REPOPILOT: Software Agents To Resolve Software Engineering Tasks at Repository-Level Scale

Anonymous ACL submission

Abstract

Coding assistants based on Large Language Models (LLMs) have recently surged in popularity. A significant challenge for LLMs is accurately responding to user queries at the scale of entire code repositories. We propose REPOPILOT , a multi-agent-based system capable of effectively navigating through source code repositories to collect relevant information, editing code and execute programs. We demonstrate the effectiveness of REPOPILOT through extensive evaluations on challenging benchmarks, including SWE-bench and an automatically collected code generation dataset. On SWE-bench Lite, REPOPILOT achieves a 17% pass rate, establishing competitive results compared to the baseline while maintains low cost and also excels in other code intelligence tasks.

1 Introduction

The remarkable advances in Large Language Models (LLMs) have driven rapid progress in various natural language processing tasks (Touvron et al., 2023; OpenAI et al., 2023) and software development tasks (Zhang et al., 2023; Austin et al., 2021; Zhang et al., 2023). Recently, multiple attempts have been made to directly exploit LLMs as agent models to address planning problems in software engineering (SE) (Bairi et al., 2023; Zhang et al., 2024a; Yang et al.; Zhang et al., 2024c). Most Software Engineering tasks are performed within the context of an integrated development environment (IDE), which requires a quick response time for interactive usage. However, most existing approaches rely on high-performance LLMs as the main agent, leading to high query costs. Additionally, these approaches often aim to solve a single task specifically with a single LLM agent, and their agent flow and tool designs are highly customized and not general enough for a variety of software development tasks. For example, AutoCodeRover's pipeline (Zhang et al., 2024c) is difficult to extend to code generation and question answering tasks due to its two-stage pipeline: localizing and patching. While SWE-agent (Yang et al.) offers more flexibility in agent design, their tools are designed to quickly localize objects and reduce API cost and context length, potentially losing necessary fine-grained information during search and editing, which can be meaningful for other software development tasks (bug tracing, question answering, etc.).

We develop REPOPILOT , an agent-based framework that is general for a wide range of software development tasks while reducing the cost of using LLMs. In REPOPILOT , a software task is decomposed into four main processes: *planning*, *navigating*, *generating*, and *executing*. Each process requires a different level of intricacy and, therefore, deserves a different level of intelligence from LLMs. One can use a lightweight and computationally cheap LLM for navigation because this process is the least complex but consumes the highest number of tokens. This approach not only reduces cost but also reduces context for other processes, thereby improving performance. Despite the promise of a multi-agent system, designing such a system for software intelligent tasks is challenging for the following reasons:

- Tool design (interface and functionality) for each agent role should be well-designed and specialized depending on the role's complexity.
- Communication between agents can cause information loss, leading to imprecise decisions in other processes, which can cause compound errors in subsequent iterations.
- Using multiple LLMs working together can substantially increase inference costs if not designed properly.

REPOPILOT follows a new centralized multi-agent system for software tasks, where the *Planner* has

the role of receiving user inputs (GitHub Issue, code generation request), planning, and iteratively deciding which child agent role (*Navigator*, *Editor*, and *Executor*) should be used in the next iteration.

- Cost-Effective: We propose a centralized multi-agent system (Section 3.1) using a high-cost LLM for the *Planner*, which subordinates child agents that are low-cost and inexpensive to execute requested tasks.

- Communication: We propose a communication mechanism (Section 3.2) to offload tasks to child agents while ensuring that the observations received by the *Planner* are succinct and relevant. This is the key to REPOPILOT in reducing the need for making API calls to the *Planner*'s LLM while ensuring the *Planner* remains grounded with information from the codebase environment.

- Tool Design: We propose a suite of new tools (Section 3.3) for efficient navigation, editing, and execution, specialized for each agent role. These tools are carefully customized for each type of agent and the difficulty of each task.

Finally, our contributions are as follows:

1. REPOPILOT : An adaptive and general multi-agent framework for different software tasks.

2. An empirical evaluation demonstrating a comparable pass rate of **17**

2 Related Work

2.1 Software Engineering Benchmarks

Code generation benchmarks, which evaluate models on the task of synthesizing code from natural language descriptions, have served as a long standing bellwether for measuring LM performance (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Lu et al., 2021). Subsequent works have built upon the code generation task formulation to contribute new benchmarks that translate problems to different (programming) languages (Cassano et al., 2022; Wang et al., 2022), incorporate third party libraries (Lai et al., 2023; Liu et al., 2023c), introduce derivative code completion tasks (Muennighoff et al., 2023), increase test coverage (Liu et al., 2023a), change the edit scope (Ding et al., 2024; Yu et al., 2024; Du et al., 2023), and add robustness to dataset contamination (Naman Jain et al., 2024). Code generation problems are largely self-contained, with short problem descriptions (∼100 lines) and corresponding solutions that are similarly brief, requiring nothing more complex than basic language primitives. Tests are either handwritten or generated synthetically via fuzz testing. In recent months, rapid development of LMs has begun to saturate many of these benchmarks. The top method solves 94.4% of HumanEval (Zhou et al., 2023).

Gauging future trends with the code generation task paradigm may be limited by the simplicity of this setting and cost of human-in-the-loop problem creation. In response, recent efforts have demonstrated that software engineering (SE) can serve as a diverse, challenging testbed for LM evaluation (Zhang et al., 2023; Jimenez et al., 2023; Liu et al., 2023b). Repository-level code editing introduces many reasoning challenges grounded in real SE subtasks such as spotting errant code, identifying cross-file relationships, and understanding codebase-specific symbols and conventions. As a field, SE has generally studied tasks in a more isolated manner; prior benchmarks tend to frame problems in isolation from the rest of a codebase (Just et al., 2014; Karampatsis and Sutton, 2020).

We use SWE-bench because it unites many separate SE tasks such as automated program repair(Xia and Zhang, 2022), bug localization (Chakraborty et al., 2018; Yang et al., 2024a) under a single task formulation that faithfully mirrors practical SE. Furthermore, SWE-bench task instances are diverse, having been collected from real GitHub issues across 12 different repositories. In addition, SWE-bench performance is based on rigorous, automatic execution-based evaluation.

2.2 Language Models as Agents

The co-emergence of stronger LMs, increasingly challenging benchmarks, and practical use cases have all together motivated a paradigm shift in LMs' inference setting. In place of traditional zero/few-shot generation, language agents (Sumers et al., 2023; Xi et al., 2023; Wang et al., 2024a), which use LMs to interact with a real/virtual world, have proliferated as the default setting for web navigation (Nakano et al., 2021; Thoppilan et al., 2022; Yao et al., 2022a,b), computer control (Wu et al., 2024; Xie et al., 2024), and code generation tasks (Wang et al., 2023).

Interaction and code generation are increasingly used together, with code as the modality of choice for actions (Yang et al., 2024b; Wang et al., 2024b),

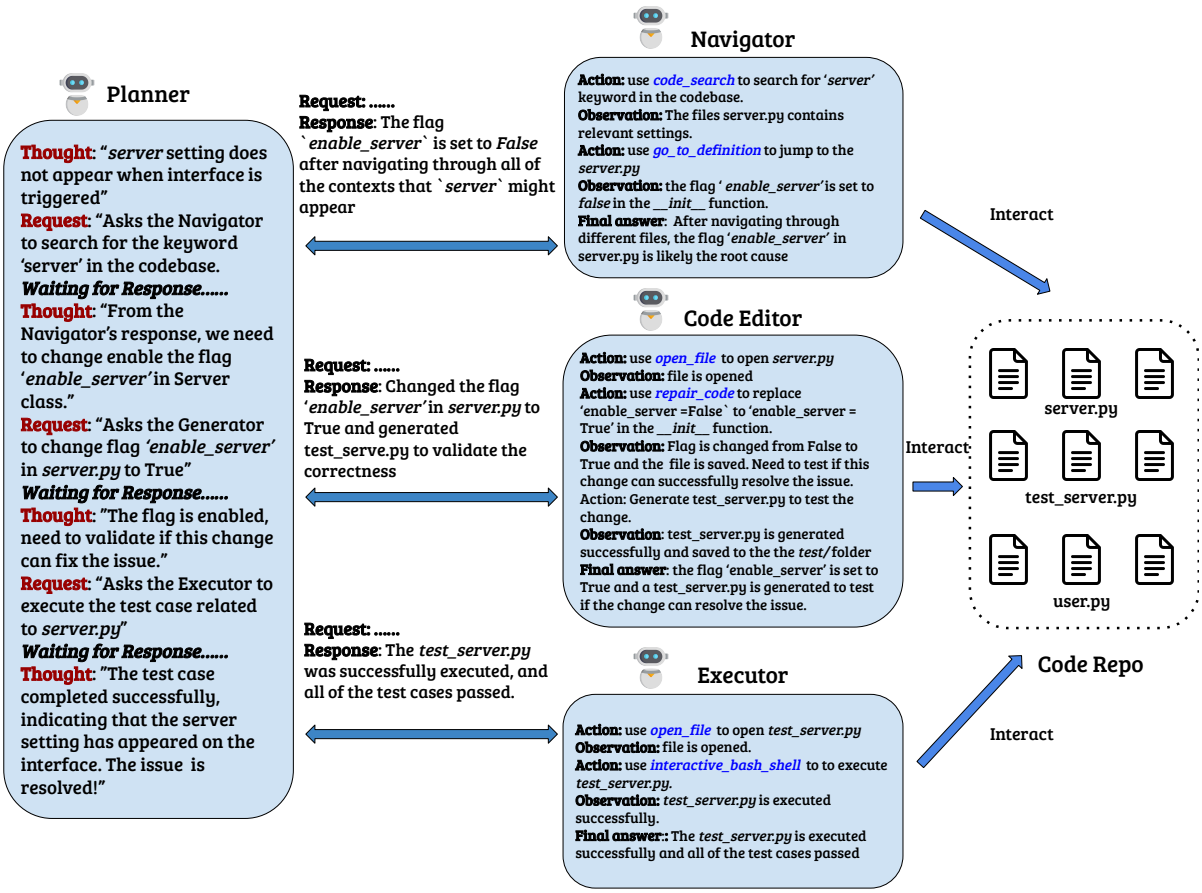

Figure 1: RepoPilot Multi-Agent Framework

tool construction (Wang et al., 2024c; Zhang et al., 2024b; Gu et al., 2024), and reasoning (Zelikman et al., 2023a,b; Shinn et al., 2024). Code language agents have also been applied to offensive security (Yang et al., 2023; Shao et al., 2024), theorem proving (Thakur et al., 2023).

To the best of our knowledge, REPOPILOT is the first to explore low-cost multi-agents system for generic, end-to-end software engineering tasks.

3 Methodology

Figure 1 illustrates our framework. The key design of REPOPILOT is centered at *the idea that only Planner requires advanced reasoning and it should dedicate easy but need long execution tasks into its child agents which can be small and fast*, therefore, reduce the inference cost and improve overall performance by eliminating redundant information out of context of the *Planner*. At each iteration, the *Planner* generates a thought consisting a plan to resolve a query, an agent request and receive a feedback from that agent from generated request. After receiving request from *Planner*, child agents which follow the design of ReAct agent (Yao et al., 2022b), will iteratively execute tools, receiving feedback from codebase environment and report its findings to the *Planner*. *Planner* will terminate when the task is fully resolved.

3.1 Centralized Multi-Agent System

The central agent *Planner* controls three supporting agents *Navigator*, `Editor` and *Executor*:

The role of the *Planner* is to directly receive human task prompts, such as resolving GitHub issues or generating code based on a given description, and address these tasks iteratively. In each iteration, it generates a plan to resolve the query, deciding which supporting agents to utilize and how to employ them in that step (see Section 3.2 for details on how the *Planner* delegates task execution to the child agents and receives feedback from them). The *Planner* also determines when to terminate the resolving process or stops after a maximum number of iterations to minimize the costs.

Navigator is a crucial agent that has responsibility to receive information-seeking query from *Planner* then iteratively navigates and collects all relevant information that can be used to answer

the query. Navigator is equipped with tools similar to ones in Integrated development environment (IDE) commonly used by programmers which are `go_to_definition` or `code_search`. Given a query from *Planner*, navigator uses a series of these provided tools to quickly traverse in the codebase and collect information. The search process will be terminated by the agent or timeout is reached. However, the information-seeking process is costly, requiring a certain number of exploratory iterations for an agent to understand the repository and address the query. This is particularly challenging when the codebase is private and not publicly available, as the LLM lacks knowledge of its structure and key design elements.

Therefore, the navigation agent should be fast and lightweight. It is crucial to have multiple simple, easy-to-use tools that, when combined, yield excellent results for navigation and search. This contrasts with the design of SWE-agent (Yang et al.), which relies solely on summarized code search. While lightweight, this approach often fails to provide concise context understanding. Consequently, LLMs such as LLaMA (Touvron et al., 2023) or Claude 3 Haiku (Chiang et al., 2024) can struggle when faced with exhaustive search results from these tools.

The *Editor* is an agent responsible for editing code in multiple files or creating new files and generating source code for them. *Editor* is equipped with several tools, including `auto_repair_editor`, `code_search`, and `open_file`. It receives the target file from the *Planner* along with the editing context, which includes related information such as the expected behavior after the change and the specific lines to be considered. *Editor* then produces a replacement block of code, which the `auto_repair_editor` inserts into the target file. If the `auto_repair_editor` confirms the edit's success, the *Editor* loop can be terminated, and control returns to the *Planner*.

The *Executor* is an agent responsible for reproducing issues or verifying that the generated patch behaves correctly and resolves the issue after changes are made. It is equipped with an `interactive_bash_shell`, which maintains its state even after executing commands, and `open_file` for reading installation instructions or documentation. The environment setup is also handled automatically by the *Executor*.

3.2 Agent Execution and Communication

This section focuses on how the *Planner* delegates repetitive, interactive but simple tasks to child agents and how these child agents communicate their findings back to the *Planner* with minimal information loss.

Firstly, *Planner* needs to provide detailed request with contextual information for child agents to perform its role and execution. The format of communication from *Planner* sending to other agents always have Context and Request fields. The Context field is more descriptive on original behavior and why *Editor* has to generate a patch while the Request field is a direct command to re-ensure code generated is behaved according to the intention of the *Planner*. Similarly, the Context field for *Navigator* is utilized for faster and direct navigation by optionally directing the target files or specific locations in advance.

```
Planner -> Generator

Request: Generate a patch for the __new__ method to
strip the 'L' suffix from the string before converting
it to an integer, ensuring compatibility between
Python 2 and Python 3.
Context: The __new__ method in `sympy/core/numbers.py`
handles long integers by converting hexadecimal
strings to long integers using the long() function.
This causes an error when unpickling in Python 3
because Python 3 does not recognize the 'L' suffix
used in Python 2 for long integers.
```

Figure 2: Request from *Planner* to *Editor*

After receiving a request and context from the *Planner*, child agents iteratively execute their tasks to resolve the request and eventually report their final answer back to the *Planner*. However, we observed significant information loss in these final reports, especially from the navigation agent. The final answers often lack code snippets or detailed information about the explored objects or codebase structure, leading to deterioration and hallucination in the *Planner* after several iterations. To address this, we propose using a lightweight LLM summarizer to compile intermediate results from the child agent's execution, incorporating this summary into the final answer communicated to the *Planner*.

3.3 Tool Design

A key distinction of REPOPILOTcompared to existing approaches lies in tool design. Several factors are important when designing a tool for each agent role. First, the feedback format from the tools to the LLM must be succinct, informative, and readable. Second, the functionality of the tool should

be considered: does it provide the most information, help the LLM achieve a higher patch apply rate, or speed up the testing environment setup? Finally, usability is crucial; the input interface must be easy-to-use and intuitive. A complex input interface could lead to the LLM failing to use these tools correctly, and in some cases, incorrect tool outputs can negatively impact overall performance.

Here, we describe how tools are designed specifically for each agent role.

Navigating. In tasks like GitHub Issue Resolving or Code Generation, efficiently finding relevant function and class definitions is crucial. To address this, we implemented the `code_search` tool using a trigram-based code search engine with symbol ranking, Zoekt (Nienhuys and Source-Graph), as the code search backend. Additionally, we introduced IDE-like features such as `go_to_definition`, `get_all_references`, and `get_all_symbols`. These tools, when used in conjunction with `code_search`, are particularly powerful because `code_search` can produce many false positives, especially in large codebases. However, these results can indicate where the target definition is called or referenced, and `go_to_definition` can quickly localize the definition effortlessly. This approach contrasts sharply with the `summarized_search` in (Yang et al.), which presents exhaustive search results in one feedback. Similarly, `get_all_references` is effective in localizing similar usages of target functions or classes, helping the system understand how these objects are typically used, leading to better and more consistent fixes.

Designing input interfaces for these tools presents significant challenges. These tools require precise specifications, including exact line numbers, column numbers, file paths, and identifier names, to function properly. Our findings indicate that LLMs often struggle to accurately localize the column number and even the line number of a target word. Therefore, we implemented a proximity search algorithm that identifies the closest exact position relative to the LLM's prediction. Consequently, the LLM now only needs to provide the 'word_list', 'file_path', and 'line_number'.

We implemented the `get_tree_structure` function, which visualizes the structure of a codebase as a tree. This function takes `relative_path` and 'level' as inputs, displaying the folder and file structure within the specified `relative_path` up to the given 'level' of depth. Moreover, we equipped Navigator with `open_file` and `find_file` functionalities. The `open_file` function can display up to 50 lines of source code, annotated with line numbers, and is integrated with a keyword search, allowing the LLM to locate sections containing specific keywords without exhaustive scrolling.

```
Action:
```
{
  "action": "open_file",
  "action_input": {
    "relative_file_path": "sympy/core/numbers.py",
    "start_line": 200,
    "end_line": 600,
    "keywords": ["__new__"]
  }
}
```
```

```
Observation: The content of sympy/core/numbers.py
is:
Results for keyword: __new__
524
525     # Used to make max(x._prec, y._prec) return
526     _prec = -1
527
528     def __new__(cls, *obj):
529         if len(obj) == 1:
530             obj = obj[0]
```

Figure 3: Open File Interface

`code_search` is the most intensively used tool for navigation, making its interface design critical. Besides using simple input specifications and a robust trigram-based search engine, we re-rank results to prioritize primary symbols like functions or classes. Additionally, we found that returned results should include a preview window. This window displays documentation for primary symbols or 5 lines surrounding the search position for other results.

Editing. We also equip the *Editor* with tools similar to those of the *Navigator*, such as `open_file` and `go_to_definition`, to observe the target editing file and jump to the definition of identifiers around the editing location, ensuring consistent fixes within the codebase. The primary focus of editing is the `auto_repair_editor` tool, which uses the 'start_line' and 'end_line' positions of the original block and 'patch' fields for the replacement code. This tool can receive a replacement patch and automatically detect and repair any indentation or syntax errors via an LLM.

Executing Executor can use `open_file` or `get_tree_structure` to find documentations related to testing instruction or environment setup. Then it can use `interactive_bash_shell` to run bash command, normally, these commands are

Model	SWE-bench		SWE-bench Lite	
	% Pass Rate	$ Avg. Cost	% Pass Rate	$ Avg. Cost
RAG				
w/ GPT-4 Turbo	1.31	0.13	2.67	0.13
w/ Claude 3 Opus	3.79	0.25	4.33	0.25
SWE-agent				
w/ GPT-4 Turbo	12.47	1.59	18.00	1.67
w/ Claude 3 Opus	10.46	2.59	13.00	2.18
AutoCodeRover*				
w/ GPT-4 Turbo	-	-	16.11	0.44
RepoPilot				
w/ `GPT-4o`	10.12	0.41	17.00	0.38
w/ `Claude Opus`	-	-	13.00	0.56
w/ `WizardLM2`	-	-	16.00	0.28

Table 1: Performance comparison of different methods on SWE-bench and SWE-bench Lite.

'pytest', 'conda installation' or 'python3'. Notes that we uses interactive bash shell to maintain execution states of bash instead of using Docker.

4 Implementation Details

Benchmarks We evaluate REPOPILOT using the SWE-bench dataset (Jimenez et al., 2023), which comprises 2,294 task instances derived from 12 popular Python repositories, such as `flask`, `numpy`, and `matplotlib`. SWE-bench is designed to assess a system's capability to automatically resolve GitHub issues by using Issue-Pull Request (PR) pairs. The evaluation process involves verifying unit tests using the post-PR behavior as the reference solution. For ablations and analysis, due to budget constraint, we can only utilize a small subset called SWE-bench Tiny consisting of 100 instances from SWE-bench Lite (Jimenez et al., 2023). Additionally, we also examine our method on a small, hand-collected and executable benchmark for repository-level code generation called ExecBench, this benchmark is similar with DevEval (Li et al., 2024) but with synthetic test cases. This benchmark aims for evaluating code generation at the repository level that emphasizes executability and correctness. The test cases are automatically generated by few-shot prompted GPT-3.5 Turbo with internal pipeline ensuring 96.25% test converage. The benchmark includes 120 samples.

Models. For *summarizer* in REPOPILOT, we used Mixtral 8x7B (Jiang et al., 2024). We use a wide variety of LMs to examine the flexibility of the framework as well as measure robustness. For *Planner* role, advanced LLMs including GPT-4o (`gpt-4o-2024-05-13`) (OpenAI et al., 2023), Claude 3 Opus (`claude-3-opus-20230229` (Chiang et al., 2024) and WizardLM2 (`wizardlm-2-8x22b`) (Tao et al., 2024) are used. For *Navigator*, we only use Claude 3 Haiku (`claude-3-haiku-20240307` for fast inference time and low cost; GPT-4o, Claude 3 Sonnet (`claude-3-sonnet-20240229`) and WizardLM2 are used for *Editor* role. Finally, for *Executor*, GPT-3.5 Turbo (`gpt-3.5-turbo-0125`) and Claude 3 Haiku are utilized. All models and agent roles utilize greedy sampling during inference with `temperature` parameter set to 0. We have 3 different multi-agent configurations denoted as following `GPT-4o` with (GPT-4o, Claude Haiku, GPT-4o, GPT-3.5 Turbo) as (planner, navigator, editor and executor), similarly, `WizardLM2` with (WizardLM2, Claude Haiku, WizardLM2, Claude Haiku) and `Claude Opus` with (Claude Opus, Claude Haiku, Claude Sonnet, Claude Haiku).

Baselines. On SWE-Bench, we compare REPOPILOT to three baselines: SWE-Agent, a bash interactive agent with Agent-Computer Interfaces (Yang et al.); AutoCodeRover, a two-stage agent pipeline focusing on bug fixing scenarios (Zhang et al., 2024c) and Retrieval Augmented Generation baselines shown in (Jimenez et al., 2023).

Metrics. We use pass@1 as our metric, which measures the percentage of instances where all tests pass successfully after applying the model-generated patch to the repository. Additionally, we report the Avg cost. Cost metric, representing the average API inference cost for all LLM usages. For ExecBench

Tools. `go_to_def`, `get_all_references` and `get_all_symbols` are implemented using Multilspy (Agrawal et al., 2024) with natural language interface for LLM. LLM inside `auto_repair_editor` is the same with *Editor* in

one configuration.

Others. For RAG, `text-embedding-3-large` is used for source code embeddings, we use code parsing and chunking implementation from Langchain [1]. The chunking size is 1000 tokens.

5 Experimental Results

Methods	pass@1	pass@5
GPT-3.5-Turbo + RAG	24.16	35.00
WizardLM2 + RAG	32.50	49.16
RepoPilot w/ `WizardLM2`	38.33	53.33

Table 2: Pass Rates of different methods on ExecBench

As shown as in Table 1, on SWE-bench Lite, in our framework, `GPT-4o` configuration achieves best performance 17% (51/300) compared to `Claude Opus` and `WizardLM2`. Furthermore, it has higher resolve rate compared to AutoCodeRover (17% versus 16.11%) and competitive with SWE-agent while having significantly lower cost ($\sim$ 4.4 times lower). Notes that REPOPILOT has a more generic architecture rather than AutoCodeRover which is a two-stage pipeline focusing on bug resolving. Moreover, we found that an open weight model `WizardLM2-8x22B` shows strong planning capabilities and when combined with other LLMs inside our framework, it shows comparable results with `GPT-4o` but with only 73% API cost of `GPT-4o` configuration. Due to buget constraint, we can only evaluate `GPT-4o` configuration on full SWE-bench testset, and, it shows superior performance over RAG approaches and positive results compared with SWE-agent (10.12% versus 12.47%).

In Table 2, REPOPILOT exhibits strong performance on ExecBench, surpassing RAG baselines in both pass@1 and pass@5 metrics. This demonstrates its versatility in handling various software engineering tasks of REPOPILOT .

6 Analysis

6.1 Ablation Studies on Agent Roles

In this experiment, we use SWE-bench Tiny to evaluate the contribution of each agent role to overall performance by replacing each child agent with the planner itself. This requires the planner to directly call the eliminated agent's toolset to execute necessary tasks. Table 3 shows a significant cost increase for all configurations when any agent role is removed. The resolving rate also decreases, varying by which role is eliminated. Removing the *Navigator* causes the most performance drop, followed by the *Editor* and the *Executor*, respectively.

Notably, in the case of a medium-long context length LLM such as `WizardLM2` acting as the *Planner* and replacing the role of *Editor* or *Navigator*, we observe a more severe drop in the resolving rate. This is because these roles require continuous interaction with the environment, necessitating a long context.

Model		SWE-bench Tiny	
		% Pass Rate	$ Avg. Cost
`GPT-4o`		15.00	0.42
	w/o Navigator.	7.00	2.81
	w/o Editor	11.00	1.92
	w/o Executor	14.00	0.75
`WizardLM2`		13.00	0.31
	w/o Navigator	4.00	1.21
	w/o Editor	7.00	0.51
	w/o Executor	13.00	0.38

Table 3: Ablation study on different agent role's contribution on SWE-bench Tiny

6.2 Analysis of Tool Design

We investigate the extent of improvements brought by our major design choices in the tool's interface and functionality. We conduct an ablation study on the functionalities of `go_to_definition`, `auto_repair_editor`, `open_file`, and `code_search` using SWE-bench Tiny. For each tool, we evaluate the overall performance when the tool is utilized versus when it is not, as shown in Table 4 (denoted as Used and No Usage, respectively).

A crucial finding for `go_to_definition` is that the LLM agent struggles to effectively use this IDE-like feature. It requires exact line and column num-

`go_to_definition`	
Used	$9.00_{\downarrow 6.0}$
w/ search	15.00
No usage	$12.0_{\downarrow 3.0}$

`open_file`	
Used	$9.00_{\downarrow 6.0}$
w/ annotated lines	$11.00_{\downarrow 4.0}$
w/ keyword summary	15.00
No usage	$4.0_{\downarrow 11.0}$

`code_search`	
Used	$8.00_{\downarrow 6.0}$
w/ preview	$11.00_{\downarrow 3.0}$
w/ ranking	14.00
No usage	$3.0_{\downarrow 11.0}$

`auto_repair_editor`	
Used	$8.00_{\downarrow 7.0}$
w/ linting feedback	$11.00_{\downarrow 4.0}$
w/ repairing	15.00
No usage	$1.0_{\downarrow 14.0}$

Table 4: Ablation result on resolving performance on SWE-Bench Tiny with different key tool designs

[1] https://github.com/langchain-ai/langchain

bers and the precise symbol name, which demands precise localization of character positions. Despite supporting annotated line numbers, the agent often fails and retries multiple times. However, incorporating a proximity-based search process, allowing the agent to approximate specifications, would significantly improve performance (from 9% without search to 15% with search).

For `open_file`, it's frequently to observe that small LLMs, such as Claude Haiku, tend to scroll up and down multiple times to find desired snippets by continuously increasing 'start_line' and 'end_line', leading to out-of-context length issues. We solved this problem by adding an additional input field 'keywords', allowing LLM to search keywords inside file and tool to quickly localize the positions of keywords inside the file and display the surrounding lines. This increases the resolving rate by 3%.

Without the utilization of `code_search`, the *Navigator* would face significant challenges in swiftly identifying the necessary objects to fulfill a request, resulting in a significantly lower performance rate of 3% compared to 8% when the tool is employed. Furthermore, merely providing file paths and keyword positions from `code_search` would not be sufficient. Enhancing the output to include partial surrounding context around the keyword enabling `Navigator` to make more informed decisions and select the most relevant results, thereby improving performance from 8% to 11%. In the majority of Software Engineering Workflow (SWE) tasks, programmers primarily focus on key objects such as functions and classes. Consequently, prioritizing search results for these objects over other results is beneficial, and re-ranking these results further enhances overall performance, increasing it from 11% to 14%.

We further observe a substantial enhancement (8% to 11%) for providing python linting feedback [2] to the *Editor* whenever it produces a patch via `auto_repair_tool`. By providing linting feedback for an internal LLM of the tool to autonomously refine the generated patch and attempt to fix any encountered error, the quality of the patch would be improved leading resolving rate increased by 4%

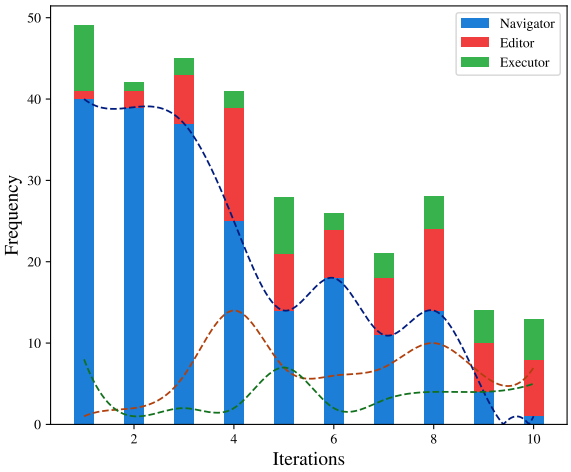

Figure 4: Agent Roles Usage Frequency in SWE-bench by *Planner*'s iterations

6.3 Agent Behavior

We analyzed the frequency of each agent role requested by the *Planner* throughout the issue resolution process. Figure 4 illustrates a typical pattern where the *Planner* is most active at the beginning of the resolution process, gathering relevant information about the codebase environment. Subsequently, the *Editor* is frequently used to generate patches, often immediately following the *Navigator*, with notable peaks at Iterations 4 and 8. Finally, the *Executor* is requested more frequently in the later iterations to verify the results by executing tests. It is noteworthy that, in the first iteration, there are a small peak indicating that *Executor* is requested to reproduce the issue.

7 Conclusion

In this paper, we present REPOPILOT, a versatile multi-agent-based system designed to tackle complex software engineering tasks at the repository level. By carefully crafting the overall framework, communication mechanisms, and interactive tools, we achieve competitive performance compared to single-agent systems while maintaining significantly lower costs. Furthermore, our design is adaptable to other intelligent software engineering tasks and various programming languages.

[2] We use flake8 for providing syntax errors.

Limitations

Despite the significant advancements demonstrated in multi-agent software development, several limitations and areas for future research remain to be addressed for REPOPILOT.

One potential avenue for future research is to apply REPOPILOT to more diversified software engineering tasks, such as repository-level question answering, which requires a deep understanding of the codebase. REPOPILOT can leverage its effective tool suite to navigate and collect relevant information efficiently.

Secondly, future work could focus on training smaller, open-sourced large language models (LLMs) using trajectories collected from more capable yet expensive models like WizardLM2. Effectively training these smaller models presents a significant challenge and an interesting research problem.

Finally, our approach may result in lengthy execution times for child agents due to the lack of contextual information from previous rounds (as the child agent's context is reset after each request from the *Planner*). Implementing a specialized adaptive memory for each agent role could enhance the efficiency and speed of subsequent executions.

References

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu Lahiri, and Sriram Rajamani. 2024. Monitor-guided decoding of code lms with static analysis of repository context. *Advances in Neural Information Processing Systems*, 36.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

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*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Saikat Chakraborty, Yujian Li, Matt Irvine, Ripon Saha, and Baishakhi Ray. 2018. Entropy guided spectrum based bug localization using statistical language model. *arXiv preprint arXiv:1802.06947*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. 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. *ArXiv*, abs/2107.03374.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. 2024. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Yu Gu, Yiheng Shu, Hao Yu, Xiao Liu, Yuxiao Dong, Jie Tang, Jayanth Srinivasa, Hugo Latapie, and Yu Su. 2024. Middleware for llms: Tools are instrumental for language agents in complex environments. *arXiv preprint arXiv:2402.14672*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories.

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

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023b. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, et al. 2023c. Ml-bench: Large language models leverage open-source libraries for machine learning tasks. *arXiv preprint arXiv:2311.09835*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Niklas Muennighoff, Qian Liu, Armel Randy 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. In *The Twelfth International Conference on Learning Representations*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

King Han Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Han-Wen Nienhuys and SourceGraph. Zoekt. https://github.com/sourcegraph/zoekt.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan,

Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report.

Minghao Shao, Boyuan Chen, Sofija Jancheska, Brendan Dolan-Gavitt, Siddharth Garg, Ramesh Karri, and Muhammad Shafique. 2024. An empirical evaluation of llms for solving offensive security challenges. *arXiv preprint arXiv:2402.11814*.

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.

Theodore R Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. 2023. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*.

Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. 2024. A survey on self-evolution of large language models. *arXiv preprint arXiv:2404.14387*.

Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):1–26.

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

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. 2022. Mconala: a benchmark for code generation from multiple natural languages. *arXiv preprint arXiv:2203.08388*.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024c. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*.

Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024a. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2024b. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36.

John Yang, Akshara Prabhakar, Shunyu Yao, Kexin Pei, and Karthik R Narasimhan. 2023. Language agents as hackers: Evaluating cybersecurity skills with capture the flag. In *Multi-Agent Security Workshop@ NeurIPS'23*.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022a. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023a. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. 2023b. Self-taught optimizer (stop): Recursively self-improving code generation. In *OPT 2023: Optimization for Machine Learning*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. 2024b. Training language model agents without modifying language models. *arXiv preprint arXiv:2402.11359*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024c. Autocoderover: Autonomous program improvement.

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

A.1 Prompt Template for Planner

Instruction Prompt Templates for Planner

Suffix Prompt:

```
Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Use tools if
↪  necessary. Respond directly if you have gathered enough information from the repository.
↪  Format is Action:```$JSON_BLOB```then Observation:. Thought:
```

Prefix Prompt:

```
You are a great developer with expertise in resolving programmer's query. You have been
↪  assigned a task to resolve a programmer's issue in a large repository. Devise a detailed
↪  plan using other language model agents to resolve the issue.
You have access into N agents, utilize them to step-by-step solve the query. Each consequent
↪  steps should be strictly based on the previous steps. Your thought process should be
↪  grounded by information collected from your agents, consider its results carefully, and
↪  make a decision based on the results and thought process.
Output the agent you want to use and the request you want to make to the agent. Respond
↪  directly and terminated=true if you have resolved the issue (code generated is verified
↪  and correct).
If you want to modify the logic of the code, or resolve the issue based on retrieved facts from
↪  code navigator, use code editor agent. Terminate if your task is executed successfully and
↪  reviewed correctly.

Top Priorities:
    1. You need to use Codebase Navigator and Code Generator agents to resolve the issue. Use
    ↪  them at least once.
    2. Maintain in mind which files need to edit.
    3. Do not repeat your actions!. After receiving the response from the agent, diversify your
    ↪  next action to get more information.
    4. Identify crucial causes of the issue, localize where the problem is before choosing the
    ↪  code generator agent.
    5. No need to edit test file or test the code. You only need to resolve the issue in the
    ↪  codebase.
    6. Do not care about any Pull Request or Existing Issue in the repository. You are only
    ↪  focused on the issue assigned to you.
    7. Give a detailed request to the agent, so that the agent can understand the context of
    ↪  the query as well.

Important Notes:
    1. Reading the issue description and understanding the problem is the first step. Make sure
    ↪  to identify the key components of the issue and the expected behavior.
    2. Reading the response from the agents carefully, think about the information you have
    ↪  collected and how it can be used to resolve the issue.
    3. Your thought process is the most important part of this task. Make sure to provide a
    ↪  detailed explanation of your reasoning with the issue, code snippets and relevant
    ↪  information collected from the agents.
    4. Stop the task when you have resolved the issue. (Final Answer)

$THOUGHT_PROCESS is your thought process about the query and previous results.
You have access description to the following agents: {formatted_agents}
```

A.2 Prompt Template for Navigator

Instruction Prompt Templates for Navigator

Suffix Prompt:

```
Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Use tools if
↪  necessary. Respond directly if you have gathered enough information from the repository.
↪  Format is Action:```$JSON_BLOB```then Observation:. Thought:
```

Prefix Prompt:

```
You are an expert in finding all relevant information insider a code repository to answer the
↪  query from a planner agent. You have the full access to the codebase of the project you're
↪  working on to resolve a query from a planner. Your tools help you navigate the codebase and
↪  find relevant information. Use them wisely to explore the repository and find the
↪  information you need to resolve the query.
If your first attempts do not provide enough information to resolve the query, try different
↪  tools or use tool with different parameters to get the information you need.
Think carefully before making a decision. Your tools can provide valuable insights to help you
↪  resolve the query. Once you have collected relevant information, you can provide a
↪  response to the query with Final Answer, put any code snippet into that summary. Do not
↪  repeat your actions.

Top Priorities:
    1. Understanding the query, think step-by-step carefully before decision to propose actions
    ↪  to collect necessary information to resolve the query.
    2. Do not repeat your actions.
    3. Try to use the tools to get the information you need. DO NOT GUESS or refuse to response
    ↪  the planner's request. Planner request is always correct. You may only see part of the
    ↪  information, but the planner sees the whole picture.
    4. If one tool does not find the information you need, try another tool. If you open a
    ↪  file, but do not find the information you need, reopen with different start_line and
    ↪  end_line or keywords.
    5. Your summarization should be relevant to the query (provide code snippet if it's
    ↪  required by query), do not provide unnecessary information.

Important Notes:
    1. Try to combine different tools to seek related information to the query inside the
    ↪  project
    2. get_all_references: Use this tool to get all references to a symbol in the codebase.
    ↪  This will help you understand how the symbol is used in the codebase.
    3. get_all_symbols: Use this tool to get all symbols in the target file, it should be used
    ↪  with a keyword.
    4. get_folder_structure: Use this tool to get the structure of the target folder. This will
    ↪  help you understand the organization of the codebase, and find the relevant files to
    ↪  use other tools.
    5. code_search: Use this tool to search for symbol name if you know the exact name of the
    ↪  symbol, this is useful to find the definition if you're not familiar with codebase yet.
    6. go_to_definition: Use this tool to navigate to the definition of an identifier, for
    ↪  example self._print in a class. (single word only, not a combination like sympy.latex),
    ↪  in this case, _print.
    7. open_file: Use this tool to open a file in the codebase, this is useful to read the
    ↪  partial content of the file (50 lines). Should be used with a keyword (single word
    ↪  only, not a combination like `sympy.latex` just `latex` only) or limited start_line and
    ↪  end_line. If your previous open does not show all the information, next turn you can
    ↪  open the same file with different start_line and end_line (incrementally scrolling).

You have access to the following tools: {formatted_tools}
```

A.3 Prompt Template for Editor

Instruction Prompt Templates for Editor

Suffix Prompt:

```
Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Use tools if
↪  necessary. Respond directly if you have gathered enough information from the repository.
↪  Format is Action:```$JSON_BLOB```then Observation:. Thought:
```

Prefix Prompt:

```
You are an expert in programming, you're excellent at editing or generate source code files.

Top priorities:
    1. Always think step-by-step carefully before decision (Thought:).
    2. DO NOT re-generate the same failed edit. Running it again will lead to the same error.
    ↪  Edit the file again if necessary based on the error message.
    3. You should not edit lines locally one part by one part. You might need to remain the
    ↪  original code in your start_line and end_line if it's not changed (regenerate the code
    ↪  if necessary).
    4. You should aware about the code context, and surrounding variables and functions. Do not
    ↪  add some undefined variables.

Important notes:
    1. Always open the file before editing to see latest code!.
    2. If you have to edit the code, ensure the code is correct with syntax, dependency,
    ↪  consistent with the file and the codebase.
    3. Returning final answer if your code is succesfully applied. You first can open a file to
    ↪  get the context and use editor to apply the patch. You have to generate code that is
    ↪  correct with syntax, ensure the dependency, consistent with the file and the codebase.
    4. If you have the exact name of the file and symbol to edit, you can use the code search
    ↪  to find the definition of the symbol. If there's no definition, you can use open
    ↪  open_file tool.
    5. Pay attention to original indentation.
    6. The patch should be a block of code that be replaced into the code.

You have access to the following tool: {formatted_tools}
```

A.4 Prompt Template for Executor

Instruction Prompt Templates for Executor

Suffix Prompt:

```
Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Use tools if
↪  necessary. Respond directly if you have gathered enough information from the repository.
↪  Format is Action:```$JSON_BLOB```then Observation:. Thought:
```

Prefix Prompt:

```
You are an expert in running bash commandlines, based on the requests, try to run commands or
↪  files.
If your request is not specified, considering setup the environment first (using conda create),
↪  cd into the project path and pip3 install -e .[dev]. Then find test folder to run all the
↪  tests. Think carefully before making a decision.

Remember these things:
    1. Your environment code name is `repopilot`, create it if it doesn't exist. Activate it
    ↪  before running the tests.
    2. No need to create new environments.
    3. Check the requirements or setup instruction via opening docs files.
    4. You should setup the developing environment before running the tests, for example
    ↪  running pip3 install -e . or something similar.

Important Tips:
    1. When you observe (y/[n]) in the terminal, you should respond only with y or n.
    ↪  Similarly, when you see (yes/[no]), you should respond with yes or no.
    2. You're already in the project directory.
    3. Run command one by one.

You have access into followng tools: {formatted_tools}
```

B RepoPilot trajectories on SWE-bench

Figure 5: Trajectory of RepoPilot for instance `django_django-16041`.

Issue **django_django-16041**

You need to identify the cause of the following github issue, collect the relevant information, and provide a solution.
Github Issue:

```
When explicitly setting form_kwargs = {'empty_permitted':True} or form_kwargs =
↪  {'empty_permitted':False} , a KeyError occurs when rendering a template that uses a
↪  formset's empty_form.
Expected Behavior
empty_permitted is ignored for formset.empty_form since empty_permitted is irrelevant for
↪  empty_form, as empty_form is not meant to be used to pass data and therefore does not need
↪  to be validated.
Steps to Reproduce
from django.shortcuts import render
from .models import MyModel
def test_view(request):
        context = {}
        ff = modelformset_factory(MyModel, fields = ['a_field'])
        context['formset'] = ff(
                queryset = MyModel.objects.none(),
                form_kwargs = {'empty_permitted':True} # or form_kwargs =
                ↪  {'empty_permitted':False}
        )
        return render(request, 'my_app/my_model_formset.html', context)
# urls.py
from django.urls import path, include
from .views import test_view
urlpatterns = [
```

```
        path('test', test_view)
]
# my_model_formset.html
{% extends "my_app/base.html" %}
{% block content %}
<form id="my-form" method="post">
 {% csrf_token %}
 {{ formset }}
 <input type="submit" value="Save">
</form>
{{ formset.empty_form }}
{% endblock %}
```

INSTRUCTIONS: (31 lines)

```
(Open file: n/a)
(Current directory: /astropy__astropy)
bash-\$
```

REPOPILOT **Planner**

```
1. **Understanding the Issue**:
   - The issue is a KeyError when rendering a template that uses a

   formset's empty_form with form_kwargs containing empty\_permitted.
   - The expected behavior is that empty_permitted should be ignored

   for empty_form since it is irrelevant for empty_form.
2. **Identifying Key Components**:
   - The issue is related to formset and empty_form.
   - The form_kwargs parameter is causing the KeyError.
3. **Steps to Investigate**:
   - Locate the implementation of modelformset_factory and BaseModelFormSet in the Django codebase.
   - Investigate how form_kwargs is processed and how empty_form is generated.
4. **Plan**:
   - Use the codebase_navigator to locate the relevant files and functions.
   - Use the code_generator to create a patch if necessary.
```

Action:

```
{
  "action": "codebase_navigator",
  "action_input": {
    "context": "Locate modelformset_factory and BaseModelFormSet",
    "request": "Find the implementation of modelformset_factory and BaseModelFormSet in the
    ↪  Django codebase."
  }
}
```

REPOPILOT **Navigator**

```
Entering new execution chain..
Thought: To find the implementation of `modelformset_factory` and

`BaseModelFormSet` in the Django codebase, I will first search for the definitions of these symbols.
```

Action:

```
{
```

```
  "action": "code_search",
  "action_input": {
    "names": ["modelformset_factory", "BaseModelFormSet"]
  }
}
```

Observation **Code Search**

```
Results for modelformset_factory:
==================
File: django/forms/models.py
Name: modelformset_factory
Documentation: Return a FormSet class for the given Django model class.

Results for BaseModelFormSet:
==================
File: django/forms/models.py
Name: BaseModelFormSet
Documentation:
    A ``FormSet`` for editing a queryset and/or adding new objects to it.

--------------------
File: django/forms/models.py
Implementation:
32     "ModelMultipleChoiceField",
33     "ALL_FIELDS",
34     "BaseModelFormSet",
35     "modelformset_factory",
--------------------
File: django/forms/models.py
Implementation:
644
645
646 class BaseModelFormSet(BaseFormSet):
647     """
--------------------
File: django/forms/models.py
Implementation:
1000     form=ModelForm,
1001     formfield_callback=None,
1002     formset=BaseModelFormSet,
1003     extra=1,
--------------------
File: django/forms/models.py
Implementation:
1067
1068
1069 class BaseInlineFormSet(BaseModelFormSet):
1070     """A formset for child objects related to a parent."""
```

REPOPILOT **Navigator**

```
Thought: The search results indicate that the implementation of `modelformset_factory`
and `BaseModelFormSet` is located in the `django/forms/models.py`

file. I will now open that file to explore the details of these implementations.

---

Action:
```
{
  "action": "open_file",
```
```

```
  "action_input": {
    "relative_file_path": "django/forms/models.py",
    "start_line": 644,
    "end_line": 684
  }
}
```

Observation **Open File**

The content of django/forms/models.py is:

```
644
645
646 class BaseModelFormSet(BaseFormSet):
647     """
648     A ``FormSet`` for editing a queryset and/or adding new objects to it.
649     """
650
651     model = None
652     edit_only = False
653
654     # Set of fields that must be unique among forms of this set.
655     unique_fields = set()
656
657     def __init__(
658         self,
659         data=None,
660         files=None,
661         auto_id="id_%s",
662         prefix=None,
663         queryset=None,
664         *,
665         initial=None,
666         **kwargs,
667     ):
668         self.queryset = queryset
669         self.initial_extra = initial
670         super().__init__(
671             **{
672                 "data": data,
673                 "files": files,
674                 "auto_id": auto_id,
675                 "prefix": prefix,
676                 **kwargs,
677             }
678         )
679
680     def initial_form_count(self):
681         """Return the number of forms that are required in this FormSet."""
682         if not self.is_bound:
683             return len(self.get_queryset())
```

REPOPILOT **Navigator**

Okay, let's summarize the key information about the implementation of

`modelformset_factory` and `BaseModelFormSet` in the Django codebase:

Final Answer:

The implementation of `modelformset_factory` and `BaseModelFormSet`

is located in the `django/forms/models.py` file.

The `modelformset_factory` function is defined in this file and is

used to return a `FormSet` class for the given Django model class. It

takes several parameters such as the model class, the form class, and

various options to customize the formset behavior.

The `BaseModelFormSet` class is also defined in this file. It is a

subclass of `BaseFormSet` and is used for editing a queryset and/or

adding new objects to it. It has several properties and methods that

are specific to model formsets, such as `queryset`, `initial_extra`,

and `initial_form_count()`.

The key parts of the `BaseModelFormSet` implementation are:

```python
class BaseModelFormSet(BaseFormSet):
    """
    A ``FormSet`` for editing a queryset and/or adding new objects to it.
    """

    model = None
    edit_only = False

    # Set of fields that must be unique among forms of this set.
    unique_fields = set()

    def __init__(
        self,
        data=None,
        files=None,
        auto_id="id_%s",
        prefix=None,
        queryset=None,
        *,
        initial=None,
        **kwargs,
    ):
        self.queryset = queryset
        self.initial_extra = initial
        super().__init__(
            **{
                "data": data,
                "files": files,
                "auto_id": auto_id,
                "prefix": prefix,
                **kwargs,
            }
        )

    def initial_form_count(self):
        """Return the number of forms that are required in this FormSet."""
        if not self.is_bound:
            return len(self.get_queryset())
```

This provides the core functionality for a model formset, including

managing the queryset, initial data, and number of forms in the
formset.

REPOPILOT **Planner**

Based on the information gathered, the issue seems to be related to how `form_kwargs` are processed when creating the `empty_form` in a formset. Specifically, the `empty_permitted` key in `form_kwargs` is causing a `KeyError` when rendering the `empty_form`.

To resolve this issue, we need to ensure that `empty_permitted` is ignored for `empty_form` since it is irrelevant for `empty_form`. This can be done by modifying the relevant part of the code where `empty_form` is created to exclude `empty_permitted` from `form_kwargs`.

Next, I will use the `codebase_navigator` to locate the part of the code where `empty_form` is created and `form_kwargs` are processed in the `BaseModelFormSet` class.

Action:
```
{
  "action": "codebase_navigator",
  "action_input": {
    "request": "Find the part of the code in BaseModelFormSet where empty_form is created and form_kwargs are processed.",
    "context": "BaseModelFormSet is in `django/forms/models.py`
  }
}
```

REPOPILOT **Navigator**

Thought: To find the part of the code in BaseModelFormSet where empty_form is created and form_kwargs are processed, I will use the following steps:

1. Use get_folder_structure to understand the overall structure of the of `django/forms` and identify the relevant file. Planner suggests we should look into `django/forms/models.py`
2. Use open_file to browse the contents of the relevant file and locate the code that creates the empty_form and processes form_kwargs.
3. Use go_to_definition to navigate to the definition of the relevant methods or attributes.
4. Use find_all_references to find all references to the relevant methods or attributes.

Action:
```
{
  "action": "get_folder_structure",
  "action_input": {
    "relative_path": "django/forms"
  }
}
```

Observation **Get Folder Structure**

The tree structure of django/forms is:
forms
 __init__.py
 jinja2

```
templates
boundfield.py
fields.py
forms.py
formsets.py
models.py
renderers.py
utils.py
widgets.py
```

REPOPILOT **Navigator**

Though: Okay, let's continue the process to find the relevant code in BaseModelFormSet and empty$_form$.

Action:
```
{
  "action": "open_file",
  "action_input": {
    "relative_file_path": "django/forms/formsets.py",
    "start_line": 1,
    "end_line": 20,
    "keywords": ["BaseModelFormSet", "empty_form"]
  }
}
```

Observation **Open File**

The content of django/forms/formsets.py is:
Results for keyword: empty_form
```
252         """Return a list of all the initial forms in this formset."""
253         return self.forms[: self.initial_form_count()]
254
255     @property
256     def extra_forms(self):
257         """Return a list of all the extra forms in this formset."""
258         return self.forms[self.initial_form_count() :]
259
260     @property
261     @property
262     def empty_form(self):
263         form_kwargs = self.get_form_kwargs(None)
264         form_kwargs.pop('empty_permitted', None)
265         form = self.form(
266             auto_id=self.auto_id,
267             prefix=self.add_prefix("__prefix__"),
268             empty_permitted=True,
269             use_required_attribute=False,
270             **form_kwargs,
271             renderer=self.renderer,
399
400     def full_clean(self):
401         """
402         Clean all of self.data and populate self._errors and
403         self._non_form_errors.
404         """
405         self._errors = []
406         self._non_form_errors = self.error_class(
407             error_class="nonform", renderer=self.renderer
408         )
409         empty_forms_count = 0
410
```

```
        if not self.is_bound:  # Stop further processing.
            return

        if not self.management_form.is_valid():
            error = ValidationError(
                self.error_messages["missing_management_form"],
                params={
                    "field_names": ", ".join(
                        for field_name in self.management_form.errors
                    ),
                },
                code="missing_management_form",
            )
            self._non_form_errors.append(error)

        for i, form in enumerate(self.forms):
            # Empty forms are unchanged forms beyond those with initial data.
            if not form.has_changed() and i >= self.initial_form_count():
                empty_forms_count += 1
            # Accessing errors calls full_clean() if necessary.
            # _should_delete_form() requires cleaned_data.
            form_errors = form.errors
            if self.can_delete and self._should_delete_form(form):
                continue
            self._errors.append(form_errors)
        try:
            if (
                self.validate_max
                TOTAL_FORM_COUNT
            ] > self.absolute_max:
                raise ValidationError(
                    self.error_messages["too_many_forms"] % {"num": self.max_num},
                    code="too_many_forms",
                )
            if (
                self.validate_min
                and self.total_form_count()
                - len(self.deleted_forms)
                - empty_forms_count
                < self.min_num
            ):
                raise ValidationError(
                    self.error_messages["too_few_forms"] % {"num": self.min_num},
                    code="too_few_forms",
                )
            # Give self.clean() a chance to do cross-form validation.
            self.clean()
        except ValidationError as e:
        return "%s-%s" % (self.prefix, index)

    def is_multipart(self):
        """
        Return True if the formset needs to be multipart, i.e. it
        has FileInput, or False otherwise.
        """
        if self.forms:
            return self.forms[0].is_multipart()
        else:
            return self.empty_form.is_multipart()

    @property
    def media(self):
        # All the forms on a FormSet are the same, so you only need to
        # interrogate the first form for media.
        if self.forms:
            return self.forms[0].media
        else:
            return self.empty_form.media
```

REPOPILOT **Navigator**

The part of the code in BaseModelFormSet where `empty_form` is created and

`form_kwargs` are processed is located in the `django/forms/formsets.py` file.

Specifically, the `empty_form` property is defined on lines 261-271, where

the form is instantiated with various keyword arguments, including

`form_kwargs`. The `form_kwargs` are processed by calling the

`get_form_kwargs()` method on line 263, which prepares the keyword arguments

for the form.

Here is the relevant code snippet:

```python
@property
def empty_form(self):
    form_kwargs = self.get_form_kwargs(None)
    form_kwargs.pop('empty_permitted', None)
    form = self.form(
        auto_id=self.auto_id,
        prefix=self.add_prefix("__prefix__"),
        empty_permitted=True,
        use_required_attribute=False,
        **form_kwargs,
        renderer=self.renderer,
    )
    return form
```

This method creates the `empty_form` that is used by the `BaseModelFormSet`

to provide an initial, empty form for the formset.

REPOPILOT **Planner**

Based on the information gathered, the issue arises because the `empty_form` property in the
↪ `BaseModelFormSet` class processes `form_kwargs` and includes the `empty_permitted` key,
↪ which is irrelevant for `empty_form`. This causes a `KeyError` when rendering the template.

To resolve this issue, we need to ensure that `empty_permitted` is ignored for `empty_form` by
↪ modifying the `empty_form` property to exclude `empty_permitted` from `form_kwargs`.

Next, I will use the `code_generator` to create a patch that modifies the `empty_form`
↪ property in the `BaseModelFormSet` class to exclude `empty_permitted` from `form_kwargs`.

Action:
```
{
  "action": "code_editor",
  "action_input": {
    "title": "Patch to fix KeyError in empty_form",
    "file_path": "django/forms/formsets.py",
```

```
    "context": "The issue is a KeyError when rendering a template that uses a formset's
    ↪  empty_form with form_kwargs containing empty_permitted. The expected behavior is that
    ↪  empty_permitted should be ignored for empty_form since it is irrelevant for
    ↪  empty_form. Ensure that form_kwargs.pop('empty_permitted', None) is called before
    ↪  passing form_kwargs to the form constructor.",
    "request": "Modify the empty_form property in the BaseModelFormSet class to exclude
    ↪  empty_permitted from form_kwargs."
  }
}
```

REPOPILOT **Editor**

Entering new execution chain..
Thought: I will look into "django/forms/formsets.py" to see relevant code snippets before editing.

Action:
```
Action:
```
```
{
  "action": "open_file",
  "action_input": {
    "relative_file_path": "django/forms/formsets.py",
    "keyword": "class",
    "preview_size": 20
  }
}
```

Observation **Open File**

The content of django/forms/formsets.py is:
```
1 from django.core.exceptions import ValidationError
2 from django.forms import Form
3 from django.forms.fields import BooleanField, IntegerField
4 from django.forms.renderers import get_default_renderer
5 from django.forms.utils import ErrorList, RenderableFormMixin
6 from django.forms.widgets import CheckboxInput, HiddenInput, NumberInput
7 from django.utils.functional import cached_property
8 from django.utils.translation import gettext_lazy as _
9 from django.utils.translation import ngettext_lazy
10
11 __all__ = ("BaseFormSet", "formset_factory", "all_valid")
12
13 # special field names
14 TOTAL_FORM_COUNT = "TOTAL_FORMS"
15 INITIAL_FORM_COUNT = "INITIAL_FORMS"
16 MIN_NUM_FORM_COUNT = "MIN_NUM_FORMS"
17 MAX_NUM_FORM_COUNT = "MAX_NUM_FORMS"
18 ORDERING_FIELD_NAME = "ORDER"
19 DELETION_FIELD_NAME = "DELETE"
20
21 # default minimum number of forms in a formset
22 DEFAULT_MIN_NUM = 0
23
24 # default maximum number of forms in a formset, to prevent memory exhaustion
25 DEFAULT_MAX_NUM = 1000
26
```

```

class ManagementForm(Form):
    """
    Keep track of how many form instances are displayed on the page. If adding
    new forms via JavaScript, you should increment the count field of this form
    as well.
    """

    template_name = "django/forms/div.html"  # RemovedInDjango50Warning.

    TOTAL_FORMS = IntegerField(widget=HiddenInput)
    INITIAL_FORMS = IntegerField(widget=HiddenInput)
    # MIN_NUM_FORM_COUNT and MAX_NUM_FORM_COUNT are output with the rest of the
    # management form, but only for the convenience of client-side code. The
    # POST value of them returned from the client is not checked.
    MIN_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)
    MAX_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)

    def clean(self):
        cleaned_data = super().clean()
        # When the management form is invalid, we don't know how many forms
        # were submitted.
        cleaned_data.setdefault(TOTAL_FORM_COUNT, 0)
        cleaned_data.setdefault(INITIAL_FORM_COUNT, 0)
        return cleaned_data


class BaseFormSet(RenderableFormMixin):
    """
    A collection of instances of the same Form class.
    """

    deletion_widget = CheckboxInput
    ordering_widget = NumberInput
    default_error_messages = {
        "missing_management_form": _(
            "ManagementForm data is missing or has been tampered with. Missing fields: "
            "%(field_names)s. You may need to file a bug report if the issue persists."
        ),
        "too_many_forms": ngettext_lazy(
            "Please submit at most %(num)d form.",
            "Please submit at most %(num)d forms.",
            "num",
        ),
        "too_few_forms": ngettext_lazy(
            "Please submit at least %(num)d form.",
            "Please submit at least %(num)d forms.",
            "num",
        ),
    }

    template_name_div = "django/forms/formsets/div.html"
    template_name_p = "django/forms/formsets/p.html"
    template_name_table = "django/forms/formsets/table.html"8 from django.utils.translation
↪  import gettext_lazy as _
9 from django.utils.translation import ngettext_lazy
10
11 __all__ = ("BaseFormSet", "formset_factory", "all_valid")
12
13 # special field names
14 TOTAL_FORM_COUNT = "TOTAL_FORMS"
15 INITIAL_FORM_COUNT = "INITIAL_FORMS"
16 MIN_NUM_FORM_COUNT = "MIN_NUM_FORMS"
17 MAX_NUM_FORM_COUNT = "MAX_NUM_FORMS"
18 ORDERING_FIELD_NAME = "ORDER"
19 DELETION_FIELD_NAME = "DELETE"
20
21 # default minimum number of forms in a formset
```

```
DEFAULT_MIN_NUM = 0

# default maximum number of forms in a formset, to prevent memory exhaustion
DEFAULT_MAX_NUM = 1000


class ManagementForm(Form):
    """
    Keep track of how many form instances are displayed on the page. If adding
    new forms via JavaScript, you should increment the count field of this form
    as well.
    """

    template_name = "django/forms/div.html"  # RemovedInDjango50Warning.

    TOTAL_FORMS = IntegerField(widget=HiddenInput)
    INITIAL_FORMS = IntegerField(widget=HiddenInput)
    # MIN_NUM_FORM_COUNT and MAX_NUM_FORM_COUNT are output with the rest of the
    # management form, but only for the convenience of client-side code. The
    # POST value of them returned from the client is not checked.
    MIN_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)
    MAX_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)

    def clean(self):
        cleaned_data = super().clean()
        # When the management form is invalid, we don't know how many forms
```


```
    template_name = "django/forms/div.html"  # RemovedInDjango50Warning.

    TOTAL_FORMS = IntegerField(widget=HiddenInput)
    INITIAL_FORMS = IntegerField(widget=HiddenInput)
    # MIN_NUM_FORM_COUNT and MAX_NUM_FORM_COUNT are output with the rest of the
    # management form, but only for the convenience of client-side code. The
    # POST value of them returned from the client is not checked.
    MIN_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)
    MAX_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)

    def clean(self):
        cleaned_data = super().clean()
        # When the management form is invalid, we don't know how many forms
        # were submitted.
        cleaned_data.setdefault(TOTAL_FORM_COUNT, 0)
        cleaned_data.setdefault(INITIAL_FORM_COUNT, 0)
        return cleaned_data


class BaseFormSet(RenderableFormMixin):
    """
    A collection of instances of the same Form class.
    """

    deletion_widget = CheckboxInput
    ordering_widget = NumberInput
    default_error_messages = {
        "missing_management_form": _(
            "ManagementForm data is missing or has been tampered with. Missing fields: "
            "%(field_names)s. You may need to file a bug report if the issue persists."
        ),
        "too_many_forms": ngettext_lazy(
            "Please submit at most %(num)d form.",
            "Please submit at most %(num)d forms.",
            "num",
        ),
        "too_few_forms": ngettext_lazy(
            "Please submit at least %(num)d form.",
            "Please submit at least %(num)d forms.",

    TOTAL_FORMS = IntegerField(widget=HiddenInput)
```

```
    INITIAL_FORMS = IntegerField(widget=HiddenInput)
    # MIN_NUM_FORM_COUNT and MAX_NUM_FORM_COUNT are output with the rest of the
    # management form, but only for the convenience of client-side code. The
    # POST value of them returned from the client is not checked.
    MIN_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)
    MAX_NUM_FORMS = IntegerField(required=False, widget=HiddenInput)

    def clean(self):
        cleaned_data = super().clean()
        # When the management form is invalid, we don't know how many forms
        # were submitted.
        cleaned_data.setdefault(TOTAL_FORM_COUNT, 0)
        cleaned_data.setdefault(INITIAL_FORM_COUNT, 0)
        return cleaned_data


class BaseFormSet(RenderableFormMixin):
    """
    A collection of instances of the same Form class.
    """

    deletion_widget = CheckboxInput
    ordering_widget = NumberInput
    default_error_messages = {
        "missing_management_form": _(
            "ManagementForm data is missing or has been tampered with. Missing fields: "
            "%(field_names)s. You may need to file a bug report if the issue persists."
        ),
        "too_many_forms": ngettext_lazy(
            "Please submit at most %(num)d form.",
            "Please submit at most %(num)d forms.",
            "num",
        ),
        "too_few_forms": ngettext_lazy(
            "Please submit at least %(num)d form.",
            "Please submit at least %(num)d forms.",
            "num",
        ),
        ),
        "too_few_forms": ngettext_lazy(
            "Please submit at least %(num)d form.",
            "Please submit at least %(num)d forms.",
            "num",
        ),
    }

    template_name_div = "django/forms/formsets/div.html"
    template_name_p = "django/forms/formsets/p.html"
    template_name_table = "django/forms/formsets/table.html"
    template_name_ul = "django/forms/formsets/ul.html"

    def __init__(
        self,
        data=None,
        files=None,
        auto_id="id_%s",
        prefix=None,
        initial=None,
        error_class=ErrorList,
        form_kwargs=None,
        error_messages=None,
    ):
        self.is_bound = data is not None or files is not None
        self.prefix = prefix or self.get_default_prefix()
        self.auto_id = auto_id
        self.data = data or {}
        self.files = files or {}
        self.initial = initial
```

```
        self.form_kwargs = form_kwargs or {}
        self.error_class = error_class
        self._errors = None
        self._non_form_errors = None

        messages = {}
        for cls in reversed(type(self).__mro__):
            messages.update(getattr(cls, "default_error_messages", {}))
        if error_messages is not None:
            messages.update(error_messages)
    template_name_ul = "django/forms/formsets/ul.html"

    def __init__(
        self,
        data=None,
        files=None,
        auto_id="id_%s",
        prefix=None,
        initial=None,
        error_class=ErrorList,
        form_kwargs=None,
        error_messages=None,
    ):
        self.is_bound = data is not None or files is not None
        self.prefix = prefix or self.get_default_prefix()
        self.auto_id = auto_id
        self.data = data or {}
        self.files = files or {}
        self.initial = initial
        self.form_kwargs = form_kwargs or {}
        self.error_class = error_class
        self._errors = None
        self._non_form_errors = None

        messages = {}
        for cls in reversed(type(self).__mro__):
            messages.update(getattr(cls, "default_error_messages", {}))
        if error_messages is not None:
            messages.update(error_messages)
        self.error_messages = messages

    def __iter__(self):
        """Yield the forms in the order they should be rendered."""
        return iter(self.forms)

    def __getitem__(self, index):
        """Return the form at the given index, based on the rendering order."""
        return self.forms[index]

    def __len__(self):
```

REPOPILOT **Editor**

Entering new execution chain..
Thought: The `empty_form` property is defined in the `BaseFormSet` class within the `django/forms/formsets.py` fi

Here is the relevant code snippet from the `BaseFormSet` class:

```python
@property
def empty_form(self):
    form_kwargs = self.get_form_kwargs(None)
    form_kwargs.pop('empty_permitted', None)
    form = self.form(
        auto_id=self.auto_id,
        prefix=self.add_prefix("__prefix__"),
```

```
        empty_permitted=True,
        use_required_attribute=False,
        **form_kwargs,
        renderer=self.renderer,
    )
    self.add_fields(form, None)
    return form
```

I will now apply the required modification to the `empty_form` property in the `BaseFormSet` class.

Action:
```
{
  "action": "autorepair_editor_file",
  "action_input": {
    "relative_file_path": "django/forms/formsets.py",
    "start_line": 261,
    "end_line": 274,
    "patch": "    @property\n    def empty_form(self):\n        form_kwargs =
    ↪  self.get_form_kwargs(None)\n        form_kwargs.pop('empty_permitted', None)\n
    ↪  form = self.form(\n            auto_id=self.auto_id,\n
    ↪  prefix=self.add_prefix(\"__prefix__\"),\n            empty_permitted=True,\n
    ↪  use_required_attribute=False,\n            **form_kwargs,\n
    ↪  renderer=self.renderer,\n        )\n        self.add_fields(form, None)\n        return
    ↪  form"
  }
}
```

Observation **Auto-repair Editor**

Successfully edited the file django/forms/formsets.py from line 261 to 274

Patch

```
diff --git a/django/forms/formsets.py b/django/forms/formsets.py
index 57676428ff..221aa405a8 100644
--- a/django/forms/formsets.py
+++ b/django/forms/formsets.py
@@ -214,8 +214,10 @@ class BaseFormSet(RenderableFormMixin):
     index will be None if the form being constructed is a new empty
     form.
     """
-    return self.form_kwargs.copy()
-
+    form_kwargs = self.form_kwargs.copy()
+    if index is None:
+        form_kwargs.pop('empty_permitted', None)
+    return form_kwargs

 def constructform(self, i, kwargs):
     """Instantiate and return the i-th form instance in a formset."""
@@ -255,14 +257,17 @@ class BaseFormSet(RenderableFormMixin):
     """Return a list of all the extra forms in this formset."""
     return self.forms[self.initial_form_count() :]

+    @property
 @property
 def empty_form(self):
+    form_kwargs = self.get_form_kwargs(None)
+    form_kwargs.pop('empty_permitted', None)
```

```
      form = self.form(
          auto_id=self.auto_id,
          prefix=self.add_prefix("prefix"),
          empty_permitted=True,
          use_required_attribute=False,
-         self.get_form_kwargs(None),
+         **form_kwargs,
          renderer=self.renderer,
      )
      self.add_fields(form, None)
```

REPOPILOT **Planner**

The modification to the `empty_form` property in the `BaseFormSet` class has

been successfully applied. The `empty_permitted` key is now excluded from

`form_kwargs` before passing it to the form constructor, addressing the

issue of the KeyError when rendering a template that uses a formset's

`empty_form`.

Action:

```
{
  "action": "Final Answer",
  "action_input": {
    "context": "The task is fully resolved.",
  }
}
```