SWE-Swiss: A Multi-Task Fine-Tuning and RL Recipe for High-Performance Issue Resolution

Zhenyu He[1*], **Qingping Yang**[2], **Wei Shen**[2], **Xiaojian Zhong**[2], **Kechi Zhang**[1], **Chenxin An**[3], **Wenlei Shi**[2], **Tianle Cai**[2], **Di He**[1†], **Jiaze Chen**[2*], **Jingjing Xu**[2*]

[1] State Key Laboratory of General Artificial Intelligence, Peking University, Beijing, China

[2] ByteDance Seed

[3] The University of Hong Kong

Abstract

Automated software engineering, particularly resolving real-world issues on benchmarks like SWE-bench, remains a significant challenge for Large Language Models (LLMs). To address this, we introduce SWE-Swiss, a two-phase training recipe that systematically develops these capabilities. Our approach first decomposes issue resolution into three core skills: Localization, Repair, and Unit Test Generation. In the first phase, we perform multi-task Supervised Fine-Tuning (SFT) on three new, meticulously curated datasets to build a versatile foundation. The second phase applies targeted Reinforcement Learning (RL), using direct feedback from test execution to boost the critical skill of code repair. The resulting model, SWE-Swiss-32B, establishes a new state-of-the-art for open-source models in its size class, achieving a 60.2% score on the SWE-bench Verified benchmark and placing it in the same top-tier performance bracket as much larger models. Finally, we show that despite its specialized training, SWE-Swiss-32B demonstrates strong generalization to other common LLM benchmarks. To accelerate research in the community, we are open-sourcing the models and our complete training datasets.

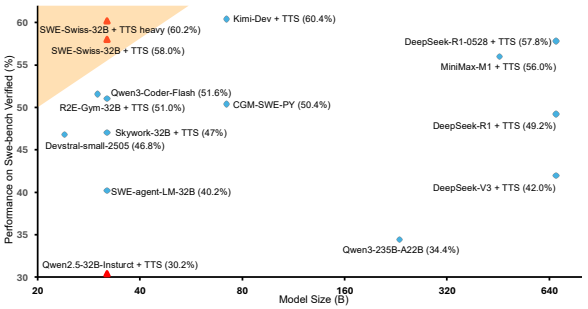

Figure 1: Performance and model size comparison on SWE-bench Verified. Our 32B model, SWE-Swiss-32B, achieves a top-tier score of 60.2% with test-time scaling. For Qwen2.5-32B-Insturct, the score is obtained via the Agentless framework. The scores for other models are reported from their respective blogs or papers.

1 Introduction

Recent advancements in autonomous software engineering (SWE) have shown the growing potential of Large Language Models (LLMs) to address complex, real-world problems like resolving GitHub issues (Yang et al., 2024; Wang et al., 2024a; Xia et al., 2024). Progress in this domain is systematically evaluated on benchmarks such as SWE-bench (Jimenez et al., 2023). A persistent trend, however, is the performance gap between leading proprietary models and their open-source counterparts. DeepSeek-R1 stands as a significant outlier, being an open-weight model that has proven capable of reaching performance levels comparable to the best proprietary models. Nevertheless, the specific "training recipes"—from the composition of supervised fine-tuning (SFT) and reinforcement learning (RL) training data to the precise training techniques and hyperparameters—that enable this top-tier performance remain opaque, hindering reproducible research and further progress within the open-source community.

This lack of a clear, replicable methodology highlights a foundational challenge. Frameworks like Agentless (Xia et al., 2024) have demonstrated the potential of breaking this challenge down into a structured workflow that mimics a human developer: first locating the problem files, then generating a fix, and finally validating the fix. While this paradigm is promising, it raises a critical question: what is the most effective way to train a model to excel at each stage of this process? To address this, we introduce the SWE-Swiss recipe, a strategy centered on decomposing the complex task of fixing a GitHub issue into a curriculum of core, train-

*Project Leads. Work done during Zhenyu's internship at ByteDance.

†Corresponding to Di He <dihe@pku.edu.cn>.

Findings of the Association for Computational Linguistics: ACL 2026, pages 16102–16114

able competencies. This comprehensive, two-phase training strategy is designed to create a powerful and versatile issue-resolution model by explicitly training it on three fundamental skills: (1) Localization: Pinpointing the exact files that require modification. (2) Repair: Generating the correct code patch to resolve the identified issue. (3) Unit Test Generation: Creating new, relevant tests to validate the proposed fix.

Our approach begins by creating high-quality datasets for each of these three tasks through a meticulous process of generation and verified rejection sampling, where only outputs that pass validation in real test environments are retained. In the first phase of our recipe, we perform multi-task Supervised Fine-Tuning (SFT) on a base model using this curated data. This embeds a broad, foundational understanding of the entire issue-resolution workflow into a single, cohesive model. In the second phase, we sharpen the most critical and challenging skill, Repair, through a targeted Reinforcement Learning (RL) phase. Using the GRPO algorithm (Shao et al., 2024) and a curriculum learning strategy, the model receives direct binary rewards based on whether its generated patches pass the actual repository test suites, allowing it to learn from environmental feedback.

Following this recipe, we train SWE-Swiss-32B, a model based on Qwen2.5-32B-Instruct (Qwen et al., 2025). Our experiments demonstrate the effectiveness of this approach, as SWE-Swiss-32B achieves a score of 60.2% on the SWE-bench Verified benchmark. This result establishes a new state-of-the-art for open-source models in its size class and places it in the same top-tier performance bracket as much larger models (e.g., DeepSeek-R1-0528 (DeepSeek, 2025)). Furthermore, to address the limitations of standard self-consistency in code generation, we introduce an Enhanced Self-consistency method that incorporates similarity-based consistency to better identify the majority of code answers.

Our primary contributions are three-fold:

1. We propose the SWE-Swiss recipe, a multi-task SFT and two-stage RL training strategy for creating high-performance issue resolution models.

2. We construct and release three high-quality, validated datasets for the core tasks of Localization, Repair, and Unit Test Generation to facilitate future research.

3. We release SWE-Swiss-32B, a state-of-the-art 32B open-source model for software engineering that is competitive with top-performing proprietary models.

2 Related Work

2.1 System design for Software Engineering

The application of Large Language Models (LLMs) to software engineering has evolved rapidly from function-level code generation (Chen et al., 2021; Austin et al., 2021) to complex repository-level problem solving (Ding et al., 2023; Liu et al., 2023c; Jimenez et al., 2023). More recently, the field has shifted towards creating autonomous systems capable of handling entire software engineering tasks, such as resolving GitHub issues. Benchmarks like SWE-bench (Jimenez et al., 2023; Yang et al., 2025a; Zan et al., 2025) have been instrumental in driving and measuring progress in this area by providing a standardized evaluation suite based on real-world software repositories. To tackle these complex tasks, agent systems (Yao et al., 2023) like SWE-agent (Yang et al., 2024) and OpenHands (Wang et al., 2024a) have emerged, which equip LLMs with tools to interact with a file system, execute code, and browse the web. A parallel line of works design workflows (Xia et al., 2024; Orwall, 2024; Ma et al., 2024; Tao et al., 2025) to tackle the problem. For example, Agentless (Xia et al., 2024) focuses on structuring the problem into a multi-stage work, typically involving identifying relevant files (localization), generating a code patch (repair), and validating the code patch (unit test). While these agent-based and workflow approaches have proven effective, they often rely on proprietary models to achieve top-tier performance. Our work proposes a full training recipe to achieve top-tier performance using open-weight models.

2.2 Training LLMs for Software Engineering

Beyond frameworks, a critical aspect of improving SWE models is the training process itself. Current training methodologies can be broadly categorized into two groups: those that use static, execution-free data and those that rely on execution-based verification.

One category of methods relies on static data from GitHub, thus forgoing live execution from docker environments. For instance, CGM-SWE (Tao et al., 2025) constructs a repository code graph and directly uses the ground-truth patch for

Supervised Fine-Tuning (SFT). Others apply more sophisticated filtering or data augmentation; SWE-fixer (Xie et al., 2025) uses a proprietary model (GPT-4o) to generate Chain-of-Thought demonstrations for the oracle patch before SFT, while Lingma SWE-GPT (Ma et al., 2024) and SWE-RL (Wei et al., 2025) use a similarity-based metric to heuristically compare generated patches against the oracle patch, using the result for rejection sampling or as an RL reward signal, respectively.

In contrast, the second line of approaches involves creating high-fidelity training data through execution-based verification. Works such as SWE-Gym (Pan et al., 2024a), R2E-Gym (Jain et al., 2025b), and Skywork-SWE (Zeng et al., 2025a) collect real-world issues from GitHub and build corresponding Docker environments to enable unit test execution. Similarly, SWE-Smith (Yang et al., 2025b) and SWE-Mirror (Wang et al., 2025b) synthesize new issues within these environments. These works typically employ a distillation strategy: a powerful proprietary model (e.g., Claude 3.7 (Anthropic, 2025)) is used within an agent scaffold (e.g., OpenHands (Wang et al., 2024a)) to generate solution trajectories. Only trajectories that successfully pass the unit tests are collected and used to fine-tune a smaller, open-weight student model.

2.3 Reinforcement Learning from Verifiable Rewards

Reinforcement Learning (RL) has become a cornerstone for aligning LLM behavior with desired outcomes. While Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022) is a widely used technique, it relies on costly and subjective human annotations. An increasingly popular alternative is to source feedback directly from a verifiable, objective environment, a paradigm referred to as Reinforcement Learning from Verifiable Rewards (Lambert et al., 2024; Luong et al., 2024). This approach has shown great promise in domains where correctness can be automatically determined. For instance, in theorem proving, models have been trained using feedback from formal proof assistants like Lean (Moura and Ullrich, 2021; Xin et al., 2024); in mathematical reasoning, rewards are derived from the correctness of the final answer when compared against an oracle (Lambert et al., 2024; Luong et al., 2024; Shao et al., 2024; Guo et al., 2025; Luo et al., 2025c); and in competitive programming, models are rewarded

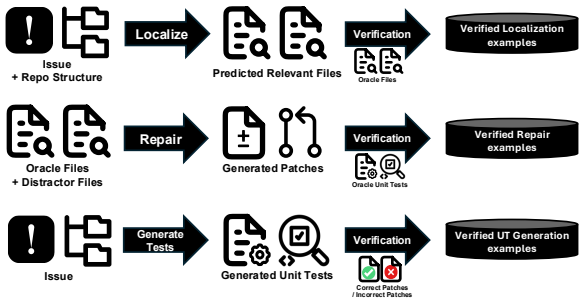

Figure 2: Data curation via rejection sampling. This diagram illustrates the three parallel pipelines used to generate and validate high-quality training data for the Localization, Repair, and Unit Test Generation tasks, respectively.

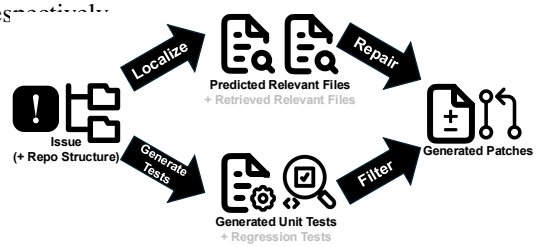

Figure 3: Illustration of the LLM-driven patch generation process, which is enabled by three core abilities: localization, repair and unit test generation.

for generating code that successfully passes a set of predefined test cases (Le et al., 2022; Liu et al., 2023a; Guo et al., 2025; Luo et al., 2025b). This principle extends naturally to software engineering, where the test suite of a repository serves as a powerful, built-in verifier. Several concurrent works (Cao et al., 2025; Da et al., 2025; kim, 2025; Luo et al., 2025a; Golubev et al., 2025) also leverage repository test suites as a reward source for RL. Our work contributes to this emerging line of research by integrating RL as the second phase of our training recipe.

3 Methods: The SWE-Swiss Recipe: Methodology

3.1 Decomposing Issue Resolution: A Three-Skill Curriculum

Our SWE-Swiss recipe is founded on a principled decomposition of this complex process. We hypothesize that by breaking it down into a curriculum of three fundamental and synergistic skills—**Localization**, **Repair**, and **Unit Test Generation**—we can train more robust models. This decomposition provides a structured approach to model training, allowing for targeted data collection and skill development for each sub-task. Figure 3 illustrates how these skills synergize within

the overall patch generation and validation process.

3.2 Data Curation via Verified Rejection Sampling

We utilize rejection sampling (Zelikman et al., 2022) as our core data curation methodology. This strategy acts as a powerful filter, ensuring that every data point in our training curriculum corresponds to a verifiably successful outcome, thereby maximizing the quality of the learning signal. The process involves two steps: first, using a powerful generator LLM (DeepSeek-R1-0528 (Guo et al., 2025)) to produce a large pool of candidate data points for each skill, and second, applying a strict, execution-based oracle (e.g., a test suite) to filter for only successful examples. The data curation pipeline is illustrated in Figure 2.

Localization: To teach the model to identify the correct files to edit, we used training sets from SWE-bench (Jimenez et al., 2023) and SWE-Gym-Raw (Pan et al., 2024a), which contain GitHub issues and their corresponding commit IDs. By analyzing the GitHub repositories, we identify the exact set of files modified in the ground-truth commit for each issue. The generator model is then prompted with an issue description and repository file structure to predict which files require modification. To prevent data contamination, instances from repositories present in the SWE-bench test set are excluded. A prediction is accepted into the dataset only if it meets two strict curation criteria: the number of predicted files is five or fewer, and the recall is 1.0 (meaning all ground-truth modified files are correctly identified). This process yields a targeted dataset of 5,302 high-quality localization examples.

Repair: To train the core task of generating a code patch, we utilize 12,097 issues and corresponding executable Docker environments from SWE-gym (Pan et al., 2024a) and SWE-smith (Yang et al., 2025b), which yields 12,087 prompts. The generator model is prompted with the issue, the oracle files (the ground-truth modified files), and plausible but incorrect "distractor" files (see Appendix A for more details). The curation criterion is stringent: a generated patch is accepted as a valid training instance only if it successfully passes the unit tests within the provided Docker environment. This rigorous, test-driven approach compiles a dataset of 3,935 successful Repair examples.

Unit Test Generation: To enable the system to validate its own fixes, we prompt the generator model with 12,097 issues from SWE-gym (Pan et al., 2024a) and SWE-smith (Yang et al., 2025b) to create new unit tests. The curation criteria require that a generated test's execution behavior be perfectly consistent with the ground-truth tests. Specifically, the generated test must pass for known correct patches and fail for known incorrect patches (collected during our RL experiments), mirroring the oracle's behavior. This process produces a dataset of 1,017 validated examples.

Together, these three curated datasets form a comprehensive curriculum designed to teach the fundamental competencies of software engineering.

3.3 Two-Phase Training Strategy

The curated data is used in a two-phase training process (Team et al., 2025) (depicted in Figure 4), designed to first build a broad foundation of skills through SFT, and then sharpen repair capabilities with RL.

3.3.1 Supervised Fine-Tuning

In the first phase, we perform supervised fine-tuning on the Qwen2.5-32B-Instruct (Qwen et al., 2025) model, combining all 10,254 examples from the Localization, Repair, and Unit Test tasks. This multi-task learning approach embeds the three distinct skills into a single, cohesive model, providing it with a comprehensive understanding of the entire issue-resolution workflow. This foundational phase alone is effective, achieving a 36.0% score on SWE-bench Verified without test-time scaling. We name this SFTed model SWE-Swiss-SFT.

3.3.2 Reinforcement Learning from Verifiable Rewards

Building on this strong SFT foundation, we initiate a second phase focused on improving the most critical task: Repair. Given the same prompts from our repair data curation, the model generates patches itself. Leveraging Docker environments, the patches are then immediately evaluated against the repository's actual test suite, providing real-world feedback on the correctness of its generated code. The reward function is a direct binary signal based on test outcomes:

$$R_{\text{repair}} = \begin{cases} 1 & \text{if patch passes unit tests} \\ -1 & \text{otherwise} \end{cases} \quad (1)$$

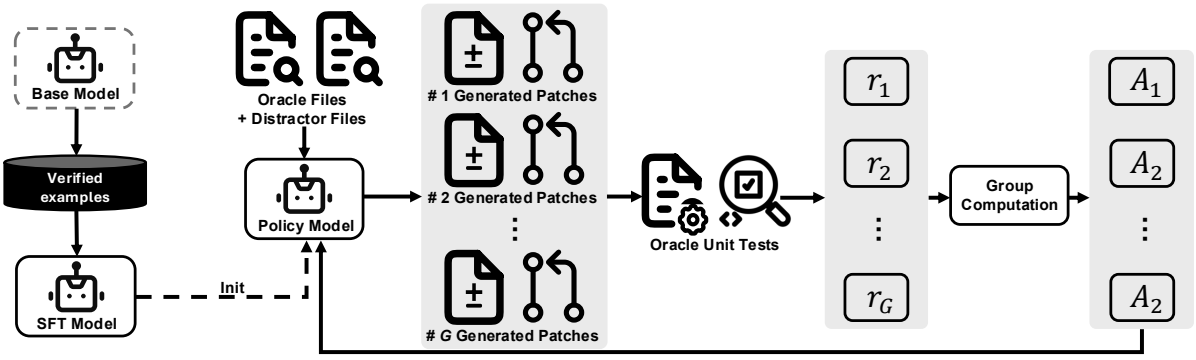

Figure 4: Overview of the two-phase training strategy. A base model is first fine-tuned via SFT on the curated multi-task dataset. The resulting SFT model then serves as the initial policy for a targeted RL phase, which iteratively improves the model's repair capabilities based on direct feedback from test execution environments.

We utilize the GRPO (Shao et al., 2024) algorithm and adopt techniques including no KL loss, clip higher, dynamic sampling, and token-level policy gradient loss from DAPO (Yu et al., 2025):

$$\mathcal{J}_{\text{DAPO}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)} \left[\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\min\Big(r_{i,t}(\theta)\hat{A}_{i,t}, \text{clip}\Big(r_{i,t}(\theta), 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}}\Big)\hat{A}_{i,t}\Big)\right], \tag{2}$$

where q refers to the prompt sampled from the dataset $\mathcal{D}$. $\{o_i\}_{i=1}^{G}$ refers to a group of G output sequences generated by the policy. $r_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}$ is the per-token importance sampling ratio and $\varepsilon_{low}, \varepsilon_{high}$ define the clipping range. $\hat{A}_{i,t}$ represent the advantage estimate, computed based on the relative rewards of outputs within each group $\frac{R_i - \text{mean}(\{R_i\}_{i=1}^{G})}{\text{std}(\{R_i\}_{i=1}^{G})}$.

Inspired by the principles of curriculum learning, as demonstrated in POLARIS (An et al., 2025), we implement a two-stage RL training process. This ensures the model focuses its learning capacity on problems where it stands to improve the most:

- **Stage One:** The model first trains on the complete set of prompts for 200 steps to build broad competency across all problems.
- **Stage Two:** This stage involves performance-based pruning of the training data, where we remove prompts that the model has already achieved over 90% accuracy (we also tried 75% but performed poorer). This creates a more challenging curriculum for a second, concentrated training phase, pushing the model to overcome its most persistent weaknesses. This stage consists of 90 steps of RL training.

The impact of this targeted RL phase is profound. On the SWE-bench Verified benchmark, the model's score jumps from a post-SFT baseline of 36.0% to 45.0% based on single-patch generation alone, surpassing DeepSeek-R1-0528 (43.2%) under the same evaluation setting. When combined with multi-patch test-time scaling methods, the final performance is further boosted to an impressive 60.2%.

4 Experiments

4.1 Experimental Setup

Training Configurations. *SFT Phase:* We fine-tune SWE-Swiss-SFT-32B from the Qwen2.5-32B-Instruct model (Qwen et al., 2025) for 2 epochs using the curated dataset from Section 3.2. This phase uses a 100k context window, a dynamic batch size averaging around 60, a learning rate of 2×10^{-5}, and a weight decay of 0.1. *RL Phase:* The SWE-Swiss-32B model is initialized from SWE-Swiss-SFT-32B and trained using a two-stage curriculum: an initial 200 steps on the full dataset, followed by 90 steps on the pruned, more challenging subset. For rollouts, we use a prompt batch size of 128 and sample 8 responses per prompt. For training, we use a mini-batch size of 128, performing 8 gradient updates per rollout step. The learning rate is set to 1×10^{-6}. Maximum prompt and response lengths are 65,536 and 32,768 tokens, respectively. The clipping parameters ε_{low}, ε_{high} are set to 0.2 and 0.28. For both training phases, we use the AdamW optimizer (Kingma and Ba, 2014; Loshchilov and Hutter, 2017).

Table 1: Model performance across different approaches on SWE-Bench Verified. Rows corresponding to Qwen-2.5-32B-Instruct are highlighted in a light cyan background. ⋆ Results are based on our own evaluation; all other results are taken from their respective blogs or papers.

Approach	#Params	Framework	Model	Resolve Rate (%)
Proprietary Models				
Agentless-1.5 (Xia et al., 2024) + GPT-4o (OpenAI, 2024a)	–	Agentless	GPT-4o	38.8
o1 (OpenAI, 2024b)	–	Agentless	o1	48.9
GPT-4.1 (OpenAI, 2025a)	–	Internal scaffold	OpenAI-GPT-4.1	54.6
OpenHands + o3-mini (OpenAI, 2025c)	–	OpenHands	o3-mini	43.7
OpenHands + o4-mini (OpenAI, 2025b)	–	OpenHands	o4-mini	56.8
o3-mini (OpenAI, 2025c)	–	Internal scaffold	o3-mini	49.3
o4-mini (OpenAI, 2025c)	–	Internal scaffold	o4-mini	**68.1**
OpenHands + Claude-3.7-Sonnet (Anthropic, 2025)	–	OpenHands	Claude-3.7-Sonnet	**60.6**
Claude-3.7-Sonnet (Anthropic, 2025)	–	Internal scaffold	Claude-3.7-Sonnet	**70.3**
OpenHands + Claude-4-Sonnet (Anthropic, 2025)	–	OpenHands	Claude-4-Sonnet	**70.4**
Claude-4-Sonnet (Anthropic, 2025)	–	Internal scaffold	Claude-4-Sonnet	**72.7**
Open-source Models				
Devstral-Small-2505 (Mistral AI, 2025)	24B	OpenHands	Mistral-Small-3.1-24B-Base-2503	46.8
SWE-Gym-32B (Pan et al., 2024b)	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	32.0
Qwen-2.5-32B-Instruct⋆ (Qwen et al., 2025)	32B	Agentless	Qwen-2.5-32B-Instruct	30.2
Qwen-2.5-32B-Instruct⋆ (Qwen et al., 2025)	32B	Agentless-Ours	Qwen-2.5-32B-Instruct	22.2
SWE-Dev-32B (Wang et al., 2025a)	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	36.6
SWE-agent-LM-32B (Yang et al., 2025b)	32B	SWE-Agent	Qwen-2.5-Coder-32B-Instruct	40.2
Skywork-SWE-32B (Zeng et al., 2025b)	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	47.0
R2E-Gym-32B (Zeng et al., 2025b)	32B	OpenHands	Qwen-2.5-Coder-32B-Instruct	51.0
SWESynInfer + Lingma SWE-GPT 72B (Ma et al., 2024)	72B	SWESynInfer	Qwen-2.5-72B-Instruct	30.2
SWE-Fixer-72B (Xie et al., 2025)	72B	RAG	Qwen-2.5-72B-Instruct	30.2
CGM-SWE-PY (Tao et al., 2025)	72B	Agentless + graph RAG	Qwen-2.5-72B-Instruct	30.2
Agentless-1.5 + DeepSeek-V3 (Liu et al., 2024)	671B	Agentless	DeepSeek-V3	42.0
Agentless-1.5 + DeepSeek-R1 (Guo et al., 2025)	671B	Agentless	DeepSeek-R1	49.2
Agentless-1.5 + DeepSeek-R1-0528 (DeepSeek, 2025)	671B	Agentless	DeepSeek-R1-0528	57.8
Ours				
SWE-Swiss-32B ⋆	32B	Agentless-Ours	Qwen-2.5-32B-Instruct	**58.0**
SWE-Swiss-32B ⋆	32B	Agentless-Ours + TTS heavy	Qwen-2.5-32B-Instruct	**60.2**

Benchmark. All evaluations are conducted on the SWE-bench Verified benchmark (Jimenez et al., 2023; OpenAI, 2024c), a high-quality subset of 500 real-world software issues that have been manually confirmed as solvable by human engineers.

Evaluation Pipeline. Our evaluation pipeline is adapted from Agentless with the following modifications: 1) Simplified localization: Inspired by Agentless Mini (Wei et al., 2025), we streamline the localization stage. Instead of the original hierarchical process (file → class/function → final location), our model is only required to identify the relevant files for a given issue. 2) Enriched prompts for unit test generation: During the unit test generation phase, we enhance the model's input prompt. In addition to the original issue description, we directly incorporate the full contents of the relevant files identified in the localization stage. 3) Enhanced self-consistency for patch selection: For the final patch selection, we replace the standard self-consistency method (majority vote) with our own enhanced self-consistency algorithm (detailed in Section 4.2). This allows for a more nuanced selection from the generated patches. Maximum prompt and response lengths are 65,536 and 32,768 tokens, respectively. More details can be found in Appendix B.

4.2 Enhanced Self-consistency for Code Generation

Vanilla self-consistency methods (Wang et al., 2022), which rely on a majority vote over exact-match outputs, are highly effective in domains with concise, canonical answers, such as mathematics. This approach, however, is less suitable for code generation. In this domain, semantically equivalent solutions can have numerous character-level variations—such as different variable names, comments, or whitespace—which prevent otherwise identical solutions from being grouped together and diluting the effectiveness of a simple majority vote. To overcome this limitation, we propose Enhanced Self-consistency, a scoring mechanism that augments the standard exact-match approach with a similarity-based score. This hybrid method allows us to identify not only the most frequent solution but also the most representative solution within a cluster of similar candidates. The final score for any given candidate code c is the sum of these two components:

$$\text{Score}(c) = \text{Score}_{\text{EM}}(c) + \text{Score}_{\text{Sim}}(c) \quad (3)$$

The first component, the Exact-Match Score (Score_{EM}) , is the standard self-consistency measure. It is simply the frequency (raw count) of a specific candidate code appearing in the complete set of generated outputs. This score rewards the solution that is generated most often in its exact form. The second, more nuanced component is the Similarity-Enhanced Score ($\text{Score}_{\text{Sim}}$). This score quantifies how well a candidate represents a cluster of similar, plausible solutions. To calculate it, we first compute the pairwise similarity between all generated code snippets using Levenshtein Distance (Lcvenshtcin, 1966) as the metric (we leave choosing other similarity functions as future work). Then, for each candidate, we identify its top-k most similar neighbors and average their similarity scores. This final average rewards candidates that exist within a dense neighborhood of plausible variations, even if they are not the most frequent exact match. In our experimental setup, the value of k is set to half of the total number of generated patches. The candidate with the highest combined score is selected as the final answer.

4.3 Main Results

The effectiveness of our SWE-Swiss training recipe is demonstrated by the performance of SWE-Swiss-32B on the SWE-bench Verified benchmark. As detailed in Table 1, our 32B parameter model achieves a final resolve rate of 60.2%. This result establishes a new state-of-the-art for open-source models in the 32B class and positions SWE-Swiss-32B as a top-performing model overall. Notably, our model surpasses the performance of significantly larger open-source models. For instance, it outperforms the 671B parameter DeepSeek-R1-0528 (DeepSeek, 2025), which scored 57.8%. Furthermore, SWE-Swiss-32B is competitive with some leading proprietary models, achieving a score comparable to Claude-3.7-Sonnet (Anthropic, 2025) (60.6%) when used with the OpenHands framework and exceeding others like o3-mini (OpenAI, 2025c) (49.3%) and GPT-4.1 (OpenAI, 2025a) (54.6%). However, our model still lags behind the best coding model Claude-4-Sonnet (Anthropic, 2025).

The progression of the model's capability is illustrated in Figure 5, which shows that the two-stage RL phase improves the single-patch generation performance from a post-SFT score of 36.0% to 45.0%.

Table 2: Ablation study of training components. R, L, and UT stand for repair, localization, and unit test generation data, respectively.

Config.	Resolve (%)	Gain (%)
Qwen2.5-32B-Instruct	11.8	-
SFT Phase		
+ R	33.1	+21.3
+ R + L	34.4	+1.3
+ R + L + UT	36.0	+1.6
RL Phase		
+ R	45.0	+9.0

The final peak score of 60.2% is achieved by leveraging our Enhanced Self-consistency method at test time, scaling the number of generated candidate patches to 120, as shown in Figure 6. The overall landscape of model performance versus size is visualized in Figure 1, where SWE-Swiss-32B is clearly positioned in the top tier for performance while having a much smaller parameter count than many of its competitors.

4.4 Ablation Study

The ablation study in Table 2 systematically quantifies the performance gains from each stage of the SWE-Swiss training recipe. Starting with the base Qwen2.5-32B-Instruct model, the initial Supervised Fine-Tuning (SFT) phase, which incorporates curated data for Repair, Localization, and Unit Test Generation, provides a substantial boost, raising the model's score to 36.0%. A subsequent, targeted Reinforcement Learning (RL) phase adds another significant gain of 9.0 percentage points, pushing the single-patch performance to 45.0%. The findings confirm that both the SFT and RL phases are critical, complementary components that contribute to the overall success of the SWE-Swiss recipe.

4.5 Generalization on Other Tasks

To assess whether the skills acquired during training on software engineering tasks translate to other domains, we evaluate the performance of SWE-Swiss-SFT-32B and SWE-Swiss-32B on a variety of general, mathematical, and coding benchmarks, comparing them against the base Qwen2.5-32B-Instruct model. The tasks include MMLU (Hendrycks et al., 2021), MMLU-pro (Wang et al., 2024b), GPQA-Diamond (Rein et al., 2024), SuperGPQA (Du et al., 2025), AIME2024 (MAA, 2024), AIME2025 (MAA,

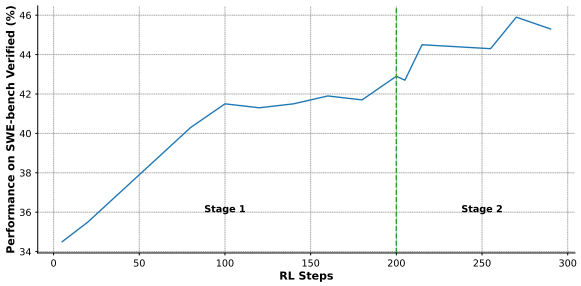

Figure 5: Performance improvement of single patch generation on SWE-bench Verified during the two-stage Reinforcement Learning phase.

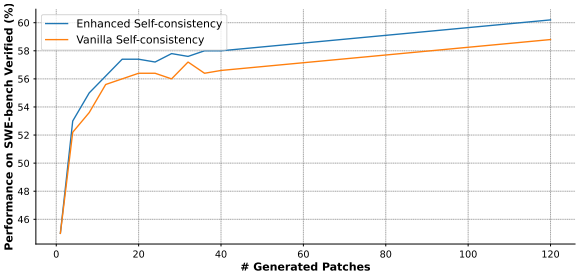

Figure 6: Test-time scaling performance of SWE-Swiss-32B. The score improves as the number of generated candidate patches increases. The model achieves its peak performance of 60.2% with 120 generated patches.

Table 3: Performance of SWE-Swiss-32B and SWE-Swiss-SFT-32B beyond SWE-bench compared with Qwen2.5-32B-Instruct.

Datasets	**Qwen2.5-32B-Instruct**	**SWE-Swiss-SFT-32B**	**SWE-Swiss-32B**
	General Tasks		
MMLU	82.0	81.4	**84.3**
MMLU-pro	70.2	73.5	**75.7**
GPQA-Diamond	44.7	48.8	**56.1**
SuperGPQA	39.1	36.3	**43.0**
	Math Tasks		
AIME2024	15.3	22.7	**26.7**
AIME2025	11.3	18.3	**19.0**
ZebraLogic	25.4	48.6	**51.1**
	Coding Tasks		
MBPP+	74.9	72.8	**79.4**
BigCodeBench (C)	53.4	52.3	**55.9**
BigCodeBench-Hard (C)	23.6	27.0	**27.7**
Aider-Polyglot	8.9	24.0	**24.9**
LiveCodeBench_250201_250501	30.5	32.1	**35.1**

2025), ZebraLogic (Lin et al., 2025), MBPP+ (Liu et al., 2023b), BigCodeBench (Zhuo et al., 2025), Aider-Polyglot[1], and LiveCodeBench (Jain et al., 2025a). The results in Table 3 indicate that the final SWE-Swiss-32B model consistently shows gains across the board. While the intermediate SWE-Swiss-SFT-32B model shows a more complex performance profile with mixed results compared to the base model, the subsequent reinforcement learning (RL) phase successfully resolves these inconsistencies.

5 Conclusion

In this paper, we introduce SWE-Swiss, a two-phase training recipe designed to systematically resolve complex software engineering tasks. Our method first builds a foundational model through multi-task Supervised Fine-Tuning (SFT) on three core skills—Localization, Repair, and Unit Test Generation—using high-quality, curated datasets. We then sharpen the repair skill via a targeted Reinforcement Learning (RL) phase that leverages direct feedback from test execution environments. The resulting model, SWE-Swiss-32B, achieves a top-tier of 60.2% on the SWE-bench Verified benchmark, demonstrating that a 32B parameter model can rival the performance of significantly larger counterparts. Alongside this recipe, we propose an Enhanced Self-consistency method for more effective test-time scaling. Despite the specialized training, SWE-Swiss-32B generalizes well to other popular benchmarks.

[1] https://aider.chat/docs/leaderboards/

Limitations

The limitations of our work are as follows:

Performance Gap with Frontier Proprietary Models: Although SWE-Swiss-32B establishes a new standard for open-source models within its parameter class, a discernible performance gap persists when compared to the leading proprietary models. As indicated in Table 1, our model's resolve rate on SWE-bench Verified has not yet reached the levels achieved by the best commercial models like Claude-4-Sonnet. Closing this gap and enhancing the competitiveness of open-source solutions remains a challenge for the research community.

Linguistic Specialization to Python: The training corpus for SWE-Swiss-32B is predominantly composed of Python-based repositories. Extending our training regimen to encompass a broader range of languages would necessitate substantial computational resources and the curation of language-specific datasets with executable environments, which was beyond the scope of the present study. We hypothesize that our methodology can be generalized effectively to other programming languages, representing a direction for future research.

Acknowledgments

DH is supported by National Science Foundation of China (NSFC62376007), National Science Foundation of China (under Key Project No. 92570203), Beijing Natural Science Foundation (Z250001) and the Beijing Major Science and Technology Project (No.Z251100008425004). This work is supported in part by the State Key Laboratory of General Artificial Intelligence.

References

2025. Introducing kimi-dev: A strong and open-source coding llm for issue resolution. https://moonshotai.github.io/Kimi-Dev. Blog.

Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. 2025. Polaris: A post-training recipe for scaling reinforcement learning on advanced reasoning models.

Anthropic. 2025. Claude 3.7 sonnet and claude code. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2025-09-04.

Anthropic. 2025. Introducing claude 3.7 sonnet.

Anthropic. 2025. Introducing claude 4. https://www.anthropic.com/news/claude-4. Accessed: 2025-09-04.

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

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. 2025. Agent-rlvr: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425*.

DeepSeek. 2025. DeepSeek-R1-0528 Release. https://api-docs.deepseek.com/news/news250528. Accessed: 2025-09-05.

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

Xinrun Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, King Zhu, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, and 1 others. 2025. Supergpqa: Scaling llm evaluation across 285 graduate disciplines. *arXiv preprint arXiv:2502.14739*.

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*.

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

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025a. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, and 1 others. 2024. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*.

VI Lcvenshtcin. 1966. Binary coors capable or 'correcting deletions, insertions, and reversals. In *Soviet physics-doklady*, volume 10.

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

Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. 2025. Zebralogic: On the scaling limits of LLMs for logical reasoning. In *Forty-second International Conference on Machine Learning*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023c. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

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. 2025a. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl. https://pretty-radio-b75.notion.site/DeepSWE-Training-a-Fully-Open-sourced-State-of-the-Art-Coding-Agent-by-Scaling-RL-22281902c1468193aabbe9a8c59bbe33. Notion Blog.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025b. Deepcoder: A fully open-source 14b coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51. Notion Blog.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025c. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2. Notion Blog.

Trung Quoc Luong, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. 2024. Reft: Reasoning with reinforced fine-tuning. *arXiv preprint arXiv:2401.08967*.

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*.

MAA. 2024. American invitational mathematics examination - aime. In *American Invitational Mathematics Examination - AIME 2024*.

MAA. 2025. American invitational mathematics examination - aime. In *American Invitational Mathematics Examination - AIME 2025*.

Mistral AI. 2025. Devstral: Introducing the best open-source model for coding agents. `https://mistral.ai/news/devstral`. Accessed: 2025-09-05.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer.

OpenAI. 2024a. Hello gpt-4o. Accessed: 2025-09-04.

OpenAI. 2024b. Introducing openai o1. Accessed: 2025-09-04.

OpenAI. 2024c. Introducing swe-bench verified.

OpenAI. 2024d. New embedding models and api updates. https://openai.com/index/new-embedding-models-and-api-updates/.

OpenAI. 2025a. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/. Accessed: 2025-09-04.

OpenAI. 2025b. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/. Accessed: 2025-09-04.

OpenAI. 2025c. Openai o3-mini. https://openai.com/index/openai-o3-mini/. Accessed: 2025-09-04.

Albert Orwall. 2024. Moatless tools. Accessed: 2025-08.

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

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

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

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

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2024. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, and 1 others. 2025. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks. *arXiv preprint arXiv:2505.16901*.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, and 75 others. 2025. Kimi k1.5: Scaling reinforcement learning with llms. *CoRR*, abs/2501.12599.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. 2025a. Swe-dev: Building software engineering agents with training and inference scaling. *arXiv preprint arXiv:2506.07636*.

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. *arXiv preprint arXiv:2509.08724*.

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

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024b. MMLU-pro: A more robust and challenging multi-task language understanding benchmark. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

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

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

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*.

Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, and 1 others. 2024. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*.

John Yang, Carlos E 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*, 37:50528–50652.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025a. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, and 1 others. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and 1 others. 2025a. Skywork-swe: Unveiling data scaling laws for software engineering in llms. *arXiv preprint arXiv:2506.19290*.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and 1 others. 2025b. Skywork-swe: Unveiling data scaling laws for software engineering in llms. *arXiv preprint arXiv:2506.19290*.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

A Repair Prompt Construction via RL-trained Localization

To create a more challenging training environment for the Repair task, this research explored a strategy for generating "distractor files". This is achieved using an auxiliary localization model (initialized from SWE-Swiss-SFT-32B) trained with Reinforcement Learning (RL), where a reward of +1 is given for predicting no more than five files with perfect recall. The files predicted by this model, minus the known oracle files, are then used as distractors in the Repair prompt.

This RL-based localization approach proved to be effective for enhancing the performance of weaker base models in our early development of this project. However, it was observed that the multi-task SFT phase already endowed the final SWE-Swiss-32B model with a sufficiently powerful localization capability when augmented with the embedding model (OpenAI, 2024d) for retrieval. Considering this, and for the sake of overall recipe simplicity, this RL phase is not included in the final presented recipe, though its utility for other contexts is noted.

B Evaluation Pipeline

Our evaluation pipeline, adapted from the Agentless (Xia et al., 2024) and Agentless mini (Wei et al., 2025) frameworks, operates in two modes depending on the desired number of output patches per issue.

Single-Patch Generation. For generating a single patch, we use a direct, two-stage workflow. First, the Localize module predicts the relevant files. Second, the Repair module uses these files, combined with files retrieved via text-embedding-3-small [11], to generate a single candidate patch.

Multi-Patch Generation with Test-Time Filtering. For generating multiple candidates, we execute the Localize module multiple times and generate multiple patches for each localization. A filtering mechanism is activated to select the final patch (as depicted in the bottom path in Figure 2):

1. Initial Filtering with Existing Regression Tests: As a preliminary screening step, all generated candidate patches are first evaluated against the repository's pre-existing regression test suite.

2. Secondary Filtering with Generated Reproduction Tests: Patches that pass the initial regression testing then undergo a more targeted validation using the LLM's own newly generated reproduction tests. The LLM generates these tests based on the issue description and predicted relevant files. The tests themselves are first filtered to ensure they correctly reproduce the issue before being used to evaluate the patches. The final test is selected via self-consistency (majority vote).

3. Final Selection: A final patch is selected via our enhanced self-consistency.