MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue ReSolution

Wei Tao
Fudan University
wtao18@fudan.edu.cn

Yucheng Zhou
University of Macau
yucheng.zhou@connect.um.edu.mo

Wenqiang Zhang
Fudan University
wqzhang@fudan.edu.cn

Yu Cheng
Rice University
yc180@rice.edu

Abstract

In software evolution, resolving the emergent issues within GitHub repositories is a complex challenge that involves not only the incorporation of new code but also the maintenance of existing functionalities. Large Language Models (LLMs) have shown promise in code generation and understanding but face difficulties in code change, particularly at the repository level. To overcome these challenges, we empirically study the reason why LLMs mostly fail to resolve GitHub issues and analyze some impact factors. Motivated by the empirical findings, we propose a novel LLM-based **M**ulti-**A**gent framework for **G**itHub **I**ssue re**S**olution, **MAGIS**, consisting of four kinds of agents customized for the software evolution: Manager, Repository Custodian, Developer, and Quality Assurance Engineer agents. This framework leverages the collaboration of various agents in the planning and coding process to unlock the potential of LLMs to resolve GitHub issues. In experiments, we employ the SWE-bench benchmark to compare MAGIS with popular LLMs, including GPT-3.5, GPT-4, and Claude-2. MAGIS can resolve **13.94%** GitHub issues, which significantly outperforms the baselines. Specifically, MAGIS achieves an eight-fold increase in resolved ratio over the direct application of GPT-4, the based LLM of our method. We also analyze the factors for improving GitHub issue resolution rates, such as line location, task allocation, etc.

1 Introduction

In real-world software development, the software application is rarely set in stone. High-quality and popular software continually evolves to address emergent bugs or adapt to new requirements arising from shifts in user needs, software environments, and physical hardware. On platforms such as GitHub [1], issues typically signify the requirement for software evolution. However, addressing these issues poses significant challenges, as it requires implementing the code change across the entire repository and maintaining the existing functionality while integrating new capabilities. Consequently, resolving GitHub issues remains a significant challenge across academia and industry [15, 4].

Large Language Models (LLMs) have demonstrated remarkable capabilities across a variety of tasks [6], including code generation and code understanding [40, 30]. Specifically, LLMs excel in generating function-level code, as evidenced by their performance on numerous benchmark datasets such as MBPP [2] and HumanEval [9]. Despite their success, LLMs remain challenged in tasks that require advanced code generation capabilities, such as the ClassEval benchmark [10]. Moreover,

[1] https://github.com

LLMs exhibit limitations in processing excessively long context inputs and are subject to constraints regarding their input context length [20]. This limitation is particularly evident in repository-level tasks, such as solving GitHub issues, where the context comprises the entire repository, thus imposing constraints on directly using the full repository as input to LLMs.

To harness the full potential of LLMs, researchers [12, 26, 36] have designed LLM-based multi-agent systems. These approaches have significantly improved LLMs' efficacy in code generation, enabling these systems to construct code repositories based on LLM. While these methods address the process of transitioning code repositories from inception to establishment, they rarely consider the handling of software evolution, e.g., resolving Github issues. For Github repositories, especially popular ones, a large number of commits are pushed every day. These commits derive from a spectrum of evolutionary requirements that span bug fixes, feature additions, performance enhancements, etc [32]. For open-source software, new requirements frequently emerge as issues in the project's repository.

To investigate the capability of LLMs in addressing GitHub issues, Jimenez et al. [15] developed a benchmark, namely SWE-bench, to conduct a comprehensive analysis of popular LLMs. The study reveals that LLMs fail to resolve over 5% of instances, even when provided with file paths that require modifications. This significantly low rate of applying solutions and resolving issues underscores the limitations inherent in LLMs. The reasons behind the suboptimal performance of LLMs and strategies to harness their potential for GitHub issue resolution remain under-explored.

In this study, we analyze the factors impacting the effectiveness of LLMs in resolving issues under two different settings (i.e., w/ Oracle and w/o Oracle). Furthermore, our empirical analysis has concluded a correlation between file location and line location within files, and performance in resolving GitHub issues. Based on these insights, we propose a novel LLM-based Multi-agent Framework, termed MAGIS, comprising four types of agents: Manager, Repository Custodian, Developer, and Quality Assurance (QA) Engineer. Our approach facilitates the resolution of GitHub issues through collaboration among agents, each fulfilling a unique role.

In our experiment, we evaluate our framework on the SWE-bench, comparing its performance against existing popular LLMs, such as ChatGPT-3.5 [24], GPT-4 [25], and Claude-2 [1]. The results demonstrate that our framework, utilizing GPT-4 as its base model, significantly outperforms the baseline, and achieves an eight-fold performance gain compared to the direct application of GPT-4. Further analysis revealed additional factors, i.e., the planning of code change, line location within the code file, and code review process, that significantly influence the resolution rate.

Our main contributions are summarized as follows:

- We conduct an empirical analysis of LLMs in resolving GitHub issues in two settings (i.e., w/ Oracle setting and w/o Oracle setting), and explore the correlation between line location, complexity of the code change, file location, and the success rate in resolving GitHub issues.
- We propose a novel LLM-based multi-agent framework, MAGIS, to alleviate the limitations of existing LLMs on GitHub issue resolution. Both our designed four-type agents and their collaboration for planning and coding unlock LLMs' potential on the repository-level coding task.
- We compare our framework and other strong LLM competitors (i.e., GPT-3.5, GPT-4, and Claude-2) on the SWE-bench dataset. The results show MAGIS significantly outperforms these competitors. Further analysis is conducted and verifies the effectiveness and necessity of our framework design.

2 Empirical Study

2.1 RQ 1: What Impacts the Performance Under With-Oracle Setting?

The with-Oracle setting is a more straightforward setup for the GitHub issue resolution by supplying the specific files requiring modifications. Despite the observation that LLMs exhibit better performance in the with-Oracle setting compared to their performance without Oracle, the proportion of issues successfully resolved under the with-Oracle setting remains modest. Specifically, in the with-Oracle setting, GPT-4 achieved a resolved rate of only 1.74% on the SWE-bench.

To delve into the factors impacting the efficacy within the with-Oracle setting, we examined failed instances' generation. This scrutiny led to the identification of two factors that could influence the resolved rate: (1) Line location (location of the modified line in the code file), and (2) The complexity of the code change. This information is important and provided in the code change of each commit in the code repository. A typical code change includes multiple changed hunks and each hunk contains the line numbers targeted for modification and the specifics of the changed code at these locations. Both the line locations and the changed content make a repository evolve.

To validate this identification, we analyze the relation between the resolved rate and the two factors mentioned above.

2.1.1 Line Location

To quantitatively analyze the accuracy of line localization, we use the line numbers' range of the modified content in the reference code change as the basis assuming that the correct modification location of the code change is uniquely determined in most cases. By calculating the overlap ratio of the line number ranges of the generated and reference, we can estimate the accuracy of line localization in the generation process. Specifically, for each instance, the line number ranges of the code change in the reference r is represented as a set of intervals $\boldsymbol{L}_r = \{[s_0, e_0], ..., [s_n, e_n]\}$, while the line number range set of the generated code change g is $\boldsymbol{L}_g = \{[s'_0, e'_0], ..., [s'_m, e'_m]\}$, where s and e respectively represent the starting and ending line number of each modification hunk in the file, with n hunks in the reference code change and m in the generated one. The formula for calculating the overlap ratio is as follows:

$$\text{Overlap Ratio} = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m}\left|[s_i, e_i] \cap [s'_j, e'_j]\right|}{\sum_{i=0}^{n}(e_i - s_i + 1)}, \tag{1}$$

where $\left|[s_i, e_i] \cap [s'_j, e'_j]\right|$ denotes the size of the intersection between the intervals $[s_i, e_i]$ and $[s'_j, e'_j]$.

For 574 instances in the SWE-bench test set that experiments GPT-4, the distribution of the overlap ratio between the results generated by three LLMs and the reference code change is shown in Figure 1 with the vertical axis representing the frequency of the range of line location overlap ratio for each group, and the horizontal axis representing the overlap ratio. From this, we observe that: (1) The distribution near the overlap ratio 0 (left side of the figure) is the highest for all three LLMs, indicating that in most cases, the content generated by these models has a very low overlap ratio with the reference in terms of line location. This means that these LLMs are most likely not able to accurately locate the lines of code that need to be modified in the process of generating the code change. (2) In the distribution near the line location overlap of 1 (right side of the figure), the three models show a consistent ranking (i.e., Claude-2 $>$ GPT-4 $>$ GPT-3.5) and this ranking is also consistent with the proportion of instances solved by the three models. This phenomenon suggests that the performance of LLMs in generating the code change is probably related to their ability to locate code lines accurately.

Furthermore, we assess the relationship between the overlap ratio and the resolution of GitHub issues by calculating their correlation coefficient. Given that the distribution of these variables exhibits skewness, and the resolution result is binary (resolved or not), logistic regression is employed for the analysis across three LLMs. However, due to the limited number of successfully generated instances on GPT-4 and GPT-3.5, a statistically significant relationship (P-value < 0.05) is detected solely in the generated result from Claude-2. This signifies that the correlation coefficient from Claude-2 is both statistically significant and meaningful. Specifically, with a positive coefficient, 0.5997, on Claude-2, there is a substantial and positive relation between improvements in the overlap ratio and the probability of successfully resolving issues. Consequently, the line location emerges as an important factor for GitHub issue resolution.

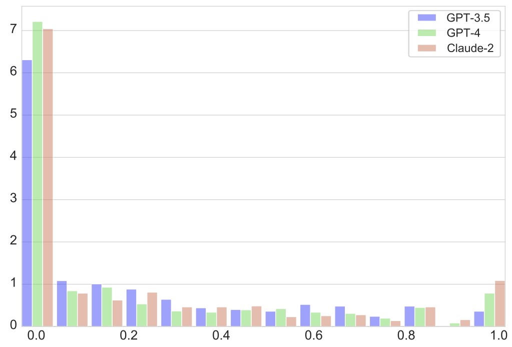


Figure 1: The Comparison of Line Location Overlap Ratio between Three LLMs On SWE-bench.

2.1.2 Complexity of the Code Change

The complexity of the code change is reflected in various indices, including the number of modified files, functions, hunks, and lines added or deleted. Firstly, we quantitatively assess the complexity by calculating the value of various indices corresponding to the reference code change. Secondly, the coefficient is calculated between the numbers in each index and the resolution of issues. Table 1 shows the correlation scores between six indices and the issue resolution, under the logistic regression.

Table 1: Correlation between the Complexity Indices and the Issue Resolution.

LLM	# Files	# Functions	# Hunks	# Added LoC	# Deleted LoC	# Changed LoC
GPT-3.5	-17.57^*	-17.57^*	-0.06^*	-0.02	-0.03	-0.53^*
GPT-4	-25.15^*	-25.15^*	-0.06	-0.10	-0.04	-0.21
Claude-2	-1.47^*	-1.47^*	-0.11^*	-0.09^*	-0.07^*	-0.44^*

* The correlation between the index and the issue resolution is significant (P-value < 0.05).

As shown in Table 1, all three LLMs, i.e., GPT-3.5, GPT-4, and Claude-2, demonstrate a statistically significant correlation with the issue resolution across several indices, i.e., p-value less than 0.05 and marked by $*$. The correlation scores for the number of files and functions modified are notably negative for all models, indicating that an increase in these indices is associated with a decreasing likelihood of issue resolution. This suggests that the more complex the code change, as indicated by a higher number of files and functions modified, may hinder the issue resolution. Compared with GPT-3.5 and GPT-4, Claude-2 exhibits a different pattern, with much lower negative correlations for the number of files and functions, which indicates it is a more efficient approach to generate the code change for GitHub issue resolution. However, it also shows significant negative correlations across other indices such as the number of hunks, added lines of code (LoC), deleted LoC, and changed LoC. The analysis reveals a relationship between the complexity, as measured by several indices, and whether to successfully resolve the issues in software evolution. The negative correlations suggest that increased complexity, particularly in terms of the number of files and functions changed, tends to hinder issue resolution.

2.2 RQ 2: What Impacts the Performance Under Without-Oracle Setting?

The difference between the with-Oracle setting and the without-Oracle setting lies in the necessity for the latter to identify the modified files. Consequently, file locating emerges as a crucial factor influencing performance under the without-Oracle setting.

Jimenez et al. [15] employ the BM25 method [28] to retrieve relevant code files that are subsequently utilized as input to the LLM. After employing retrieval methods, it is necessary to select the top-K files or truncate the content based on the maximum context length of the LLM. Incorporating more files can enhance recall scores. However, it also imposes significant demands on the capabilities of LLMs. As demonstrated by the study [15], Claude-2 exhibits a reduction at the resolved ratio as recall scores increase. This decline may be attributable to the inclusion of irrelevant files or the limited capacity of LLMs to process longer contexts effectively. Consequently, exploiting the performance of LLMs can be better achieved by striving for higher recall scores with a minimized set of files, thus suggesting a strategic balance between recall optimization and the number of chosen files.

3 Methodology

In this section, we elaborate on our LLM-based multi-agent framework, MAGIS, for GitHub issue resolution. The framework resolves each issue by applying the generated code change to the repository. As shown in Figure 2, we first provide the overview of our framework (§3.1), followed by the role design of each type agent and how these roles differ from human workflows (§3.2). Lastly, we detail the collaborative process among these roles within our framework (§3.3).

3.1 Overview

The architecture of our framework is illustrated in Figure 2, encompassing four key roles of agents working collaboratively in the workflow.

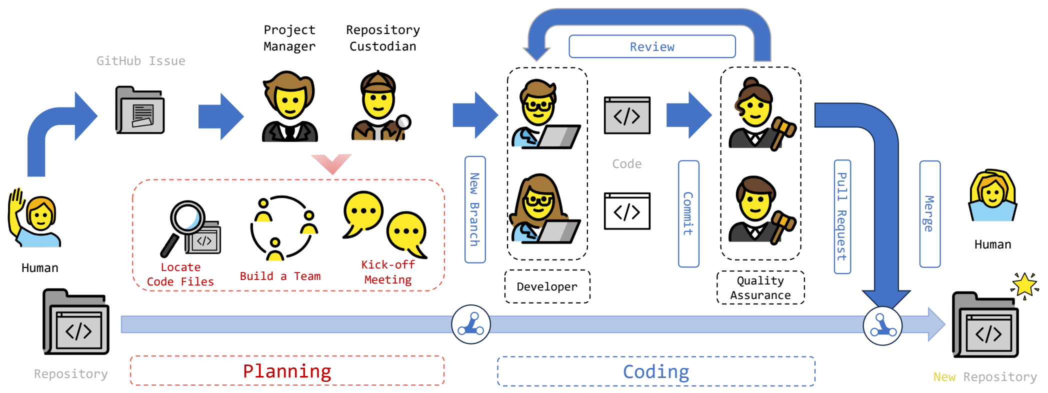


Figure 2: Overview of Our Framework, MAGIS.

- **Manager**. This role tasks with team assembly, meeting organization, and plan formulation.
- **Repository Custodian**. The custodian is responsible for locating the relevant files in the repository according to the GitHub issue and recording the change of the repository.
- **Developer**. This role participates in planning discussions and completes tasks from the manager.
- **Quality Assurance (QA) Engineer**. The QA engineer reviews the code change from developers to ensure the quality of the whole repository.

The collaborative process involves planning and coding. In the planning, a GitHub issue is assigned to the project manager and the repository custodian. The repository custodian identifies candidate files relevant to the issue for modification. With the issue description and a list of candidate files, the manager defines tasks and assembles a team, where each member is a developer specifically designed for the defined task. The manager holds a kick-off team meeting with developers and devises a plan. During the coding, developers undertake their assigned tasks from the manager, and the QA engineer reviews each code change. If a change fails to meet quality standards, the QA engineer provides feedback, prompting further revisions until the change satisfies quality criteria or a set iteration limit is reached.

3.2 Agent Role Design

Our workflow draws inspiration from the GitHub Flow [2], an effective human workflow paradigm, adopted by many software teams. Both the human workflow and our LLM-based agent framework prioritize collaboration among individuals with diverse skills. While the underlying principles are similar, there are notable differences. Accordingly, we have tailored the roles as follows:

- **Manager**. The manager's role is pivotal in planning. In conventional setups, managers decompose the issue into tasks according to the pre-formed team and allocate these tasks for members with different skills. In contrast, our manager agent can first decompose the issue into tasks and then design developer agents to form a team. This setup improves team flexibility and adaptability, enabling the formation of teams that can meet various issues efficiently.

- **Repository Custodian**. Considering extensive files in a repository, the custodian agent's task is to locate files relevant to the issue. Different from humans, who can browse through the entire repository, the LLM-based agent faces challenges in browsing. Although LLMs, including GPT-4 [23], have extended context limits, their application is constrained in two aspects. First, it is a high computational cost to query the whole repository for each update while some repositories update frequently. Second, the performance of LLMs degrades when relevant information occurs in the middle of the long context input [19, 42].

[2] `https://docs.github.com/en/get-started/using-github/github-flow`

- **Developer**. In contrast to human developers, the developer agent can work continuously, and complete a task in minutes even seconds. Therefore, scheduling the agent to work in parallel is easier than scheduling humans as the latter requires considering other than work while the former only needs to ensure the tasks are independent. Additionally, although there are numerous developer agents capable of generating code [12, 26], their ability to modify existing code is not equally proficient. To address this issue, our framework decomposes the code modification process into atomic operations, which encompass the code generation. This approach enables developers to leverage the benefits of automatic code generation thereby generating applicable code changes.
- **QA Engineer**. In software evolution, QA engineers, especially code reviewers, play a crucial role in maintaining software quality [21, 17]. Despite its critical role, code review practices are often undervalued or even overlooked [3]. Such neglect can hinder software development, illustrated by instances where developers may experience delays of up to 96 hours awaiting code review feedback [5]. To address this problem, our framework pairs each developer agent with a QA engineer agent, designed to offer task-specific, timely feedback. This personalized QA approach aims to boost the review process thereby better ensuring the software quality.

3.3 Collaborative Process

3.3.1 Planning

Three types of role agents engage in the planning process: the Repository Custodian, the Manager, and the Developer. The process comprises three phases: locating code files, team building, and the kick-off meeting.

Locating Code Files. Firstly, the repository custodian employs the BM25 algorithm [28] to rank the files in the repository based on the GitHub issue description. Subsequently, the top k files are selected as potential candidates for further coding. However, as described in §2.2, this simple retrieval method can introduce irrelevant files, increasing the cost and reducing the effectiveness of subsequent code changing by LLMs. Therefore, it is necessary to filter them according the relevance. While it is feasible to directly assess the relevance between each file and the issue by LLMs, many queries are redundant as some code snippets in the previous request are duplicated, leading to unnecessary costs. Considering that applying the code change often modifies a specific part of the file rather than the whole file, we propose a method to reuse the previously requested information.

Algorithm 1 Locating

```
1: Input: repository: $\mathcal{R}_i$ include file $f_i$, GitHub issue description: $q_x$, LLM: $\mathcal{L}$
2: Config: filter top width: $k$, prompts: $\mathcal{P}$
3: Output: candidate files: $\mathcal{C}_i^k \leftarrow \emptyset$, repository evolution memory: $\mathcal{M} \leftarrow \emptyset$
4: $\mathcal{R}_i \leftarrow$ BM25($\mathcal{R}_i$, $q_x$)
5: $\mathcal{C}_i^k \leftarrow \mathcal{R}_i$[:$k$]
6: for $f_i \in \mathcal{C}_i^k$ do
7:     $f_h, s_h \leftarrow$ find($f_i, \mathcal{M}$)
8:     if $\exists\, f_h$ and len($s_h$) $<$ len($f_i$) then
9:         if $h$ is $i$ then
10:            $s_i \leftarrow s_h$
11:        else
12:            $\Delta d \leftarrow$ diff($f_h$, $f_i$)
13:            $m \leftarrow \mathcal{L}(\Delta d, \mathcal{P}_1)$
14:            $s_i \leftarrow s_h \cup m$
15:        end if
16:    else
17:        $s_i \leftarrow \mathcal{L}(f_i, \mathcal{P}_2)$
18:    end if
19:    $\mathcal{M} \leftarrow \mathcal{M}$.update($\{f_i : s_i\}$)
20:    if $\mathcal{L}((s_i, q_x), \mathcal{P}_3)$ is false then
21:        $\mathcal{C}_i^k \leftarrow \mathcal{C}_i^k$ - $f_i$
22:    end if
23: end for
```

As Algorithm 1 demonstrates, when a file f_i is compared with the issue q_x for the first time, the LLM $\mathcal{L}$ compresses it into a shorter code summary s_i, where i denotes the file's version number. Upon modification on f_i to a new version, denoted as f_j, the difference, $d_{i,j}$, is obtained via the "`git diff`" command. If the length of $d_{i,j}$ is less than that of f_j, indicating that the version differences are smaller than the new file itself, $\mathcal{L}$ can summarize the code changes as a "commit" message $m_{i,j}$. s_i together with $m_{i,j}$ makes up the description of the newer version f_j, enabling the LLM $\mathcal{L}$ to determine relevance r_j in fewer length. Based on this relevance, the custodian agent filters candidate files, allowing the manager agent to define tasks with these relevant files.

Team Building. In this process, the manager agent has the flexibility to "recruit" team members as the issue needs. Firstly, upon receiving the candidate files, the manager begins with analyzing the GitHub issue for the repository and breaks them into detailed file-level tasks. Specifically, for each code file f_i in the candidate set $\mathcal{C}_i^k$, the project manager leverages the LLM $\mathcal{L}$ with the prompt $\mathcal{P}_4$ and the issue description q_x to define the corresponding file-level task t_i. One issue can be converted to multiple tasks. These tasks, along with the associated code file, are stored in a task set $\mathcal{T}_i^k$. Once a task is clarified, the project manager defines the personality role r_i of the developer by invoking LLM $\mathcal{L}$ with the prompt $\mathcal{P}_5$ and the task t_i. By iterating through these candidate code files, the project manager agent ultimately designs a collection of developer agent role descriptions $\mathcal{D}_i^k$, thus forming the development team. The details of the team building are shown in the part of Algorithm 2.

Kick-off Meeting. After building the team, the project manager organizes a kick-off meeting. This meeting serves multiple purposes: (1) To confirm whether the tasks assigned by the project manager are reasonable and ensure that all developers in the team can collaboratively resolve the issue q_x, and (2) To determine which developers' tasks can be executed concurrently and which tasks have dependencies need to be sorted. The meeting takes the form of a circular speech, and the project manager is responsible for opening the speech, guiding the discussion, and summarizing the results. After the meeting, developers adjust their role descriptions $\mathcal{D}_i^k$ based on the discussion, and the project manager, leveraging the LLM $\mathcal{L}$ and the prompt $\mathcal{P}_7$, generates a main work plan c_{main}. This work plan is presented as code, and embedded into the main program for execution.

Algorithm 2 Making the Plan

```
1: Input: candidate files: $\mathcal{C}_i^k$, issue: $q_x$, LLM: $\mathcal{L}$,
2: Config: prompts: $\mathcal{P}$
3: Output: tasks: $\mathcal{T}_i^k \leftarrow \emptyset$, developer agents' role
   description: $\mathcal{D}_i^k \leftarrow \emptyset$, plan: $c_{main}$
4: for $f_i \in \mathcal{C}_i^k$ do
5:     $t_i \leftarrow \mathcal{L}((f_i, q_x), \mathcal{P}_4)$
6:     $\mathcal{T}_i^k \leftarrow \mathcal{T}_i^k \cup (f_i, t_i)$
7:     $r_i \leftarrow \mathcal{L}((t, q_x), \mathcal{P}_5)$
8:     $\mathcal{D}_i^k \leftarrow \mathcal{D}_i^k \cup r_i$
9: end for
10: recording = kick_off_meeting($\mathcal{D}_i^k$)
11: $\mathcal{D}_i^k \leftarrow \mathcal{L}((\mathcal{D}_i^k$, recording$), \mathcal{P}_6)$
12: $c_{main} \leftarrow \mathcal{L}($recording$, \mathcal{P}_7)$
```

3.3.2 Coding

Two types of agents participate in the coding process: developers and QA engineers. As outlined in Algorithm 3, for each task t_i in the task set $\mathcal{T}_i^k$ and its associated code file f_i, the developer agent generates the role description of the QA engineer a_i by leveraging the LLM $\mathcal{L}$ with the prompt $\mathcal{P}_8$. Subsequently, developers collaborate with their QA engineers to execute the coding tasks. During each execution of the developer, the range of lines of code that need to be modified is firstly determined as a set of intervals $\{[s_i', e_i']\}$ where s_i' represents the starting line number in the i-th hunk, and e_i' is the ending line number. The determination is generated by analyzing the task content t_i and file content f_i using the LLM $\mathcal{L}$ with the prompt $\mathcal{P}_9$. These intervals split the original code file f_i into parts to be modified, *old_code_part*, and parts to be retained. Developers then generate new code snippets, *new_code_part*, by the LLM $\mathcal{L}$ with the prompt $\mathcal{P}_{10}$. The code snippets replace *old_code_part*, resulting in a new version of the code file f_i'. Utilizing Git tools, the code change Δd_i for this file f_i is generated. With the code change Δd_i, QA engineer produce *review_comment* and *review_decision*, by the LLM $\mathcal{L}$ with the prompt $\mathcal{P}_{11}$. If the decision, *review_decision*, is negative (i.e., $false$), the feedback, *review_comment*, prompts developers to revise the code in the next attempt. This iterative process continues until the code change meets the quality standards or reaches a predefined maximum number of iterations. After the iteration, the final version of the code change, Δd, is fixed and it represents

Algorithm 3 Coding Task Execution

```
1: Input: file-task set: $\mathcal{T}_i^k$, LLM: $\mathcal{L}$
2: Config: prompts: $\mathcal{P}$, the max of iteration: $n_{\max}$
3: Output: code changes: $\mathcal{D}$
4: for $f_i, t_i \in \mathcal{T}_i^k$ do
5:     $a_i \leftarrow \mathcal{L}((f_i, t_i), \mathcal{P}_8)$
6:     for $j \in [\ 0, n_{\max}\ )$ do
7:         if $j > 0$ then
8:             $t_i$ = ($t_i$, review_comment)
9:         end if
10:        $\{[s_i', e_i']\} \leftarrow \mathcal{L}((f_i, t_i), \mathcal{P}_9)$
11:        $f_i$, old_code_part $\leftarrow$ split($f_i$, $\{[s_i', e_i']\}$)
12:        new_code_part $\leftarrow \mathcal{L}((f_i, t_i,$ old_code_part$), \mathcal{P}_{10})$
13:        $f_i' \leftarrow$ replace($f_i$, $\{[s_i', e_i']\}$, new_code_part)
14:        $\Delta d_i \leftarrow$ diff($f_i$, $f_i'$)
15:        review_comment = $\mathcal{L}((t_i, \Delta d_i), \mathcal{P}_{11})$
16:        review_decision = $\mathcal{L}(($review_comment$), \mathcal{P}_{11})$
17:        if review_decision is true then
18:            break
19:        end if
20:    end for
21:    $\Delta d \leftarrow$ diff($f_i'$, $f_i$)
22:    $\mathcal{D} \leftarrow \mathcal{D} \cup \Delta d$
23: end for
```

the ultimate modification result on each file. All generated final-version code changes during this process are merged into the repository-level code change $\mathcal{D}$ as the issue solution.

4 Experiments and Analysis

To validate the performance of our framework, we conduct experiments and compare it against other popular Large Language Models (LLMs) to demonstrate its overall effectiveness (§4.2). In addition, to further investigate the effectiveness of our framework, we assessed its performance in two processes: planning (§4.3) and coding (§4.4).

4.1 Setup

4.1.1 Dataset and Experimental Settings

In the experiments, we employ the SWE-bench dataset as the evaluation benchmark because it is the latest dataset specifically designed for evaluating the performance of the GitHub issue resolution. SWE-bench comprises $2,294$ issues extracted from 12 popular Python repositories, representing real software evolution requirements. It includes two context settings: an Oracle retrieval and sparse retrieval. In the former setting (w/ Oracle), the LLM receives the files to modify, whereas, in the latter (w/o Oracle), the method needs to retrieve the files for modification.

Given the observation that experimental outcomes on the 25% subset of SWE-bench align with those obtained from the entire dataset [15], we opt for the same 25% subset previously utilized in experiments for GPT-4 according to their materials [3]. Moreover, the experimental scores for the five LLMs, have been made available by them [4].

Our framework is flexible to integrate various LLMs. To compare with the scores reported by SWE-bench, we select GPT-4 as the base LLM. Another reason for the selection is that GPT-4 shows remarkable performance on code generation and understanding, which has been demonstrated on benchmarks such as MBPP [2] and HumanEval [9]. Claude-2 is not chosen due to the unavailability of API access.

4.1.2 Metric

Following the SWE-bench [15], we use the applied and resolved ratio to evaluate the performance under the with-Oracle setting. Specifically, the applied ratio indicates the proportion of instances where the code change is successfully generated and can be applied to the existing code repository using Git tools, i.e.,

$$\text{Applied Ratio} = \frac{|\mathcal{D}|}{|\mathcal{I}|}, \tag{2}$$

where $\mathcal{D}$ represents the set of instances in the generated code change set that could be applied to the original code repository using the "`git apply`" operation, and $\mathcal{I}$ is the set of all instances in the test set. The resolved ratio refers to the proportion of instances in which the code change is successfully applied and passed a series of tests, i.e.,

$$\text{Resolved Ratio} = \frac{\left|\sum_{i=0}^{l}(\{T_{old}(d_i)\} \cap \{T_{new}(d_i)\})\right|}{|\mathcal{I}|}, \tag{3}$$

where T_{old} denotes all the test cases that the old version of the code repository could pass, T_{new} represents all the test cases designed for new requirements, and d_i denotes the code change generated to resolve the issue in the i-th instance. Furthermore, $T(d) = \text{True}$ means that the code change d can pass all the test cases in T.

The recall score versus file number curve is used to measure the effectiveness of file locating for the without-Oracle setting. The recall score refers to the proportion of files that are successfully located

[3] `https://drive.google.com/drive/folders/1EnrKzGAnsb_NmZKyECGmA2DrAc8ZuJ80`

[4] `https://openreview.net/forum?id=VTF8yNQM66&noteId=lfJF38VxJr`

out of all the files that require modification. The formula for calculating the file location recall score for the i-th instance is as follows:

$$\text{Recall} = \frac{|\mathcal{G}_i \cap \mathcal{R}_i|}{|\mathcal{R}_i|} \times 100\%, \tag{4}$$

where $\mathcal{G}_i = \sum_{j=0}^{n} g_{i,j}$ represents the set of file paths located by our framework, with each file path in the set denoted as $g_{i,j}$ and the total number of files as n; $\mathcal{R}_i = \sum_{k=0}^{m} r_{i,k}$ denotes the paths of the files that need to be modified, with each reference file path denoted as $r_{i,k}$ and the total file number as m. In this curve, "file number" refers to the average number of files that need to be processed across all instances to achieve the given recall score. Specifically, it illustrates how many files averagely need to be located by our framework before reaching the recall score denoted by the curve at any point. This metric represents both the effectiveness and efficiency of file locating.

4.2 RQ 3: How Effective Is Our Framework?

The comparative performance analysis between our framework and other LLMs on the same dataset is presented in Table 2. The results indicate that our framework significantly outperforms other LLMs. Notably, with a resolved ratio of 13.94%, our framework's effectiveness is eight-fold that of the base model, GPT-4. This substantial increase underscores our framework's capability to harness the potential of LLMs more effectively. Furthermore, when contrasted with the previous state-of-the-art LLM, Claude-2, our framework's resolved ratio exceeds that benchmark by more than two-fold. This superior performance unequivocally establishes the advance of our method. As Devin [34] uses a different subset from the evaluated subset of GPT-4 mentioned in the SWE-bench paper [15], the comparison between it and ours is discussed in § 5.2.

The ablation study is designed to simulate two scenarios: (1) Without QA (w/o QA): Considering the QA engineer agent as optional within our framework, we directly evaluate the code changes generated by the developer agent, bypassing the QA process. This scenario aims to investigate the effectiveness and necessity of QA engineer review. (2) Without hints (w/o hints): Hints refer to the textual content found in the comments section of pull requests, which are typically created prior to the first commit of the pull request. The absence of hints (w/o hints) means our framework operates without any clarifications except for the issue, despite such information being available on GitHub before the issue resolution process begins. This analysis aims to explore if the participation of humans could potentially improve the success rate of issue resolution.

Table 2: The Comparison of Overall Performance between MAGIS and Baselines on SWE-bench.

Method	% Applied	% Resolved
GPT-3.5	11.67	0.84
Claude-2	49.36	4.88
GPT-4	13.24	1.74
SWE-Llama 7b	51.56	2.12
SWE-Llama 13b	49.13	4.36
Devin [34][*]	–	13.86[*]
MAGIS	**97.39**	**13.94**
MAGIS (w/o QA)	92.71	10.63
MAGIS (w/o hints)	94.25	10.28
MAGIS (w/o hints, w/o QA)	91.99	8.71

[*] Note that this work is evaluated on a randomly chosen 25% test set, but this subset differs from the 25% subset experimented on GPT-4 in the SWE-bench.

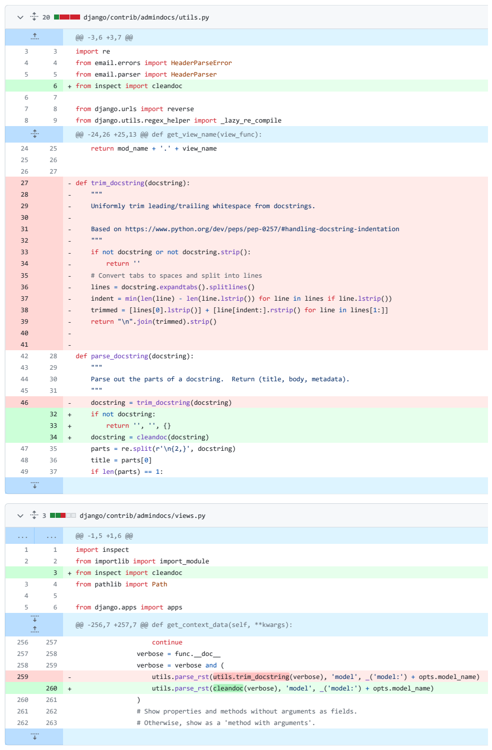
Our framework demonstrates the capability to significantly enhance issue resolution, even without the QA engineer or hints. It achieves a resolved ratio of 8.71, which is five times higher than that of the base model, GPT-4. This increase underscores the contribution of other agents in our framework to its overall performance. Furthermore, integrating cooperation with QA or hints separately can further elevate the resolved ratio by 1.92 or 1.57, respectively. These findings serve to underscore the value of QA engineers and the participation of humans, as demonstrated by the resolved rates achieved through their integration.

For instance, to resolve the issue [5] from the repository `Django` [6], the developer modifies four hunks in two files [7], as shown in Figure 3. Despite the availability of two files in the with-Oracle setting, our

[5] `https://code.djangoproject.com/ticket/30255`

[6] `https://github.com/django/django/`

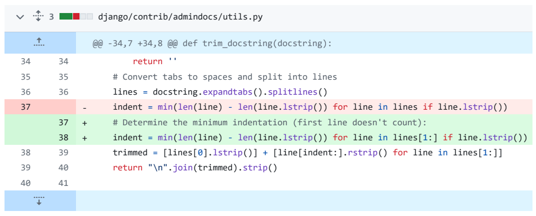
[7] `https://github.com/django/django/pull/12155/files`



```
20 django/contrib/admindocs/utils.py
@@ -3,6 +3,7 @@
  import re
  from email.errors import HeaderParseError
  from email.parser import HeaderParser
+ from inspect import cleandoc

  from django.urls import reverse
  from django.utils.regex_helper import _lazy_re_compile
@@ -24,26 +25,13 @@ def get_view_name(view_func):
      return mod_name + '.' + view_name


- def trim_docstring(docstring):
-     """
-     Uniformly trim leading/trailing whitespace from docstrings.
-
-     Based on https://www.python.org/dev/peps/pep-0257/#handling-docstring-indentation
-     """
-     if not docstring or not docstring.strip():
-         return ''
-     # Convert tabs to spaces and split into lines
-     lines = docstring.expandtabs().splitlines()
-     indent = min(len(line) - len(line.lstrip()) for line in lines if line.lstrip())
-     trimmed = [lines[0].lstrip()] + [line[indent:].rstrip() for line in lines[1:]]
-     return "\n".join(trimmed).strip()
-
-
  def parse_docstring(docstring):
      """
      Parse out the parts of a docstring.  Return (title, body, metadata).
      """
-     docstring = trim_docstring(docstring)
+     if not docstring:
+         return '', '', {}
+     docstring = cleandoc(docstring)
      parts = re.split(r'\n{2,}', docstring)
      title = parts[0]
      if len(parts) == 1:
```

```
3 django/contrib/admindocs/views.py
@@ -1,5 +1,6 @@
  import inspect
  from importlib import import_module
+ from inspect import cleandoc
  from pathlib import Path

  from django.apps import apps
@@ -256,7 +257,7 @@ def get_context_data(self, **kwargs):
                  continue
              verbose = func.__doc__
              verbose = verbose and (
-                 utils.parse_rst(utils.trim_docstring(verbose), 'model', _('model:') + opts.model_name)
+                 utils.parse_rst(cleandoc(verbose), 'model', _('model:') + opts.model_name)
              )
              # Show properties and methods without arguments as fields.
              # Otherwise, show as a 'method with arguments'.
```

Figure 3: Case from Django (Gold).



```
3 django/contrib/admindocs/utils.py
@@ -34,7 +34,8 @@ def trim_docstring(docstring):
          return ''
      # Convert tabs to spaces and split into lines
      lines = docstring.expandtabs().splitlines()
-     indent = min(len(line) - len(line.lstrip()) for line in lines if line.lstrip())
+     # Determine the minimum indentation (first line doesn't count):
+     indent = min(len(line) - len(line.lstrip()) for line in lines[1:] if line.lstrip())
      trimmed = [lines[0].lstrip()] + [line[indent:].rstrip() for line in lines[1:]]
      return "\n".join(trimmed).strip()

```

Figure 4: Case from Django (Ours).

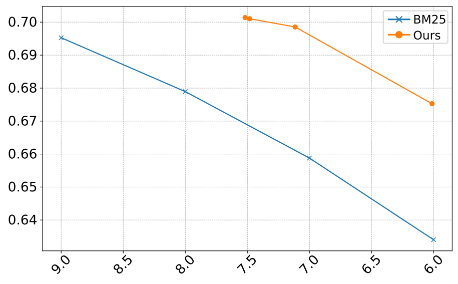


Figure 5: Comparison of Recall Scores between MAGIS and BM25.

proposed framework opts for modifications in only one file, as illustrated in Figure 4. Remarkably, this simpler code change enables the repository to successfully pass all requisite test cases.

4.3 RQ 4: How Effective Is Our Planning Process?

To investigate the effectiveness of the planning process, we analyze the repository custodian and project manager agent, respectively. The performance of the repository custodian agent is observed in the recall score versus the file number curve, as shown in Figure 5 with the horizontal axis representing the average number of files and the vertical axis denoting the recall score associated with that number of files. This curve illustrates that our method consistently outperforms the BM25 baseline across different numbers of selected files, which suggests that our custodian can find as many relevant code files as possible with the lowest possible number of files.

For the project manager agent, we examined the alignment of its generated task descriptions with the reference code change. A higher correlation score indicates a better alignment and thus, a more accurate and effective planning direction.

The correlation scores are determined by GPT-4 based on a set of criteria defined in Table 3, which spans from a score of 1, indicating no relevance, to a score of 5, indicating perfect alignment with high accuracy and completeness. The distribution of these correlation scores is presented in Figure 6, where the vertical axis represents the frequency of each score. Notably, most of the scores are 3 or above, which implies that the majority of task descriptions are in the right direction concerning planning. Furthermore, the higher scores correlate with a higher probability of issue resolution, indicated by a

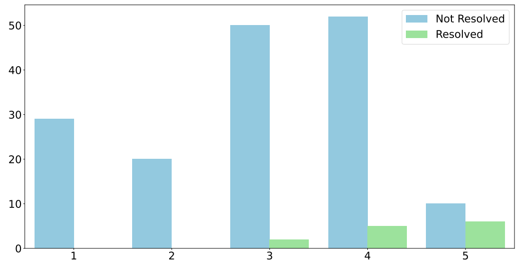


Figure 6: Distribution of the Correlation Score Between the Generated Task Description and the Reference Code Change.

Table 3: The Meaning of Scores in GPT-4 Evaluation on the Correlation Between the Generated Task Description and the Reference Code Change.

Score	Meaning
1	The code changes are unrelated to the task description.
2	The code changes address a minor part of the task but are largely irrelevant.
3	The code changes partially meet the task requirements but lack completeness or accuracy.
4	The code changes are relevant and mostly complete, with minor discrepancies from the task description.
5	The code changes perfectly align with the task description, fully addressing all specified requirements with high accuracy and completeness.

larger proportion of "resolved" outcomes in scores 4 and 5. This signifies that when the generated task description closely aligns with the reference, there is a higher possibility to resolve the issue.

The analysis above demonstrates the effectiveness of both the repository custodian and the project manager agent in the planning process of our framework.

4.4 RQ 5: How Effective Is Our Coding Process?

To investigate the effectiveness of the coding process in our framework, we analyze the performance of the developer's line locating and the issue resolving across instances of varying complexities.

Figure 7 illustrates the distribution of the line location overlap ratio of MAGIS and the baseline models (GPT-4 and Claude-2). The vertical axis quantifies the frequency of occurrences within specific ranges of line location overlap ratios for each group. This visualization reveals that our developer agent frequently attains a line location overlap ratio nearing 1. Compared with baseline models, the developer agent demonstrates a pronounced preference for higher distribution values close to 1, and conversely, a reduced preference for lower distribution values near 0. Such a distribution validates the superior performance of MAGIS in line location.

Further analysis is provided in Figure 8, which illustrates the relationship between the line location overlap ratio and the issue resolved ratio within those overlaps. In the figure, the horizontal axis represents the range of overlap ratios for each bar's corresponding interval, while the height of each bar indicates the resolved ratio for instances within that interval. These resolved ratios correspond to the scale on the left vertical axis. The orange curve represents the cumulative frequency of instances that can be resolved under different overlap ratio thresholds, with the cumulative frequency corresponding to the scale on the right side.

As shown in Figure 8, the right four bars are higher than the five left, which indicates the resolved ratio can increase with the line location overlap. This observation also suggests that accurate line location is helpful for issue resolution. The cumulative frequency curve, shown in orange, provides an additional analysis, indicating the cumulative proportion of issues resolved ratio up to each point along the line location overlap. A steady increase in cumulative frequency accompanies the increase in line location overlap, reinforcing the idea that resolving issues is more successful in areas of high overlap. The slope of the curve's left half is lower than that of the right half, indicating that the benefits of increasing the overlap ratio are less pronounced at lower overlap ratios than at higher ones. Therefore, the developer agent should prioritize improving its capability of line location.

Moreover, as shown in Table 4, we present a logistic regression analysis that quantifies the correlation between several complexity indices and issue resolution. The results show that GPT-4 has significant negative correlations across the number of files and functions, suggesting that as these indices increase, the likelihood of issue resolution decreases. Conversely, the negative correlations are less pronounced with our model, MAGIS, particularly in the number of files and functions, suggesting mitigation of challenges corresponding to these complexity indices.

To evaluate the performance of the QA engineer, the ablation experiment is conducted and the results are shown in Table 2. As the table shows, in settings with and without hints, the presence of the QA engineer can increase the resolved ratio by 1.57% and 3.31%, respectively. This overall enhancement substantiates the QA engineer's contribution to improving outcomes.

Table 4: Correlation Between the Complexity Indices and the Issue Resolution.

Method	# Files	# Functions	# Hunks	# Added LoC	# Deleted LoC	# Changed LoC
GPT-4	-25.15^*	-25.15^*	-0.06	-0.10	-0.04	-0.21
MAGIS	-1.55^*	-1.55^*	-0.12^*	-0.04^*	-0.06^*	-0.57^*

* The correlation between the index and the issue resolution is significant (P-value < 0.05).

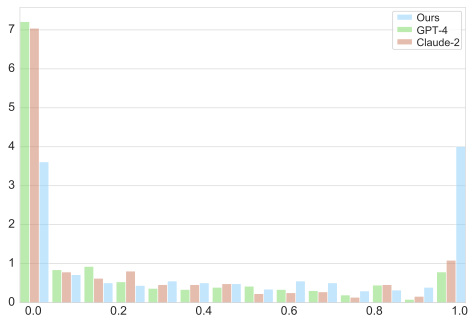


Figure 7: Comparison of Line Location Overlap between MAGIS (Ours) and Baselines.

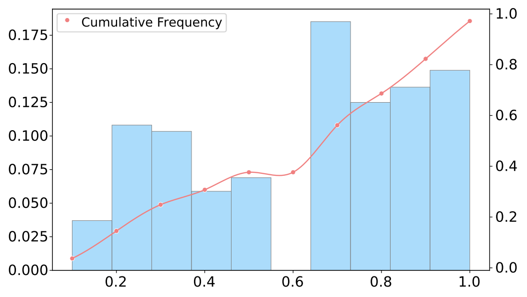


Figure 8: Resolved Ratio in Different Line Location Overlap Intervals.

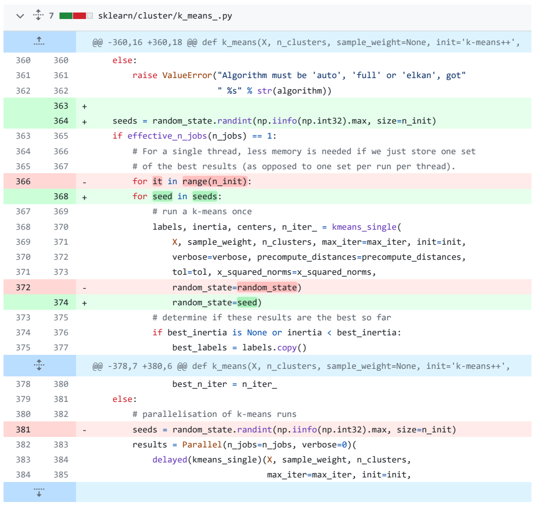
Specifically, there is an issue [8] from the repository `scikit-learn` [9] and the reference code change [10] is shown in Figure 5. During the flow of our framework, the developer firstly modifies the code as shown in Figure 9 but the parameter`random_state` (Line 371 in the new-version code) of the function `kmeans_single` is not assigned the right number in `seeds`. After the erroneous modification was made, the QA engineer identified the mistake and provided feedback. Their commentary highlighted the issue: "This code change modifies the implementation of K-means algorithm and doesn't seem entirely correct". They further elaborated, "Running the algorithm just one time could lead to worse results, compared to running it multiple times (n_init times) and choosing the best result, as was originally done" This critique specifically targets the flaw associated with the iterative process ("running times"). With the help of the QA engineer, the developer further revise the code, and the final code change is shown in Figure 10. All of the necessary test cases are passed after applying this code change. Both the ablation study and the case study underscore the QA engineer's efficacy.

5 Statistics and Discussion

This section provides statistics on code changes corresponding to resolved issues and those applicable but unresolved using our framework.

5.1 Complexity of Code Changes.

The statistics on the code change for instances with resolved issues are presented in Table 6. Overall, the statistical information of the generated code changes for these instances, such as the average number of code files, functions, hunks, and deleted lines, all differ slightly (not exceeding 0.3) from the reference solutions written by humans. This indicates that for these instances, the complexity of the code change



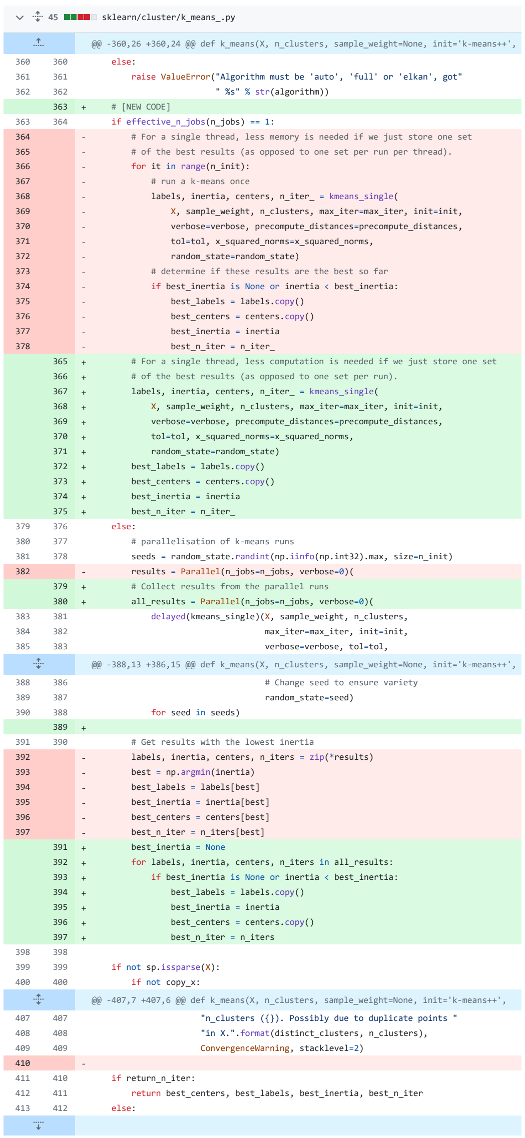
```
7 sklearn/cluster/k_means_.py
@@ -360,16 +360,18 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
360 360          else:
361 361              raise ValueError("Algorithm must be 'auto', 'full' or 'elkan', got"
362 362                               " %s" % str(algorithm))
    363   +
    364   +      seeds = random_state.randint(np.iinfo(np.int32).max, size=n_init)
363 365          if effective_n_jobs(n_jobs) == 1:
364 366              # For a single thread, less memory is needed if we just store one set
365 367              # of the best results (as opposed to one set per run per thread).
366       -          for it in range(n_init):
    368   +          for seed in seeds:
367 369                  # run a k-means once
368 370                  labels, inertia, centers, n_iter_ = kmeans_single(
369 371                      X, sample_weight, n_clusters, max_iter=max_iter, init=init,
370 372                      verbose=verbose, precompute_distances=precompute_distances,
371 373                      tol=tol, x_squared_norms=x_squared_norms,
372       -                  random_state=random_state)
    374   +                  random_state=seed)
373 375                  # determine if these results are the best so far
374 376                  if best_inertia is None or inertia < best_inertia:
375 377                      best_labels = labels.copy()
@@ -378,7 +380,6 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
378 380                      best_n_iter = n_iter_
379 381          else:
380 382              # parallelisation of k-means runs
381       -      seeds = random_state.randint(np.iinfo(np.int32).max, size=n_init)
382 383              results = Parallel(n_jobs=n_jobs, verbose=0)(
383 384                  delayed(kmeans_single)(X, sample_weight, n_clusters,
384 385                                         max_iter=max_iter, init=init,
```

Table 5: Case from `scikit-learn` (Gold).

[8] `https://github.com/scikit-learn/scikit-learn/issues/9784`

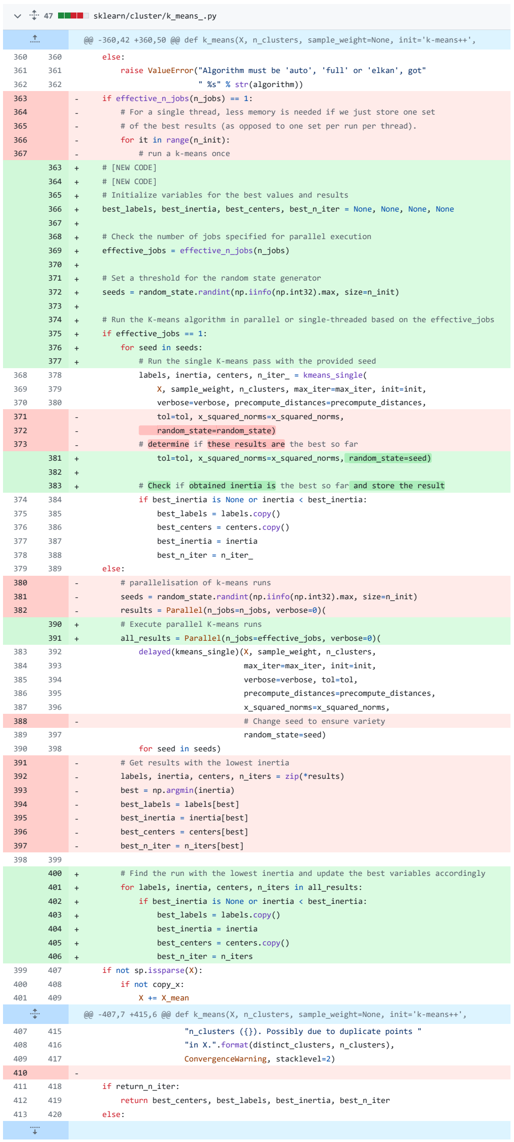
[9] `https://github.com/scikit-learn/scikit-learn`

[10] `https://github.com/scikit-learn/scikit-learn/pull/9288`



```diff
45 sklearn/cluster/k_means_.py
@@ -360,26 +360,24 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
     else:
         raise ValueError("Algorithm must be 'auto', 'full' or 'elkan', got"
                          " %s" % str(algorithm))
+    # [NEW CODE]
     if effective_n_jobs(n_jobs) == 1:
-        # For a single thread, less memory is needed if we just store one set
-        # of the best results (as opposed to one set per run per thread).
-        for it in range(n_init):
-            # run a k-means once
-            labels, inertia, centers, n_iter_ = kmeans_single(
-                X, sample_weight, n_clusters, max_iter=max_iter, init=init,
-                verbose=verbose, precompute_distances=precompute_distances,
-                tol=tol, x_squared_norms=x_squared_norms,
-                random_state=random_state)
-            # determine if these results are the best so far
-            if best_inertia is None or inertia < best_inertia:
-                best_labels = labels.copy()
-                best_centers = centers.copy()
-                best_inertia = inertia
-                best_n_iter = n_iter_
+        # For a single thread, less computation is needed if we just store one set
+        # of the best results (as opposed to one set per run).
+        labels, inertia, centers, n_iter_ = kmeans_single(
+            X, sample_weight, n_clusters, max_iter=max_iter, init=init,
+            verbose=verbose, precompute_distances=precompute_distances,
+            tol=tol, x_squared_norms=x_squared_norms,
+            random_state=random_state)
+        best_labels = labels.copy()
+        best_centers = centers.copy()
+        best_inertia = inertia
+        best_n_iter = n_iter_
     else:
         # parallelisation of k-means runs
         seeds = random_state.randint(np.iinfo(np.int32).max, size=n_init)
-        results = Parallel(n_jobs=n_jobs, verbose=0)(
+        # Collect results from the parallel runs
+        all_results = Parallel(n_jobs=n_jobs, verbose=0)(
             delayed(kmeans_single)(X, sample_weight, n_clusters,
                                    max_iter=max_iter, init=init,
                                    verbose=verbose, tol=tol,
@@ -388,13 +386,15 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
                                    # Change seed to ensure variety
                                    random_state=seed)
             for seed in seeds)
+
         # Get results with the lowest inertia
-        labels, inertia, centers, n_iters = zip(*results)
-        best = np.argmin(inertia)
-        best_labels = labels[best]
-        best_inertia = inertia[best]
-        best_centers = centers[best]
-        best_n_iter = n_iters[best]
+        best_inertia = None
+        for labels, inertia, centers, n_iters in all_results:
+            if best_inertia is None or inertia < best_inertia:
+                best_labels = labels.copy()
+                best_inertia = inertia
+                best_centers = centers.copy()
+                best_n_iter = n_iters
 
     if not sp.issparse(X):
         if not copy_x:
@@ -407,7 +407,6 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
                           "n_clusters ({}). Possibly due to duplicate points "
                           "in X.".format(distinct_clusters, n_clusters),
                           ConvergenceWarning, stacklevel=2)
-
     if return_n_iter:
         return best_centers, best_labels, best_inertia, best_n_iter
     else:
```

Figure 9: Case from `scikit-learn` (Ours, before review).



```diff
47 sklearn/cluster/k_means_.py
@@ -360,42 +360,50 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
     else:
         raise ValueError("Algorithm must be 'auto', 'full' or 'elkan', got"
                          " %s" % str(algorithm))
-    if effective_n_jobs(n_jobs) == 1:
-        # For a single thread, less memory is needed if we just store one set
-        # of the best results (as opposed to one set per run per thread).
-        for it in range(n_init):
-            # run a k-means once
+    # [NEW CODE]
+    # [NEW CODE]
+    # Initialize variables for the best values and results
+    best_labels, best_inertia, best_centers, best_n_iter = None, None, None, None
+
+    # Check the number of jobs specified for parallel execution
+    effective_jobs = effective_n_jobs(n_jobs)
+
+    # Set a threshold for the random state generator
+    seeds = random_state.randint(np.iinfo(np.int32).max, size=n_init)
+
+    # Run the K-means algorithm in parallel or single-threaded based on the effective_jobs
+    if effective_jobs == 1:
+        for seed in seeds:
+            # Run the single K-means pass with the provided seed
             labels, inertia, centers, n_iter_ = kmeans_single(
                 X, sample_weight, n_clusters, max_iter=max_iter, init=init,
                 verbose=verbose, precompute_distances=precompute_distances,
-                tol=tol, x_squared_norms=x_squared_norms,
-                random_state=random_state)
-            # determine if these results are the best so far
+                tol=tol, x_squared_norms=x_squared_norms, random_state=seed)
+
+            # Check if obtained inertia is the best so far and store the result
             if best_inertia is None or inertia < best_inertia:
                 best_labels = labels.copy()
                 best_centers = centers.copy()
                 best_inertia = inertia
                 best_n_iter = n_iter_
     else:
-        # parallelisation of k-means runs
-        seeds = random_state.randint(np.iinfo(np.int32).max, size=n_init)
-        results = Parallel(n_jobs=n_jobs, verbose=0)(
+        # Execute parallel K-means runs
+        all_results = Parallel(n_jobs=effective_jobs, verbose=0)(
             delayed(kmeans_single)(X, sample_weight, n_clusters,
                                    max_iter=max_iter, init=init,
                                    verbose=verbose, tol=tol,
                                    precompute_distances=precompute_distances,
                                    x_squared_norms=x_squared_norms,
-                                   # Change seed to ensure variety
                                    random_state=seed)
             for seed in seeds)
-        # Get results with the lowest inertia
-        labels, inertia, centers, n_iters = zip(*results)
-        best = np.argmin(inertia)
-        best_labels = labels[best]
-        best_inertia = inertia[best]
-        best_centers = centers[best]
-        best_n_iter = n_iters[best]
 
+        # Find the run with the lowest inertia and update the best variables accordingly
+        for labels, inertia, centers, n_iters in all_results:
+            if best_inertia is None or inertia < best_inertia:
+                best_labels = labels.copy()
+                best_inertia = inertia
+                best_centers = centers.copy()
+                best_n_iter = n_iters
     if not sp.issparse(X):
         if not copy_x:
             X += X_mean
@@ -407,7 +415,6 @@ def k_means(X, n_clusters, sample_weight=None, init='k-means++',
                           "n_clusters ({}). Possibly due to duplicate points "
                           "in X.".format(distinct_clusters, n_clusters),
                           ConvergenceWarning, stacklevel=2)
-
     if return_n_iter:
         return best_centers, best_labels, best_inertia, best_n_iter
     else:
```

Figure 10: Case from `scikit-learn` (Ours, after review).

Table 6: The Statistical Analysis of Our Framework on Resolved Instances.

	MAGIS			Gold		
	Min	**Max**	**Avg.**	**Min**	**Max**	**Avg.**
# Code Files	1	2	1.02	1	2	1.04
# Functions	1	2	1.02	1	2	1.04
# Hunks	1	4	1.45	1	6	1.66
# Added Lines	1	146	9.75	0	38	4.34
# Deleted Lines	0	77	5.27	0	115	5.16
Change Start Index	1	1,655	270.12	1	1,657	256.09
Change End Index	22	1,665	301.68	0	1,666	315.05
# Changed Lines	2	190	15.02	1	115	9.50

generated by our framework is similar to that of humans. Furthermore, the maximum values observed in the table reveal that our framework can implement code modifications involving two files, four hunks, and 1,655 lines modification, with single modifications reaching up to 190 lines. Results demonstrate the effectiveness of our method in resolving complex issues that need to modify the code file on multiple locations and with long context.

Specifically, the distribution of the number of modified lines for the resolving instances is shown in Figure 11. We observe that the distribution of the number of modified lines in our framework for the solved instances exceeds that of the reference solution, especially in terms of the number of added lines being significantly higher than the reference. Upon manual inspection, we found that the generation results provided by our framework often contained more comment information, which led to an increase in the total number of modified lines. For example, Figure 10 displays the generation result of our framework. Lines 365, 368, 371, 374, 383 in the new version file correspond to the comment for the added code. These natural language descriptions are valuable in actual software evolution [14, 22]. In contrast, Figure 5 shows a human-written solution lacking such explanatory comments, which might disadvantage software maintainers in reading and understanding.

The statistics on the code change for instances without resolved issues are shown in Table 7. From the table, our framework can generate applicable code changes including up to 13 files and 28 hunks, and the location of the modifications can be as far as line 7,150, with a single modification reaching up to 9,367 lines. These results suggest that our method has a strong adaptability in generating applicable code changes. However, considering that these code changes have not passed all the potential test cases they could pass, which indicates that there is still room for improvement.

To further analyze the reasons behind the failure of test cases in these instances, we have quantified the distribution of the lengths of code changes in the unresolved instances, as shown in Figure 12. From the figure, we observe that for unresolved instances, the framework tends to delete a larger number of lines while adding fewer lines, in contrast to the distribution of human-written changes.

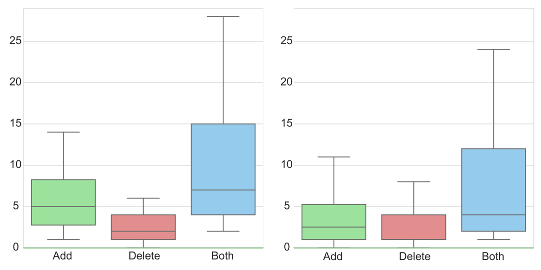


Figure 11: Distribution of the LoC in the Resolved Instances.

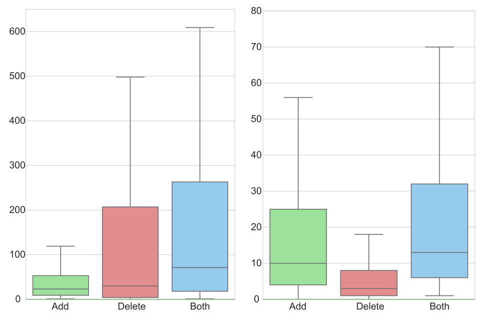


Figure 12: Distribution of the LoC in the Applied but Not Resolved Instances.

Table 7: The Statistical Analysis of Our Framework on Applied but Not Resolved Instances.

	MAGIS			Gold		
	Min	**Max**	**Avg.**	**Min**	**Max**	**Avg.**
# Code Files	1	13	1.50	1	18	1.61
# Functions	1	13	1.50	1	18	1.61
# Hunks	1	28	2.52	1	52	3.72
# Added Lines	1	920	40.38	0	$3,050$	28.27
# Deleted Lines	0	$9,160$	327.27	0	$2,975$	14.51
Change Start Index	1	$4,568$	424.84	0	$6,651$	485.01
Change End Index	9	$7,150$	513.13	0	$6,658$	728.96
# Changed Lines	1	$9,367$	367.65	1	$6,025$	42.79

This discrepancy may point to different repair strategies or attitudes towards problem-solving, where the framework presented herein might prefer to reduce errors by removing potentially problematic code, whereas human developers may lean towards adding new code to address issues.

Moreover, a comparison between Table 6 and Table 7 reveals that the latter contains a higher overall number of files, hunks, and changed lines of code. These instances, involving more modification locations, correspond to more complex scenarios. This phenomenon suggests that the performance of our framework in resolving such complex issues requires further enhancement.

Furthermore, the variability in difficulty across different software repositories may influence the effectiveness of code changes. To this end, we compile statistics on the resolved ratios in various software repositories, as shown in Figure 13. From the figure, we observe that there is a significant variation in the resolved ratios across different repositories in our framework. Some repositories have a resolved ratio as high as 40%, while others are close to 0%. This suggests that the differences among various software such as code structure and coding style can impact the generation and application of the code change.

5.2 Comparison with Devin.

Devin is a novel agent for software development [34], and its performance has also been assessed using the SWE-bench. However, the evaluation dataset employed by Devin differs from the subset used for experiments with GPT-4 reported by the paper of SWE-bench [15]. An analysis of the repository name and pull request ID of each instance reveals that only 140 instances overlap between the two datasets.

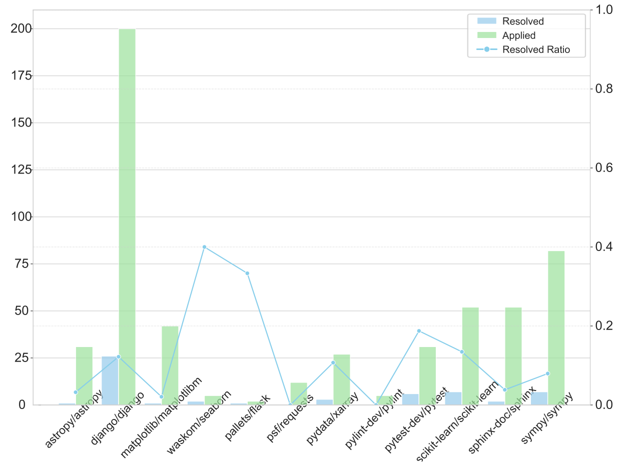


Figure 13: The Number of Applied and Resolved Instances in Different Repositories.

Within the shared pool of 140 instances, our framework successfully resolves 21 (15%) issues, surpassing Devin's resolution of 18 (12.86%) issues [11]. This comparison, however, may not be entirely equitable. Devin's possible underlying LLM is unknown, and it possesses the capability to integrate feedback from the environment. Moreover, Devin's reported scores are under the setting given the entire repository, and it operates with "common developer tools including the shell, code editor, and browser", and "agents with internet access could

[11]https://github.com/CognitionAI/devin-swebench-results/tree/main/output_diffs/pass

potentially find external information through other methods" as detailed at the report [12]. In contrast, our approach solely relies on the shell, without the need of any additional external tools.

For running time, 72% of instances resolved by Devin require greater than 10 minutes to complete. In contrast, our framework finalizes each resolved issue within approximately 3 minutes. On average, our framework completes the processing of each instance in under 5 minutes, demonstrating its capability to assist in resolving GitHub issues with minimal time expenditure.

6 Limitation

Prompt. The design of prompt words may impact the performance of LLMs, thereby affecting the validity and fairness of the results [8]. While this paper focuses on innovative aspects of the proposed framework design and relies on practical guidelines for the design of prompt word templates [29] to reduce the emergence of design biases, the complete elimination of the prompt bias is extremely difficult due to the inherent biases in the dataset instances and the limitations of API resources.

Dataset. The dataset contains a limited variety of software types. The evaluating dataset, SWE-bench, encompasses 12 repositories, which cover the Python programming language. However, this quantity remains insufficient compared to the diverse software projects available on GitHub. The code style, architectural design, and implementation techniques of these selected repositories, while representative, cannot fully reflect the diversity of all code repositories. In particular, the current dataset may fail to encompass some specialized fields or different programming paradigms, such as microservice architecture [41] and functional programming [16]. This limitation implies that, although our framework is designed to be independent of any specific software, the validation of its effectiveness and general applicability might be affected by this limited sample scope. Therefore, applying the findings of this paper to other code repositories may require further validation.

7 Related Work

7.1 Large Language Models

Large Language Models (LLMs) refer to the pre-trained language models that contain a large number of parameters [38]. The parameter counts of these models typically range in the tens or hundreds of billions. Popular LLMs include the Generative Pre-trained Transformer (GPT) series, such as GPT-3 [27], GPT-4 [25], and the open-source LLaMA [35] which publicly shares its weight information. The first version of the open-source model LLaMA has parameters ranging from 7 billion to 65 billion. Many researchers [33, 11] have built upon the foundation of LLaMA, implementing enhancements to forge new LLMs. These LLMs have demonstrated formidable natural language generation capabilities in general scenarios, with GPT-4, in particular, standing out [18, 39]. It has consistently maintained the top position in several rankings, including code generation, reflecting its significant potential in tasks related to software engineering [13].

7.2 LLM-Based Multi-Agent System

With the powerful text generation capabilities of LLMs, many researchers [12, 31, 7, 37, 26, 36] have explored the construction of LLM-based Multi-Agent Systems, enabling them to accomplish tasks beyond the capabilities of the LLMs themselves. For example, MetaGPT [12], which simulates the Standardized Operating Procedures (SOPs) of a programming team, completing tasks including definition, design, planning, coding, and testing through constructed roles (e.g., product managers, architects, project managers, etc.). This framework has achieved leading scores on the HumanEval [9] and MBPP [2], outperforming many LLMs, and researchers show its ability to complete a software establishment (e.g., a code repository to play Gomoku game), indicating that a multi-agent framework can better leverage the capabilities of LLMs in code generation tasks. Moreover, Qian et al. [26] designed ChatDev, a virtual development company simulating a human development team, which decomposes requirements into atomic tasks assigned to the developers. Developers mitigate the hallucination that may arise with the LLM through mutual communication and self-reflection mechanisms. Experimental results show that ChatDev can complete the establishment of some small

[12] `https://www.cognition-labs.com/introducing-devin`

software (averaging no more than 5 files per software) in a relatively short time (less than 7 minutes on average). However, these works focus on the transformation from the requirement to software and overlook the code change generation during software evolution which needs not only understanding the requirement but also dealing with the large repository.

8 Conclusion

In conclusion, this paper illuminates the potential of large language models in software evolution, particularly in resolving GitHub issues. Our study identifies the challenges of direct LLM application. To address them, we proposed a novel LLM-based multi-agent framework, MAGIS, enhancing issue resolution through well-designed agents' collaboration. The superiority of MAGIS on the SWE-bench dataset against popular LLMs highlights its effectiveness, pointing towards a promising direction for integrating LLMs into software evolution workflows. This work not only shows the potential of LLMs in GitHub issue resolution but also explores an LLM-based paradigm for software evolution.

References

[1] Anthropic. Claude 2. `https://www.anthropic.com/news/claude-2`, 2023.

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv Preprint*, abs/2108.07732, 2021. URL `https://arxiv.org/abs/2108.07732`.

[3] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. Factors influencing code review processes in industry. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 85–96. ACM, 2016. doi: 10.1145/2950290.2950323. URL `https://doi.org/10.1145/2950290.2950323`.

[4] Tegawendé F. Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillère, Jacques Klein, and Yves Le Traon. Got issues? who cares about it? A large scale investigation of issue trackers from github. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 188–197. IEEE Computer Society, 2013. doi: 10.1109/ISSRE.2013.6698918. URL `https://doi.org/10.1109/ISSRE.2013.6698918`.

[5] Amiangshu Bosu and Jeffrey C. Carver. Impact of developer reputation on code review outcomes in OSS projects: an empirical investigation. In Maurizio Morisio, Tore Dybå, and Marco Torchiano, editors, *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pages 33:1–33:10. ACM, 2014. doi: 10.1145/2652524.2652544. URL `https://doi.org/10.1145/2652524.2652544`.

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv Preprint*, abs/2303.12712, 2023. doi: 10.48550/ARXIV.2303.12712. URL `https://doi.org/10.48550/arXiv.2303.12712`.

[7] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv Preprint*, abs/2308.07201, 2023. doi: 10.48550/ARXIV.2308.07201. URL `https://doi.org/10.48550/arXiv.2308.07201`.

[8] Lichang Chen, Jiuhai Chen, Heng Huang, and Minhao Cheng. PTP: boosting stability and performance of prompt tuning with perturbation-based regularizer. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 13512–13525. Association for Computational Linguistics, 2023. URL `https://aclanthology.org/2023.emnlp-main.833`.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. Evaluating large language models trained on code. *arXiv Preprint*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`.

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

[11] Xinyang Geng and Hao Liu. Openllama: An open reproduction of llama, May 2023. URL `https://github.com/openlm-research/open_llama`.

[12] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2023.

[13] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv Preprint*, abs/2308.10620, 2023. doi: 10.48550/ARXIV.2308.10620. URL `https://doi.org/10.48550/arXiv.2308.10620`.

[14] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. Practitioners' expectations on automated code comment generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1693–1705. ACM, 2022. doi: 10.1145/3510003.3510152. URL `https://doi.org/10.1145/3510003.3510152`.

[15] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=VTF8yNQM66`.

[16] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer, 1987. doi: 10.1007/3-540-18317-5_10. URL `https://doi.org/10.1007/3-540-18317-5_10`.

[17] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In Rainer Koschke, Jens Krinke, and Martin P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 111–120. IEEE Computer Society, 2015. doi: 10.1109/ICSM.2015.7332457. URL `https://doi.org/10.1109/ICSM.2015.7332457`.

[18] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html`.

[19] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv Preprint*, abs/2307.03172, 2023. doi: 10.48550/ARXIV.2307.03172. URL `https://doi.org/10.48550/arXiv.2307.03172`.

[20] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv Preprint*, abs/2307.03172, 2023. doi: 10.48550/ARXIV.2307.03172. URL `https://doi.org/10.48550/arXiv.2307.03172`.

[21] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In Premkumar T. Devanbu, Sung Kim, and Martin Pinzger, editors, *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 192–201. ACM, 2014. doi: 10.1145/2597073.2597076. URL `https://doi.org/10.1145/2597073.2597076`.

[22] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. Developer-intent driven code comment generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 768–780. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00073. URL `https://doi.org/10.1109/ICSE48619.2023.00073`.

[23] OpenAI. GPT-4 technical report, 2023. URL `https://doi.org/10.48550/arXiv.2303.08774`.

[24] OpenAI. Gpt-3.5 turbo fine-tuning and api updates. `https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates`, 2023.

[25] OpenAI. Gpt-4. `https://openai.com/research/gpt-4`, 2023.

[26] Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv Preprint*, 2023.

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training, 2018.

[28] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In Donna K. Harman, editor, *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994*, volume 500-225 of *NIST Special Publication*, pages 109–126. National Institute of Standards and Technology (NIST), 1994. URL `http://trec.nist.gov/pubs/trec3/papers/city.ps.gz`.

[29] Jessica Shieh. Best practices for prompt engineering with openai api. *OpenAI, February https://help. openai. com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api*, 2023.

[30] Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu. A survey of neural code intelligence: Paradigms, advances and beyond, 2024.

[31] Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent LLM agents. *arXiv Preprint*, abs/2306.03314, 2023. doi: 10.48550/ARXIV.2306.03314. URL `https://doi.org/10.48550/arXiv.2306.03314`.

[32] Wei Tao, Yucheng Zhou, Yanlin Wang, Hongyu Zhang, Haofen Wang, and Wenqiang Zhang. Kadel: Knowledge-aware denoising learning for commit message generation. *ACM Trans. Softw. Eng. Methodol.*, jan 2024. ISSN 1049-331X. doi: 10.1145/3643675. URL `https://doi.org/10.1145/3643675`.

[33] LLaMA-MoE Team. Llama-moe: Building mixture-of-experts from llama with continual pre-training, Dec 2023. URL `https://github.com/pjlab-sys4nlp/llama-moe`.

[34] The Cognition Team. Swe-bench technical report, 2024. URL `https://www.cognition-labs.com/post/swe-bench-technical-report`.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[36] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Autodev: Automated ai-driven development, 2024.

[37] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *arXiv Preprint*, abs/2308.08155, 2023. doi: 10.48550/ARXIV.2308.08155. URL `https://doi.org/10.48550/arXiv.2308.08155`.

[38] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *arXiv Preprint*, abs/2303.18223, 2023. doi: 10.48550/ARXIV.2303.18223. URL `https://doi.org/10.48550/arXiv.2303.18223`.

[39] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html`.

[40] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. Towards an understanding of large language models in software engineering tasks. *arXiv Preprint*, abs/2308.11396, 2023. doi: 10.48550/ARXIV.2308.11396. URL `https://doi.org/10.48550/arXiv.2308.11396`.

[41] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Software Eng.*, 47(2):243–260, 2021. doi: 10.1109/TSE.2018.2887384. URL `https://doi.org/10.1109/TSE.2018.2887384`.

[42] Yucheng Zhou, Xiubo Geng, Tao Shen, Chongyang Tao, Guodong Long, Jian-Guang Lou, and Jianbing Shen. Thread of thought unraveling chaotic contexts. *arXiv Preprint*, abs/2311.08734, 2023. doi: 10.48550/ARXIV.2311.08734. URL `https://doi.org/10.48550/arXiv.2311.08734`.