SWE-GPT: A Process-Centric Language Model for Automated Software Improvement

YINGWEI MA, RONGYU CAO, YONGCHANG CAO, YUE ZHANG, JUE CHEN, YIBO LIU, YUCHEN LIU, BINHUA LI, FEI HUANG, and YONGBIN LI*, Tongyi Lab, Alibaba Group, China

Large language models (LLMs) have demonstrated remarkable performance in code generation, significantly enhancing the coding efficiency of developers. Recent advancements in LLM-based agents have led to significant progress in end-to-end automatic software engineering (ASE), particularly in software maintenance (e.g., fixing software issues) and evolution (e.g., adding new features). Despite these encouraging advances, current research faces two major challenges. First, state-of-the-art performance primarily depends on closed-source models like GPT-4, which significantly limits the technology's accessibility, and potential for customization in diverse software engineering tasks. This dependence also raises concerns about data privacy, particularly when handling sensitive codebases. Second, these models are predominantly trained on static code data, lacking a deep understanding of the dynamic interactions, iterative problem-solving processes, and evolutionary characteristics inherent in software development. Consequently, they may face challenges in navigating complex project structures and generating contextually relevant solutions, which can affect their practical utility in real-world scenarios.

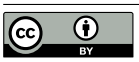
To address these challenges, our study adopts a software engineering perspective. We recognize that real-world software maintenance and evolution processes encompass not only static code data but also developers' thought processes, utilization of external tools, and the interaction between different functional personnel. Our objective is to develop an open-source large language model specifically optimized for software improvement, aiming to match the performance of closed-source alternatives while offering greater accessibility and customization potential. Consequently, we introduce the **Lingma SWE-GPT** series, comprising Lingma SWE-GPT 7B and Lingma SWE-GPT 72B. By learning from and simulating real-world code submission activities, Lingma SWE-GPT systematically incorporates the dynamic interactions and iterative problem-solving inherent in software development process—such as repository understanding, fault localization, and patch generation—thereby achieving a more comprehensive understanding of software improvement processes. We conducted experimental evaluations using SWE-bench-Verified benchmark (comprising 500 real GitHub issues), recently proposed by OpenAI. The results demonstrate that **Lingma SWE-GPT 72B successfully resolves 30.20% of the GitHub issues**, marking a significant improvement in automatic issue resolution (22.76% relative improvement compared to Llama 3.1 405B), approaching the performance of closed-source models (31.80% issues of GPT-4o resolved). Notably, Lingma SWE-GPT 7B resolves 18.20% of the issues, surpassing the 17.20% resolution rate of Llama 3.1 70B, highlighting the potential for applying smaller models to ASE tasks.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Multi-agent planning**.

Additional Key Words and Phrases: Automatic Software Engineering (ASE), Large Language Models (LLMs), Software Engineering Agents, Fault Localization, Automated Program Repair

*Corresponding Author.

Authors' Contact Information: Yingwei Ma; Rongyu Cao; Yongchang Cao; Yue Zhang; Jue Chen; Yibo Liu; Yuchen Liu; Binhua Li; Fei Huang; Yongbin Li, mayingwei.myw@alibaba-inc.com, Tongyi Lab, Alibaba Group, Beijing, China.



ACM 2994-970X/2025/7-ARTISSTA104
https://doi.org/10.1145/3728981

ACM Reference Format:
Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2025. SWE-GPT: A Process-Centric Language Model for Automated Software Improvement. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA104 (July 2025), 22 pages. https://doi.org/10.1145/3728981

1 Introduction

Automated software engineering (ASE) has long been a vision pursued by both the software engineering (SE) and artificial intelligence (AI) communities. Recent advancements in large language models (LLMs) have shown significant potential in advancing this field. Initially, the HumanEval benchmark [9] was developed to assess LLMs' capabilities in function-level code generation. Both closed-source (e.g., GPT-4o [39]) and open-source models (e.g., Llama 3.1 405B [36]) have performed well on these tasks, solving more than 90% of problems. However, function-level code generation represents only a fraction of the challenges encountered in real-world software development.

To evaluate LLMs' capabilities in more realistic scenarios, the SWE-bench benchmark [19] series was introduced. The evaluation process in SWE-bench is designed to simulate real-world software improvement tasks: given a natural language description of an issue and the corresponding Github repository, the model is expected to generate a patch that resolves the issue. This approach tests not only code generation capabilities but also the model's ability to understand real-world software issue, locate relevant code across multiple files, and make appropriate modifications while maintaining the integrity of the entire codebase. In response to these challenges, both SE and AI communities have focused on developing sophisticated software engineering agents [8, 11, 29, 34]. Systems such as SWE-agent [63] and AutoCodeRover [70] exemplify these approaches, leveraging the general capabilities of LLMs to iteratively and autonomously formulate plans, execute tool calls, and observe feedback. For example, SWE-agent employs an agent-computer interface to execute operations like opening files, searching code lines, and running tests. In contrast, approaches like Agentless [55] utilize LLMs within predefined workflows that guide the problem-solving process through fixed, expert-designed steps. While this method limits the autonomy and generality of LLM, it also shows promising results by effectively structuring tasks. However, our analysis reveals that current ASE approaches still face significant limitations due to two primary factors.

Over-reliance on Closed-Source Models. The top-performing submissions on the SWE-bench [19] leaderboard are exclusively based on closed-source models like GPT-4/4o [1, 39] and Claude 3.5 Sonnet [3]. In contrast, submissions utilizing open-source models, such as SWE-Llama 13B, solve only 1.00% of the problems. Although recent research has shown progress in open-source model performance—with Qwen2 72B instrut achieving 9.34% on SWE-bench Lite [28]—this advancement, while encouraging, still lags significantly behind closed-source models. Moreover, considering the privacy of user code repositories in software development, utilizing closed-source models for code analysis and modification may raise security concerns. This factor restricts the applicability of closed-source models in real-world software engineering scenarios, highlighting the urgent need for high-performance open-source models.

Lack of Comprehensive Development Process Data. General-purpose and code-specific large models are typically trained on vast amounts of static code data [17, 36, 75]. While this approach has significantly enhanced their code generation capabilities, it overlooks the dynamic interactions and development process crucial to comprehensively understanding real-world software engineering practices. Real-world software improvement encompasses a complex reasoning cycle of issue identification, tool utlization, code navigation, and iterative problem-solving, which is not captured by merely training on static codebases. Although some existing models [30, 37] utilize pull request submission processes recorded on GitHub, these data only include commit messages and final patches, lacking the detailed thought processes and reasoning of developers during problem-solving.

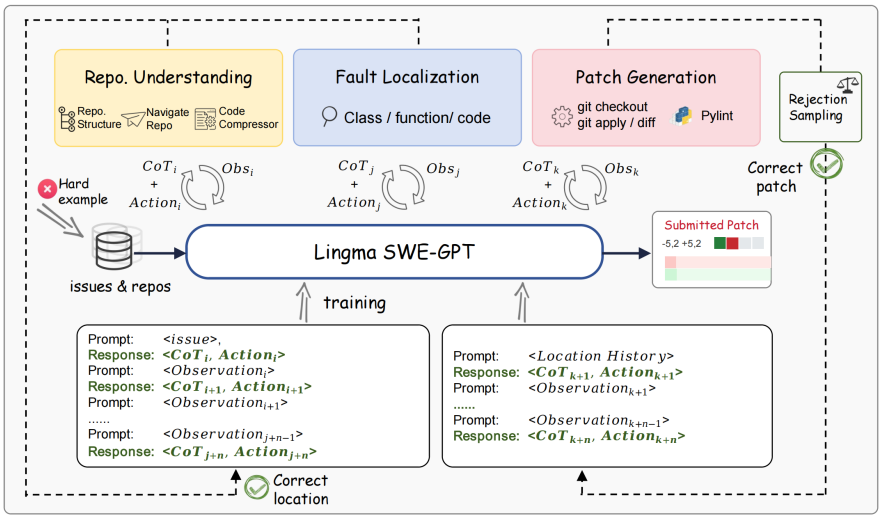


Fig. 1. Overview of an iterative development-process training framework for automated software improvement.

Furthermore, current training methods primarily focus on isolated code snippets or single files, neglecting broader project structures and cross-file information. This results in models lacking a global perspective when dealing with complex software systems, making it challenging to accurately understand and handle cross-module interactions.

Our Approach. To address these challenges, we introduce the Lingma SWE-GPT series, which includes models of two size, 7B and 72B, specifically designed for software improvement. As shown in Figure 1, our approach simulates the software improvement process through a three-stage process: repository understanding, fault localization, and patch generation. Each stage operates in a Chain-of-Thought (CoT) manner. To address real-world GitHub issues, Lingma SWE-GPT initially conducts a comprehensive analysis of the software repository, examining the codebase hierarchically from the overall file structure to specific classes and functions. Specifically, Lingma SWE-GPT identifies a set of potentially relevant files based on the natural language description of the issue and the repository's directory tree structure. It then identifies relevant classes and functions based on the skeletons of these files and formulates a plan for issue resolution. Following this repository understanding, Lingma SWE-GPT retrieves pertinent code context by invoking specialized search APIs (e.g., search_func(*'resize'*)). These APIs, leveraging Abstract Syntax Tree analysis, extract contextual information such as method and class implementations from the codebase. The model iteratively refines its understanding of both the issue and the repository, strategically selecting which APIs to use in subsequent iterations until potential fault locations are identified. In the concluding phase, Lingma SWE-GPT generates and applies patches to address the localized issues. This phase encompasses concrete solution generation, code replacement, and iterative debugging processes based on syntax validation and git operations, ensuring the development of applicable patches. To further enhance the model's capabilities, we employ a development process-centric iterative training strategy. This involves optimizing model performance through maximizing the conditional probability of generating development process outputs, including thought processes, tool utilization record and final results. By incorporating curriculum learning, the model progressively tackles increasingly complex tasks, establishing a robust foundation on simpler ones. Additionally, we

implement a rejection sampling process to ensure the quality of synthesized data, selectively retaining high-quality instances that closely mimic real-world software development practices.

Our extensive evaluation on SWE-bench Verified and SWE-bench Lite demonstrates the effectiveness of Lingma SWE-GPT: the 72B version successfully resolves 30.20% of issues on SWE-bench Verified, marking a significant improvement over existing open-source models(22.76% relative improvement compared to Llama 3.1 405B) and approaching the performance of leading closed-source alternatives (31.80% issues of GPT-4o resolved). Additionally, the 7B version achieves an impressive 18.20% success rate, surpassing the 17.20% resolution rate of Llama 3.1 70B, showcasing the potential of smaller, more efficient models in automated software engineering tasks.

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

- We introduce Lingma SWE-GPT, a novel series of open-source large language models specifically optimized for automated software improvement[1].
- We propose a development process-centric training approach that captures the dynamic nature of software engineering, including tool utilizing, reasoning, and interactive problem-solving capabilities.
- We demonstrate the effectiveness of our approach through comprehensive evaluations on SWE-bench Lite and Verified, showing significant improvements over existing open-source models and competitive performance against closed-source alternatives.
- We provide insights into the model's fault localization capabilities and performance consistency, offering valuable directions for future research in AI-assisted software engineering.

2 Related Work

2.1 Large Language Models for Code

Generative models such as ChatGPT have exhibited significant capabilities in code generation and comprehension. These models have substantially impacted various aspects of software engineering, enabling tasks such as code generation [18, 33, 42, 52, 58, 71, 74] from natural language requirements, test generation [24, 27, 56, 60], and code editing and refactoring [2, 7, 23, 48, 68, 69]. Furthermore, developers and researchers have applied these models to more complex software engineering tasks, including code translation [13, 25, 41], debugging [10, 12, 38, 47, 61], and automated program repair [19, 22, 70].

The effectiveness of these models can be attributed to their pretraining on extensive general-purpose data and open-source code repositories. This extensive training has facilitated the development of sophisticated code generation and reasoning capabilities. Notable examples of such models include GPT-4 [1], Claude 3.5 Sonnet [3], CodeX [9], Code Llama [45], StarCoder [30], DeepSeek-Coder [75], Qwen2.5 Coder [17], and CodeGemma [49]. These models have shown considerable proficiency in comprehending and generating code across various programming languages and paradigms. In addition to pretrained LLMs, researchers have developed instruction-tuned models specifically tailored for code-related tasks. Examples of such models include CodeLlama-Instruct [45], Pangu-coder [46], WizardCoder [32], Magicoder [54], WaveCoder [64] and Opencodeinterpreter [72]. These models undergo additional fine-tuning with carefully curated instructions, enhancing their performance on specific downstream code-related tasks. Despite their advanced capabilities, current LLMs are primarily trained on static code data, limiting their understanding of the dynamic interactions nature of software development. This paper proposes developing models that can simulate these dynamic aspects, including reasoning about tool usage and mimicking thought processes, to better address the challenges of real-world software engineering tasks.

[1] https://github.com/LingmaTongyi/Lingma-SWE-GPT

2.2 LLM-based Software Engineering Agents

In recent years, LLM-based AI agents have advanced the development of automatic software engineering (ASE). AI agents improve the capabilities of project-level software engineering tasks through running environment awareness [14, 20, 53, 57], planning & reasoning [11, 31, 53], and tool construction [15, 21, 35, 59, 66]. Surprisingly, Devin [11] is a milestone that explores an end-to-end LLM-based agent system to handle complex SE tasks. Concretely, it first plans the requirements of users, then adopts the editor, terminal and search engine tools to make independent decisions and reasoning, and finally generates codes to satisfy the needs of users in an end-to-end manner. Its promising designs and performance swiftly ignited unprecedented attention from the SE and AI community to automatic software engineering (ASE) [8, 26, 28, 29, 34, 55, 63, 70, 73]. For example, SWE-agent [63] carefully designs an Agent Computer Interface (ACI) to empower the SE agents capabilities of creating & editing code files, navigating repositories, and executing programs. Besides, AutoCodeRover [70] extracts the abstract syntax trees in programs, then iteratively searches the useful information according to requirements and extracted ASTs, and eventually generates program patches. RepoUnderstander [34] develops an exploration strategy based on the Monte Carlo Tree Search algorithm for software repository understanding and fault localization. While these agents have made significant strides in advancing ASE, it is important to note that the majority of these systems rely heavily on closed-source models. Open-source alternatives, while more accessible, have generally struggled to match the performance of their closed-source counterparts in complex software engineering tasks. Our work addresses this critical gap by developing an open-source model that aims to bridge the performance divide.

2.3 Evaluation of Real-world Software Engineering Tasks

Benefiting from the strong general capability of LLMs, LLM-based software engineering agents can handle many important SE tasks, e.g., code generation [53] and code debugging [14]. More recently, SWE-bench team[19, 63] develop a unified dataset named SWE-bench to evaluate the capability of the agent system to resolve real-world GitHub issues automatically. Specifically, it collects the task instances from real-world GitHub issues from twelve repositories. Consistent with previous evaluation methods, SWE-bench is based on the automatic execution of the unit tests. Differently, the presented test set is challenging and requires the agents to have multiple capabilities simultaneously, including repository navigation, fault locating, debugging, code generation and program repairing, so as to solve a given issue end-to-end. Besides, SWE-bench Lite [5] and SWE-bench Verified [40] are subsets of SWE-bench, and they have a similar diversity and distribution of repositories as the original version. Due to the smaller test cost and more detailed human filtering, SWE-bench Lite and Verified are officially recommended as the benchmark of LLM-based SE agents. Therefore, consistent with previous methods [55, 63, 70], we report our performance on SWE-bench Lite and SWE-bench Verified.

2.4 Large Language Model for Code Generation

Generative models such as ChatGPT [1] and Claude [3] have exhibited significant capabilities in code comprehension and generation. The effectiveness of these models can be attributed to their pre-training on extensive general-purpose data and code repositories. Notable examples of such models include CodeX [9], CodeLlama [45], StarCoder [30], DeepSeek-Coder [75], Qwen2.5 Coder [17], and CodeGemma [49]. In addition to pretrained LLMs, researchers have developed instruction-tuned models specifically tailored for code-related tasks. Examples of such models include CodeLlama-Instruct [45], WizardCoder [32], Magicoder [54], WaveCoder [64] and Opencodeinterpreter [72].

These models undergo additional fine-tuning with carefully curated instructions, enhancing their performance on specific downstream code-related tasks.

2.5 Trustworthy code generation

Trustworthy Code Generation. An emerging concern is that LLM-generated code may inadvertently violate security best practices. Because these models learn from massive public code repositories—many of which include buggy or outdated patterns—they can propagate known vulnerabilities or poor coding habits. In response, researchers have begun exploring security-focused code generation techniques. For instance, SaferCode [ref] augments LLM training with carefully curated secure code datasets or explicitly fine-tunes models to avoid unsafe coding patterns. Other work [ICL] proposes incorporating security examples directly into prompts (in-context learning), demonstrating that such guidance can indeed reduce vulnerabilities in generated outputs.

Beyond security issues, VersiCode [xx] introduces a version-specific benchmark to evaluate how models handle API changes across different releases, revealing that version-controllable code generation poses significant challenges. Failing to adapt can lead to functional errors and long-term maintenance issues. Similarly, SecuCoGen [xx] systematically analyzes deprecated API usage in LLM-based code completion. Testing seven advanced models on over 28k prompts and 145 API changes, they observe a substantial rate of outdated suggestions, indicating that many LLMs fail to migrate to newer APIs. These limitations underscore the need for vigilant oversight and continuing refinement; despite their remarkable progress, LLM-based code generation have yet to meet developers' expectations for consistently secure and up-to-date code. Our work addresses these gaps by providing a robust framework for unlearning undesired knowledge—ranging from vulnerable code patterns to deprecated APIs—without sacrificing the model's overall utility.

3 Lingma SWE-GPT

In this section, we present our novel approach for training LLM to perform automated program improvement. Our method comprises three main phases: **issue data collection** (Figure 2), **development process data synthesis** (Figure 3) and **model training** (Figure 1). Leveraging an instruction-tuned model as the foundation model, our approach begins with the input of an issue description and the associated project codebase, then engages in a multi-stage workflow mimicking expert programmers' cognitive processes. This workflow consists of three key stages: repository understanding, fault localization, and patch generation. In the repository understanding stage, the model analyzes the repository structure, navigates the codebase, and utilizes a code compressor to grasp relevant project files and code snippets. The fault localization stage builds upon this understanding to identify potential fault locations at the class, function, or code level. Finally, in the patch generation stage, the model generates and applies patches using git-related operations. Throughout these stages, the model operates in a Chain-of-Thought (CoT) manner, where each step outputs a reasoning process (CoT_i) and an action ($Action_i$). This action is then executed, generating an observation (Obs_i). Based on this observation, the model proceeds to the next step (CoT_{i+1} + $Action_{i+1}$), creating an iterative feedback loop for continuous refinement.

3.1 Issue Data Collection

The foundation of our approach lies in leveraging high-quality, real-world software development data. GitHub Issues serve as valuable resources for understanding bugs, feature requests, and enhancements, often providing natural language descriptions of problems or desired functionalities. Pull Requests (PRs) contain the corresponding code changes, including commit histories and code diffs, which directly address the issues raised. By leveraging this combination of data, we can simulate real-world programming scenarios, capturing the context and reasoning behind code

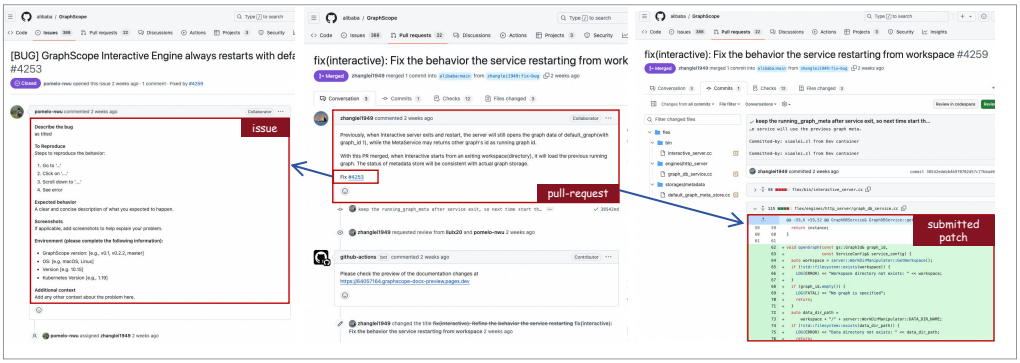


Fig. 2. Example of issue and corresponding pull-request data collection process.

modifications. To effectively train SWE-GPT for automatic program improvement, we constructed a comprehensive dataset by collecting issues, corresponding PRs, and codebases from public GitHub repositories. Below we will describe our data collection process in detail.

First, we selected Github repositories with at least 50 stars on GitHub, thereby ensuring a basic level of community recognition and activity. To avoid potential data leakage, we carefully filtered out any repositories that overlapped with those used in SWE-bench [19]. For each selected repository, we retrieved all issues and their linked PRs, focusing specifically on those PRs that had been merged by the developers (as illustrated in Figure 2). Specifically, we utilized GitHub's API to fetch PRs with the state "merged" and their associated issues with the state "closed", ensuring that we captured only completed and approved code changes. Additionally, we stored snapshots of the codebase at the time of the PR to provide sufficient context for the code changes.

To further ensure the quality of the issues and PRs, we applied a set of heuristic filtering rules, similar to the approach used in OctoPack [37]. **For issues**, we retained only those with textual descriptions containing at least 20 characters, thus excluding trivial or insufficiently detailed issues. Additionally, to avoid issues that primarily reference external resources without providing adequate context, we filtered out those with more than three hyperlinks. Lastly, we retained only issues with at least 80% English content in their textual descriptions, to maintain language consistency across the dataset. **For PRs**, we applied two main criteria. First, we selected PRs that involved modifications to between one and five code files. This ensured that each PR represented a substantive but not overly complex change. Second, we excluded PRs that only modified test files, focusing instead on changes that directly impacted the primary functionality of the codebase.

After applying these filtering criteria, we constructed a dataset consisting of approximately 90,000 PRs collected from 4,000 repositories. Notably, the codebases we processed were typically of substantial size, often containing hundreds of files, reflecting the complexity inherent in real-world software improvements.

3.2 Development Process Data Synthesis

Building upon the comprehensive issue data collection, we introduce a novel data synthesis process designed to capture the dynamic nature of software development. This process addresses the limitations of current LLM-based approaches by simulating the complex, interactive aspects of real-world software maintenance and evolution. Figure 1 illustrates our data synthesis approach, with its algorithm detailed in Algorithm 1. This approach mimics the cognitive workflow of expert programmers across three key stages: Repository Understanding, Fault Localization, and Patch

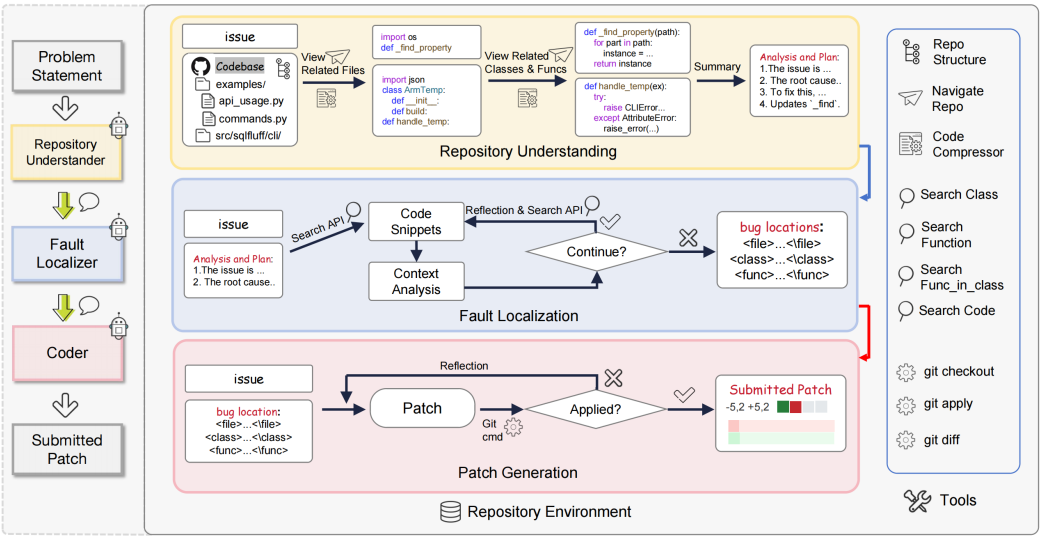


Fig. 3. The overview of three-stage data synthesis and inference workflow.

Generation. Each stage involves a series of Chain-of-Thought reasoning steps, allowing the model to iteratively refine its understanding and output to the given issue.

Figure 3 illustrates our three-stage SoftWare Engineering process data Synthesis and Inference workflow(line 9-18, Algo. 1), named SWESynInfer. This workflow extends the publicly available AutoCodeRover [70] framework. AutoCodeRover provides baseline processes for context retrieval and patch generation stages, our work further introduces crucial enhancements to more accurately simulate the cognitive processes of expert developers.

Repository Understanding. This stage focuses on comprehending the structure and content of a code repository, crucial for establishing the context necessary for issue resolution. The Repository Understanding Agent (RepoUer) employs a hierarchical approach inspired by Agentless [55], utilizing three key tools: Repo Structure tool, Navigate Repo tool, and Code Compressor tool (see Figure 3). First, RepoUer uses the Repo Structure tool to create a concise representation of the repository's directory structure. This tool transforms the directory structure into a tree format suitable for LLM input, starting from the repository's root folder and organizing code files and folder names. Files and folders at the same directory level are vertically aligned, with subdirectory contents indented. Next, the Navigate Repo tool is employed to traverse the repository and locate potentially relevant files and elements. RepoUer leverages the issue description and the generated directory tree to identify N potentially relevant files. The Code Compressor tool then transforms these files into a skeleton format. This compressed representation preserves global references, class definitions, function signatures, associated comments, and variable declarations, effectively reducing the overall context length while maintaining essential structural information. Using these compressed file representations, RepoUer further refines its search to identify specific classes or functions that are potentially relevant to the issue. Finally, RepoUer analyzes the collected information and formulates a comprehensive plan for issue resolution. This plan typically includes: a restatement of the issue, an analysis of the root cause, and proposed steps to fix the issue. This multi-step process emulates a developer's initial exploration of a project, providing a comprehensive understanding of the codebase's architecture and relevant components.

Fault Localization. Building on insights from the repository understanding stage, the Fault Localization phase aims to identify specific problem areas within the codebase. This phase employs the Fault Localizer (FLer), which simulates the diagnostic steps developers take when resolving issues, combining tool utlization with iterative problem-solving. FLer initially uses specialized search APIs to retrieve relevant code snippets at the file, class, function, and snippet levels. These APIs (see Tools in Figure 3), informed by the repository summary and issue description, extract contextual information necessary for pinpointing potential fault locations. FLer then systematically evaluates the retrieved code snippets, performing context analysis to understand the relationships and dependencies within the codebase. Finally, it assesses whether the fault location is identified and decides whether to continue searching or proceed to the patch generation phase. FLer continues this cycle until it either successfully identifies the fault locations or reaches a iteration limit. FLer ensures that after each observation of results, it performs a summary and reflection. By documenting these intermediate reasoning steps, the model learns to emulate the cognitive processes involved in fault localization, enhancing its ability to tackle complex software engineering tasks.

Patch Generation. In the final stage, the Patch Generation Agent (Coder) generates and applies patches to address the localized issues. This process involves patch generation and the use of git-related operations and lint tool to implement and validate changes. Specifically, Coder first generates a concrete solution based on the issue description and identified fault code snippets. It then replaces the fault code snippets with the new solution. If the generated patch fails to conform to the specified format or cannot be syntactically applied to the original program (i.e., if lint tools detect syntax errors in the generated code, or if the git apply command fails), Coder debugs based on the error type until a correctly formatted patch is generated or the maximum retry limit is reached. This stage embodies the iterative nature of real-world software development scenarios, allowing for multiple refinement cycles to produce high-quality, applicable patches.

By enhancing each stage with detailed intermediate reasoning steps and incorporating tools that mirror real-world development practices, SWESynInfer provides a more comprehensive and realistic simulation of the software maintenance and evolution process. This approach enables the generation of high-quality training data that captures the complex, interactive aspects of software development.

Rejection Sampling. To ensure the quality of the synthesized data, we implement a rejection sampling process based on two key metrics. This approach allows us to selectively retain high-quality instances that closely mimic real-world software development practices.

Fault localization accuracy. We compare the predicted fault location with the actual fault location using a similarity threshold m_1. Specifically, we first extract the modified locations from the submitted patch in the pull request as the actual fault location (line 7, Algo. 1). We map the patch to the current version of the repository and then use the abstract syntax tree to extract the corresponding functions and classes for the modified locations. For global code modifications, we select the surrounding code (3 lines above and below the modified line) as the modification location. We then use the same method to extract the modification location from the model-generated patch (line 20). To quantify the accuracy of fault localization, we calculate the Jaccard similarity coefficient [51] between these two sets of modification locations. Specifically, we divide the size of the intersection of the two sets by the size of their union. This ratio is then compared with the threshold m_1 (line 21, Algo. 1). If the calculated ratio exceeds m_1, we consider the fault localization to be sufficiently accurate.

Patch similarity. We evaluate the similarity between the predicted patch and the developer submitted patch using a threshold m_2. Following the approach in Agentless [55], we first normalize both the model-generated and developer-written patches to ignore surface-level differences (e.g.,

Algorithm 1: Lingma SWE-GPT Training Algorithm

Input: Instruct LLM M, Dataset $D = \{(\text{issue}, \text{codebase}, \text{pull\_request})\}$, Fault localization threshold m_1, Patch similarity threshold m_2, Number of iterations N
Output: Optimized model M'

```
Sort D by pull-request timestamps to prevent data leakage;
M' ← M ;                                          // Initialize the model to be optimized
for iteration ← 1 to N do
    B ← {} ;                                      // Initialize batch for current iteration
    foreach (issue, codebase, pull_request) ∈ D do
        submitted_patch ← extract_patch(pull_request);
        actual_fault_location ← extract_fault_location(submitted_patch);
        Obs_CoT_Actions ← [];
        foreach stage ∈ {"Repo_Understanding", "Fault_Localization", "Patch_Generation"} do
            stage_Obs_CoT_Actions ← [];
            observation ← initial_observation(issue, codebase, stage, Obs_CoT_Action);
            // Model performs autonomous analysis and reflection
            while not stage_completed(stage, observation) do
                cot_action ← generate_cot_action(M', observation, stage);
                Append (observation, cot_action) to stage_Obs_CoT_Actions;
                next_observation ← execute_action(cot_action, codebase, stage);
                observation ← next_observation;
            Extend Obs_CoT_Actions with stage_Obs_CoT_Actions;
        // Synthetic data filtering and selection
        predicted_fault_location ← extract_predicted_location(Obs_CoT_Actions);
        if similarity(predicted_fault_location, actual_fault_location) ≥ m1 then
            predicted_patch ← extract_predicted_patch(Obs_CoT_Actions);
            if patch_similarity(predicted_patch, submitted_patch) ≥ m2 then
                B ← B ∪ {(issue, Obs_CoT_Actions)};
            else
                Obs_CoT_Actions_without_patch ← remove_patch_related_steps(Obs_CoT_Actions);
                B ← B ∪ {(issue, Obs_CoT_Actions_without_patch)};
    // Optimize model using maximum likelihood estimation
    θ' ← arg max_θ Σ_{(issue,Obs_CoT_Actions)∈B} Σ_{(obs_i, cot_action_i)∈Obs_CoT_Actions} log P_θ(cot_action_i | issue, obs_i);
    M' ← update_model_parameters(M, θ');
return M'
```

extra spaces, newlines, and comments). We then calculate the similarity between the model-generated patch and the developer-written patch using both n-gram [6] and CodeBLEU [44] scores (line 23, Algo. 1). If any similarity score exceeds m_2, we retain the patch.

Data instances meeting both criteria are added to the training batch B (line 27, Algo. 1). For instances that accurately localize the fault but generate dissimilar patches, we retain the fault localization steps while removing patch-related actions, preserving valuable intermediate reasoning. These rigorous filtering criteria ensure that our synthesized data closely resembles real-world software development practices, also guaranteeing the reliability of the intermediate reasoning process.

3.3 Model training

After collecting a set of training examples B_i, we implement an iterative optimization strategy to train our model. In each iteration, the model optimizes its performance by maximizing the conditional probability of generating the target CoT and corresponding action given the current state observation (line 29, Algo. 1). To further enhance the robustness of the training process, we incorporate a curriculum learning approach. As iterations progress, we accumulate problems that the current version of the model fails to solve and incrementally incorporate them into subsequent training. The complexity of the accumulated training examples increases progressively with the number of model iterations. This approach enables the model to establish a robust foundation on simpler tasks before addressing more complex challenges. Drawing inspiration from STaR [65] and NExT [38], we adopt a similar strategy to mitigate the potential negative impact of low-quality samples that may exist in early iterations. Specifically, we initialize the model from its original checkpoint M at the beginning of each iteration (line 30). This approach mitigates the risk of overfitting to potentially low-quality training examples in the early stages, thereby ensuring the stability of the training process and enhancing the generalization capability of the final model.

4 Experiment Setup

To evaluate the capabilities of Lingma SWE-GPT in resolving real-world Github issues, we answer the following research questions.

RQ1: How does Lingma SWE-GPT compare to state-of-the-art models in solving real-world software issues?

RQ2: What is the performance of Lingma SWE-GPT compared to open-source models in automated program improvement task?

RQ3: How effective is Lingma SWE-GPT in fault localization within the essential steps required for issue resolution?

RQ4: To what extent does the inherent randomness of large language models impact the consistency of Lingma SWE-GPT's performance?

4.1 Benchmark and Evaluation Metric

SWE-bench Verified and SWE-bench Lite. We evaluated Lingma SWE-GPT on the recently proposed benchmarks SWE-bench Verified [40] and SWE-bench Lite [5], comprising 500 and 300 real-world GitHub issues, respectively. 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. For detailed information on SWE-bench Verified and SWE-bench Lite, refer to Section 2.3.

Evaluation Metric. We use (1) the percentage of resolved task instances, (2) average inference cost of requesting closed-source model API. These evaluation metrics represent overall effectiveness, and economic efficacy in resolving real-world GitHub issues. Due to the public accessibility of open-source models, we assigns their API calls costs to NULL (-). This approach disregards potential deployment costs, which is left to future research to more comprehensively evaluate various factors, including indirect expenses. In addition, to mitigate the natural randomness of LLM, we repeat our experiments three times. Following AutoCodeRover [70] and SWE-agent [63], we report the results with the SWE-GPT @3 annotations (i.e., pass@3 metric).

4.2 Baselines

To thoroughly evaluate the performance of Lingma SWE-GPT in resolving real-world GitHub issues, we compared our model against both state-of-the-art closed-source and open-source models.

Closed-Source Models. We included leading closed-source models whose results are reported on the SWE-bench leaderboard [19] and in recent related research [75]. These models primarily consist of OpenAI's GPT and Anthropic's Claude series. For these closed-source models, we utilized the results reported by SWE-bench [5, 40].

- **GPT-4** [1] and **GPT-4o** [39]: GPT-4 represents one of the most advanced language models available, exhibiting strong capabilities in understanding and generating human-like text and code. GPT-4o is OpenAI's flagship model that can reason across audio, vision, and text in real time.
- **Claude 3.5 Sonnet** [3] and **Claude 3 Ops** [4], developed by Anthropic. The Claude models are designed with a focus on alignment and safe AI practices, and have shown proficiency in various tasks, particularly in code understanding and generation.
- **Gemini-1.5 Pro** [50], developed by Google. Gemini-1.5 Pro is a state-of-the-art multimodal model that excels in long-context understanding. It can process up to one million tokens in a single prompt, allowing for unprecedented context analysis.

Open-Source Models. For open-source baselines, we selected the most advanced models from the Llama and Qwen series. We obtained the results by both using existing research reports [28] and deploying and running them ourselves.

- **Llama Series** [36]: We evaluated Llama 3.1 instruction-tuned models with 70B and 405B parameters, representing the most advanced and largest open-source models in Llama 3.1 series.
- **Qwen Series** [62]: We included Qwen2-72B-Instruct, Qwen2.5-72B-Instruct, and Qwen2.5-Coder 7B (maximum coder version [17]) models. This series demonstrated superior performance on the HuggingFace open-source comprehensive leaderboard [16].

4.3 Implementation Details

We implemented Lingma SWE-GPT using Qwen2.5 72B instruct [62] and Qwen2.5 Coder 7B [17] as the foundation LLMs for the 72B and 7B versions, respectively. All training was conducted on a cluster of 64 NVIDIA A100 GPUs running Ubuntu 20.04, with a global batch size of 512 throughout the training process. For inference, we employed temperature sampling with a temperature (T) of 0.3. The training process consisted of 90 iterations in total. To mitigate the risk of the model converging on low-quality synthetic data during the initial stages, we utilized GPT-4o [39] to generate the initial training data for the first 10 iterations before transitioning to model-generated data. To ensure training efficiency, we updated the model with synthesized data every 10 iterations. In Algorithm 1, the parameters m_1 and m_2 were set to 0.6 and 0.5, respectively. For evaluation purposes, we configured the inference process with a temperature of 0.3 and a maximum token limit of 1024 to generate results for each round. During the Repository Understanding stage, the number of potentially relevant files (N) identified was set to 5. In the Fault Localization stage, the predefined iteration limit was set to 5. For the Patch Generation stage, the maximum retry limit was set to 3. To ensure consistency and reproducibility, all tests were conducted using the official SWE-bench Docker environment provided by the SWE-bench team [19].

Agent	LLM	Verified	Lite	API Cost
RAG [19]	GPT-4	2.80%	2.67%	$0.13
RAG [19]	Claude 3 Opus	7.00%	4.33%	$0.25
AutoCodeRover [28]	Qwen2 72B instruct	-	9.34%	$ -
SWE-agent [19]	Claude 3 Opus	18.20%	11.67%	$3.42
SWESynInfer	Lingma SWE-GPT 7B	18.20%	12.00%	$ -
SWE-agent [63]	GPT-4	22.40%	18.00%	$2.51
SWE-agent [40]	GPT-4o	23.00%	18.30%	*Unknown*
Refined OpenDevin [75]	Gemini-1.5-Pro	-	18.70%	*Unknown*
AutoCodeRover [70]	GPT-4	-	19.00%	$0.45
AppMap Navie [19]	GPT-4o	26.20%	21.67%	*Unknown*
AutoCodeRover [40]	GPT-4o	28.80%	22.70%	*Unknown*
SWESynInfer	Lingma SWE-GPT 72B	30.20%	22.00%	$ -
SWESynInfer	GPT-4o	31.80%	20.67%	$0.78
Agentless [40]	GPT 4o	33.20%	**24.30%**	$0.34
SWE-agent [63]	Claude 3.5 Sonnet	33.60%	23.00%	*Unknown*
SWESynInfer	**Claude 3.5 Sonnet**	**35.40%**	23.67%	$0.42

Table 1. Performance comparison of Lingma SWE-GPT and other models on SWE-bench Verified and Lite benchmarks. Note: open-source models are denoted by , and closed-source models are indicated by . "-" signifies unreported results for the respective benchmark. "Unknown" indicates unreported API costs. For open-source models, "$-" represents no direct API call costs.

5 Evaluation

5.1 RQ1: Overall Effectiveness in Resolving Real-life Software Issues

To comprehensively evaluate the performance of Lingma SWE-GPT in resolving real-world software issues, we conducted extensive experiments using two challenging benchmarks: SWE-bench Verified and SWE-bench Lite. These benchmarks provide a realistic runtime testing environment that closely simulates real-world software development scenarios. The input for each task consists solely of the natural language description of GitHub issues and the corresponding repository code at the time the issue was reported. The expected output is a corrective patch that resolves the issue. We measure the overall effectiveness of Lingma SWE-GPT and baseline models by quantifying the number of successfully resolved issue instances. Table 1 presents the comprehensive results of Lingma SWE-GPT (7B and 72B) alongside various state-of-the-art models on both SWE-bench Verified and SWE-bench Lite. The results reveal several significant findings.

Existing Open-Source vs. Closed-Source Models. From table 1, we can see that the majority of top-performing submissions are based on closed-source models. For instance, SWE-agent utilizing Claude 3.5 Sonnet achieves success rates of 33.60% and 23.00% on Verified and Lite benchmarks, respectively. Similarly, Agentless combined with GPT-4o resolves 33.20% of issues on Verified and 24.30% on Lite. These results underscore the current dominance of closed-source models in complex software engineering tasks. In contrast, open-source models have traditionally underperformed compared to their closed-source counterparts. For example, AutoCodeRover using Qwen2 72B instruct achieves a 9.34% success rate on SWE-bench Lite, while the same framework with GPT-4o resolves 22.70% of issues, highlighting a significant performance gap.

Agent	LLM	Size	Verified	Lite
SWESynInfer (∗)	Qwen2.5-coder	7B	5.80%	4.33%
SWESynInfer (∗)	Llama-3.1-instruct	70B	17.20%	7.00%
SWESynInfer	Lingma SWE-GPT	7B	18.20%	12.00%
SWESynInfer	Qwen2-instruct	72B	20.40%	13.70%
SWESynInfer	Llama-3.1-instruct	405B	24.60%	15.66%
SWESynInfer (∗)	Qwen2.5-instruct	72B	25.40%	18.00%
SWESynInfer	**Lingma SWE-GPT**	**72B**	**30.20% (18.90%↑)**	**22.00% (22.22%↑)**

Table 2. Comparative analysis of Lingma SWE-GPT and other open-source LLMs on SWE-bench Verified and Lite benchmarks. Note: (∗) denotes models with limited instruction-following capabilities, necessitating manual prompt engineering, which may yield results that overestimate actual performance.

Performance of Lingma SWE-GPT. Lingma SWE-GPT 72B demonstrates competitive performance compared to state-of-the-art closed-source models. On SWE-bench Verified, it successfully resolves 30.20% of the issues, closely approaching the performance of GPT-4o (31.80%) under the same inference process. This marks a significant milestone as it is the first time an open-source model has surpassed the 30% threshold in resolving these complex software issues. On SWE-bench Lite, Lingma SWE-GPT 72B even outperforms GPT-4o, resolving 22.00% of issues compared to GPT-4o's 20.67%. While Claude 3.5 Sonnet achieves the best overall performance on both benchmarks (35.40% on Verified and 23.67% on Lite), it's worth noting that Lingma SWE-GPT 72B's performance is also highly competitive, especially considering its open-source nature.

API Cost Considerations. A critical consideration in deploying large language models for software engineering tasks is the associated API costs. Due to the complexity of software repositories and the multi-round reasoning process, solving the 500 problems in SWE-bench Verified using GPT-4o incurs an approximate cost of $390. This translates to an average of $0.78 per issue, which can be prohibitively expensive for large-scale applications or continuous integration scenarios.

In contrast, open-source models like Lingma SWE-GPT do not incur direct API costs due to their public accessibility. This cost-effectiveness, combined with the competitive performance, presents a case for the adoption of open-source models in automated software engineering tasks. Interestingly, even the smaller Lingma SWE-GPT 7B model shows promising results, resolving 18.20% and 12.00% of issues in SWE-bench Verified and Lite, respectively. This performance underscores the potential of smaller models in automated issue resolution, particularly in resource-constrained scenarios. It opens up possibilities for more widespread integration of AI-assisted bug fixing and code improvement in software development pipelines, especially for organizations with budget constraints or privacy concerns.

5.2 RQ2: Performance Against State-of-the-Art Open-Source Models

To evaluate the performance of Lingma SWE-GPT in comparison with state-of-the-art open-source models for automated program improvement, we conducted a comprehensive analysis using the same inference process (SWESynInfer) across various models. Table 2 presents the overall results of this comparison.

Experimental Challenges and Setup. During our evaluation, we encountered significant challenges with certain open-source models due to their limited instruction-following capabilities. To optimize the assessment of these models' effectiveness in software engineering tasks (rather than their general instruction-following ability), we implemented model-specific prompt engineering and

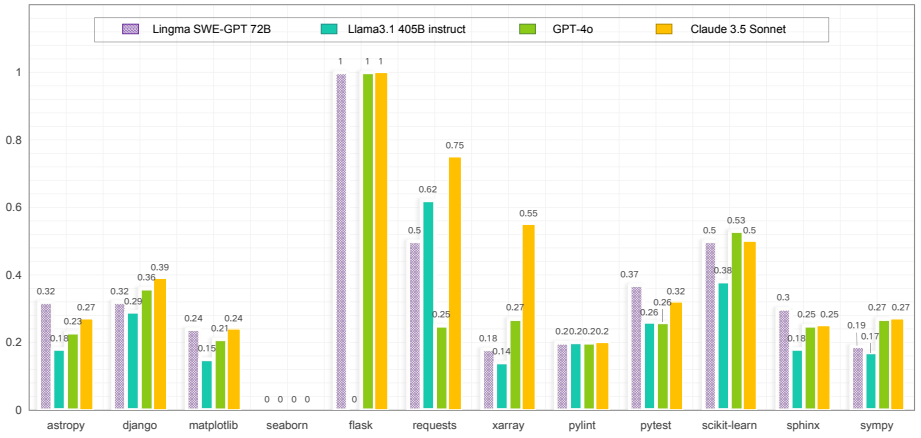


Fig. 4. Comparison of issue resolution rates between Lingma SWE-GPT and other LLMs across different repositories in SWE-bench Verified.

tool customization. For instance, Qwen2.5-instruct 72B consistently generated fixed JSON formats ```` ```json\n{actual_content}``` ````, while Llama-3.1-instruct 70B output ```` ```\n{actual_content}``` ````, both of which led to JSON extraction failures(should be $\{actual\_content\}$). These issues likely originate from excessively constrained format requirements during the models' alignment processes, resulting in diminished adaptability to complex tasks. In contrast, larger models like Llama-3.1-instruct 405B and closed-source models demonstrated superior instruction-following abilities, correctly generating JSON formats as required. To ensure a fair comparison, we adjusted prompts and customized tool invocations for affected models, and indicated by $(*)$ in Table 2.

Results on SWE-bench. The results demonstrate that Lingma SWE-GPT 72B significantly outperforms the latest open-source models, including Qwen2.5-instruct 70B and the largest open-source model, Llama-3.1-instruct 405B. On SWE-bench Verified, Lingma SWE-GPT 72B achieves a 30.20% success rate, marking an 18.90% relative improvement over Qwen2.5-instruct 72B (25.40%) and a 22.76% relative improvement over Llama-3.1-instruct 405B (24.60%). This performance gap highlights the effectiveness of Lingma SWE-GPT's training approach, which focuses on understanding and generating dynamically evolving processes in software development tasks.

Notably, even the smaller Lingma SWE-GPT 7B model outperforms Llama-3.1-instruct 70B (18.20% vs. 17.20% on SWE-bench Verified, 12.00% vs. 7.00% on SWE-bench Lite), further validating the efficacy of our process-oriented data training methodology. This result demonstrates that smaller, more efficient models can also attain competitive performance when trained on high-quality, process-oriented data.

Among other open-source models, Llama-3.1-instruct 405B exhibits the best performance without prompt-specific adjustments, highlighting its general capabilities and specialized abilities in software engineering tasks. This observation aligns with the scaling law hypothesis, which posits that larger models tend to perform better across a wide range of tasks. This insight provides valuable guidance for the future development of more advanced software engineering models.

Performance Across Different Repositories. Figure 4 illustrates the performance of Lingma SWE-GPT 72B across 12 diverse software repositories, compared to the best-performing open-source model (Llama 3.1 405B instruct) and leading closed-source models (GPT-4o and Claude 3.5 Sonnet). Lingma SWE-GPT 72B outperforms Llama 3.1 405B instruct in the majority of repositories (9/12) and approaches the performance of closed-source models in numerous instances. This consistent

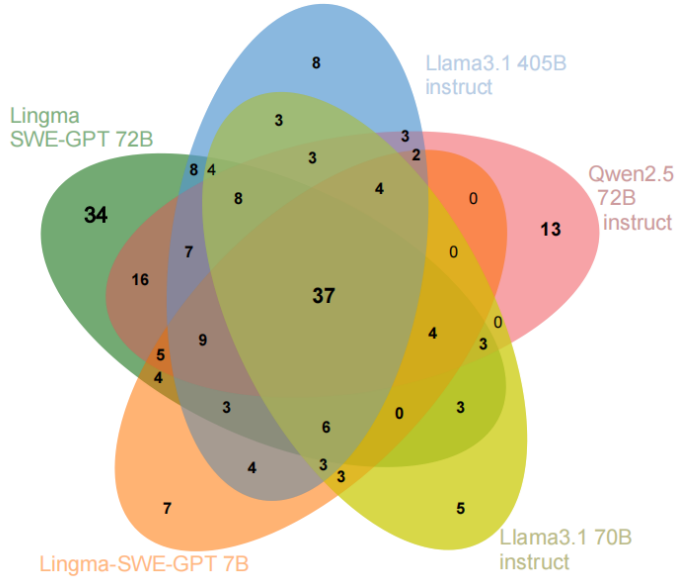


Fig. 5. Venn diagram of issue instances solved by Lingma SWE-GPT and other open-source models on SWE-bench Verified.

performance across various domains demonstrates the model's robust generalization capabilities and its potential for extensive application in diverse software engineering contexts.

Complementarity of Models. Figure 5 presents a venn diagram illustrating the overlap and uniqueness of issues resolved by different models. Notably, Lingma SWE-GPT 72B uniquely resolves the highest number of issues (34), demonstrating its superior comprehension and analytical capabilities in software engineering processes. In addition, Lingma SWE-GPT 7B solves 7 tasks that are not solved by other models, surpassing the 5 independently solved tasks of Llama 3.1 70B-instruct, demonstrating the potential of smaller models. The diagram also reveals a degree of complementarity among different models, with each solving some unique issues. This observation suggests potential for future research in model ensemble approaches to further improve overall performance in automated program improvement tasks, a direction already began to explore in some existing studies [43, 67].

5.3 RQ3: Fault Localization Effectiveness

Fault localization is a critical step in resolving software issues in real-world development scenarios. Accurate identification of edit locations is not only crucial for automated issue resolution and invaluable for assisting human developers in debugging tasks. To evaluate the fault localization capabilities of Lingma SWE-GPT, we conducted a comprehensive analysis comparing the locations of patches generated by various models to the actual patch locations.

Evaluation Methodology. We employed a rigorous process to ensure a thorough evaluation. We mapped the patches to the current version of the repository and used abstract syntax trees to extract the functions and classes corresponding to the modified locations. For chunk-level analysis, which not only encompasses function/class modifications but also global code changes, we considered the surrounding code (3 lines above and below the modified line) as the global code modification location. Following the approach outlined in Agentless [55], we acknowledge that while errors can

LLM	Model Size	Verified	Chunk	Function	File
Qwen2.5-coder-instruct	7B	5.80%	13.15%	13.91%	19.20%
Llama-3.1-instruct	70B	17.20%	47.81%	51.01%	70.67%
Lingma SWE-GPT	7B	18.20%	39.10%	42.20%	58.82%
Qwen2-instruct	72B	20.40%	38.87%	40.80%	55.21%
Llama-3.1-instruct	405B	24.60%	44.92%	47.80%	67.49%
Qwen2.5-instruct	72B	25.40%	42.84%	45.66%	61.34%
Lingma SWE-GPT (run_1)	72B	30.20%	51.16%	54.30%	72.29%
Lingma SWE-GPT (run_2)	72B	29.00%	51.88%	53.90%	72.02%
Lingma SWE-GPT (run_3)	72B	30.20%	53.24%	55.17%	72.90%
Lingma SWE-GPT (*pass*@3)	**72B**	**39.80%**	**61.63%**	**66.28%**	**80.85%**
GPT-4o	*Unknown*	31.80%	52.18%	55.68%	72.49%
Claude 3.5 Sonnet	***Unknown***	**35.40%**	**54.90%**	**58.08%**	**74.27%**

Table 3. Comparative analysis of fault localization accuracy across various language models at different granularities.

be fixed at locations different from the actual patch, comparing with the real patch serves as an effective approximate measure. Table 3 shows the localization accuracy at the chunk, function, and file levels with each model.

Results and Analysis. Our investigation reveals several key insights. Lingma SWE-GPT demonstrates significantly better fault localization capabilities compared to other open-source models across all granularity levels (chunk, function, and file). Its performance closely approaches that of closed-source models, underscoring its effectiveness in fault localization. In addition, we observe a general positive correlation between fault localization success rate and issue resolution rate. This relationship underscores the critical role of accurate fault localization in successful issue resolution. However, even the best-performing model, Claude 3.5 Sonnet, achieves localization accuracy of only 54.90% and 58.08% at chunk and function levels, respectively. This observation suggests that fault localization remains a challenging aspect of automated issue resolution. This presents an opportunity for future research to focus on improving localization techniques, potentially through enhanced code understanding mechanisms or more sophisticated static and dynamic analysis methods.

5.4 RQ4: Model Consistency and Randomness Impact

Large language models (LLMs) inherently exhibit stochastic behavior in their outputs, which can potentially affect their consistency and reliability in practical applications. To address this concern, we conducted a comprehensive investigation into the consistency of Lingma SWE-GPT's performance across multiple executions on the SWE-bench Verified. This analysis aims to quantify the impact of the model's inherent variability on its ability to consistently solve software engineering tasks.

Evaluation Methodology. We executed Lingma SWE-GPT 72B three times (denoted as run_1, run_2, run_3) on the SWE-bench Verified dataset. This approach allows us to isolate and observe the effects of the model's inherent randomness on its performance. We then analyzed the results in terms of issue resolution rates and fault localization accuracy across different granularity levels (chunk, function, and file).

Results and Analysis. Table 3 presents the detailed results of our consistency analysis. Across the three runs, Lingma SWE-GPT 72B demonstrated remarkable consistency in its issue resolution capability, achieving success rates of 30.20%, 29.00%, and 30.20% for run_1, run_2, and run_3, respectively. The fault localization performance also exhibited stability across runs, with narrow ranges at chunk, function, and file levels, further underscoring the model's consistency in pinpointing issue locations. Follow AutoCodeRover [70] and SWE-agent [19], we also implemented a pass@3 metric, which considers an issue as resolved if any of the three runs successfully addresses it. This approach significantly boosted the overall performance, increasing the issue resolution rate to 39.80%. Notably, this surpasses the performance of Claude 3.5 Sonnet (35.40%), a leading closed-source model. The pass@3 approach also substantially enhanced fault localization accuracy, achieving 61.63% at the chunk level, 66.28% at the function level, and 80.85% at the file level. This is particularly relevant in scenarios where test suites or other validation mechanisms are available, opening avenues for more accurate automated software improvement techniques.

6 Limitation and Threats to Validity

While Lingma SWE-GPT has demonstrated promising results in automated software improvement, it is crucial to acknowledge the limitations of our approach and potential threats to the validity of our findings. This section discusses these aspects and outlines directions for future research.

Automatic Solution Verification. A limitation of our current approach is the lack of comprehensive solution verification step. Although Lingma SWE-GPT has shown impressive performance on the SWE-bench dataset, we have not implemented an automated process yet to verify the correctness of the generated patches through unit testing. This verification step is important for ensuring the reliability and practical applicability of the model's outputs. The primary challenge in addressing this limitation stems from the scarcity of suitable training data. Real-world scenarios rarely provide readily available datasets that encompass the entire process of testing and debugging. While it is possible to synthesize buggy code from existing repository data and simulate the patch verification and debugging process, this approach may not fully capture the complexity of real-world software issues. Furthermore, automatically constructing complex project environments and resolving dependencies poses a significant challenge. To address this, we propose developing an agent capable of automating environment setup, which we contend is crucial for scalable process data acquisition. We posit that future work will further enhance the end-to-end effectiveness of program improvement by incorporating solution verification mechanisms.

Model Evaluation. Another limitation of our study lies in the evaluation methodology. Our evaluation of Lingma SWE-GPT primarily utilizes the SWE-bench Verified [40] and SWE-bench Lite [5] benchmarks. While these benchmarks provide valuable insights into the model's ability to address real-world issues by assessing proficiency in various tasks such as fault localization, debugging, and program repair, they may not fully capture the complexity and real-world applicability of the model compared to human developers. We acknowledge that a more comprehensive evaluation approach would involve directly applying Lingma SWE-GPT to resolve open issues in open-source software projects on platforms like GitHub. This approach would offer a more direct validation of the model's efficacy in real-world software improvement scenarios. Furthermore, we are exploring more sophisticated evaluation methodologies to assess the model's performance on complex software engineering tasks that better reflect the nuances of real-world development processes.

Despite these limitations, Lingma SWE-GPT constitutes a significant step forward in automated software improvement. Each delineated limitation not only highlights the challenges in this field but also provides clear directions for future research and improvement. We aim to harness these

findings to evolve Lingma SWE-GPT into a more robust, adaptable, and effective system for assisting developers throughout the software development lifecycle.

7 Conclusion

In this paper, we introduce Lingma SWE-GPT, a novel open-source large language model series designed to address complex software improvement tasks. This series includes two variants: Lingma SWE-GPT 7B and Lingma SWE-GPT 72B, catering to different computational resource requirements while maintaining high performance. By focusing on simulating the dynamic nature of software development, including tool usage reasoning and interactive problem-solving capabilities, Lingma SWE-GPT distinguishes itself apart from models trained solely on static code data. Our evaluation, utilizing the challenging SWE-bench Verified and Lite benchmarks demonstrates Lingma SWE-GPT's effectiveness in automated software improvement. The 72B variant consistently outperforms existing open-source models and approximates the performance of closed-source alternatives, achieving a remarkable 30.20% success rate on SWE-bench Verified. Notably, the 7B variant resolves 18.20% of issues, highlighting the potential of smaller models in resource-constrained environments. Lingma SWE-GPT achieves exceptional fault localization capabilities across chunk, function, and file levels. The model's consistent performance across multiple runs and the significant enhancements achieved through our pass@3 approach highlight its reliability and potential for ensemble methods in automated software engineering. As we continue to refine and expand its capabilities, we posit that Lingma SWE-GPT will play an increasingly significant role in supporting developers, enhancing productivity, and improving software quality, thereby advancing the field of AI-assisted software engineering.

8 Data Availability

We release our model checkpoints (Lingma SWE-GPT 7B & 72B) and data synthesis and inference (SWESynInfer) code to encourage further exploration in this direction. The artifact that supports the results discussed in this paper is available at: https://github.com/LingmaTongyi/Lingma-SWE-GPT

Acknowledgments

We would like to express our gratitude to Zhipeng Xue and Ke Liu for their invaluable feedback and suggestions on the manuscript.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 202–206.

[3] Anthropic. 2024. *Introducing Claude 3.5 Sonnet.* https://www.anthropic.com/news/claude-3-5-sonnet

[4] Anthropic. 2024. *Introducing the next generation of Claude.* https://www.anthropic.com/news/claude-3-family

[5] Carlos E. Jimenez, John Yang, Jiayi Geng. 2024. *SWE-bench Lite: A Canonical Subset for Efficient Evaluation of Language Models as Software Engineers.* https://www.swebench.com/lite.html

[6] William B Cavnar, John M Trenkle, et al. 1994. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, Vol. 161175. Ann Arbor, Michigan, 14.

[7] Saikat Chakraborty and Baishakhi Ray. 2021. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 443–455.

[8] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *arXiv preprint arXiv:2406.01304* (2024).

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[10] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[11] Cognition. 2023. *Introducing Devin.* https://www.cognition.ai/introducing-devin

[12] Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 392–418.

[13] Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: Structure-Aware Pretraining for Code Generation and Understanding. *arXiv preprint arXiv:2401.03003* (2024).

[14] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[15] Xiangbing Huang, Yingwei Ma, Haifang Zhou, Zhijie Jiang, Yuanliang Zhang, Teng Wang, and Shanshan Li. 2023. Towards Better Multilingual Code Search through Cross-Lingual Contrastive Learning. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 22–32.

[16] Huggingface Open LLM Leaderboard. 2024. *Dataset Card for Evaluation run of Qwen.* https://huggingface.co/datasets/open-llm-leaderboard/Qwen__Qwen2-72B-details

[17] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[18] Zhijie Jiang, Haixu Xiong, Yingwei Ma, Yao Zhang, Yan Ding, Yun Xiong, and Shanshan Li. 2023. Automatic Code Annotation Generation Based on Heterogeneous Graph Structure. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 497–508.

[19] 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* (2023).

[20] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. ContrastRepair: Enhancing Conversation-Based Automated Program Repair via Contrastive Test Case Pairs. *arXiv preprint arXiv:2403.01971* (2024).

[21] Cheryl Lee, Chunqiu Steven Xia, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R Lyu. 2024. A Unified Debugging Approach via LLM-Based Multi-Agent Synergy. *arXiv preprint arXiv:2404.17153* (2024).

[22] Jiaying Li, Yan Lei, Shanshan Li, Haifang Zhou, Yue Yu, Zhouyang Jia, Yingwei Ma, and Teng Wang. 2023. A two-stage framework for ambiguous classification in software engineering. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 275–286.

[23] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. Codeeditor: Learning to edit source code with pre-trained models. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–22.

[24] Meiziniu Li, Dongze Li, Jianmeng Liu, Jialun Cao, Yongqiang Tian, and Shing-Chi Cheung. 2024. DLLens: Testing Deep Learning Libraries via LLM-aided Synthesis. *arXiv preprint arXiv:2406.07944* (2024).

[25] Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, et al. 2024. Mftcoder: Boosting code llms with multitask fine-tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5430–5441.

[26] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *arXiv preprint arXiv:2409.02977* (2024).

[27] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[28] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Wenmeng Zhou, Fei Wang, and Michael Shieh. 2024. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. *arXiv preprint arXiv:2408.03910* (2024).

[29] Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024. MarsCode Agent: AI-native Automated Bug Fixing. *arXiv preprint arXiv:2409.00899* (2024).

[30] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

[31] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. *arXiv preprint arXiv:2402.16667* (2024).

[32] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[33] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At Which Training Stage Does Code Data Help LLMs Reasoning? *arXiv preprint arXiv:2309.16298* (2023).

[34] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? *arXiv preprint arXiv:2406.01422* (2024).

[35] Yingwei Ma, Yue Yu, Shanshan Li, Zhouyang Jia, Jun Ma, Rulin Xu, Wei Dong, and Xiangke Liao. 2023. Mulcs: Towards a unified deep representation for multilingual code search. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 120–131.

[36] Meta. 2024. *Introducing Llama 3.1.* https://ai.meta.com/blog/meta-llama-3-1/

[37] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124* (2023).

[38] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NExT: Teaching Large Language Models to Reason about Code Execution. *arXiv preprint arXiv:2404.14662* (2024).

[39] OpenAI. 2024. *Introducing GPT-4o.* https://openai.com/index/hello-gpt-4o/

[40] OpenAI. 2024. *Introducing SWE-bench Verified.* https://openai.com/index/introducing-swe-bench-verified/

[41] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109* (2023).

[42] Zhenyu Pan, Rongyu Cao, Yongchang Cao, Yingwei Ma, Binhua Li, Fei Huang, Han Liu, and Yongbin Li. 2024. Codev-Bench: How Do LLMs Understand Developer-Centric Code Completion? *arXiv preprint arXiv:2410.01353* (2024).

[43] Paul Gauthier. 2024. *Aider is ai pair programming in your terminal.* https://aider.chat/2024

[44] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[45] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[46] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936* (2023).

[47] Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging. *arXiv preprint arXiv:2410.01215* (2024).

[48] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 151–160.

[49] CodeGemma Team. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409* (2024).

[50] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[51] Vikas Thada and Vivek Jaglan. 2013. Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm. *International Journal of Innovations in Engineering and Technology* 2, 4 (2013), 202–205.

[52] Yuchen Tian, Weixiang Yan, Qian Yang, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. 2024. CodeHalu: Code Hallucinations in LLMs Driven by Execution-based Verification. *arXiv preprint arXiv:2405.00253* (2024).

[53] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. arXiv:2402.01030 [cs.CL]

[54] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

[55] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
[56] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[57] Yifan Xie, Zhouyang Jia, Shanshan Li, Ying Wang, Jun Ma, Xiaoling Li, Haoran Liu, Ying Fu, and Xiangke Liao. 2024. 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*. 694–705.
[58] Ruiyang Xu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2024. CRUXEval-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution. *arXiv preprint arXiv:2408.13001* (2024).
[59] Zhipeng Xue, Zhipeng Gao, Xing Hu, and Shanping Li. 2023. ACWRecommender: A Tool for Validating Actionable Warnings with Weak Supervision. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1876–1880.
[60] Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shanping Li. 2024. SelfPiCo: Self-Guided Partial Code Execution with LLMs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1389–1401.
[61] Jiwei Yan, Jinhao Huang, Chunrong Fang, Jun Yan, and Jian Zhang. 2024. Better Debugging: Combining Static Analysis and LLMs for Explainable Crashing Fault Localization. *arXiv preprint arXiv:2408.12070* (2024).
[62] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* (2024).
[63] 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. *arXiv preprint arXiv:2405.15793* (2024).
[64] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187* (2023).
[65] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems* 35 (2022), 15476–15488.
[66] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. *arXiv preprint arXiv:2401.07339* (2024).
[67] Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, et al. 2024. Diversity empowers intelligence: Integrating expertise of software engineering agents. *arXiv preprint arXiv:2408.07060* (2024).
[68] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2023. Lampr: Boosting the Effectiveness of Language-Generic Program Reduction via Large Language Models. *arXiv preprint arXiv:2312.13064* (2023).
[69] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–273.
[70] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
[71] Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. 2024. CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding? *arXiv preprint arXiv:2408.10718* (2024).
[72] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658* (2024).
[73] Ming Zhu and Yi Zhou. 2024. MOSS: Enabling Code-Driven Evolution and Context Management for AI Agents. *arXiv preprint arXiv:2409.16120* (2024).
[74] Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2024. DOMAINEVAL: An Auto-Constructed Benchmark for Multi-Domain Code Generation. *arXiv preprint arXiv:2408.13204* (2024).
[75] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).

accepted 2025-03-31; accepted 2025-03-31