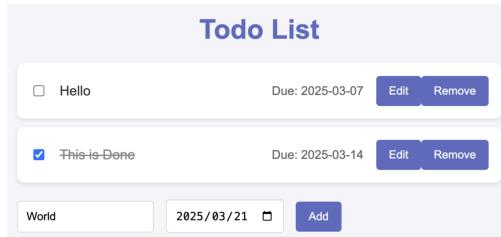
**Table 7: User Experience Evaluation on Real-World Software Engineering Tasks.** Results compare Qwen2.5-Coder-Instruct-14B and SEAlign-14B. Each task is scored by five developers on a scale from 1 to 5, where 5 represents the best, and the average score is calculated. Tasks are ordered from top to bottom based on difficulty, and ranked from simplest to most complex according to evaluator assessments.

Task	Model	Func.	Quality	Aesthetic Appeal
Homepage	Qwen	1.7	3.3	2.7
	SEAlign	3.0	3.7	3.0
Weather	Qwen	1.0	2.0	1.0
	SEAlign	3.0	3.0	2.3
News	Qwen	4.0	4.3	3.3
	SEAlign	4.3	4.3	4.0
To-do	Qwen	1.3	3.0	1.7
	SEAlign	3.7	3.3	4.3
Snake	Qwen	1.0	1.0	1.3
	SEAlign	1.7	3.0	2.3
Avg.	Qwen	1.8	2.7	2.0
	SEAlign	3.1	3.5	3.2

**Qwen2.5-Coder-Instruct-14B:**



**Our 14B:**



**Figure 3: Case study on creating a to-do list web application with OpenHands agentic framework compared with Qwen2.5-Coder-Instruct-14B and SEAlign-14B.**

we conduct experiments to evaluate the generalization capability of SEAlign. Specifically, a model optimized for the OpenHands framework using SEAlign is evaluated on the AutoCodeRover framework [46]. While OpenHands is a flexible and general-purpose agentic framework capable of dynamic tool calls, AutoCodeRover relies on domain-specific, pre-defined workflows that provide the model with different tools at various stages to solve corresponding subproblems. We still employ Qwen2.5-Coder-Instruct-14B and SEAlign-14B to

**Table 8: Alignment Generalization Across OpenHands and AutoCoderRover on SWE-Bench-Lite. SEAlign only use OpenHands framework to construct training dataset.**

Agentic Framework	Model	Resolve Rate (%) ↑
OpenHands	Qwen-14B	3.7
	SEAlign-14B	17.7
	GPT-4o	21.0
AutoCodeRover	Qwen-14B	2.0
	SEAlign-14B	5.7
	GPT-4o	30.7

evaluate their performance. In addition, we also compare with the close-sourced powerful LLM, GPT-4o [7].

The experimental results, presented in Table 8, show that while the aligned LLM, transferred from OpenHands to AutoCodeRover, achieves reasonable performance improvements over the unaligned baseline, its performance remains significantly lower than that of powerful closed-source models such as GPT-4o. Notably, SEAlign-14B achieves comparable performance with GPT-4o on the OpenHands framework, while on the AutoCodeRover framework, the performance gap widens substantially. These observations indicate that while proprietary alignment methods are highly effective within specific agentic workflows, additional training strategies are still essential to enhance cross-framework adaptability.

## 7.2 Overhead for Execution Validation

Although SEAlign requires minimal training overhead and a small number of data samples, one of the main bottlenecks lies in the inefficiency and significant computational overhead associated with **execution validation**. This challenge impacts both the scalability of SEAlign and the feasibility of applying similar alignment training methods to large-scale datasets for software engineering agentic tasks. During our experiments, substantial computational and storage resources are required. ① Approximately 1TB of free disk space is needed for intermediate outputs. ② Using a 16-CPU server, it takes nearly 5 minutes to complete execution validation for a single sample with the official docker platform [12, 28].

As a result, constructing training datasets and running experimental evaluations involved significant time and resource costs. Scaling alignment methods like SEAlign to larger datasets or more environments will therefore require solving the challenge of execution validation efficiency. Optimizing this step, perhaps by employing lightweight performance estimations or other efficiency strategies, can be critical for future improvements in scalability and practicality.

## 7.3 Comparison with Advanced RL Techniques

While our study demonstrates the effectiveness of SEAlign, we have not extensively compared it with advanced reinforcement learning (RL) alignment techniques, such as DeepSeek-R1 [8]. RL-based alignment models, including those trained with methods like GRPO [31], have shown strong performance in reasoning scenarios such as mathematics and competition-level code tasks. These techniques

typically utilize online training environments, leveraging high-quality datasets and carefully designed reward systems. However, this comes at the cost of requiring substantial training resources and computational overhead.

In contrast, offline alignment techniques [30], which is adopted by SEAlign, approximate similar optimization objectives while introducing necessary simplifications. For example, SEAlign leverages pre-generated trajectory datasets and critical node optimization methods instead of conducting resource-intensive online sampling. This design choice reduces computational costs and enables straightforward implementation in constrained environments.

To further explore the potential of advanced RL-based techniques, we perform a simplified experiment using an online iterative DPO approach [29]. We use the additional 50% on-policy dataset based on SEAlign and results are shown in Table 6. However, results suggest that this method shows limited improvement. A detailed comparison of how SEAlign performs relative to advanced RL alignment techniques in both effectiveness and efficiency remains an open question.

## 7.4 Threats to Validity

In this section, we outline potential threats to the validity of our study and address measures taken to mitigate them.

**Threats to Internal Validity.** Threats to internal validity are related to the dataset collection process and hyperparameter settings used during training. In our experiments, the quality and diversity of the trajectories could influence the model’s ability to generalize across different tasks. Additionally, hyperparameter choices such as the learning rate may impact the final performance. While we conduct ablation studies to ensure robustness, there remains a possibility that certain experimental configurations may bias results.

**Threats to External Validity.** External validity threats are related to the tasks and datasets selected for evaluation. Our study relies on benchmarks such as SWE-Bench-Lite, SWE-Bench-Verified, and HumanEvalFix, which simulate real-world software engineering scenarios. However, these benchmarks may not fully capture the diversity and complexity of all possible software engineering tasks. The results may therefore not generalize to other tasks or repositories beyond those included in the benchmarks.

**Threats to Construct Validity.** Construct validity threats are related to the evaluation metrics used to assess model performance. In this work, we primarily rely on resolve rate, which measures the percentage of successfully completed tasks. While this metric provides a direct assessment of task success, it may not fully capture other aspects of model performance, such as efficiency, code quality, or usability in real-world scenarios.

## 8 Conclusion

In this paper, we introduce SEAlign, a novel alignment framework designed to address the gap between existing code model training methods and the requirements of real-world software engineering agents. SEAlign leverages unique characteristics of software development processes, including the identification of critical decision points, fine-grained alignment and preference optimization, enabling models to better handle complex software engineering

tasks within agentic frameworks. Our experimental results demonstrate that SEAlign achieves state-of-the-art performance across multiple benchmarks designed to simulate real-world software engineering challenges. Additionally, we apply SEAlign to real-world software engineering scenarios, successfully automating the creation of several small applications. We are convinced that ongoing advancements in alignment techniques, scalability, and broader applicability will significantly accelerate the realization of fully automated software engineering powered by LLMs.

## References

- [1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
- [2] Authors of this paper. 2025. Available reproduction package of this paper. <https://anonymous.4open.science/r/SWEAlign>.
- [3] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [4] Codeforces. 2009. <https://codeforces.com>. (2009).
- [5] Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2024. Code-Optimise: Self-Generated Preference Data for Correctness and Efficiency. *CoRR abs/2406.12502* (2024). doi:10.48550/ARXIV.2406.12502 arXiv:2406.12502
- [6] GPT-4. 2023. <https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo>. OpenAI (2023).
- [7] GPT-4o. 2024. <https://openai.com/index/hello-gpt-4o/>. OpenAI (2024).
- [8] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
- [9] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
- [10] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. *CoRR abs/2409.12186* (2024). doi:10.48550/ARXIV.2409.12186 arXiv:2409.12186
- [11] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).
- [12] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*.
- [13] Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip H. S. Torr, Salman Khan, and Fahad Shahbaz Khan. 2025. LLM Post-Training: A Deep Dive into Reasoning Large Language Models. *arXiv:2502.21321 [cs.CL]* <https://arxiv.org/abs/2502.21321>
- [14] Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. 2024. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *arXiv preprint arXiv:2406.18629* (2024).
- [15] LeetCode. 2018. <https://leetcode.com>. (2018).
- [16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustín Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
- [17] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
- [18] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. 2024. Codexgraph: Bridging large language models and code repositories via code graph databases. *arXiv preprint arXiv:2408.03910* (2024).
- [19] Llama. 2023. <https://huggingface.co/meta-llama/Llama-2-70b-chat>. (2023).
- [20] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).
- [21] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).
- [22] Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622* (2024).
- [23] Moatless. 2024. <https://github.com/aorwall/moatless-tools>. (2024).
- [24] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayan Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. OctoPack: Instruction Tuning Code Large Language Models. *arXiv preprint arXiv:2308.07124* (2023).
- [25] OpenAI. 2024. <https://openai.com/index/introducing-swe-bench-verified>. (2024).
- [26] OpenAI-03. 2024. <https://openai.com/index/openai-o3-mini>. (2024).
- [27] Siru Ouyang, Wenhai Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2024. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. *arXiv preprint arXiv:2410.14684* (2024).
- [28] Jiayi Pan, Xingya Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Zihye Zhang. 2024. Training Software Engineering Agents and Verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139* (2024).
- [29] Richard Yuanzhe Pang, Weizhe Yuan, He He, Kyunghyun Cho, Sainbayar Sukhbaatar, and Jason Weston. 2024. Iterative reasoning preference optimization. *Advances in Neural Information Processing Systems* 37 (2024), 116617–116637.
- [30] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2024).
- [31] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Huawei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).
- [32] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.
- [33] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.
- [34] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560* (2022).
- [35] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).
- [36] Yue Wu, Zhiqing Sun, Huizhuo Yuan, Kaixuan Ji, Yiming Yang, and Quanquan Gu. 2024. Self-play preference optimization for language model alignment. *arXiv preprint arXiv:2405.00675* (2024).
- [37] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
- [38] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution. *arXiv preprint arXiv:2501.05040* (2025).
- [39] 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. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
- [40] Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. 2025. LIMO: Less is More for Reasoning. *arXiv preprint arXiv:2502.03387* (2025).
- [41] Dylan Zhang, Shizhe Diao, Xueyan Zou, and Hao Peng. 2024. PLUM: Preference Learning Plus Test Cases Yields Better Code Language Models. *CoRR abs/2406.06887* (2024). doi:10.48550/ARXIV.2406.06887 arXiv:2406.06887
- [42] Huangzhao Zhang, Kechi Zhang, Zhuo Liu, Jia Li, Jia Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, et al. 2024. Deep learning for code generation: a survey. *Science China Information Sciences* 67, 9 (2024), 191101.
- [43] Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024. CodeDPO: Aligning Code Models with Self Generated and Verified Source Code. *CoRR abs/2410.05605* (2024). doi:10.48550/ARXIV.2410.05605
- [44] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 13643–13658.
- [45] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhua Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. 2023. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792* (2023).

- [46] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009