MASAI: Modular Architecture for Software-engineering AI Agents

Anonymous EMNLP submission

Abstract

A common method to solve complex problems in software engineering, is to divide the problem into multiple sub-problems. Inspired by this, we propose a Modular Architecture for Software-engineering AI (MASAI) agents, where different LLM-powered sub-agents are instantiated with well-defined objectives and strategies tuned to achieve those objectives. Our modular architecture offers several advantages: (1) employing and tuning different problem-solving strategies across sub-agents, (2) enabling sub-agents to gather information from different sources scattered throughout a repository, and (3) avoiding unnecessarily long trajectories which inflate costs and add extraneous context. MASAI enabled us to achieve the highest performance (28.33% resolution rate) on the popular and highly challenging SWE-bench Lite dataset consisting of 300 GitHub issues from 11 Python repositories. We conduct a comprehensive evaluation of MASAI relative to other agentic methods and analyze the effects of our design decisions and their contribution to the success of MASAI.

1 Introduction

Software engineering is a challenging activity which requires exercising various skills such as coding, reasoning, testing, and debugging. The ever growing demand for software calls for better support to software engineers. Recent advances in AI offer much promise in this direction.

Large language models (LLMs) have shown remarkable ability to code (Chen et al. (2021); Roziere et al. (2023); CodeGemma Team (2024), *inter alia*), reason (Kojima et al., 2022) and plan (Huang et al., 2022). Iterative reasoning, structured as chains (Wei et al., 2022) or trees (Yao et al., 2024) of thought, further enhance their ability to solve complex problems that require many interrelated steps of reasoning. When combined with tools or environment actions (Yao et al., 2023; Patil et al., 2023; Schick et al., 2024) and feedback from the environment (Zhou et al., 2023; Shinn et al., 2024), they enable autonomous agents capable of achieving specific goals (Zhang et al., 2023).

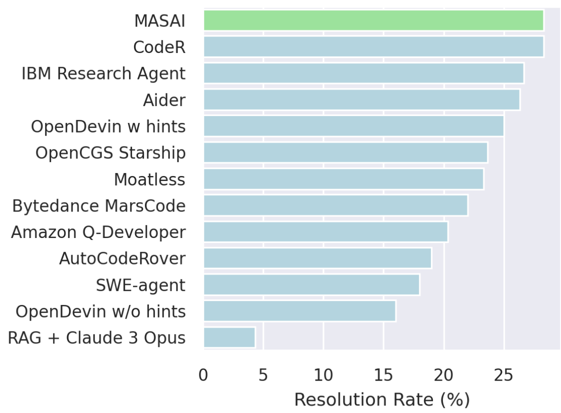

Figure 1: Comparison of MASAI with existing methods. *Resolution rate* refers to the percentage of issues in SWE-bench Lite that are resolved.

As the problem complexity increases, it becomes difficult to devise a single, over-arching strategy that works across the board. Indeed, when faced with a complex coding problem, software engineers break it down into sub-problems and use different strategies to deal with them separately. Inspired by this, we propose a Modular Architecture of Software-engineering AI (MASAI) agents, where different LLM-powered sub-agents are instantiated with well-defined objectives and strategies tuned to achieve those objectives.

Our modular architecture consists of 5 different sub-agents: **Test Template Generator** which generates a template test case and instructions on how to run it, **Issue Reproducer** which writes a test case to reproduce the issue, **Edit Localizer** which finds files to be edited, **Fixer** which fixes the issue by generating multiple possible patches, and finally **Ranker** which ranks the patches based on the generated test. When combined, all these individual sub-agents work in tandem to resolve complex real-world software engineering issues.

Our approach offers several advantages: (1) employing and tuning different problem-solving strategies across sub-agents (e.g., ReAct or CoT), (2) enabling sub-agents to gather information from different sources scattered throughout a repository (e.g., from a README or a test file), and (3) avoiding unnecessarily long trajectories which inflate inference costs and pass extraneous context which could degrade performance (Shi et al., 2023).

We evaluate MASAI on the popular and highly challenging SWE-bench Lite dataset (Jimenez et al., 2024) of 300 GitHub issues from 11 Python repositories. Due to its practical relevance and challenging nature, SWE-bench Lite has attracted significant efforts from academia, industry and start-ups. As shown in Figure 1, with the highest resolution rate of 28.33%, MASAI achieves state-of-the-art results on SWE-bench Lite. The field of AI agents, and specifically software-engineering AI agents, is nascent and rapidly evolving. In fact, all the existing methods in Figure 1 have been developed within the past three months. Nevertheless, we do compare against them thoroughly.

AI agents for software engineering would encounter many common sub-problems, such as autonomously understanding testing infrastructure and code organization of a repository, writing new tests, localizing bugs, editing large files without introducing syntactic/semantic errors, synthesizing fixes and writing new code. We believe that it is crucial to understand how different strategies perform on these sub-problems. Therefore we conduct a thorough investigation into the performance of MASAI and existing methods on SWE-bench Lite, and present the impact of key design decisions.

In summary, our contributions are:

(1) Propose a modular architecture, MASAI, that allows optimized design of sub-agents separately while combining them to solving larger, end-to-end software engineering tasks.

(2) Show the effectiveness of MASAI by achieving the highest resolution rate on SWE-bench Lite.

(3) Conduct a thorough investigation into key design decisions of MASAI and the existing methods which can help inform future research and development in this rapidly evolving space.

(4) Contribute our results to the SWE-bench Lite leaderboard (MASAI) for validation. For reproducibility, we provide our prompts in the Appendix and detailed logs as supplementary material.

2 MASAI Agent Architecture

Solving a problem in a code repository requires understanding the problem description and the codebase, gathering the necessary information scattered across multiple files, locating the root cause, fixing it and verifying the fix. Instead of treating this as one long chain of reasoning and actions, we propose modularizing the problem into sub-problems and delegating them to different sub-agents.

2.1 Agent Specification and Composition

A MASAI *agent* is a composition of several MASAI *sub-agents*. A MASAI *sub-agent* is specified by a tuple $\langle Input, Strategy, Output \rangle$ where

(1) *Input* to the sub-agent comprises of the code repository, information obtained from other sub-agents as necessary, a set of allowed actions and task instructions.

(2) *Strategy* is the problem-solving strategy to be followed by the sub-agent in using the LLM to solve its given sub-problem. This could be vanilla completion, CoT (Wei et al., 2022), ReAct (Yao et al., 2023), RAG (Lewis et al., 2020), etc.;

(3) *Output* is the specification of the content that the sub-agent must return upon completion as well the format it must be presented in.

Compared to multi-agent frameworks (Wu et al., 2023; Qian et al., 2023; Hong et al., 2024), the MASAI architecture is simpler, in that, the sub-agents are given modular objectives that do not require explicit one-to-one or group conversations between sub-agents. The sub-agents are composed by passing the output from one sub-agent to the input of another sub-agent.

2.2 Action Space

All the sub-agents are presented with a set of actions which allows them to interact with the environment. The actions we use in this work are:

(1) `READ(file, class, function)`: Query and read a specific function, class or file. All three attributes are not necessary; the agent can specify only a function and a file or even a single file. If there exists only one exactly matching code segment with these attributes, then that code is returned. If there are multiple matches, all their names are returned and the query can be refined if necessary. The `READ` action returns a **lazy representation** that aims to keep the output concise. When reading a file, only signatures of the top level definitions are presented; when reading a class, the

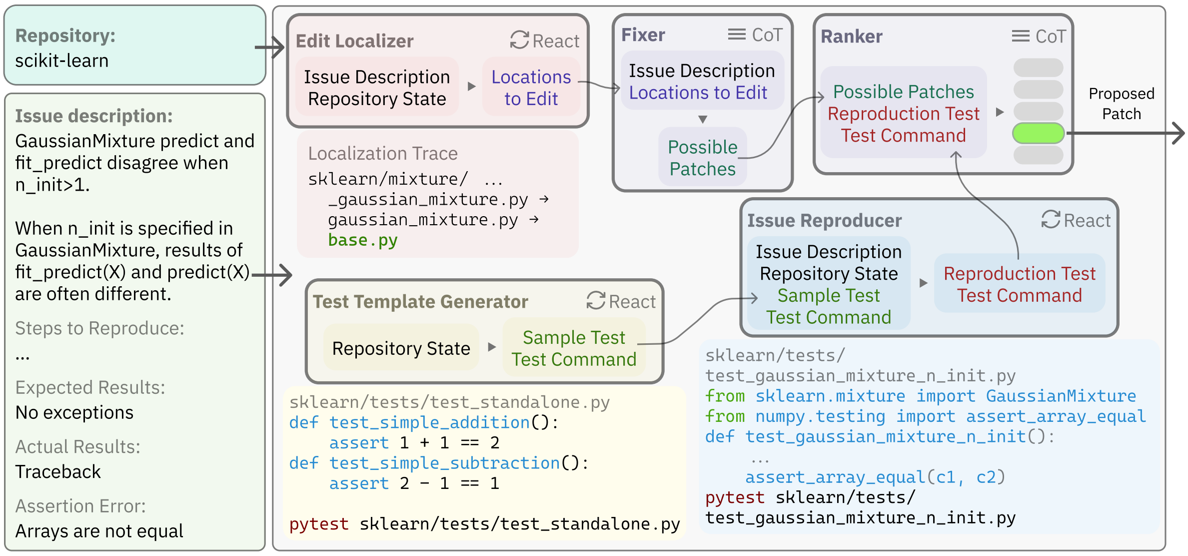

Figure 2: Overview of MASAI applied to the task of repository-level issue resolution on an example issue 13142 from `scikit-learn`. MASAI takes a repository and an issue description as input, and produces a single patch. The 5 sub-agents (shown in thick boxes) tackle different sub-problems. The information flow between them is shown by directed edges. The sub-agents are marked with the solution strategy and input–output pairs.

signature of the class (class name and member signatures) are presented and when reading a function, its complete body is presented.
(2) `EDIT(file, class, function)`: Marks a code segment for editing. Just like `READ`, this marks a code segment only when a unique match exists. Otherwise, the set of partial matches are returned which may be refined further.
(3) `ADD(file)`: Marks a file for code addition. The file must exist for the action to succeed.
(4) `WRITE(file, contents)`: Writes the specified content to a file. The specified file can be new or a file that the agent has created earlier.
(5) `LIST(folder)`: Lists folder contents if it exists.
(6) `COMMAND(command)`: Executes the command in a shell with timeout and truncation of large results.
(7) `DONE`: Used by the agent to signal that it has completed its assigned objective.

2.3 Agent Instantiation

In this work, we focus on the general task of resolving repository-level issues, as exemplified by the SWE-bench Lite dataset. A problem statement consists of an issue description and a repository. The agent is required to produce a patch so that the issue is resolved. Issue resolution is checked by ensuring that the relevant, held-out test cases pass.

Below, we refer to ReAct (Yao et al., 2023) which is a problem-solving strategy that alternates between generating an action to take using an LLM followed by executing the action and using the resulting observations as input for the subsequent action generation. Chain of Thought (CoT) (Wei et al., 2022) generates solutions to a problem using an LLM while asking it to generate specific intermediate reasoning steps.

We instantiate 5 sub-agents to collectively resolve repository-level issues. Figure 2 shows the overall architecture of our MASAI agent on a concrete example, along with the information flow between the sub-agents (shown by the solid edges). We describe each of the sub-agents below with detailed prompts in Appendix A.

(1) Test Template Generator: Discovers how to write and run a new test by analyzing the testing setup specific to the repository.

- *Input*: The repository state (within its execution environment) is provided. `READ`, `LIST`, `COMMAND`, `WRITE` and `DONE` actions are provided.
- *Strategy*: ReAct.
- *Output*: The code for a template test case (which is issue independent) for the repository along with the command to run it. This is used to aid the Issue Reproducer sub-agent described next.

Test Template Generator is instructed to explore the documentation and existing tests within the repository to complete its objective and to keep trying until it comes up with a template and a command that passes without exceptions. Test Template Generator evaluates the output of its ReAct loop to determine whether the generated test passes without exceptions. It retries upto a specified limit or until it finds a template that works.

(2) Issue Reproducer: Writes a test that reproduces the behaviour reported in the given issue.

- *Input*: In addition to the repository state and issue description, the sample test file and the command to run it, generated by the Test Template Generator, are provided. Actions available are `READ`, `LIST`, `COMMAND`, `WRITE` and `DONE`.
- *Strategy*: ReAct.
- *Output*: The code for a test case which reproduces the issue and would show a change in status (pass vs. fail) when the issue is fixed. It also outputs the shell command to run the test.

(3) Edit Localizer: Navigates the repository and identifies code locations (files, classes, functions) that need to be edited to resolve the issue.

- *Input*: The repository state and the issue description are provided. Available actions are `READ`, `LIST`, `EDIT`, `ADD`, `COMMAND` and `DONE`.
- *Strategy*: ReAct.
- *Output*: List of code locations (specified through the `EDIT` and `ADD` commands) to edit.

If no locations have been marked at the end of the ReAct loop, then the Edit Localizer selects a set of locations from all of the ones it has read so far.

(4) Fixer: Suggests multiple potential patches to the code locations marked by Edit Localizer that may resolve the issue.

- *Input*: Issue description along with contents of the code locations required to be edited. No actions are given to this sub-agent.
- *Strategy*: CoT.
- *Output*: Multiple possible candidate patches to the provided suspicious code.

When prompting the LLM for a possible patch, Fixer asks for the edit in the form of a **minimal rewrite** instead of rewriting the full sections. Similar to Deligiannis et al. (2023), the content of the locations to edit are provided by Fixer with line numbers. For each edit, the Fixer expects the LLM to output the original version of the code snippet (*pre*) followed by the edited version of this snippet (*post*). Both these snippets are expected to have a line number for each line. Fixer then searches for the *pre* snippet using line numbers in the target file to replace with the *post* version. If an exact match is not found, it uses **fuzzy matching** to find the closest matching span for the *pre* snippet. After replacing with the *post* span, it computes the `diff` of the target file with its contents before the edit. Syntactically incorrect edits are rejected and the resultant patches are used downstream.

(5) Ranker: Ranks the candidate patches from the Fixer, using the test generated by Issue Reproducer.

- *Input*: Issue description, candidate patches from Fixer, and the reproduction test (as well as the command to run it) from Issue Reproducer. No environment actions are allowed.
- *Strategy*: CoT.
- *Output*: Ranking of the candidate patches in the order of likelihood to resolve the issue.

For each of the patches, Ranker first runs the test on each of the patches and then asks the LLM to determine whether the application of that patch to the repository has caused the provided test to change status (go from failing to passing or vice versa) given the test results. Based on the output of this, the LLM is then asked to rank the patches. The top ranked patch is selected as the issue resolution. If the Issue Reproducer sub-agent could not generate a test, then the Ranker ranks the patches using only the issue description.

3 Experimental Setup

Dataset: As stated earlier, we perform experiments on SWE-bench Lite (Jimenez et al., 2024) (MIT license). The objective is to produce a patch given a repository and an issue description, so that the repository after the patch is applied, passes the issue-specific tests (that are never revealed to the agent).

Metrics: We report three metrics: (1) *Resolution rate*, the percentage of issues successfully resolved (i.e., pass the issue-specific tests); (2) *Localization rate*, the percentage of issues where the patch proposed by a method fully covers the ground-truth patch files, i.e., where recall is 100% at the file level; and (3) *Application rate*, the percentage of issues where the patch proposed by a method successfully applies on the repository (i.e., the Linux command `patch` does not raise an error).

Competing methods: We compare with all the existing methods that are also evaluated on SWE-bench Lite (with logs here):

(1) SWE-agent (Yang et al., 2024a): Utilizes a single ReAct loop along with specialized environment

interface with multiple tools. Uses GPT-4 (1106).
(2) AutoCodeRover (Zhang et al., 2024) (ACR): Uses ReAct loops for localization and for generating patches. Uses specialized tools for searching specific code elements (class, method) within other code elements and presenting them as signatures whenever appropriate. Uses GPT-4 (0125).
(3) OpenDevin (OpenDevin): Uses the CodeAct (Wang et al., 2024a) framework where the agent (a single ReAct loop) can execute any bash command along with using various helper commands. The version of OpenDevin with highest reported performance `v1.3_gpt4o` makes use of `hints_text` in SWE-bench Lite, conversation transcript of developers on an issue in GitHub. While we include results from this version, we compare in detail with the highest performing version that does *not* use hints, `v1.5_gpt4o_nohints`.
(4) Aider (Aider): Uses static analysis to provide a compact view of the repository and, in turn, to determine the file(s) to edit. Uses ReAct loop for editing the identified file(s) until a valid patch that passes *pre-existing* tests is obtained. Uses GPT-4o and Claude 3 Opus on alternate runs.
(5) CodeR (Chen et al., 2024): A multi-agent solution which reproduces and resolves the issue iteratively. Uses BM25 along with test coverage statistics for fault localization. Uses GPT-4 (1106).
(6) Moatless (Moatless Tools): Uses a ReAct loop to localize and another to fix the code. Leverages semantic search to query for relevant code chunks.
(7) RAG: Uses BM25 to retrieve relevant files which are used to prompt an LLM to generate a patch. We compare with the best-performing RAG model from the SWE-bench Lite leaderboard (SWE-bench): `RAG + Claude 3 Opus`.
(8) Along with the above, commercial offerings **Amazon Q-Developer** (Amazon), **Bytedance MarsCode** (Bytedance), **OpenCGS Starship** (OpenCGS) and **IBM Research Agent-101** (IBM) have also reported results on SWE-bench Lite. While we report metrics for these, we are unable to conduct further comparisons with them due to non-availability of detailed logs or any information about their approaches. We do not compare with Devin (Devin) as it reports performance a subset of SWE-bench different from SWE-bench Lite.

Implementation: We evaluate MASAI by setting up a fresh development environment with all the requirements and providing the issue description. MASAI generates a single patch which is then evaluated using the SWE-bench Lite testing harness. The `tree-sitter==0.21.1` package is used to implement the lazy representation part of the `READ` function. We use the GPT-4o model throughout our pipeline. For Test Template Generator, we start with a temperature of 0 and increase by 0.2 for each attempt. For Issue Reproducer, Edit Localizer, and Ranker, we use a temperature of 0; for Fixer, we use 0.5 and sample 5 candidate patches. We limit the ReAct loops of the Test Template Generator, Issue Reproducer, and Edit Localizer to 25 steps and limit Test Template Generator to 3 retries. After the ranker selects the patch, we run an auto-import tool to add missing imports. We discard any edits to pre-existing tests which the agent might have made. The per-issue cost for MASAI is \$1.96 on average. We estimate the total cost of our experiments to be <10k USD.

4 Results

We first present comprehensive results on the SWE-bench Lite dataset. Then we provide supporting empirical observations and examples that bring out the effectiveness of our design choices.

4.1 RQ1: Performance on software engineering tasks in SWE-bench Lite

We present our main results in Table 1. Multiple remarks are in order.
(1) Our method, MASAI, achieves the highest resolution rate of 28.33% on the dataset, thereby establishing a state-of-the-art on the benchmark leaderboard alongside CodeR (MASAI).
(2) Standard RAG baseline (first row) performs substantially poor on the dataset, as has also been established in recent works (Jimenez et al., 2024; Chen et al., 2024); which is a strong indication of the complexity of the SWE-bench Lite dataset.
(3) MASAI localizes the issue (at a file-level) in 75% of the cases; the best method in terms of localization rate, OpenCGS Starship, at nearly 91%, however achieves only 23.67% resolution rate.
(4) The (edit) application rate is generally high for all LLM-based agents; MASAI's patches, in particular, successfully apply in over 95% of the cases.

4.2 RQ2: Assumptions by different methods

High autonomy and less dependence on external signals (e.g., expert hints) is desirable from software-engineering agents. In the standard SWE-bench Lite setup, all agents are provided the issue description along with the repository. However,

Method	Resolv. rate (%)	Locl. rate (%)	Appl. rate (%)
RAG	4.33	48.00	51.67
SWE-agent	18.00	61.00	93.67
ACR	19.00	62.33	80.00
Q-Dev	20.33	71.67	97.33
MarsCode	22.00	67.00	83.67
Moatless	23.33	73.00	97.00
Starship	23.67	**90.67**	**99.00**
OpenDevin	25.00	77.00	90.00
– hints	16.00	63.00	81.33
Aider	26.33	69.67	96.67
Agent-101	26.67	72.67	97.33
CodeR	**28.33**	66.67	74.00
MASAI	**28.33**	75.00	95.33

Table 1: Performance of baseline and competing methods on SWE-bench Lite (best in **bold**). Our proposed method, MASAI, achieves the best resolution rate (% issues resolved). Row "– hints" indicates executing OpenDevin without using `hints_text` in the dataset.

we observe that different methods make different assumptions about available auxiliary information.

- All methods apart from RAG and Moatless require that for each task, an environment be set up with the appropriate requirements installed beforehand so that code can be executed.
- OpenDevin avails `hints_text` provided by SWE-bench Lite as discussed in Section 3.
- Aider, when running pre-existing tests, uses predetermined test commands consist of (1) the testing framework used to run tests in the task repository and (2) specific unit tests that target the code pertaining to the issue at hand. The former assumes information about the repository-specific testing framework which is not present in the standard SWE-bench Lite setup. In the case of the latter, providing output from only the target test (and not the whole test suite) during ReAct steps, inadvertently provides additional information about which part of the repository is relevant to the issue.
- CodeR uses coverage-based code ranking (Wong et al., 2016) for fault localization. As in Aider, this would require repository-specific commands to run pre-existing tests, and instrumentation of the full repository to get coverage information.

MASAI aims for high autonomy by avoiding dependence on additional inputs, only relying on the original setup proposed by Jimenez et al. (2024). SWE-agent and AutoCodeRover operate at a similar level of autonomy to MASAI. Results in Table 1 show that MASAI outperforms all other approaches without making additional assumptions.

4.3 RQ3: How does MASAI perform effective fault localization from issue description?

Localization requires multi-step reasoning to identify the root cause of the error from issue descriptions which are often vague and usually only describe the problem being observed. We observe that (1) the choice of ReAct as the strategy, (2) the specificity of its objective (to only identify files to edit) and (3) the designs of tools available enables the Edit Localizer to perform the required multi-step reasoning in a flexible and robust manner. Note that (1) and (2) are results of the modularity of MASAI. SWE-agent and OpenDevin, methods that do not employ a separate localization sub-agent, achieve 61% and 63% localization rates respectively, compared to 75% achieved by MASAI's Edit Localizer.

We observe the advantages of using a ReAct sub-agent, by comparing with Aider which uses a single step CoT approach. In the 27 issues solved by MASAI but not by Aider, Aider failed to localize in 10 (37%) issues whereas among the 21 issues solved by Aider but not by MASAI, MASAI only failed to localize in 3 (14%) issues. This shows that better localization plays a role in superior resolution rate. Comparing the average search steps (as proxy for complexity) required for problems that both Aider and MASAI solved (10.9) and those that only MASAI solved (12.8), we further see that MASAI's ReAct based Edit Localizer has the flexibility to scale to more complex localization challenges.

[Example 1]: MASAI performs **multi-step reasoning** required for localization in the task `scikit-learn__scikit-learn-13142` (described in Fig. 2). Edit Localizer finds the class mentioned in the issue and then traces symbols and inheritance links to identify the root cause.

[Example 2]: The ability of the `READ` action to return **approximate matches** (Section 2) helps in the issue `astropy__astropy-14995`. When the LLM asks for a non-existent `NDDataRef.multiply` method in a file, the action responds with an approximate match `NDArithmeticMixin.multiply` in a different file. Then the sub-agent traces 3 callee links to get to the actual faulty function.

[Example 3]: Access to basic **shell commands** helps the Edit Localizer in the issue

Selection Strategy	1 Sample	5 Samples
Oracle	23.33%	35.00%
Random	-	22.28%
LLM w/o test	-	23.33%
LLM w/ test (Ranker)	-	28.33%

Table 2: Resolution rates of MASAI on SWE-bench Lite, with different number of Fixer samples (i.e., candidate patches), using different sample selection strategies (rows, discussed in Section 4.4).

`matplotlib__matplotlib-25332`. `grep` is used to look for occurrences of an attribute within a large file which helps identify the faulty function.

Neither Aider nor CodeR localized faulty functions correctly in any of the 3 examples. OpenDevin localized Example 2; SWE-agent Examples 2 and 3. Links to the agent logs are in Appendix B.

4.4 RQ4: How does MASAI's sampling and ranking compare to iterative repair?

We observe that sampling multiple repair patches from the Fixer significantly increases the possibility of generating a correct patch, as reported in Table 2 (Oracle selection 23.33% at 1 sample vs 35% at 5 samples). However the LLM alone is unable to select amongst theses patches (LLM w/o test). This can be overcome by using the output from the generated issue-reproduction test on each patch for ranking the patches (LLM w/ test (Ranker)).

MASAI exploits the above observations through its modularity by (1) leveraging a CoT sampling strategy for Fixer and (2) instantiating independent sub-agents for test generation and repair. Other methods rely on an iterative approach to extract multiple edits from the LLM asking it to iteratively fix any mistakes it has made.

We evaluate the benefits of our approach empirically in Table 3. By controlling for localization, we are comparing the effectiveness of completing the repair. MASAI is substantially more effective at this than most methods, barring CodeR and Aider.

As as example, consider the issue `django__django-14787` where CodeR, Aider, OpenDevin and MASAI all correctly localize the issue, but only MASAI solves it correctly. While iterative methods sample one candidate and keep refining it without success, MASAI's Fixer sub-agent generates 5 samples out of which only one is correct – demonstrating the importance for diverse sampling. MASAI's Ranker correctly ranks these by utilizing outputs from running the generated reproduction test. Aider submits patch which passes pre-existing tests but is actually incorrect, showing the importance of the generated reproduction test to eliminate false positives.

Method	Both locl.	Method resolv.	MASAI resolv.
RAG	126	12	**52** (+ 31.7%)
ACR	166	51	**73** (+ 13.2%)
Q-Dev	191	55	**75** (+ 10.5%)
SWE-agent	166	48	**65** (+ 10.2%)
Starship	220	62	**81** (+ 8.6%)
OpenDevin	187	60	**74** (+ 7.5%)
– hints	164	39	**67** (+ 17.1%)
Moatless	193	62	**75** (+ 6.7%)
MarsCode	182	59	**71** (+ 6.6%)
Agent-101	193	69	**72** (+ 1.6%)
Aider	189	**71**	**71** (=)
CodeR	174	**77**	72 (- 0.3%)

Table 3: Number of issues resolved by a method (Method resolv.) named in the rows and by MASAI (MASAI resolv.) among the issues that are successfully localized by both MASAI and the method ("Both locl." column, out of 300). Row-wise max. in bold.

4.5 RQ5: How does MASAI perform effective issue reproduction?

As discussed in the previous RQ, the ability to generate tests that reproduce the stated issue is critical to select Fixer samples. Often repositories employ uncommon testing frameworks, that makes this task hard. Consider the issue `django__django-14672`. This repository proved hard to write tests for since it uses a custom testing framework, which involved having all new test classes derive from a certain base class to run. OpenDevin was unable to reproduce the test; in its attempt to install pytest, it ran out of budget and failed to solve this issue.

To remedy this, we decompose test reproduction into two steps: (1) Test Template Generator reads documentation/existing tests to **generate a sample test template** and instructions to run; (2) Issue Reproducer then uses the template as an example to **create an issue specific test** . This improves the overall capability of reproducing tests in MASAI, as seen in our logs (see Supplementary Material) for the above example — Test Template Generator first goes through the repository, creates a template file demonstrating an example test case as well as the correct command to run it; the Issue Reproducer subsequently reproduces the issue correctly, without running into problems that OpenDevin faced.

4.6 RQ6: How does MASAI generate edits that can be applied successfully?

The representation used to encode edits can have a large impact on the performance. As discussed in Section 2, MASAI prompts the LLM for edits, in the form of a **minimal rewrite** — to reproduce the current state of the code snippet it wants to edit, followed by the edited version of this snippet. Recall that we also employ **fuzzy matching** to find the relevant span in the file, by searching for the snippet that best fuzzily matches with the one provided by the model. This mitigates copying or line counting mistakes by the LLM, significantly reducing the number of syntax errors introduced when editing. Our edit representation and fuzzing matching together yield 96.33% edit application rate (Table 1) which is among the highest.

5 Related Work

We have already discussed competing methods evaluated on SWE-bench Lite, in Sections 3 and 4. We now highlight other related work on LLM-powered agents.

Software-engineering agents: Language Agent Tree Search (Zhou et al., 2023) synergizes reasoning, planning, and acting abilities of LLMs. Their strategy relies on determining partial or full termination of the search (e.g., by running provided golden test cases for successful code generation as in HumanEval (Chen et al., 2021)) and backtracking if necessary; this is often infeasible in complex software engineering tasks we tackle in this paper. CodePlan (Bairi et al., 2023) combines LLMs with static analysis-backed planning for repository-level software engineering tasks such as package migration. It relies on compiler feedback and dependency graphs to guide the localization of edits; unlike in our general setting, where the agents are more autonomous, and are equipped to discover localization strategies. AlphaCodium (Ridnik et al., 2024) differs from MASAI in that (1) it uses public *and* AI-generated test cases for filtering; (2) is evaluated in the generation (NL2Code) setting.

Conversational and multi-agent frameworks: In this line of work (Guo et al., 2024; Yang et al., 2024b), (1) the focus is often on the high level aspects of agent design such as conversation protocols. AutoGen (Wu et al., 2023) and AgentVerse (Chen et al., 2023) provide abstractions for agent interactions and conversational programming for design of multi-agent systems; similarly, Dynamic agent networks (Liu et al., 2023) focuses on inference-time agent selection and agent team optimization; and (2) the frameworks are typically instantiated on standard RL or relatively simpler code generation datasets. For instance, AutoDev (Tufano et al., 2024) can execute actions like file editing, retrieval, testing, but is evaluated on the HumanEval (Chen et al., 2021) NL2Code dataset. Similarly, MetaGPT (Hong et al., 2024) and ChatDev (Qian et al., 2023), dialogue-based cooperative agent frameworks, are instantiated on generation tasks involving a few hundred lines of code.

In contrast, we focus on designing a modularized agent architecture for solving complex, real-world software engineering tasks, as exemplified by the SWE-bench Lite dataset.

Divide-and-Conquer approaches: In this line of work, the given complex task is broken down into multiple sub-goals that are solved individually, and then the solution for the task is synthesized. Multi-level Compositional Reasoning (MCR) Agent (Bhambri et al., 2023) uses compositional reasoning for instruction following in environments with partial observability and requiring long-horizon planning, such as in robotic navigation. Compositional T2I (Wang et al., 2024b) agent uses divide-and-conquer strategy for generating images from complex textual descriptions. SwiftSage (Lin et al., 2024) agent, inspired by the dual-process theory of human cognition for solving tasks, e.g., closed-world scientific experiments (Wang et al., 2022), uses finetuned SLM policy (“Swift”) to decide and execute fast actions, and an LLM (“Sage”) for deliberate planning of sub-goals and for backtracking when necessary.

6 Conclusions

As divide-and-conquer helps humans overcome complexity, similar approaches to modularize tasks into sub-tasks can help AI agents as well. In this work, we presented a modular architecture, MASAI, for software-engineering agents. Encouraged by the effectiveness of MASAI on SWE-bench Lite, we plan to extend it to a larger range of software-engineering tasks, which will also involve building realistic and diverse datasets.

7 Limitations

Our evaluation is centered on the widely-used SWE-bench Lite dataset for evaluating software-engineering AI agents. It allowed us to do head-to-head comparison with many agents. However, the breadth of issues covered in SWE-bench Lite is limited to those that can be validated using tests. In future, we expect us and the community to expand the scope to more diverse issues.

There are a number of LLMs that support code understanding and generation. The modularity of MASAI permits use of different language models in different sub-agents. Due to the time and cost constraints, we have instantiated all sub-agents with GPT-4o. The cost-performance tradeoff of using different LLMs and possibly, even small language models (SLMs) is an interesting research problem. The competing methods that we compared against do employ different LLMs, but this still leaves out direct comparison of different LLMs on a fixed solution strategy.

The issue descriptions in SWE-bench Lite are all in English. This leaves out issues from a large segment of non-English speaking developers. The increasing support for the diverse world languages by LLMs should enable multi-lingual evaluation even in the software engineering domain, which is a problem that we are excited about.

8 Broader Concerns

Agentic frameworks with the ability to use tools like shell commands can lead to unintended side-effects on the user's system. Appropriate guardrails and sandboxing can mitigate such problems.

Our approach contributes towards the development of tools to autonomously perform software development tasks. This raises various security concerns. The tool may not always follow best practices when writing or editing code, leading to introduction of security vulnerabilities and bugs. Therefore, it is important for code changes suggested by the tool to be reviewed by expert developers before being deployed to real world systems.

As mentioned in the Section 7, the dataset we evaluate on (SWE-bench Lite) as well as the model we use (GPT-4o) are primarily in English. This limits the usability of our tool to software engineers proficient in English. Further work is necessary in developing methods for non-English speaking developers in order to prevent this population from being marginalized.

References

Aider. https://aider.chat/2024/06/02/main-swe-bench.html.

Amazon. https://aws.amazon.com/q/developer/.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, et al. 2023. CodePlan: Repository-level coding using LLMs and planning. *arXiv preprint arXiv:2309.12499*.

Suvaansh Bhambri, Byeonghwi Kim, and Jonghyun Choi. 2023. Multi-level compositional reasoning for interactive instruction following. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 223–231.

Bytedance. https://www.marscode.com/.

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*.

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.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*.

CodeGemma Team. 2024. CodeGemma: Open Code Models Based on Gemma.

Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing rust compilation errors using llms. *arXiv preprint arXiv:2308.05177*.

Devin. Introducing Devin, the first AI software engineer. https://www.cognition.ai/blog/introducing-devin.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024. MetaGPT: Meta programming for Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR.

IBM. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240612_IBM_Research_Agent101.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Bill Yuchen Lin, Yicheng Fu, Karina Yang, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Prithviraj Ammanabrolu, Yejin Choi, and Xiang Ren. 2024. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *Advances in Neural Information Processing Systems*, 36.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2023. Dynamic LLM-agent network: An LLM-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*.

MASAI. https://github.com/swe-bench/experiments/pull/20.

Moatless Tools. https://github.com/aorwall/moatless-tools.

OpenCGS. https://opencsg.com/product?class=StarShip.

OpenDevin. https://opendevin.github.io/OpenDevin/.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, et al. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

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

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

SWE-bench. https://www.swebench.com/.

Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-Driven Development. *arXiv preprint arXiv:2403.08299*.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. 2022. Scienceworld: Is your agent smarter than a 5th grader? *Preprint*, arXiv:2203.07540.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*.

Zhenyu Wang, Enze Xie, Aoxue Li, Zhongdao Wang, Xihui Liu, and Zhenguo Li. 2024b. Divide and conquer: Language models can plan and self-correct for compositional text-to-image generation. *arXiv e-prints*, pages arXiv–2401.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024a. SWE-agent: Agent computer interfaces enable software engineering language models.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, et al. 2024b. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *arXiv e-prints*, pages arXiv–2401.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*.

Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, et al. 2023. Igniting language intelligence: The hitchhiker's guide from chain-of-thought reasoning to language agents. *arXiv preprint arXiv:2311.11797*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

A Prompts used in MASAI sub-agents

Test Template Generator Sub-agent Prompt

```
You are an expert developer who can reproduce GitHub issues.

Your goal is to generate a report on how to write a standalone test(using an example already present
in the repository) and run it.

Here is the structure of the repository:
{{repo_structure}}
{% if testing_docs %}

Here are some relevant files and guidelines for testing in this repository:

{{ testing_docs }}
{% else %}
{% endif %}
You can perform the following actions while trying to figure this out:

1. LIST: List all the files in a folder
2. READ: Read the code of a function, class or file
3. WRITE: Write to a new file in the repository.
4. COMMAND: Run a shell command in the repository
5. DONE: Once you have resolved the issue, respond with the DONE action

You should specify which action to execute in the following format:

If you want to READ a function 'ABC' in class 'PQR' in file 'XYZ', respond as

<reasoning>...</reasoning>
<action>READ</action>
<file>XYZ</file>
<class>PQR</class>
<function>ABC</function>.


It's okay if you don't know all the three attributes. Even 2 attributes like function name and class
name is okay.

If you don't know the location of a file, you can LIST or 'ls' a folder FGH by saying:

<reasoning>...</reasoning>
<action>LIST</action>
<folder>FGH</folder>

As an example, if you want to READ the function get_symbolic_name from class ASTNode, then respond:

<reasoning>The function get_symbolic_name appears to be faulty when run with the verbose=False flag
and doesn't log the stacktrace. Reading it might give more hints as to where the underlying problem
would be.</reasoning>
<action>READ</action>
<class>ASTNode</class>
<function>get_symbolic_name</function>

Note that reading a file will not give the full functions inside the file. To read the full body of a
function, specify the name of the function explicitly.

Or, if you want to LIST a folder src/templates, respond:

<action>LIST</action>
<folder>src/templates</folder>



You need to write a testing script to reproduce this issue.

To write a script, you can use the WRITE action
```

```
<reasoning>...</reasoning>
<action>WRITE</action>
<file>XYZ</file>
<contents>
...
</contents>
Write perfectly correct code in the contents. Do not use ... in the code.
However, remember that WRITE will overwrite a file if it already exists.

For examples to write a script in the tests/ directory of the project to call a simple function from a
 repository, you could
<reasoning>Test whether function apply_operators works as expected</reasoning>
<action>WRITE</action>
<file>tests/my_script.py</file>
<contents>
from src.main import Generator

generator = Generator(name='start')
generator.apply_operators('+', '*')
</contents>

You can also execute shell actions using the COMMAND action like so
<reasoning>...</reasoning>
<action>COMMAND</action>
<command>XYZ</command>

For example if you want to run tests/my_script.py in the root directory of the repository, then
respond as
<reasoning>...</reasoning>
<action>COMMAND</action>
<file>python tests/my_script.py</file>

You can also make use of various shell utilities like grep, cat, etc... to debug the issue. For
example

<reasoning>...</reasoning>
<action>COMMAND</action>
<command>grep -r "get_symbolic_name" .</command>

<reasoning>...</reasoning>
<action>COMMAND</action>
<command>ls src/utils</command>

The COMMAND action can also be used to execute arbitrary executables present in either the PATH or the
 repo.

You can read the documentation to figure out how the test files look like. If you figure that out, try
 to integrate the test into the framework. Then, figure out how to run the tests and run them to
verify that the test case runs properly.
Only output one action at a time. Do not edit/overwrite any existing files.

Also, if a bash command is not available, try to find the right testing framework instead of assuming
its presence. A non-working report is NOT ACCEPTABLE. Keep trying if it doesn't work.

You can accomplish this task by doing the following activities one by one:

1. Find the folder/files which contains the tests.
2. You should read documentation such as README/docs/testing guides and figure out how tests are run.
This step is really important as there are custom functions to run tests in every repository.
3. READ an existing test file.
4. Run the existing test file using the commands discovered previously. This is a very important step.
5. WRITE a new standalone test to a new file. Try to make sure it is as simple as possible.
6. Run the test using the COMMAND action and verify that is works.
7. Keep trying to edit your scripts unless your test works PERFECTLY.

Ensure that the test you have written passes without any errors.
Once, you are done, use the DONE action like so along with a report of how to run the test.
```

```
<report>
<file>file_name</file>
<code>
...
</code>
<command>
....
</command>
</report>
<action>DONE</action>

For instance, if the repo requires pytest to be used on a file called tests/new_test.py to test the
capitalize function, then you can say:

<report>
<file>tests/new_test.py</file>
<code>
def test_dummy():
    assert True == True
</code>
<command>
pytest tests/new_test.py
</command>
</report>
<action>DONE</action>

If the test that you write doesn't emit any output, you can add print statements in the middle to make
 sure that it is actually executing.
Do not attempt to install any packages or load an environment. The current environment is sufficient
and contains all the necessary packages.
```

Issue Reproducer Sub-agent Prompt

```
You are an expert developer who can reproduce GitHub issues.

<issue>
{{ problem_statement }}
</issue>

Your goal is to generate a report on how to write a test to reproduce the bug/feature request present
in the issue and run it.

Here is the structure of the repository:
{{ repo_structure }}

{% if reproduction_report %}
Here is an example of how tests can be generated and run in the repository:

### Example:
{{ reproduction_report }}

### Instructions:
The command in <command>...</command> denotes how to run the test and <code>...</code> denotes the
example test.
{% endif %}

You can perform the following actions while trying to figure this out:

1. LIST: List all the files in a folder
2. READ: Read the code of a function, class or file
3. WRITE: Write to a new file in the repository.
4. COMMAND: Run a shell command in the repository
5. DONE: Once you have resolved the issue, respond with the DONE action

You should specify which action to execute in the following format:
```

```
If you want to READ a function 'ABC' in class 'PQR' in file 'XYZ', respond as

<reasoning>...</reasoning>
<action>READ</action>
<file>XYZ</file>
<class>PQR</class>
<function>ABC</function>.


It's okay if you don't know all the three attributes. Even 2 attributes like function name and class
name is okay.

If you don't know the location of a file, you can LIST or 'ls' a folder FGH by saying:

<reasoning>...</reasoning>
<action>LIST</action>
<folder>FGH</folder>

As an example, if you want to READ the function get_symbolic_name from class ASTNode, then respond:

<reasoning>The function get_symbolic_name appears to be faulty when run with the verbose=False flag
and doesn't log the stacktrace. Reading it might give more hints as to where the underlying problem
would be.</reasoning>
<action>READ</action>
<class>ASTNode</class>
<function>get_symbolic_name</function>

Note that if you read a file, it will list function in their folded form. To read a specific function,
 you need to specify the function parameter while doing a READ.

Or, if you want to LIST a folder src/templates, respond:

<action>LIST</action>
<folder>src/templates</folder>



You need to write a testing script to reproduce this issue.

To write a script, you can use the WRITE action
<reasoning>...</reasoning>
<action>WRITE</action>
<file>XYZ</file>
<contents>
...
</contents>
Write perfectly correct code in the contents. Do not use ... in the code.
However, remember that WRITE will overwrite a file if it already exists.

For examples to write a script in the tests/ directory of the project to call a simple function from a
 repository, you could
<reasoning>Test whether function apply_operators works as expected</reasoning>
<action>WRITE</action>
<file>tests/my_script.py</file>
<contents>
from src.main import Generator

generator = Generator(name='start')
generator.apply_operators('+', '*')
</contents>

You can also execute shell actions using the COMMAND action like so
<reasoning>...</reasoning>
<action>COMMAND</action>
<command>XYZ</command>

For example if you want to run tests/my_script.py in the root directory of the respository, then
respond as
```

```
<reasoning>...</reasoning>
<action>COMMAND</action>
<file>python tests/my_script.py</file>

You can also make use of various shell utilities like grep, cat, etc... to debug the issue. For
example

<reasoning>...</reasoning>
<action>COMMAND</action>
<command>grep -r "get_symbolic_name" .</command>

<reasoning>...</reasoning>
<action>COMMAND</action>
<command>ls src/utils</command>

The COMMAND action can also be used to execute arbitrary executables present in either the PATH or the
 repo.

You should take a look at how tests are generated. You can also read other existing test files to see
how to instrument the test case to reproduce this issue.
Only output one action at a time. Do not edit/overwrite any existing files. Always write your test in
a new file.

Also, if a bash command is not available, you can install it using pip. The non-working test is NOT
ACCEPTABLE. Keep trying if it doesn't work.

{% if reproduction_report %}
You can accomplish this task by doing the following activities one by one:
1. Read the example on how to write the test{% if reproduction_report %}(see the #Example){% endif %}.
2. Write a test to replicate the issue.
3. Execute the test until it is able to replicate the issue.
4. If you're stuck about how to execute, read other test files.
{% endif %}
Once, you are done, use the DONE action like so along with a report of how to run the test.

<report>
<file>new_file_name</file>
<code>
...
</code>
<command>
....
</command>
</report>
<action>DONE</action>


For instance, if the repo requires pytest to be used on a file called tests/issue_reproduction.py to
test the capitalize function, then you can say:

<report>
<file>tests/issue_reproduction.py</file>
<code>
# Code for a test case that replicates the issue. It should pass when the repository is fixed.
</code>
<command>
pytest tests/issue_reproduction.py
</command>
</report>
<action>DONE</action>

For reference, use the ### Example above. Start by writing the test for this issue and then try to get
 it running. Use the <command>...</command> to run the tests. Do not try to use other commands.
Do not explore the testing framework. Only if you are stuck, you should see some of the already
written tests to get a reference. Do not write on any files other than the test files. Don't try to
solve the issue yourself. Only write the test.
```

Edit Localizer Sub-agent Prompt

```
You are an expert developer who can understand issues raised on a repository. You task is to find the
root cause of the issue and identify which parts of the resposoitory require edits to resolve the
issue.
Search the repository by going through code that may be related to the issue. Explore all the
necessary code needed to fix the issue and look up all possible files, classes and functions that are
used and can be used to fix the issue. Also search for other potential functions that solve the issue
to ensure code consistency and quality.
The issues raised can be about using the code from the provided repository as a framework or library
in the user code.
Keep this in mind when understanding what might be going wrong in the provided repository (framework/
library) rather than in the user code.

Follow the above steps to debug the following issue raised in the repository named: {{ repo }} -

<issue>
{{ problem_statement }}
</issue>
{% if issue_hints %}
{{ issue_hints }}
{% endif %}
Your end goal is to identify which parts of the resposoitory require edits to resolve the issue.

Here is the structure of the repository:
{{repo_structure}}

You can perform the following actions while debugging this issue -

1. READ: Read the code of a function, class or file
2. COMMAND: Run a shell command in the repository.
3. EDIT: Mark a file, class or file in the repository for editing.
4. ADD: Mark a new function, class or file to be added to the repository.
5. DONE: Once you have identified all code requiring edits to resolve the issue, respond with the DONE
.

You should specify which action to execute in the following format -

If you want to EDIT/READ a function 'ABC' in class 'PQR' in file 'XYZ', respond as

<reasoning>...</reasoning>
<action>EDIT/READ</action>
<file>XYZ</file>
<class>PQR</class>
<function>ABC</function>.


It's okay if you don't know all the three attributes. Even 2 attributes like function name and class
name is okay.
Also, do not EDIT a function before you READ it.

If you want to add some code(maybe a function) to a file, then use the ADD action like so

<reasoning>...</reasoning>
<action>ADD</action>
<file>XYZ</file>
<class>PQR</class>
<function>function_to_be_added</function>

If you don't know the location of a file, you can LIST or 'ls' a folder FGH by saying:

<action>LIST</action>
<folder>FGH</folder>

As an example, if you want to READ the function get_symbolic_name from class ASTNode, then respond:

<reasoning>The function get_symbolic_name appears to be faulty when run with the verbose=False flag
and doesn't log the stacktrace. Reading it might give more hints as to where the underlying problem
```

```
would be.</reasoning>
<action>READ</action>
<class>ASTNode</class>
<function>get_symbolic_name</function>

Or, if you want to add a function validate_params to a file src/validator.py, respond:
<action>ADD</action>
<file>src/validator.py</file>
<function>validate_params</function>

Or, if you want to LIST a folder src/templates, respond:

<action>LIST</action>
<folder>src/templates</folder>

Or, if you want to READ a file name symbolic_solver/src/templates/numerics.py and a function
get_string_repr in the repository, then use the -AND- tag to separate the two responses as follows:

<reasoning>The file symbolic_solver/src/templates/numerics.py seems to contain important classes which
 extend BaseSymbol along with their implementations of get_symbolic_name and solve_symbolic_system</
reasoning>
<action>READ</action>
<file>symbolic_solver/src/templates/numerics.py</file>
-AND-
<reasoning>The function get_string_repr is used in the code and might be causing the issue. Reading it
 might give more hints as to where the underlying problem would be.</reasoning>
<action>READ</action>
<function>get_string_repr</function>

You can also execute shell actions using the COMMAND action like so
<reasoning>...</reasoning>
<action>COMMAND</action>
<command>XYZ</command>

For example if you want to run my_script.py in the root directory of the respository, then respond as
<reasoning>...</reasoning>
<action>COMMAND</action>
<file>python my_script.py</file>

You can also make use of various shell utilities like ls, grep, cat, etc... to debug the issue. For
example

<reasoning>...</reasoning>
<action>COMMAND</action>
<command>grep -r "get_symbolic_name" .</command>

<reasoning>...</reasoning>
<action>COMMAND</action>
<command>ls src/utils</command>

The COMMAND action can also be used to execute arbitrary executables present in either the PATH or the
 repo that may be required for debugging.


Try and read all possible locations which can have buggy code or can be useful for fixing the issue.
Ensure that you don't query for the same function or class again and again. While giving a file/class/
function to read/edit, make sure that you only query for item at a time. Make sure you dont mark
pieces of code for editing unnecessarily. Do not try to edit tests. They will be fixed later.

Once you have made the identified all the parts of the code requiring edits to resolve the issue, you
should respond with the DONE action.

<reasoning>...</reasoning>
<action>DONE</action>
```

Fixer Sub-agent Prompt

```
You are given the following {{ language }} code snippets from one or more '{{ extension }}' files:
<codebase>
{{ code_snippets }}
</codebase>
Instructions: You will be provided with a partial codebase containing a list of functions and an issue
 statement explaining a problem to resolve from the repo {{ repo_name }}.

### Issue:
{{ issue_description }}
{% if issue_hints %}
{{ issue_hints }}
{% endif %}
{% if localization %}{{ localization }}
{% endif %}

{% if testcase %}
### Testcase:
Here are testcases that should pass on correct resolution of the issue.
{{ testcase }}
{% endif %}
{% if feedback %}{{ feedback }}
{% endif %}

Solve the issue by giving changes to be done in the functions using all the information given above in
 the format mentioned below. All the necessary information has already been provided to you.
---
For your response, return one or more ChangeLogs (CLs) formatted as follows. Each CL must contain one
or more code snippet changes for a single file. There can be multiple CLs for a single file. Each CL
must start with a description of its changes. The CL must then list one or more pairs of (OriginalCode
, ChangedCode) code snippets. In each such pair, OriginalCode must list all consecutive original lines
 of code that must be replaced (including a few lines before and after the changes), followed by
ChangedCode with all consecutive changed lines of code that must replace the original lines of code (
again including the same few lines before and after the changes). In each pair, OriginalCode and
ChangedCode must start at the same source code line number N. Each listed code line, in both the
OriginalCode and ChangedCode snippets, must be prefixed with [N] that matches the line index N in the
above snippets, and then be prefixed with exactly the same whitespace indentation as the original
snippets above. See also the following examples of the expected response format.
---
Plan: Step by step plan to make the edit and the logic behind it.
ChangeLog:1@<complete file path>
Description: Short description of the edit.
OriginalCode@4:
[4] <white space> <original code line>
[5] <white space> <original code line>
[6] <white space> <original code line>
ChangedCode@4:
[4] <white space> <changed code line>
[5] <white space> <changed code line>
[6] <white space> <changed code line>
OriginalCode@9:
[9] <white space> <original code line>
[10] <white space> <original code line>
ChangedCode@9:
[9] <white space> <changed code line>
...
Plan: Step by step plan to make the edit and the logic behind it.
ChangeLog:K@<complete file path>
Description: Short description of the edit.
OriginalCode@15
[15] <white space> <original code line>
[16] <white space> <original code line>
ChangedCode@15:
[15] <white space> <changed code line>
[16] <white space> <changed code line>
[17] <white space> <changed code line>
OriginalCode@23:
```

```
[23] <white space> <original code line>
ChangedCode@23:
[23] <white space> <changed code line>
---

For instance, consider the following code snippet:

Code snippet from file 'runner/src/orchestrator.py' (lines: 0 to 22):
[0]"""
[1]Orchestrator for experimental pipeline
[2]"""
[3]
[4]if __name__ == "__main__":
[5]
[6]    import argparse
[7]    import dotenv
[8]    from pathlib import Path
[9]
[10]    from masai.config import ExpConfig
[11]    from masai.pipeline import pipeline
[12]
[13]    dotenv.load_dotenv()
[14]
[15]    parser = argparse.ArgumentParser()
[16]    parser.add_argument("--config", type=Path, default=Path("pipeline-config.yaml"))
[17]    args = parser.parse_args()
[18]
[19]    config_path = Path(args.config)
[20]    config = ExpConfig.from_yaml_file(config_path=config_path)
[21]    pipeline(config)
[22]

If the issue wants the path of the config to be validated before hand and the final looks like this:

[0]"""
[1]Orchestrator for experimental pipeline
[2]"""
[3]import os
[4]
[5]def sanity_check(config_path):
[6]    """
[7]    Check if the config_path is a valid path.
[8]    """
[9]    return os.path.exists(config_path)
[10]
[11]if __name__ == "__main__":
[12]
[13]    import argparse
[14]    import dotenv
[15]    from pathlib import Path
[16]
[17]    from masai.config import ExpConfig
[18]    from masai.pipeline import pipeline
[19]
[20]    dotenv.load_dotenv()
[21]
[22]    parser = argparse.ArgumentParser()
[23]    parser.add_argument("--config", type=Path, default=Path("pipeline-config.yaml"))
[24]    args = parser.parse_args()
[25]    # Check if path passes the sanity_check
[26]    if not sanity_check(args.config):
[27]        raise ValueError("Invalid config path provided.")
[28]
[29]    config_path = Path(args.config)
[30]    config = ExpConfig.from_yaml_file(config_path=config_path)
[31]    pipeline(config)
[32]
```

```
Then, your output should be:

Plan: First, we need to add a function called sanity_check which will check if the file exists. Then,
we will edit the code to perform the check after the arguments have been processed.
ChangeLog:1@runner/src/orchestrator.py
Description: Added sanity_check for checking config path.
OriginalCode@3:
[3]
[4]if __name__ == "__main__":
ChangedCode@3:
[3]import os
[4]
[5]def sanity_check(config_path):
[6]    """
[7]    Check if the config_path is a valid path.
[8]    """
[9]    return os.path.exists(config_path)
[10]
[11]if __name__ == "__main__":
OriginalCode@17:
[17]    args = parser.parse_args()
[18]
[19]    config_path = Path(args.config)
ChangedCode@17:
[17]    args = parser.parse_args()
[18]    # Check if path passes the sanity_check
[19]    if not sanity_check(args.config):
[20]        raise ValueError("Invalid config path provided.")
[21]
[22]    config_path = Path(args.config)

Now try to solve the issue given above.

Make sure to follow these rules while giving changelog response:
1. Ensure that your changelogs are always less that 10 lines for each change that is made.
2. Ensure that OriginalCode and ChangedCode pairs always start from the same line number.
3. Give comments on every change you make in the ChangedCode explaining what change you made.
4. OriginalCode and ChangedCode pairs should always have some difference.
5. Do not add any text after the changelog.


Make sure you plan out the edit first before giving the Changelog.
```

Ranker Sub-agent Prompt

```
You are an senior software developer who can review solutions to issues raised on large repository.
You should first consider the description of the issues to understand the problem and then carefully
consider multiple solutions that have been proposed.
{% if testcase %}
Here are some example of how you can rank solutions to issues.


# Example 1:

### Issue:
bin_search doesn't work accurately on edge-cases such as single element arrays or None inputs.
Here is an example:


>>> from utils import bin_search
>>> bin_search([5], 5)
-1
>>> bin_search(None, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "/home/utils.py", line 23, in bin_search
    left, right = 0, len(arr)-1
                     ^^^^^^^^
TypeError: object of type 'NoneType' has no len()


### Possible buggy code:

File: utils.py

def bin_search(arr, key):
    # Returns index of the key in sorted array
    # If element is not present, returns -1.
    left, right = 0, len(arr)-1
    while left < right:
        mid = (left + right) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            left = mid + 1
        else:
            right = mid - 1
    return -1


### Test case:
A junior has proposed the following test case. It might be useful for you in making your judgement.

import pytest

def test_bin_search():
    assert bin_search([5], 5) == 0
    assert bin_search(None, 5)
    assert bin_search([1,2,3,4], 4) == 3

On running the test case on the EARLIER state of the repository, the output obtained was(note that
empty output generally means that the tests passed):

### Initial Test Status:

================================================================== test session starts
==================================================================
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/
plugins: anyio-4.2.0
collected 1 item

utils_test.py F
                                            [100%]

======================================================================= FAILURES
=======================================================================
____________________________________________________________________ test_bin_search
____________________________________________________________________

    def test_bin_search():
>       assert bin_search([5], 5) == 0
E       assert -1 == 0
E        +  where -1 = bin_search([5], 5)

utils_test.py:21: AssertionError
================================================================ short test summary info
================================================================
FAILED utils_test.py::test_bin_search - assert -1 == 0
=================================================================== 1 failed in 0.04s
===================================================================
```

```
### Proposed solution patches:

### Proposed patch number 1:

--- a/utils.py
+++ b/utils.py
@@ -1,4 +1,6 @@
 def bin_search(arr, key):
+    if len(arr) == 1:
+        return 0
     # Returns index of the key in sorted array
     # If element is not present, returns -1.
     left, right = 0, len(arr)-1

After incorporating this change, the test output is:

### New Test Status 1:
================================================================== test session starts
==================================================================
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/
plugins: anyio-4.2.0
collected 1 item

utils_test.py F
                                            [100%]

======================================================================= FAILURES
=======================================================================
____________________________________________________________________ test_bin_search
____________________________________________________________________

    def test_bin_search():
        assert bin_search([5], 5) == 0
>       assert bin_search(None, 5)

utils_test.py:22:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

arr = None, key = 5

    def bin_search(arr, key):
>       if len(arr) == 1:
E       TypeError: object of type 'NoneType' has no len()

utils.py:2: TypeError
================================================================ short test summary info
================================================================
FAILED utils_test.py::test_bin_search - TypeError: object of type 'NoneType' has no len()
=================================================================== 1 failed in 0.04s
===================================================================
---

### Proposed patch number 2:

--- a/utils.py
+++ b/utils.py
@@ -2,7 +2,7 @@ def bin_search(arr, key):
     # Returns index of the key in sorted array
     # If element is not present, returns -1.
     left, right = 0, len(arr)-1
-    while left < right:
+    while left <= right:
         mid = (left + right) // 2
         if arr[mid] == key:
             return mid
```

951

```
After incorporating this change, the test output is:

### New Test Status 2:
---
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/
plugins: anyio-4.2.0
collected 1 item

utils_test.py F
                                            [100%]

======================================================================= FAILURES
========================================================================
____________________________________________________________________ test_bin_search
____________________________________________________________________

    def test_bin_search():
        assert bin_search([5], 5) == 0
>       assert bin_search(None, 5)

utils_test.py:22:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

arr = None, key = 5

    def bin_search(arr, key):
        # Returns index of the key in sorted array
        # If element is not present, returns -1.
>       left, right = 0, len(arr)-1
E       TypeError: object of type 'NoneType' has no len()

utils.py:4: TypeError
================================================================ short test summary info
================================================================
FAILED utils_test.py::test_bin_search - TypeError: object of type 'NoneType' has no len()
=================================================================== 1 failed in 0.04s
===================================================================
---

### Proposed patch number 3:

--- a/utils.py
+++ b/utils.py
@@ -1,8 +1,10 @@
 def bin_search(arr, key):
     # Returns index of the key in sorted array
     # If element is not present, returns -1.
+    if arr is None:
+        return -1
     left, right = 0, len(arr)-1
-    while left < right:
+    while left <= right:
         mid = (left + right) // 2
         if arr[mid] == key:
             return mid

After incorporating this change, the test output is:

### New Test Status 3:
================================================================== test session starts
==================================================================
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/
plugins: anyio-4.2.0
collected 1 item
```

```
utils_test.py .
                                            [100%]

=================================================================== 1 passed in 0.00s
===================================================================

### Response:

### Test Description:
The test runs the function on different values of the key and the array arr. All of these should pass
when the function is bug-free.

### Test Status:
Failing Initially

### Patch description:
[
    {
        "patch_number": 1,
        "test_effect": "The test still fails, but a new TypeError is raised instead of the old error
.",
        "test_status": "FAIL_TO_FAIL",
        "patch_effect": "The patch adds a special edge case for single length error. However it doesn'
t fix the fundamental error in the step where the left < right is wrong."
    },
    {
        "patch_number": 2,
        "test_effect": "The test still fails, but the TypeError is no longer raised.",
        "test_status": "FAIL_TO_FAIL",
        "patch_effect": "The patch fixed the most important part of the testcase where the left <
right was fixed however, the None array case is not handled properly which leads to the TypeError."
    },
    {
        "patch_number": 3,
        "test_effect": "The test passes.",
        "test_status": "FAIL_TO_PASS",
        "patch_effect": "The patch fixed left < right condition and handled the the None array case as
 well."
    }
]

### Ranking description:
Patch 1 doesn't fix the root cause of the problem and is only a superficial solution. Patch 2 and 3
both fix the root problem in the binary search function, however patch 3 handled the additional case
where a None object can be passed as well. Therefore the ranking should be [3] > [2] > [1]

### Ranking:
[3] > [2] > [1]




# Example 2:

### Issue:
Mailer fails when username contains an '@' symbol.

For example:
>>> from mailer import send_notification
>>> send_notification("Test message", "user@invalid@google.com")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/mailer.py", line 16, in send_notification
    return mailer.send_mail(email_id.split("@")[0], email_id.split("@")[1], title="Notification", body
=msg)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/mailer.py", line 10, in send_mail
```

```
    raise InvalidDomainException(f"Domain: {domain} doesn't exist.")
mailer.InvalidDomainException: Domain: invalid doesn't exist.

### Possible buggy code:

File: mailer.py

def send_notification(msg, email_id):
    mailer = Mailer()
    return mailer.send_mail(email_id.split("@")[0], email_id.split("@")[1], title="Notification", body
=msg)


### Test case:
A junior has proposed the following test case. It might be useful for you in making your judgement.

from mailer import send_notification
import pytest

def test_send_notification():
    with pytest.raises(Exception):
        assert send_notification("Test message", "user@invalid@example.com") == 0

On running the test case on the EARLIER state of the repository, the output obtained was(note that
empty output generally means that the tests passed):
### Initial Test Status:

================================================================================= test session
starts =================================================================================
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/testcase
plugins: anyio-4.2.0
collected 1 item

test_mailer.py .
                                                                              [100%]

================================================================================== 1 passed in 0.01s
 ==================================================================================

### Proposed solution patches:

### Proposed patch number 1:

--- a/mailer.py
+++ b/mailer.py
@@ -22,4 +22,4 @@ class Mailer:

 def send_notification(msg, email_id):
     mailer = Mailer()
-    return mailer.send_mail(email_id.split("@")[0], email_id.split("@")[1], title="Notification",
body=msg)
+    return mailer.send_mail(email_id.split("@")[0], email_id.split("@")[-1], title="Notification",
body=msg)

After incorporating this change, the test output is:

### New Test Status 1:
================================================================================================ test
session starts
================================================================================================
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/testcase
plugins: anyio-4.2.0
collected 1 item

test_mailer.py .
```

954

```
                        [100%]

=============================================================================================== 1
passed in 0.00s
===============================================================================================

### Proposed patch number 2:

--- a/mailer.py
+++ b/mailer.py
@@ -22,4 +22,6 @@ class Mailer:

 def send_notification(msg, email_id):
     mailer = Mailer()
-    return mailer.send_mail(email_id.split(”@”)[0], email_id.split(”@”)[1], title=”Notification”,
body=msg)
+    if ”@” in email_id:
+        domain = email_id.split(”@”)[-1]
+        return mailer.send_mail(email_id[:-len(domain)], domain, title=”Notification”, body=msg)

After incorporating this change, the test output is:

### New Test Status 2:
---
=============================================================================================== test
session starts
===============================================================================================
platform linux -- Python 3.11.7, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/testcase
plugins: anyio-4.2.0
collected 1 item

test_mailer.py F

                        [100%]

===============================================================================================
FAILURES
===============================================================================================
_______________________________________________________________________________________________
test_send_notification
_______________________________________________________________________________________________

    def test_send_notification():
>       with pytest.raises(Exception):
E       Failed: DID NOT RAISE <class ’Exception’>

test_mailer.py:5: Failed
=============================================================================================== short
test summary info
===============================================================================================
FAILED test_mailer.py::test_send_notification - Failed: DID NOT RAISE <class ’Exception’>
=============================================================================================== 1
failed in 0.05s
===============================================================================================
---

### Response:

### Test description:
The test confirms that an exception is being raised when the Mailer is used for send_notification.
This behaviour should NOT happen when the issue is fixed.

### Test Status:
Passing Initially

### Patch description:
[
```

```
    {
        ”patch_number”: 1,
        ”test_effect”: ”The test passes as before because an exception is still being raised.”,
        ”test_status”: ”PASS_TO_PASS”,
        ”patch_effect”: ”The patch modifies the computation of the domain by saying that the last
element after splitting on ’@’ should be the domain. This is correct but the username isn’t computed
correctly.”
    },
    {
        ”patch_number”: 2,
        ”test_effect”: ”The test fails indicating correct behaviour of the code now.”,
        ”test_status”: ”PASS_TO_FAIL”,
        ”patch_effect”: ”The patch fixes the issue now by splitting on the last ’@’ symbol but also
computes the username correctly.”
    }
]

### Ranking description:
Patch 1 tries to solve the problem but still hits an exception and the test cases passes which is not
the desired behaviour. Patch 2 works perfectly and an exception is not raised which is why the test
fails.
Since patch 2 is also PASS_TO_FAIL, it is more probable that it is a useful change therefore it should
 be ranked higher.

### Ranking:
[2] > [1]


Now use the same principles to solve this issue:
{% endif %}

### Issue:
{{ issue }}
</issue>

### Possible buggy code:

{% for bc in buggy_code %}
---
File: {{ bc[’file’] }}

{{ bc[’body’] }}

---
{% endfor %}

{% if testcase %}
### Test case:

A junior has proposed the following test case. It might be useful for you in making your judgement.
{{ testcase }}

On running the test case on the EARLIER state of the repository, the output obtained was(note that
empty output generally means that the tests passed):
### Initial Test Status:

{{ initial_test_output }}

{% endif %}

### Proposed solution patches:

{% for patch in patches %}

### Proposed patch number {{ loop.index }}:

{{ patch[’patch’] }}
{% if patch[’test_output’] %}
```

```
After incorporating this change, the test output is:

### New Test Status {{loop.index}}:

{{ patch[’test_output’] }}
{% endif %}

---
{% endfor %}

Your job is to rank these these solutions from most likely to least likely to fix the issue.
We want to carefully reason about whether the patch would truly fix the issue and in what way.
{%if testcase%}Use the test case outputs to determine which patch might be useful in resolving the
issue.
Note that the test case might be wrong as well.{% endif %}
Do not worry about import errors, they can be fixed easily.
Reason carefully about what solving the issue requires.

Your job is to summarize and rank each patch based on it’s correctness and effect on test cases if
provided.
You should first describe the patches in the following manner:
{% if testcase %}First, describe the status of the test BEFORE any patch was run: {% endif %}
[
    {
        ”patch_number”: 1,
{% if testcase %}        ”test_effect”: <Change in the test case status if any>,
        ”test_status”: <FAIL_TO_PASS/PASS_TO_FAIL/FAIL_TO_FAIL/PASS_TO_PASS>,{% endif %}
        ”patch_effect”: <Describe what the patch does and its effects. Does it solve the issue? Why or
 why not?>
    },
    ...
    {
        ”patch_number”: N,
{% if testcase %}        ”test_effect”: <Change in the test case status if any>,
        ”test_status”: <FAIL_TO_PASS/PASS_TO_FAIL/FAIL_TO_FAIL/PASS_TO_PASS>,{% endif %}
        ”patch_effect”: <Describe what the patch does and its effects. Does it solve the issue? Why or
 why not?>
    },
]

Then, give a ranking of the patches as follows:
For instance if there are 5 patches and you believe the order should be: patch #2 > patch #3 > patch
#1 > patch #5 > patch #4, then output: [2] > [3] > [1] > [5] > [4].

A complete example would be(assume there are 5 patches):

{%if testcase %}
### Initial Test description:
What does the test case check?(for this read the logs in ”### Test case”)

### Initial Test Status:
Passing Initially/Failing Initially(for this read the logs in ”### Initial Test Status”){% endif %}
### Patch description:
[
    {
        ”patch_number”: 1,
{% if testcase %}        ”test_effect”: <Change in the test case status if any>,
        ”test_status”: <FAIL_TO_PASS/PASS_TO_FAIL/FAIL_TO_FAIL/PASS_TO_PASS>(read ”### New Test Status
 1” for this),{% endif %}
        ”patch_effect”: <Describe what the patch does and its effects. Does it solve the issue? Why or
 why not?>
    },
    ...
    {
        ”patch_number”: N,
{% if testcase %}        ”test_effect”: <Change in the test case status if any>,
        ”test_status”: <FAIL_TO_PASS/PASS_TO_FAIL/FAIL_TO_FAIL/PASS_TO_PASS>(read ”### New Test Status
 N” for this),{% endif %}
```

957

```
        "patch_effect": <Describe what the patch does and its effects. Does it solve the issue? Why or
 why not?>
    }
]
### Ranking description:
<description>
### Ranking:
[2] > [3] > [1] > [5] > [4]

Now try on the issue given above. Do not give any justifications while giving the ### Ranking. Also do
 not use = between any patch indices. Break ties using code quality.
Also, note that passing tests is not a requirement. Use the tests like a heuristic instead.
{% if testcase %}Changes in the test status is a good indication that the patch is useful.
PASS_TO_FAIL or FAIL_TO_PASS indicates that the test is useful and that the patch should be ranked
higher. FAIL_TO_FAIL or PASS_TO_PASS patches should be ranked lower.{% endif %}
Carefully look at what was happening before any patch was applied versus what happens after the patch
is applied.

### Response:
```

B Links for logs

Logs for various methods can be found here:

Aider: https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240523_aider

CodeR: https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240604_CodeR/

Open-Devin: https://huggingface.co/spaces/OpenDevin/evaluation/tree/main/outputs/swe_bench_lite/CodeActAgent/gpt-4o-2024-05-13_maxiter_30_N_v1.5-no-hint

SWE-agent: https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240402_sweagent_gpt4/logs