Thinking Longer, Not Larger: Enhancing Software Engineering Agents via Scaling Test-Time Compute

Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Yanhao Li, Yue Liu,
Rongyu Cao, Jue Chen, Fei Huang, Binhua Li
Tongyi Lab, Alibaba Group

***Abstract*—Recent advancements in software engineering agents have demonstrated promising capabilities in automating program improvements. However, their reliance on closed-source or resource-intensive models introduces significant deployment challenges in private environments, prompting a critical question: *How can personally deployable open-source LLMs (e.g., 32B models running on a single GPU) achieve comparable code reasoning performance?* To this end, we propose a unified Test-Time Compute (TTC) scaling framework that leverages increased inference-time computation instead of larger models. Our framework incorporates two complementary strategies: internal TTC and external TTC. Internally, we introduce a *development-contextualized trajectory synthesis* method leveraging real-world software repositories to bootstrap multi-stage reasoning processes, such as fault localization and patch generation. We further enhance trajectory quality through rejection sampling, rigorously evaluating trajectories along accuracy and complexity. Externally, we propose a novel *development-process-based search* strategy guided by reward models and execution verification. This approach enables targeted computational allocation at critical development decision points, overcoming limitations of existing "end-point only" verification methods.**

Evaluations on SWE-bench Verified demonstrate our 32B model achieves a 46% issue resolution rate, surpassing significantly larger models such as DeepSeek R1 671B and OpenAI o1. Additionally, we provide the empirical validation of the test-time scaling phenomenon within SWE agents, revealing that models dynamically allocate more tokens to increasingly challenging problems, effectively enhancing reasoning capabilities. We publicly release all training data, models, and code to facilitate future research.[1]. *In fact, our method has been deployed in Tongyi Lingma, an IDE-based coding assistant developed by Alibaba Cloud, where it helps developers solve real-world programming problems.*

***Index Terms*—Software Improvement, Test Time Scaling, Code Agent, SWE-bench**

I. Introduction

Large language model (LLM)-based agents have emerged as promising tools for automating various software engineering tasks, particularly in software maintenance (e.g., bug fixing) and evolution (e.g., adding new features). The SWE-bench [1] has become a critical benchmark for evaluating the capabilities of SWE agents, specifically designed to simulate real-world software improvement tasks. Given a natural language description of an issue and the corresponding GitHub repository, SWE agent is tasked with generating a patch that resolves the issue. The typical framework in code agent research involves locating the relevant code, generating a patch, and verifying its correctness [2, 3, 4].

The main driver of progress in the field has been scaling model parameters and training data, leading to notable improvements in model capabilities. However, this scaling introduces critical deployment challenges. For instance, DeepSeek V3 671B requires 436GB of VRAM, even with 4-bit quantization, and demands multi-GPU setups (e.g., 6 NVIDIA A100 80GB) [5], making such systems impractical for most organizations. Additionally, closed-source models like Claude 3.5 raise privacy concerns when used via API services, particularly regarding private code repositories. These challenges lead to our central research question: *How can we unlock the code reasoning potential of deployable LLMs, achieving comparable performance?* For example, the 4-bit quantized 32B model requires only 21GB of VRAM and can run on a single NVIDIA RTX4090 card [5].

To address this challenge, we propose shifting the scaling paradigm from model size to increasing the inference time inspired by emerging Test-Time Compute Scaling approaches [6, 7]. Current TTC implementations take two forms: Internal TTC, where models are trained to enhance reasoning depth through longer Chain-of-Thought (CoT); and External TTC, where multiple outputs are generated in parallel, and the optimal solution is selected using search-based strategies. Despite the potential of these approaches, technical difficulties, including resource constraints and proprietary strategies, have limited further exploration in this area. Specifically, the following issues remain underexplored:

- *Proprietary Implementation Barriers*: While models like OpenAI o1 [6] and DeepSeek R1 [8] have demonstrated the effectiveness of long CoT reasoning, their methodologies remain proprietary and rely heavily on non-public training data and requires substantial computational resources and data collection efforts, making replication challenging. Given the privacy concern surrounding software repositories, there is a pressing need for transparent and computationally efficient methods, enabling strong reasoning capabilities even within resource-constrained, private development environments.
- *Search Strategies Limitations*: Existing external TTC approaches employ simplistic selection mechanisms like majority voting [4], which prove inadequate for software tasks requiring precise understanding of development

[1]Model: https://github.com/yingweima2022/SWE-Reasoner/tree/6627eba7215425ecfef65a40a9c516b2feca1bc7, Code: https://github.com/yingweima2022/AnonymousSWESynInferpro

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00309

context. Few studies have systematically analyzed the impact of different search strategies—such as outcome and process reward models, or test-driven verification—on guiding the issue resolution process.

Our Approach. To answer these questions, we conduct a systematic exploration on the challenging SWE-bench Verified [9] proposed by OpenAI. We build upon an open-source SWE framework (SWESyninfer [10]) to generate initial single-solution proposals, which divide the issue resolution process into three key steps: (1) identifying relevant codebase context (repository understanding), (2) fault localization, and (3) generating candidate code edits. We then explore both internal and external TTC methods to enhance agent performance.

For internal TTC, we propose a *development-contextualized trajectory synthesis* method to address limited due to a lack of realistic multi-stage reasoning data aligned with actual software development workflows. Specifically, we first scrape ¡issue, repository, pull-request¿ triplets from high-quality GitHub repositories (¿1000 stars) and construct executable verification environments; we then use DeepSeek R1 as a bootstrapping model to generate comprehensive reasoning trajectories spanning repository understanding, fault localization, patch generation, and patch verification. These trajectories are refined through *Development-Contextualized Rejection Sampling*, which ensures quality via multi-dimensional filtering that evaluates both accuracy and complexity (filtering out problems solvable by small base model without refinement). Finally, our *Reasoning Training* preserves both the think component (capturing planning, reflection, and correction processes) and the answer component (final solutions) at each reasoning step, enabling the model to internalize the multi-step decision-making process essential for complex software engineering tasks. This approach resolves 37.6% of issues on SWE-bench Verified with trained 32B model, surpassing Llama 3.1 405B [11]. Our results demonstrate that smaller models can achieve comparable capabilities to much larger models when trained on high-quality, multi-step reasoning trajectories derived from real software development scenarios.

For *External TTC*, we introduce a *development-process-based search* strategy that strategically focuses computational resources on critical decision points in the software engineering workflow. Unlike existing approaches that either validate only at the final solution stage [4, 12], our framework applies targeted search at three crucial development phases: repository understanding, fault localization, and patch generation. We train specialized Process Reward Model (PRM) to evaluate intermediate outputs at these critical junctures, effectively pruning less promising solution paths early while maintaining a manageable beam width. At the patch generation stage, we implement execution verification through automatically generated reproduction code, providing concrete feedback on patch correctness. For final solution selection, we employ an Outcome Reward Model (ORM) trained via Direct Preference Optimization on verified patch pairs, enabling effective ranking of candidate solutions without requiring access to intermediate reasoning steps. Our experiments demonstrate that this development-process-based search strategy significantly improves performance with fixed model size, and when combined with our Internal TTC approach, yields even greater performance gains. These results highlight how strategic test-time computation allocation can achieve performance comparable to much larger models while maintaining computational efficiency. Additionally, we provide the first empirical validation of the test-time scaling phenomenon within SWE agents, revealing that models dynamically allocate more tokens to increasingly challenging problems, effectively enhancing reasoning capabilities.

Contributions. In summary, we make the following novel contributions:

- We propose a unified scaling TTC approach tailored specifically for software engineering agents, including Internal TTC and External TTC.
- Our method achieves state-of-the-art open source results on the challenging SWE-bench Verified benchmark, resolving 46% of issues with a 32B model. Notably, our approach surpasses larger models, demonstrating the effectiveness of targeted inference-time scaling.
- We present the empirical validation of the test-time scaling phenomenon within SWE agents, showing that increased inference-time computation improves performance on challenging software engineering problems.
- We open-source our model checkpoints, data, and code to support further research and development in this field.

II. Test-Time Computation Explored: Internal and External Strategies

In this section, we explore two core strategies for enhancing SWE-agent performance through scaling TTC: Internal and External TTC. Figure 3 presents our unified framework, illustrating how these approaches improve software engineering task. We first provide an overview of these two strategies and then delve into their specific implementations and results.

A. Internal TTC in Software Engineering

Internal TTC aims to enhance the reasoning depth during inference by leveraging extended CoT. While OpenAI o1 and DeepSeek R1 achieve strong performance via large-scale Reinforcement Learning (RL) and massive datasets, we hypothesize that training smaller models (e.g., 32B parameters) using bootstrapped long reasoning trajectories, augmented by development-contextualized rejection sampling, can activate comparable reasoning capabilities. This is primarily because the model has already encoded a wealth of software engineering knowledge during pre-training. By utilizing high-quality, multi-step reasoning trajectories derived from real software development scenarios during post-training, we provide effective multi-step decision supervision, which helps unlock the model's reasoning potential. To validate this, we introduce a systematic approach for synthesizing high-quality reasoning trajectories, consisting of three primary stages: data curation, trajectory bootstrapping, and development-contextualized rejection sampling.

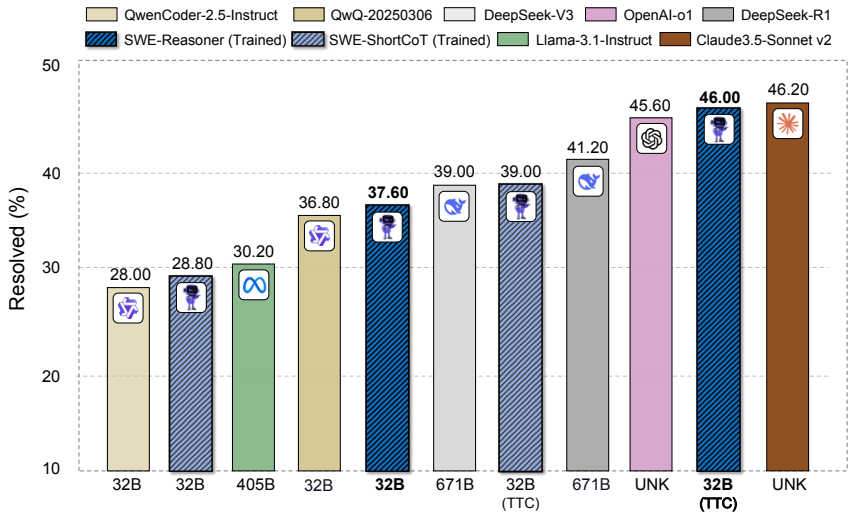

Fig. 1. Comparison between the performance of smaller LLMs with extended Test-Time Compute and larger models on SWE-Bench Verified.

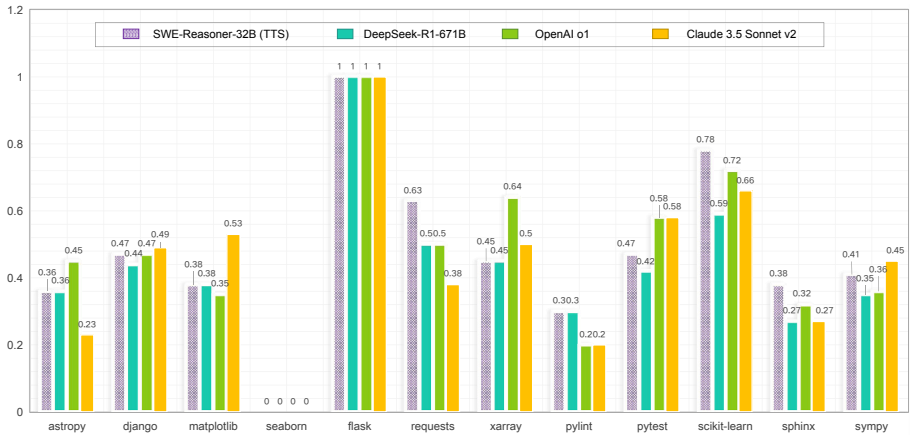

Fig. 2. Comparison of issue resolution rates between our unified TTC framework (32B) and other LLMs across different repositories in SWE-bench Verified.

1) High-Quality Trajectory Synthesis: The foundation of our approach lies in high-quality, real-world software development data. In **Data Curation** stage, we begin by scraping ¡issue, pull-request, codebase¿ triplets from GitHub using SWE-bench's data collection procedure [1], focusing on repositories with high star ratings (¿1000 stars) to ensure code quality. *We filter out repositories already present in the SWE-bench dataset to avoid data leakage.* [13] For each selected repository, we collect issues and linked pull requests (PRs) that were merged by developers. To further enhance the quality of the data, we apply a set of heuristic filtering rules, similar to those used in OctoPack [14]. For issues, we retain only those with textual descriptions containing at least 20 characters to exclude overly vague or incomplete issues. Additionally, we filter out issues containing more than three hyperlinks, as these are often references to external resources rather than detailed descriptions of the issue at hand. For pull requests, we focus on those that modify between one and five code files, excluding those that only modify test files. This ensures that the changes are substantive. To ensure that each repository is suitable for patch verification, we use ExecutionAgent [15] to automatically construct the execution environment, ensuring the necessary dependencies and execution contexts are properly set up. We filter out repositories where the environment cannot be built or run, resulting in a final dataset of 9,000 issues from 300 repositories, with verified executable environments capable of real-time patch validation.

In **Trajectory Bootstrapping** stage, we employ a bootstrapping strategy to synthesize detailed problem-solving trajectories. This approach builds upon the open-source SWE framework (SWE-SynInfer [10]), which has achieved superior results in open-source models. SWE-SynInfer divides the issue resolution process into three steps: (1) repository understanding to identify relevant codebase files, (2) fault localization to pinpoint problematic code segments, and (3) patch generation to produce candidate code edits. We extend this framework

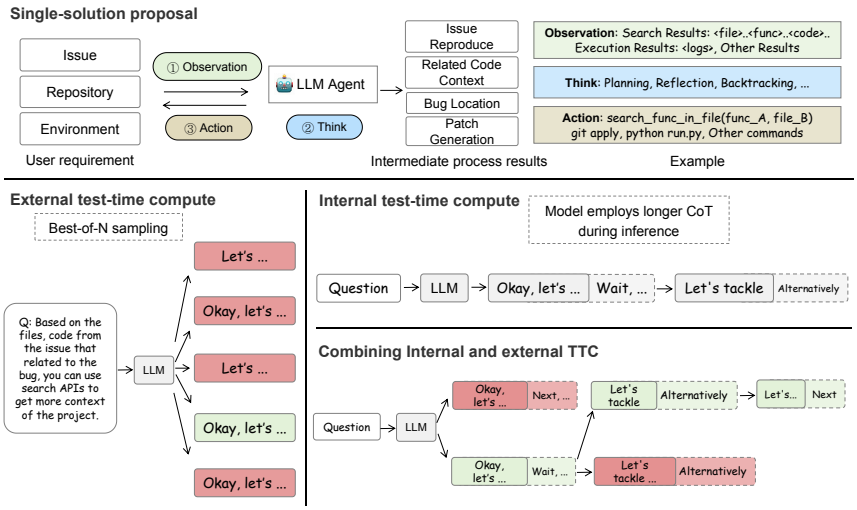

Fig. 3. A unified view of Test-Time Scaling strategies for SWE agents. Internal TTC enhances reasoning depth through extended chain-of-thought training, while External TTC employs reward-guided search and verification to select optimal solutions. The hybrid approach combines both paradigms through iterative refinement.

to include a Patch Verification phase, following Agentless [4], and call it *SWE-SynInfer+*. In this enhanced phase, the model generates reproduction code based on the issue description, and then verifies the correctness of the generated patch by executing the reproduction code. If the patch is deemed incorrect, the model iterates, refining the solution until it either meets the verification criteria or reaches the maximum threshold of iterations. We use a open-source reasoning model (DeepSeek R1 [8]) to bootstrap these long reasoning trajectories, as R1 iterates and refines its internal reasoning multiple times by utilizing more inference computation before producing the final output. Each trajectory step in the bootstrapping process includes two primary components: the *think* component, which captures the planning, reflection, and correction processes, and the *answer* component, which represents the final solution for that step. The trajectory bootstrapping process is summarized in Algorithm 1, which outlines how the model generates a sequence of reasoning steps. This algorithm mirrors real-world software development practices, where each stage builds upon previous reasoning in an iterative manner, progressively refining the solution. The environment is updated as the reasoning process progresses, and the model continues until either the patch is successfully verified by reproduce code or the maximum number of steps is reached.

We use **Development-Contextualized Rejection Sampling** to ensuring the quality of generated reasoning trajectories, which contain **accuracy** and **complexity** of each trajectory.

- Repository Understanding: We verify that the model correctly identifies the files that need modification. Specifically, we compare the model's output in the Repository Understanding phase with the files changed in the developer's patch, ensuring alignment with the actual code modifications.
- Fault Localization: The generated patch must focus on the correct locations within the code (e.g., relevant classes, functions, and surrounding code blocks). We check that the model's patch includes changes at these same locations as those in the developer's patch.
- Issue Reproduce: We validate the generated reproduction code's correctness against the developer's patch. A valid reproduction code should output *issue reproduced* when executed on the original codebase and *issue resolved* when executed after applying the developer's patch. This two-stage verification ensures that the reproduction code correctly captures the essence of the issue and can reliably

Algorithm 1 Trajectory Bootstrapping Process

Input: Issue I, Repository R, Base Model M
Initialize trajectory $\tau = []$, Environment $\mathcal{E}$
procedure GENERATETRAJECTORY(I, R, M)
  for step $t \in \{1, \ldots, T_{\max}\}$ **do**
    $s^t_{\text{think}}, s^t_{\text{answer}} \leftarrow M(\text{CoT-Prompt}(I, R, \tau[1:t-1]))$
    `ActionType, Params` $\leftarrow$ ANALYZE(s^t_{answer})
    if parsing failed **then**
      τ.append(fallback_error_handling)
      continue
    end if
    $s^t_{\text{output}} \leftarrow$ EXECUTEACTION(`ActionType, Params`, $\mathcal{E}$)
    τ.append$\big(((s^t_{\text{think}}, s^t_{\text{answer}}), s^t_{\text{output}})\big)$
    Update $\mathcal{E}$ with s^t_{output} outcomes
    if Resolved($\mathcal{E}$) **or** Failed($\mathcal{E}$) **then**
      break
    end if
  end for
  return τ
end procedure

detect when the issue has been fixed.

- Patch Correctness: We assess whether the patch resolves the issue. We apply the model's patch to the repository and run the SWE agent's reproduction code to check if the issue is fixed. For cases where the LLM fails to generate correct reproduction code, we follow the approach [10] by evaluating the similarity between the model-generated patch and the developer's patch as a filtering criterion. We also run existing unit tests to ensure the patch does not break other functionalities, verifying the correctness and stability of the solution.
- Complexity Filtering: To focus on challenging problems that activate deeper reasoning capabilities, we filter out simpler issues that Qwen2.5 Coder 32B [16] can solve in a single attempt without refinement. This ensures our training data consists of problems requiring sophisticated long CoT reasoning.

By incorporating development context into the rejection sampling process, we ensure that only high-quality trajectories are retained, ultimately enhancing the model's reasoning depth and performance. Additionally, if a patch is incorrect but the preceding reasoning stages are accurate, we discard the erroneous patch data while preserving the correct stage data. This allows us to retain valuable reasoning steps, ensuring that useful problem-solving knowledge is not lost during the filtering process.

2) Training: We train our model using supervised learning on the synthesized long CoT trajectories dataset. Our objective is to enable the model to internalize structured multi-round reasoning. We follow a standard maximum likelihood estimation objective, optimizing the conditional probability of generating correct reasoning actions given an issue and prior observations. The training loss is computed over both the *think* and *answer* components at each step, ensuring that the model learns both intermediate reasoning steps and final predictions. To enhance efficiency in multi-round inference, we adopt a history pruning mechanism inspired by DeepSeek R1 [8]. Specifically, for each reasoning step i, we discard the *think* component of the previous response and retain only the final *answer* in the historical context. Formally, given a training instance consisting of issue and the corresponding step-wise trajectory:

$$\theta' \leftarrow \underset{\theta}{\operatorname{argmax}} \sum_{(s^i_{\text{obs}}, s^i_{\text{think}}, s^i_{\text{ans}}) \in \text{traj}} \log P_\theta(s^i_{\text{think}}, s^i_{\text{ans}} \mid \text{issue}, s^i_{\text{obs}}, \mathcal{H}_{i-1}) \tag{1}$$

$$\mathcal{H}_i = \mathcal{H}_{i-1} \cup \{s^i_{\text{obs}}, s^{i-1}_{\text{ans}}\} \tag{2}$$

where s^i_{obs} represents the structured observations at step i, capturing relevant code snippets, execution logs, or other extracted information crucial for reasoning. s^i_{think} represents the model's internal reasoning process, and s^i_{ans} represents the actionable output from the model at each step, such as the search_api, the specific patch to apply, or a command to run. $\mathcal{H}_{i-1}$ denotes the historical trajectory context up to step $i-1$, ensuring that the model conditions on prior reasoning states when generating the next step.

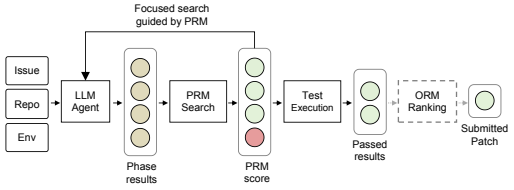

Fig. 4. Overview of Development-Process-Based Search Strategy.

B. Effective Search Strategies for External TTC

External TTC explores ways to leverage multiple inference outputs to identify the best solution (see Figure 3). Existing methods typically generate several candidate patches at once and then rely solely on a final correctness check (e.g., by running unit tests [3, 17], regression tests [4], or outcome-based reward models [12]) to select the best candidate. However, such "end-point only" methods often underutilize the available search budget because they do not intervene intermediate reasoning steps. This is particularly problematic for SE tasks, which involve lengthy reasoning chains with multiple interdependent decisions. Moreover, classical tree search [18] (like beam search) applied at *every* intermediate step (i.e., "step-by-step" validation) is also infeasible for extensive software development pipelines, due to the computational overhead of verifying. To address this, we propose a development-process-based search strategy that focuses on the critical decision phases of software development.

1) Development-Process-Based Search Strategy: We decompose the agent's problem-solving process into three essential phases: (1) repository understanding, (2) fault localization, and (3) patch generation. These phases represent crucial decision points in the development process, where errors can propagate and dramatically affect subsequent steps. By focusing our search at these junctures, we ensure that the agent's decisions are evaluated at critical stages, not at every single action within the process. Figure 4 presents our overview framework.

Focused Search with Process Reward Model (PRM). At the repository understanding and fault localization stages, we apply a lightweight beam search strategy, guided by PRM. For each stage, we generate N candidate outputs and use the PRM to score each candidate based on its likelihood of correctness. The top-k highest-scoring candidates are retained and used as input for the next stage, effectively pruning less promising solution paths. This approach maintains a manageable beam width while focusing computational resources on the most promising solution trajectories.

Patch Generation and Execution Verification. At this stage, the agent generates potential patches to resolve the identified bug. To ensure the correctness of the generated patches, we apply execution verification, where the agent generates reproduction code to check if the patch successfully

fixes the issue. This verification process also ensures that the patch does not introduce new bugs by running regression tests on the repository to confirm that existing functionality is unaffected.

Final Ranking with Outcome Reward Model (ORM). After executing the verification checks, we prioritize keeping patches that pass more tests, and then we select the most promising patches from among these (in case of a tie). Here, we apply the ORM, which evaluates the quality of the final patches. The ORM ranks multiple candidate patches and the highest score from the ORM is selected as the final solution to be submitted. Importantly, our ORM design requires only the issue description and the candidate patch as inputs, without depending on intermediate reasoning steps or specific agent architectures. This design choice ensures that our ORM can be seamlessly integrated with various SWE agent systems or CI/CD pipelines.

2) Reward Model Training: **Process Reward Model (PRM)** The PRM aims to assess the intermediate correctness at critical development phases, namely repository understanding and fault localization. To train the PRM, we construct a labeled dataset by leveraging the high-quality bootstrapped trajectories generated during the trajectory synthesis phase. For each trajectory step, we formulate a binary classification task where the PRM learns to distinguish between correct and incorrect intermediate outputs. Specifically, for repository understanding, the model predicts whether the identified files align with the actual developer's modified files. For fault localization, it predicts whether the model-generated patch aligns with the developer-edited detailed locations. We use the contextual information from issues and intermediate trajectory reasoning outputs as inputs, enabling the PRM to contextualize and effectively evaluate partial solutions. We fine-tune a base model using a standard next-token prediction objective with cross-entropy loss, guiding the model to output tokens corresponding to binary labels (i.e., "+" for correct and "-" for incorrect):

$$\mathcal{L}_{PRM} = -\sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)] \quad (3)$$

where y_i is the binary correctness label (1 for correct, 0 for incorrect), and p_i is the PRM's predicted probability of correctness.

Outcome Reward Model (ORM). The ORM performs final sorting of the generated patches. For ORM training, we curate a dataset comprising pairs of candidate patches labeled according to their verification outcomes. Specifically, patches that pass all execution verification and regression tests are considered superior (winning response), while those failing any verification steps are inferior (losing response). To effectively capture relative patch quality, we apply the Direct Preference Optimization (DPO) loss [19] for training:

$$\mathcal{L}_{ORM}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_w, y_l)} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)} \right) \right] \quad (4)$$

Here, y_w represents the winning patch (passes verification), y_l is the losing patch (fails verification), and x is the associated issue description. We fine-tune a smaller base model as the ORM reference model (π_{ref}) to maintain fast inference during the external TTC phase. The hyperparameter β controls the reward sharpness, and we chose a common value of 0.5.

C. Putting It Together

We propose a unified framework by seamlessly integrating internal and external Test-Time Scaling (TTC), emphasizing enhanced performance of software engineering agents through allowing models to *think longer* and *search more*, instead of increasing model size. Figure 3 illustrates this unified TTC framework, clearly demonstrating the integration of internal and external scaling strategies. All models in our experiments are based on Qwen2.5 Coder 32B [16]. Our approach ultimately shows that careful inference-time scaling can achieve or even surpass the performance of significantly larger models, thus enabling advanced software engineering reasoning capabilities even under constrained computational resources. The effectiveness of this approach will be thoroughly validated through subsequent experiments.

III. Evaluation

A. Benchmark and Evaluation Metric

SWE-bench Verified. We evaluated our method on the recently proposed benchmarks SWE-bench Verified [9], comprising 500 real-world GitHub issues. The model receives only the natural language description of the original GitHub issue and its corresponding code repository as input. These benchmarks employ developer-written unit tests to verify the correctness of model-generated patches, ensuring a rigorous assessment of the model's performance.

Evaluation Metric. We use (1) the percentage of resolved task instances, (2) fault location success rate. These evaluation metrics represent overall effectiveness in resolving real-world GitHub issues. In addition, we evaluate the effectiveness of solving issues at different difficulty levels and different generation budgets to verify the test-time scaling phenomenon of our method.

B. Overall Effectiveness of Unified TTC Framework

We evaluate the effectiveness of our unified TTC framework on the SWE-bench Verified benchmark. We first assess various base models under our SWE-SynInfer+ framework. Figure 1 illustrates the comparative performance results. Notably, our 32B SWE-Reasoner model, which employs Internal TTC strategies, achieves an issue-resolution accuracy of 37.60%. When combined with External TTC (budget=8), our model's

Agent	LLM	Verified
Model size unknown or size $>$ 100B		
SWE-agent [9]	GPT-4o	23.00%
AutoCodeRover [9]	GPT-4o	28.80%
SWE-SynInfer [10]	GPT-4o	31.80%
Agentless [9]	GPT-4o	33.20%
SWE-agent [3]	Claude3.5-Sonnet-v1	33.60%
SWE-SynInfer [10]	Claude3.5-Sonnet-v1	35.40%
OpenAI Tools [20]	GPT-4.5	38.00%
Agentless [7]	OpenAI-o3-mini	40.00%
Agentless [6]	OpenAI-o1-1217	41.00%
Anthropic Tools [21]	Claude3.5-Sonnet-v2	49.00%
OpenAI Tools [7]	OpenAI-o3-mini	61.00%
Anthropic Tools [22]	**Claude3.7-Sonnet**	**62.30%**
Model size $\leq$ 100B		
Agentless [23]	Qwen2.5-Coder 32B	25.60%
SWE-Gym [12]	SWE-Gym 32B	29.80%
SWE-SynInfer [10]	SWE-GPT 72B	30.20%
Agentless [23]	SoRFT-Qwen 32B	30.80%
SWE-Fixer [10]	SWE-Fixer 72B	32.80%
NebiusAI [18]	NebiusAI 72B&70B	40.60%
Agentless Mini [17]	Llama3-SWE-RL 70B	41.00%
SWE-SynInfer+	**SWE-Reasoner 32B**	**46.00%**

TABLE I
PERFORMANCE COMPARISON OF OUR METHOD AND OTHER MODELS ON SWE-BENCH VERIFIED BENCHMARK.

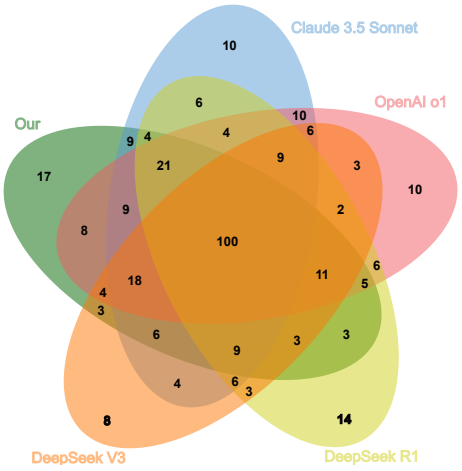

Fig. 5. Venn diagram of issue instances solved by our unified TTC framework and other models on SWE-bench Verified.

performance further increases to 46.00%. This unified approach closely matches the performance of the significantly larger proprietary Claude 3.5 Sonnet v2 model (46.20%) and surpasses OpenAI-o1 (45.60%) and DeepSeek-R1 (41.20%), clearly demonstrating the effectiveness of our unfied TTC strategies. We further benchmark our approach against leading state-of-the-art SWE agent frameworks reported in existing literature (see Table I). Within the $\leq$ 100B model-size category, our method achieves the highest issue-resolution accuracy, establishing a new state-of-the-art. Importantly, our method achieves this performance with substantially lower computational demands, emphasizing that careful inference-time computation strategies effectively leverage smaller models to reach competitive results.

Additionally, to evaluate the generalization and robustness of our unified TTC framework across different software domains, we analyzed its performance on a diverse set of repositories. Figure 2 illustrates the issue-resolution rates of our SWE-Reasoner-32B (TTC) model across 12 representative software repositories, compared with the strongest open-source baseline (DeepSeek-R1 671B) and two leading closed-source models (OpenAI-o1 and Claude 3.5 Sonnet v2). Notably, SWE-Reasoner-32B (TTC) matches or surpasses the performance of DeepSeek-R1 671B in the majority of repositories, and closely approaches the performance of larger closed-source models in numerous instances. This consistent cross-domain performance underscores our method's robust generalization capabilities, highlighting its potential applicability and effectiveness across a wide spectrum of real-world software engineering scenarios.

As illustrated in Figure 5, we further analyzed the overlap of solved issue instances among different models on the SWE-bench Verified benchmark through a Venn diagram. The diagram reveals that our unified TTC framework uniquely solves 17 issue instances that other models fail to address, while also sharing a substantial number of successfully resolved issues with major models. This indicates that our approach not only achieves competitive performance quantitatively but also demonstrates unique problem-solving capabilities in terms of coverage.

C. Analysis of Internal TTC Strategies

We conducted two detailed analyses to comprehensively assess the effectiveness of our Internal TTC strategies:

Effectiveness of Internal TTC via Ablation Study. We performed ablation studies to assess the individual contributions of key Internal TTC components, particularly evaluating their impact on issue-resolution rates and fault localization accuracy. The results are summarized in Table II. Upon removing the Long Chain-of-Thought (Long CoT) component (*-w/o. LongCoT*), we observed a significant reduction in issue resolution accuracy from 37.60% to 28.80%. Specifically, in the *-w/o. LongCoT* experiment, we omitted the *think* labels from training data, instead prompting Claude 3.5 Sonnet v2 [21] to explicitly generate short-CoT reasoning and corresponding action predictions on the same dataset. We then applied the repository-aware rejection sampling method to this short-CoT data and trained the same base model (Qwen2.5-Coder 32B [16]). Despite leveraging the stronger Claude model for short-CoT generation, the trained smaller model underperformed compared to our original Long CoT strategy. This result highlights the unique advantage of Long CoT in activating deeper reasoning capabilities in smaller models. Additionally, we evaluated the impact of our repository-aware rejection sampling method by removing this filtering step (*-w/o. Rejection*). Although using unfiltered synthesized data increased the overall volume of training data, issue-resolution performance decreased from 37.60% to 33.00%. This decline underscores the importance of carefully curated, high-quality reasoning trajectories for effective training.

Ablation	Resolved	Chunk	Func	File
SWE-Reasoner	**37.60%**	**51.00%**	**54.49%**	**72.19%**
-w/o. LongCoT	28.80%	49.05%	51.68%	69.18%
-w/o. Rejection	33.00%	48.76%	51.94%	71.38%
-w/o. All	28.00%	44.22%	47.25%	60.69%

TABLE II
ABLATION EXPERIMENT OF THE INTERNAL TTC METHOD, WHERE RESOLVED IS THE ISSUE RESOLUTION RATE ON SWE-BENCH VERIFIED, AND CHUNK, FUNC, AND FILE ARE THE FAULT LOCATION SUCCESS RATES AT THREE DIFFERENT LEVELS.

To further clarify the benefits of explicitly Long CoT reasoning trajectories training, we compared performance across internalized Long CoT (SWE-Reasoner), internalized Short CoT (*w/o. Long CoT*), and prompt-based CoT (*w/o. All*). Specifically, we categorized SWE-bench Verified issues into five difficulty buckets based on their resolution frequency among the top 30 submissions on the SWE-bench leaderboard [1]. Level 1 includes issues resolved by 25–30 agent submissions (easiest), level 2 by 20–25 submissions, level 3 by 15–20 submissions, level 4 by 10–15 submissions, and level 5 by 5–10 submissions (hardest). Issues resolved fewer than five times were excluded due to their infrequency and high variance. As shown in Figure 6, models employing internalized CoT (both Long and Short) consistently outperform Prompt-CoT-based methods. Crucially, our internalized Long CoT approach significantly surpasses Short CoT performance on the hardest bucket (level 5), achieving an issue-resolution rate approximately six times higher. These findings confirm that explicitly internalizing long reasoning trajectories is highly effective, particularly in enabling small models to tackle complex tasks by effectively leveraging test-time computational resources.

Analysis of the Test-Time Scaling Phenomenon. We further investigated whether the SWE-Reasoner dynamically allocates computational resources based on task complexity, as indicated by longer inference trajectories (measured by output token counts). Using the previously defined difficulty buckets (level 1 being easiest and level 5 hardest), we compared average output tokens generated by SWE-Reasoner, OpenAI o1, ShortCoT model, and Claude 3.5 Sonnet v2 across different issue-difficulty levels (Figure 7). From the figure, we observe that both SWE-Reasoner and OpenAI o1 continue to adaptively allocate more reasoning tokens to increasingly challenging tasks, demonstrating a clear test-time computation scaling phenomenon. Interestingly, we also observed that Claude3.5 Sonnet v2, despite being a model not explicitly trained for inference-intensive computation, exhibited a similar scaling trend, whereas ShortCoT model did not show this behavior clearly. This empirical evidence strongly supports the existence of test-time compute scaling in advanced reasoning models, further validating that our Internal TTC strategy effectively enables dynamic computational resource allocation tailored to task complexity.

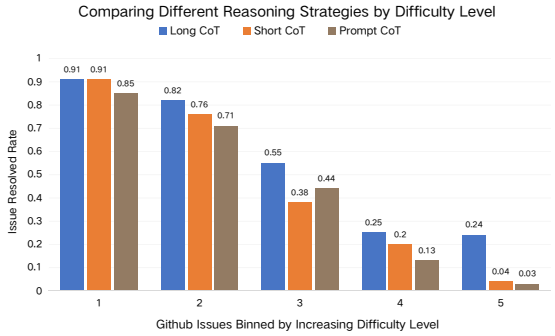

Fig. 6. Comparison of Issue Resolution Rates by Reasoning Strategies across Difficulty Levels. The graph shows the performance of three approaches: Long CoT (SWE-Reasoner), Short CoT (*w/o. Long CoT*), and Prompt CoT (*w/o. All*).

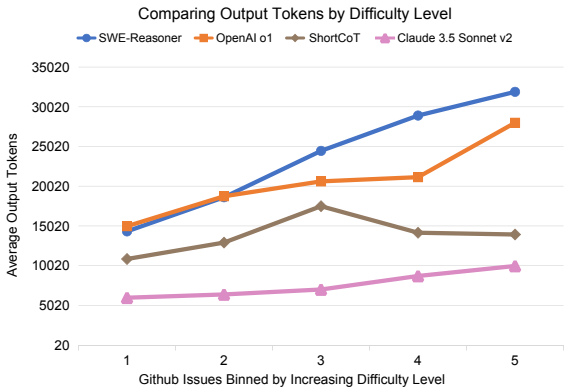

Fig. 7. Average Number of Output Tokens by Difficulty Level. We categorize SWE-bench Verified issues into five difficulty buckets based on their resolution frequency by top-performing agents (bucket 1: resolved by 25–30 agents, bucket 5: resolved by 5–10 agents).

D. Analysis of External TTC Strategies

We further evaluated the effectiveness of our proposed External Test-Time Compute (TTC) strategies, specifically the Development-Process-Based Search Strategy, through two targeted experiments. Due to the computational resources and significant time required for scaling experiments, we randomly sampled 100 issues from the SWE-bench Verified benchmark for these analyses.

Effectiveness of Development-Process-Based Search Strategy. Our first experiment aimed to systematically evaluate the effectiveness of our proposed external search strategy, labeled as **Dev-Search**, against three alternative baselines under varying inference budgets (Generation Budget). Specifically, Dev-Search utilizes our proposed Process Reward Model (PRM)-guided beam search at the repository understanding and fault localization stages, combined with execution-based patch verification and ORM-based final ranking. For budgets of 2 and 4, we set the beam search width to 2; for budget 8, we expanded it to 4. We compared Dev-Search with the following baselines. **Exec** strategy uses only execution verification (regression tests and issue reproduction) to select a

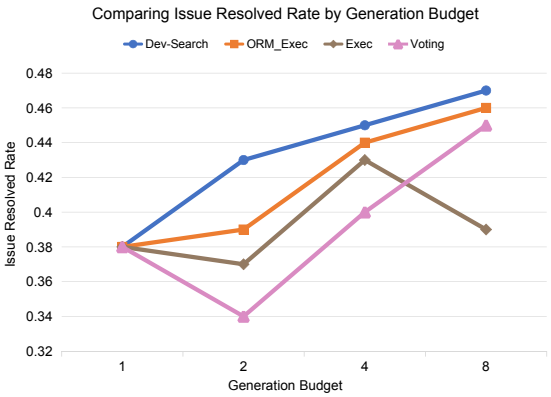

Fig. 8. The comparative issue-resolution rates under various inference budgets (1, 2, 4, and 8 rollouts).

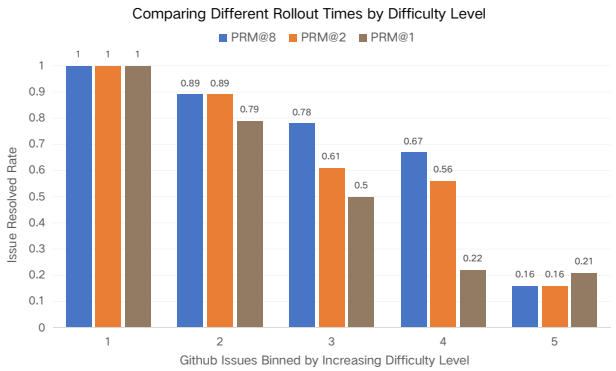

Fig. 9. Comparison of Issue Resolution Rates: By Difficulty Level and Rollout Time.

final patch. If multiple patches passed execution verification, a patch was randomly selected to resolve the tie. In **ORM_Exec** approach, we employed our Outcome Reward Model (ORM) to break ties among multiple patches that passed execution verification, rather than selecting randomly. In **Voting** strategy, following Agentless [4], we normalized patches to abstract syntax tree representations, standardized their format (ignoring comments, extra whitespace, and surface-level differences), and then selected the patch appearing most frequently.

Figure 8 illustrates the comparative issue-resolution rates under various inference budgets (1, 2, 4, and 8 rollouts). We observe several key findings. First, our proposed Dev-Search strategy consistently achieves the highest resolution rate across all budget conditions, clearly demonstrating its overall effectiveness. Moreover, a distinct test-time compute scaling phenomenon emerges, evidenced by steadily improving performance as inference budgets increase. Conversely, the Exec baseline exhibited an unexpected drop in performance at the highest budget (budget=8). A potential explanation for this performance decline is that execution-based verification alone (specifically, reproducing code functionality) might occasionally yield false positives due to limited coverage and incomplete reproducibility, leading to instability when randomly selecting among candidate patches. Importantly, incorporating the ORM-based tie-breaking method in the ORM_Exec variant mitigates this issue, achieving stable improvements with increased budgets.

Influence of Generation Budgets Across Difficulty Levels. Our second experiment analyzed how varying generation budgets impacted agent performance across different issue difficulty buckets. The issues were categorized into five difficulty levels based on their resolution frequency among existing top-ranked agent submissions from the SWE-bench Verified leaderboard (bucket 1: easiest, solved by 25–30 agents; bucket 5: hardest, solved by 5–10 agents). Figure 9 summarizes these results.

As expected, increasing inference budgets consistently improved issue-resolution performance for difficulty levels 1 through 4, particularly notable in difficulty levels 3 and 4. This clearly indicates that additional test-time compute can indeed enhance model performance, allowing the agent to explore broader reasoning trajectories and effectively handle moderately challenging problems. However, at the highest difficulty level (bucket 5), we observed a slight reduction in resolution accuracy when using higher inference budgets. This counterintuitive finding suggests that, for extremely challenging tasks, the effectiveness of external compute strategies may reach inherent limitations imposed by the model's reasoning capabilities. In other words, beyond certain complexity thresholds, merely allocating more computational budget to external search may offer limited gains without commensurate improvements in underlying model reasoning abilities. Future work should explore combining external compute strategies with complementary internal training improvements to further extend effectiveness on highly challenging tasks.

IV. Related Works

A. LLM-based Software Engineering Agents

Generative models have exhibited significant capabilities in code generation. These models have substantially impacted various aspects of software engineering, enabling tasks such as code generation [24, 25, 26, 27, 28, 29, 30, 31], test generation [32, 33, 34, 35], and code editing and refactoring [37, 38, 39, 40, 41]. In recent years, AI agents have significantly advanced ASE [42]. These agents enhance project-level SE tasks by integrating diverse capabilities, such as awareness of the running environment [43, 44, 45, 46], structured planning and reasoning [44, 47, 48], and leveraging external tools [49, 50, 51, 52, 53]. Devin [47] notably introduced a milestone end-to-end ASE framework, capable of autonomously planning requirements, utilizing tools such as code editors, terminals, and search engines, and ultimately generating functional code to fulfill user specifications. Its promising capabilities have sparked significant attention within the SE community, inspiring subsequent works, such as SWE-Agent [3], AutoCodeRover [2], and RepoUnderstander [54]. Recently, SWE-SynInfer [10] provided an effective open-

source framework to systematically handle software issues, decomposing issue resolution into stages of repository understanding, fault localization, and patch generation. Building upon this framework, we propose *SWE-SynInfer+*, an enhanced version introducing an explicit patch verification phase, where reproduction code is generated to automatically verify and iteratively refine candidate solutions. Besides, a major limitation across existing ASE agents remains their heavy reliance on larger models, which restricts accessibility in real-world deployments. Our work directly addresses this limitation by proposing a scalable inference-time compute framework, explicitly designed to strengthen open-source ASE agents through enhanced reasoning depth and systematic exploration of candidate solutions.

B. Training Software Agents

Recent advancements have demonstrated the significant potential of leveraging LLMs to tackle complex SE tasks. Existing approaches rely predominantly on proprietary or resource-intensive models such as OpenAI o1 [6] or DeepSeek R1 [8], achieving strong results but facing barriers [55], such as model accessibility, data transparency, and deployment costs. Efforts have begun to develop open-source alternatives explicitly tailored for emerging SWE tasks. For instance, Lingma-SWEGPT [10] proposes iterative, development-process-centric methods and introduces open model variants derived from Qwen2.5 [16], achieving improved performance on SWE-bench. SWE-Gym [12] further advances open-source SWE agent training by providing an environment designed to enhance the Qwen2.5-Coder series (7B and 32B) on SWE-bench tasks. Similarly, SWE-Fixer [56] fine-tunes Qwen2.5 models into specialized retrievers and editors for more efficient issue resolution. SWE-RL [17] uses reinforcement learning to improve the Llama model and achieve better issue resolution. In contrast, our work proposes a distinct approach focused explicitly on scalable inference-time compute (TTC) rather than merely scaling model size. Our framework achieves superior or comparable performance to existing state-of-the-art models, while significantly reducing computational demands and resource constraints.

C. Scaling Test-Time Compute

Recent advancements in software engineering agents, leverage external tools like parallel trajectory generation, voting mechanisms [4], and execution verification [17, 57] to enhance solution quality. For example, SWE-Gym [12] trains an ORM to select the highest-scoring trajectory from parallel generations, while Agentless [4] employs a voting mechanism to normalize and rank candidate patches, choosing the most frequent one. Although effective, these methods do not address intermediate steps in the workflow. Extensions like CodeMonkeys [57] and SWE-RL [17] generate reproduction code to validate patch correctness, offering functional feedback. Similarly, Nebius [18] introduces PRMs to guide action selection. Yet, these frameworks still emphasize either trivial or final actions rather than systematically addressing all critical stages of development. Moreover, they fail to explore the potential of internal TTC to dynamically scale reasoning capabilities. Our work bridges these gaps by proposing a novel framework that integrates targeted search at three pivotal development phases alongside Long CoT training. To the best of our knowledge, this is the first empirical demonstration of test-time scaling within software engineering agents.

V. Limitation and Threats to Validity

While SWE-Reasoner 32B (TTC) demonstrates promising results in automated software improvement, it is important to acknowledge several limitations that affect both the current approach and the broader generalizability of our findings:
Inference Efficiency. Although test-time compute scaling substantially improves model performance, it can also degrade inference efficiency, particularly for interactive tasks such as real-time code completion or conversational code assistance. In contrast, for end-to-end software issue resolution where short delays are acceptable, test-time scaling remains practical. We also observe that SWE-Reasoner and OpenAI-o1 partially adapt their reasoning depth to problem complexity, suggesting that future work could explore more fully adaptive inference-time mechanisms—automatically adjusting the extent of reasoning based on task difficulty or runtime constraints.

VI. Conclusion

In this work, we introduced a unified Test-Time Compute (TTC) scaling framework to enhance the code reasoning capabilities of software engineering agents using personally deployable open-source LLMs. Internally, we proposed a development contextualized trajectory synthesis method, leveraging realistic multi-stage reasoning trajectories extracted from high-quality GitHub repositories. This method, combined with repository-aware rejection sampling, significantly improves the model's internal reasoning capabilities. Externally, we developed a development-process-based search strategy that focuses computational resources at critical decision-making points, utilizing specialized reward models and execution verification to efficiently prune less promising trajectories. Evaluations conducted on the challenging SWE-bench Verified demonstrate that our 32B model, with targeted inference-time scaling, achieves a state-of-the-art 46% issue resolution rate, outperforming larger models. Additionally, we provided the first empirical validation of the test-time scaling phenomenon within SWE agents, revealing effective dynamic allocation of computational resources to address increasingly complex software engineering tasks.

References

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[2] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International*

Symposium on Software Testing and Analysis, 2024, pp. 1592–1604.

[3] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[4] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[5] Wei Ming T. (2025) Gpu system requirements for running deepseek-r1. [Online]. Available: https://apxml.com/posts/gpu-requirements-deepseek-r1

[6] A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney *et al.*, "Openai o1 system card," *arXiv preprint arXiv:2412.16720*, 2024.

[7] OpenAI. (2025) Openai o3-mini system card. [Online]. Available: https://cdn.openai.com/o3-mini-system-card-feb10.pdf

[8] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[9] OpenAI. (2024) Introducing swe-bench verified. [Online]. Available: https://openai.com/index/introducing-swe-bench-verified/

[10] Y. Ma, R. Cao, Y. Cao, Y. Zhang, J. Chen, Y. Liu, Y. Liu, B. Li, F. Huang, and Y. Li, "Lingma swe-gpt: An open development-process-centric language model for automated software improvement," *arXiv preprint arXiv:2411.00622*, 2024.

[11] Meta. (2024) Introducing llama 3.1. [Online]. Available: https://ai.meta.com/blog/meta-llama-3-1/

[12] J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang, "Training software engineering agents and verifiers with swe-gym," 2024. [Online]. Available: https://arxiv.org/abs/2412.21139

[13] Y. Dong, X. Jiang, H. Liu, Z. Jin, B. Gu, M. Yang, and G. Li, "Generalization or memorization: Data contamination and trustworthy evaluation for large language models," in *ACL (Findings)*. Association for Computational Linguistics, 2024, pp. 12 039–12 050.

[14] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. Von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," *arXiv preprint arXiv:2308.07124*, 2023.

[15] I. Bouzenia and M. Pradel, "You name it, i run it: An llm agent to execute tests of arbitrary projects," *arXiv preprint arXiv:2412.10133*, 2024.

[16] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[17] Y. Wei, O. Duchenne, J. Copet, Q. Carbonneaux, L. Zhang, D. Fried, G. Synnaeve, R. Singh, and S. I. Wang, "Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution," *arXiv preprint arXiv:2502.18449*, 2025.

[18] A. Golubev, S. Polezhaev, K. Zainullina, M. Trofimova, I. Badertdinov, Y. Anapolskiy, D. Litvintseva, S. Karasik, F. Fisin, S. Skvortsov, M. Nekrashevich, A. Shevtsov, S. Abramov, and B. Yangel, "Leveraging training and search for better software engineering agents," *Nebius blog*, 2024, https://nebius.com/blog/posts/training-and-search-for-software-engineering-agents.

[19] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, pp. 53 728–53 741, 2023.

[20] OpenAI. (2025) Openai gpt-4.5 system card. [Online]. Available: https://openai.com/index/gpt-4-5-system-card/

[21] Anthropic. (2024) Introducing claude 3.5 sonnet. [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[22] ——. (2025) Claude 3.7 sonnet and claude code. [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet

[23] Z. Ma, C. Peng, P. Gao, X. Meng, Y. Zou, and B. Xie, "Sorft: Issue resolving with subtask-oriented reinforced fine-tuning," *arXiv preprint arXiv:2502.20127*, 2025.

[24] Y. Ma, Y. Liu, Y. Yu, Y. Zhang, Y. Jiang, C. Wang, and S. Li, "At which training stage does code data help llms reasoning?" *arXiv preprint arXiv:2309.16298*, 2023.

[25] Z. Jiang, H. Xiong, Y. Ma, Y. Zhang, Y. Ding, Y. Xiong, and S. Li, "Automatic code annotation generation based on heterogeneous graph structure," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 497–508.

[26] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 182:1–182:30, 2024.

[27] Z. Pan, R. Cao, Y. Cao, Y. Ma, B. Li, F. Huang, H. Liu, and Y. Li, "Codev-bench: How do llms understand developer-centric code completion?" *arXiv preprint arXiv:2410.01353*, 2024.

[28] Q. Zhu, J. Cao, Y. Lu, H. Lin, X. Han, L. Sun, and S.-C. Cheung, "Domaineval: An auto-constructed benchmark for multi-domain code generation," *arXiv preprint arXiv:2408.13204*, 2024.

[29] R. Xu, J. Cao, Y. Lu, H. Lin, X. Han, B. He, S.-C. Cheung, and L. Sun, "Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution," *arXiv preprint arXiv:2408.13001*, 2024.

[30] Y. Tian, W. Yan, Q. Yang, Q. Chen, W. Wang, Z. Luo, and L. Ma, "Codehalu: Code hallucinations in llms driven by execution-based verification," *arXiv preprint arXiv:2405.00253*, 2024.

[31] Y. Zhao, Z. Luo, Y. Tian, H. Lin, W. Yan, A. Li, and J. Ma, "Codejudge-eval: Can large language models

be good judges in code understanding?" *arXiv preprint arXiv:2408.10718*, 2024.

[32] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[33] Z. Xue, Z. Gao, S. Wang, X. Hu, X. Xia, and S. Li, "Selfpico: Self-guided partial code execution with llms," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1389–1401.

[34] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[35] M. Li, D. Li, J. Liu, J. Cao, Y. Tian, and S.-C. Cheung, "Dllens: Testing deep learning libraries via llm-aided synthesis," *arXiv preprint arXiv:2406.07944*, 2024.

[36] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, pp. 77:1–77:22, 2025.

[37] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[38] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 151–160.

[39] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, and A. Ouni, "How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 202–206.

[40] M. Zhang, Y. Tian, Z. Xu, Y. Dong, S. H. Tan, and C. Sun, "Lampr: Boosting the effectiveness of language-generic program reduction via large language models," *arXiv preprint arXiv:2312.13064*, 2023.

[41] ——, "Lpr: Large language models-aided program reduction," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 261–273.

[42] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 189:1–189:38, 2024.

[43] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

[44] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, "Executable code actions elicit better llm agents," 2024.

[45] J. Kong, M. Cheng, X. Xie, S. Liu, X. Du, and Q. Guo, "Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs," *arXiv preprint arXiv:2403.01971*, 2024.

[46] Y. Xie, Z. Jia, S. Li, Y. Wang, J. Ma, X. Li, H. Liu, Y. Fu, and X. Liao, "How to pet a two-headed snake? solving cross-repository compatibility issues with hera," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 694–705.

[47] Cognition. (2023) Introducing devin. [Online]. Available: https://www.cognition.ai/introducing-devin

[48] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang *et al.*, "Repoagent: An llm-powered open-source framework for repository-level code documentation generation," *arXiv preprint arXiv:2402.16667*, 2024.

[49] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," *arXiv preprint arXiv:2401.07339*, 2024.

[50] C. Lee, C. S. Xia, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via llm-based multi-agent synergy," *arXiv preprint arXiv:2404.17153*, 2024.

[51] Z. Xue, Z. Gao, X. Hu, and S. Li, "Acwrecommender: A tool for validating actionable warnings with weak supervision," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1876–1880.

[52] Y. Ma, Y. Yu, S. Li, Z. Jia, J. Ma, R. Xu, W. Dong, and X. Liao, "Mulcs: Towards a unified deep representation for multilingual code search," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 120–131.

[53] X. Huang, Y. Ma, H. Zhou, Z. Jiang, Y. Zhang, T. Wang, and S. Li, "Towards better multilingual code search through cross-lingual contrastive learning," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 22–32.

[54] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, "How to understand whole software repository?" *arXiv preprint arXiv:2406.01422*, 2024.

[55] Y. Dong, X. Jiang, Y. Tao, H. Liu, K. Zhang, L. Mou, R. Cao, Y. Ma, J. Chen, B. Li, Z. Jin, F. Huang, Y. Li, and G. Li, "RL-PLUS: countering capability boundary collapse of llms in reinforcement learning with hybrid-policy optimization," *CoRR*, vol. abs/2508.00222, 2025.

[56] C. Xie, B. Li, C. Gao, H. Du, W. Lam, D. Zou, and K. Chen, "Swe-fixer: Training open-source llms for effective and efficient github issue resolution," *arXiv preprint arXiv:2501.05040*, 2025.

[57] R. Ehrlich, B. Brown, J. Juravsky, R. Clark, C. Ré, and A. Mirhoseini, "Codemonkeys: Scaling test-time compute for software engineering," *arXiv preprint arXiv:2501.14723*, 2025.