AutoCodeRover: Autonomous Program Improvement

Yuntong Zhang
National University of Singapore
yuntong@comp.nus.edu.sg

Haifeng Ruan
National University of Singapore
hruan@comp.nus.edu.sg

Zhiyu Fan
National University of Singapore
zhiyufan@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Abstract

Researchers have made significant progress in automating the software development process in the past decades. Automated techniques for issue summarization, bug reproduction, fault localization, and program repair have been built to ease the workload of developers. Recent progress in Large Language Models (LLMs) has significantly impacted the development process, where developers can use LLM-based programming assistants to achieve automated coding. Nevertheless, software engineering involves the process of program improvement apart from coding, specifically to enable software maintenance (e.g. program repair to fix bugs) and software evolution (e.g. feature additions). In this paper, we propose an automated approach for solving Github issues to autonomously achieve program improvement. In our approach called AutoCodeRover, LLMs are combined with sophisticated code search capabilities, ultimately leading to a program modification or patch. In contrast to recent LLM agent approaches from AI researchers and practitioners, our outlook is more software engineering oriented. We work on a program representation (abstract syntax tree) as opposed to viewing a software project as a mere collection of files. Our code search exploits the program structure in the form of classes/methods to enhance LLM's understanding of the issue's root cause, and effectively retrieve a context via iterative search. The use of spectrum-based fault localization using tests, further sharpens the context, as long as a test-suite is available. Experiments on the recently proposed SWE-bench-lite (300 real-life Github issues) show increased efficacy in solving Github issues (19% on SWE-bench-lite), which is higher than the efficacy of the recently reported Swe-agent. Interestingly, our approach resolved 57 GitHub issues in about 4 minutes each (pass@1), whereas developers spent more than 2.68 days on average. In addition, AutoCodeRover achieved this efficacy with significantly lower cost (on average, $0.43 USD), compared to other baselines. We posit that our workflow enables autonomous software engineering, where, in future, auto-generated code from LLMs can be autonomously improved.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680384

CCS Concepts

• **Software and its engineering → Automatic programming**; **Maintaining software**; **Software testing and debugging**; • **Computing methodologies → Natural language processing**.

Keywords

large language model, automatic program repair, autonomous software engineering, autonomous software improvement

ACM Reference Format:
Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680384

1 Beyond Automatic Programming

Automating software engineering tasks has long been a vision among software engineering researchers and practitioners. One of the key challenges has been the handling of ambiguous natural language requirements, in the process of automatic programming. In addition, there has been progress in some other software engineering activities such as automated test generation [5, 7], automated program repair [13], and so on.

Recent progress in large language models (LLMs) and the appearance of tools like Github Copilot [32] hold significant promise in automatic programming. This progress immediately raises the question of whether such automatically generated code can be trusted to be integrated into software projects, and if not, what improvements to the technology are needed. One possibility is to automatically repair generated code to achieve *trust*. This brings out the importance of automating program repair tasks towards achieving the vision of autonomous software engineering.

Given this motivation of automating program repair, and the large number of hours developers often spend manually fixing bugs, we looked into the possibility of fully autonomous program improvement. Specifically, we feel that bug fixing and feature addition are the two key categories of tasks that a development team may focus on when maintaining an existing software project. To achieve this goal, we proposed an approach that augments LLM with context knowledge from the code repository. We call our tool AutoCodeRover.

Technically our solution works as follows. Given a real-life GitHub issue, LLM first analyzes the attached natural language description to extract keywords that may represent files/classes/methods/code snippets in the codebase. Once these keywords are identified, we employ a stratified strategy for the LLM agent to retrieve

code context by invoking multiple necessary code search APIs at one time with the keyword combinations as arguments (e.g., search_method_in_file). These code search APIs are running locally based on AST analysis and are responsible for retrieving code context such as class signatures and method implementation details from a particular location in the codebase. By collecting the project context with code search APIs, LLM refines its understanding of the issue based on the currently available context. Note that our code context retrieval will proceed in an iterative fashion. The LLM agent directs the navigation and decides which code search APIs to use (i.e. where/what to retrieve code) in each iteration based on the current available context returned from the previous API calls. AutoCodeRover then enquires whether there is sufficient project context, and subsequently uses the collected context to derive the buggy locations. The patch construction is then handled by another LLM agent which considers the buggy locations as well as all the context collected so far for those locations.

AutoCodeRover also can leverage debugging techniques such as spectrum-based fault localization (SBFL) [44] to decide more precise code search APIs for context retrieval if a test suite accompanying the project is available. SBFL primarily considers the control flow of the passing and failing tests and assigns a suspiciousness score to the different methods of the program. The LLM agent may prioritize retrieving context from particular methods and classes if the fault localization result is provided, e.g., when a method appears both in the issue description and in the output of fault localization. In the last step, AutoCodeRover may perform patch validation using available tests, to determine whether the patch produced by AutoCodeRover passes the tests in the given test-suite. Otherwise, AutoCodeRover will rerun the patch generation with a retry limit until a correct patch is found.

Contributions. Our contribution lies in the effective use of code search to make software engineering processes like program repair autonomous. Instead of viewing the codebase as a collection of files, we posit AutoCodeRover as an approach that gleans specification from software structure, which is used to guide the patching. The code search in AutoCodeRover is a vehicle for inferring specification from program structure. We demonstrate this capability by autonomously solving GitHub issues. We report favorable experimental results on the SWE-bench lite dataset [17]. We achieve 19% efficacy on the SWE-bench lite with 300 GitHub issues. Overall, we see the need to endow a *software engineering oriented outlook* to the recent flurry of activity on LLM agents for software engineering, which mostly has only an AI flavor. The angle of software engineering can be conveyed by the following five dimensions.

- We work on program representations (abstract syntax tree or AST) as opposed to viewing a software project as a collection of files. We posit that working on program representations like AST will be useful in autonomous software engineering workflows.
- To solve GitHub issues, we focus on code search in a way that resembles the activity of a human software engineer. So we try to use the program structure - classes, methods, code snippets - in searching for relevant code context. This leads to a more effective usage of the context provided to LLM.
- We posit that higher efficacy of automated repair is more important than time efficiency, *as long as the time is within a threshold.* It is well-known in empirical software engineering research that time limits of 30-60 minutes for automated repair are tolerable, based on extensive developer surveys in the field [31]. We thus report 19% efficacy on SWE-bench lite in solving GitHub issues, within 4 minutes. We can compare this time limit of 4 minutes to the average time of 2.68 days to fix the GitHub issues manually.
- One should be able to exploit debugging techniques like test-based fault localization to guide the search for code in resolving GitHub issues. We show that the use of fault localization in setting the code context leads to an increase in the efficacy of AutoCodeRover in solving GitHub issues.
- Finally, it is also useful to examine how many of the solved GitHub issues are producing acceptable patches. We study the patches produced by AutoCodeRover and report that 2/3 of the autonomously produced patches from AutoCodeRover are correct and acceptable. We note that this aspect has not been reported by Devin [24] and SWE-agent [47].

2 Relevant Literature

2.1 Program Repair

Test-suite based automated program repair (APR) has attracted significant attention in the last decade [13]. These techniques aim to generate a patch for a buggy program to pass a given test-suite. APR techniques typically include search-based, semantic-based, and pattern/learning-based APR. Search-based APR techniques like GenProg [42] take a buggy program and generate patches using predefined code mutation operators, or search for a patch over the patch space that passes the given test suite. Semantics-based APR techniques [28, 30] generate patches by formulating a repair constraint that needs to be satisfied based on a given test suite specification, and then solving the repair constraint to generate patches. Learning-based APR techniques [26, 38, 48] often train a deep learning model with large code repositories and are guided by a specific representation of code syntax and semantics to predict the next tokens that are most likely to be correct patch. There are also works [23, 29, 39] that tried to leverage GitHub issues and bug reports to improve APR effectiveness.

Recent work [10, 15, 46] have shown the use of LLMs for automated program repair. This line of work often assumes the buggy program statements are given (i.e. perfect fault localization assumption) and focuses on constructing APR-specific prompts that guide LLM to generate a patch for the selected buggy program statements multiple times until a patch that passes all tests is found. The typical work targets repairing functional [10, 35, 41], type [9] bugs, and even software vulnerabilities [11, 18, 33] in different languages with open-source/commercialised models (e.g., CodeT5, ChatGPT, GPT-4, etc). However, obtaining buggy locations for a large project is an essential and challenging task in resolving real-life bug reports.

APR techniques have been successfully deployed in industries for domain-specific bug fixing [4, 27, 43]. However, a long-standing challenge for the APR techniques is to resolve general real-life software issues from scratch. The above APR techniques rely on a high-quality test suite which is not always available in the real world and they do not leverage the valuable natural language specification from the original problem description. To address these

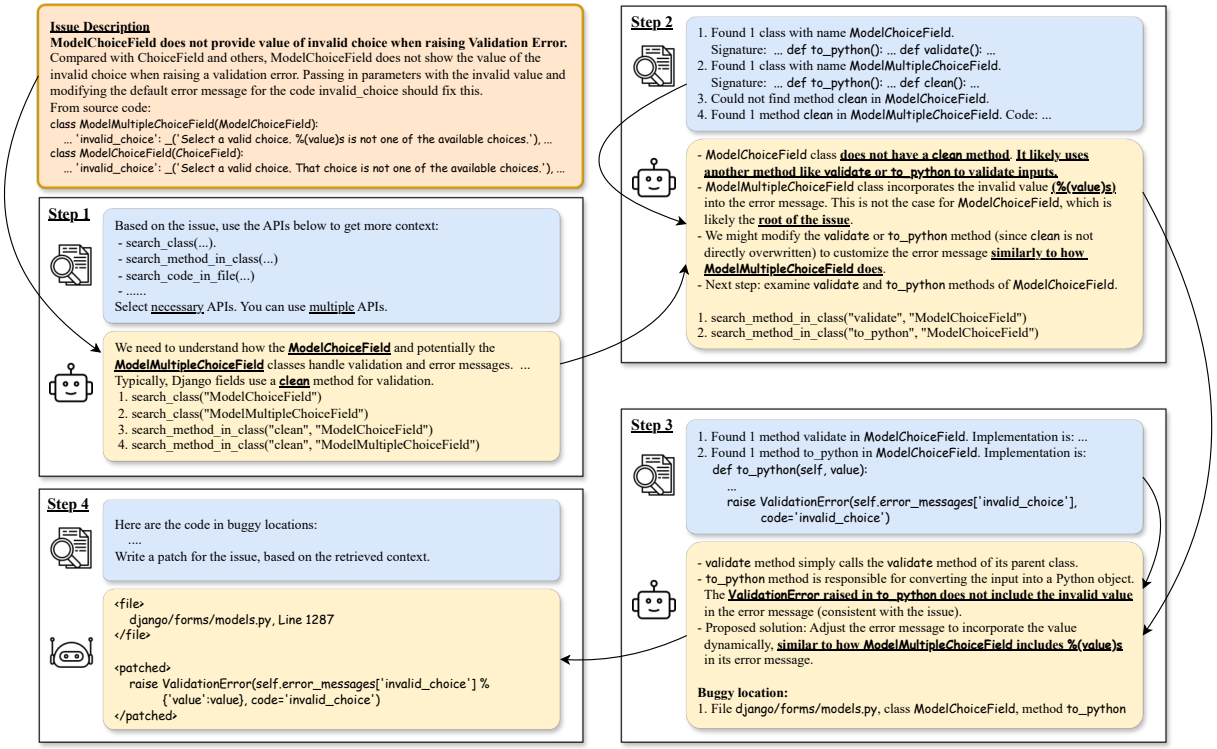

Figure 1: Issue description of django-13933 and AutoCodeRover's workflow on it.

challenges and achieve autonomous software engineering, we focus on resolving GitHub issues from a real-life dataset.

2.2 LLM Agents for SE and Dataset

SWE-bench lite [17] is a benchmark that aims to evaluate the capabilities of large language models in resolving end-to-end real-life software engineering tasks. The benchmark consists of 300 real-life software engineering task instances collected from the repositories of 11 popular large Python projects [1] (e.g., django, sympy). Each SWE-bench lite task instance contains a pair of Github issue and corresponding pull requests. The GitHub issue either reports a bug to be fixed or requests to implement new features. The pull request includes the code changes made by human developers to resolve the issue and test cases that prevent the issue. Unlike traditional code generation tasks in HumanEval [8] and MBPP [25] benchmark, resolving a SWE-bench lite instance is particularly challenging because it requires automatically generating code changes that address the problem in a GitHub issue for a matured large code repository based only on the issue description. More specifically, the process may involve a series of complex tasks like reasoning the target bug location across files in the code repository, analyzing the root cause of the issue, proposing bug-fixing strategies, and eventually writing a patch that passes all the test cases added in the pull request. There are a few primary attempts at tackling tasks in SWE-bench lite. Devin [24] is named the first AI software engineer that can solve various software engineering tasks, including building a project from scratch, bug-fixing/feature-addition for existing projects. However, it is a close-sourced commercial tool, and its details are not available. Swe-agent [47] is a concurrent work against AutoCodeRover. It designed an agent-computer interface (ACI) that allows LLM agents to execute basic file operations via shell commands to achieve interaction between the LLM engine and a software repository. Compared to Swe-agent, we posit AutoCodeRover as an approach that gleans specification from software structure, which is used to guide the patching. Instead of viewing the codebase as a collection of files.

3 Motivating Example

In this section, we illustrate how our tool AutoCodeRover can collect code context and generate a patch from an issue description and the corresponding project code-base. We show an example of a feature addition task. Figure 1 demonstrates the workflow of AutoCodeRover on an issue submitted to the Django issue tracker[2]. This issue is classified as "New feature" in the issue tracker, and is included in SWE-bench lite with the id "django-13933". The first part of Figure 1 shows the issue description (the code part is simplified for brevity). This issue requests adding support to the `ModelChoiceField` class, so that it "shows the value of the invalid choice when raising a validation error".

With this issue description, AutoCodeRover operates in two stages - *context retrieval* (Step 1-3 in Figure 1) and *patch generation*

[1] https://www.swebench.com/index.html

[2] https://code.djangoproject.com/ticket/32347

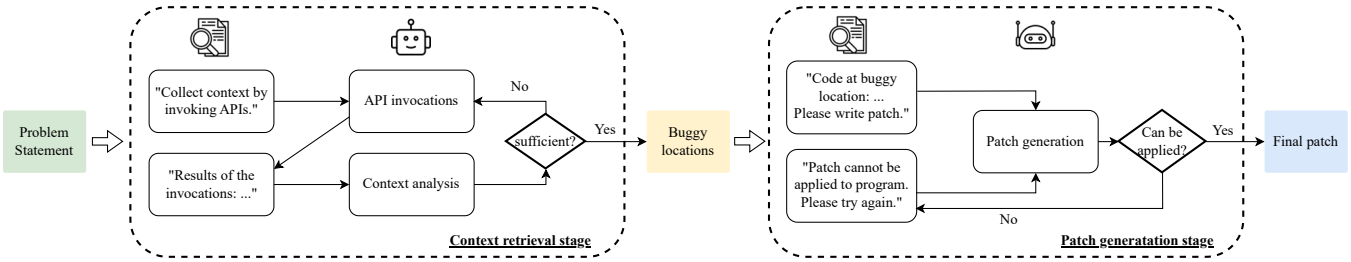

Figure 2: Overall workflow of AutoCodeRover.

```
""" AutoCodeRover patch. """
 except (ValueError, TypeError, self.queryset.model.DoesNotExist):
-    raise ValidationError(self.error_messages['invalid_choice'],
-        code='invalid_choice')
+    % Include the invalid value in the error message
+    raise ValidationError(
+        self.error_messages['invalid_choice'] % {'value': value},
+        code='invalid_choice')
""" Developer patch. """
 except (ValueError, TypeError, self.queryset.model.DoesNotExist):
-    raise ValidationError(self.error_messages['invalid_choice'],
-        code='invalid_choice')
+    raise ValidationError(
+        self.error_messages['invalid_choice'],
+        code='invalid_choice',
+        params={'value': value},)
```

Figure 3: AutoCodeRover and developer patch for Django-13933.

(Step 4). First, a context retrieval LLM agent is instructed to collect the relevant code context related to this issue, from a local copy of the Django project codebase. This retrieval is achieved by inferring relevant names (e.g. `ModelChoiceField`) and searching for them in the Abstract Syntax Tree (AST) of the project. A set of retrieval APIs are provided to the agent. In Step 1, the agent identifies the classes `ModelChoiceField` and `ModelMultipleChoiceField` might be relevant based on the issue description, and infers that a `clean` method might be also of interest for validation-related issues. It then invokes the `search_class` and `search_method_in_class` APIs to retrieve more information about them.

In Step 2, API invocations return the signature of the classes and implementation of the methods found. Results of invocation #3 reveals the absence of method `clean` in the class `ModelChoiceField`, which helps to *refine* the agent's understanding from Step 1. The returned class signature from invocation #1 also reveals two more relevant methods `to_python` and `validate`, which were not mentioned in the issue description. This suggests the retrieval should be performed *iteratively* in multiple steps, so that results from a previous search can become arguments of the following search. In this example, the agent then iteratively invokes `search_method_in_class` on the two newly revealed methods. Furthermore, by referencing results from multiple invocations, the agent can infer that `ModelMultipleChoiceField` incorporates the invalid value into the message with `%(value)s`, and methods in `ModelChoiceField` can be modified *similarly* to `ModelMultipleChoiceField`.

In Step 3, the agent receives the implementation of `validate` and `to_python` methods. Among the two methods, it selects `to_python` to be the more suitable place to make changes, since `to_python` raises the relevent exception and does not include the invalid value. At this point, the retrieval agent deems the collected code context as sufficient for understanding the issue and drafting the patch. The identified buggy location, together with the gathered context and analysis so far, is passed to another patch generation agent. This agent is instructed to write patches following the format described in Step 4 (see yellow box) of Figure 2. In Step 4, a patch is written to allow a `value` to be integrated into the error message, by utilizing %-formatting in Python. Figure 3 shows the patch generated by AutoCodeRover and the developer. Although written in a different way compared to the developer patch, this patch achieves similar effect and passes the developer provided test-suite for this issue.

4 AI Program Improvement Framework

In this section, we discuss the design of AutoCodeRover. AutoCodeRover is a system incorporating AI agents for program improvement tasks in large software projects. AutoCodeRover is designed to work in a realistic software development lifecycle, in which users submit issue reports to a software repository describing a bug, and the project maintainers craft a patch to resolve the issue. With a submitted issue, AutoCodeRover autonomously analyzes the submitted issue, retrieves the relevant code context in the software project, and generates a patch. This patch can then be vetted by human developers. If such a tool can automatically handle a certain percentage of the issues awaiting developers' attention, manual efforts are reduced.

4.1 Overview

We first describe the overall stages AutoCodeRover operates in, and will proceed in more details in the rest of this section. The overall workflow of AutoCodeRover is shown in Figure 2. AutoCodeRover takes in as input a problem statement P of the issue to be resolved, and a codebase C of the corresponding software project. This problem statement P contains the title and description of the issue, as shown in Section 3. From a problem statement written in natural language, AutoCodeRover analyzes the requirement from it and proceeds in two main stages, which are *context retrieval* and *patch generation*.

In the *context retrieval* stage, AutoCodeRover employs an LLM agent to navigate through a potentially large codebase C and extract the relevant code snippets relevant to P. This navigation is facilitated by a set of *context retrieval APIs* (Section 4.2), which enables an LLM to retrieve information about the project (e.g. class

Table 1: List of Context Retrieval APIs.

API name	Description	Output
search_class (cls)	Search for class `cls` in the codebase.	Signature of the searched class.
search_class_in_file (cls, f)	Search for class `cls` in file `f`.	Signature of the searched class.
search_method (m)	Search for method `m` in the codebase.	Implementation of the searched method.
search_method_in_class (m, cls)	Search for method `m` in class `cls`.	Implementation of the searched method.
search_method_in_file (m, f)	Search for method `m` in file `f`.	Implementation of the searched method.
search_code (c)	Search for code snippet `c` in the codebase.	+/- 3 lines of the searched snippet `c`.
search_code_in_file (c, f)	Search for code snippet `c` in file `f`.	+/- 3 lines of the searched snippet `c`.

signatures) and actual code snippets (e.g. method implementations). The context retrieval agent directs this navigation, and decides which retrieval APIs to use based on the current available context. In order to make the LLM-directed navigation more "controlled", we devise a stratified stategy of invoking the retrieval APIs (Section 4.3). This stratified strategy instructs the LLM to only invoke necessary retrieval APIs based on the available information, and iteratively changes the set of retrieval APIs used when more code-related context are returned from the previous API calls.

Once the context retrieval agent has gathered sufficient context about the issue, AutoCodeRover proceeds to the *patch generation* stage (Section 4.5). In this stage, AutoCodeRover employs another LLM agent to extract more precise code snippets from the retrieved context, and craft a patch based on the extracted code snippets. This patch generation agent is instructed to craft a patch in a specific format, and if the produced patch does not follow the format specification or cannot be applied to the original codebase, the agent enters a retry-loop which terminates after a pre-configured number of attempts. Finally, AutoCodeRover outputs a patch that attempts to resolve the original issue.

We note that the workflow of AutoCodeRover discussed so far only requires the problem statement P and codebase C as input, and do not require any program specifications such as testcases. However, when testcases are available (e.g. provided by developers or generated from another tool), execution-based information and program analysis techniques can be integrated into the AutoCodeRover framework (Section 4.4). For example, statistical fault localization [44] tool can be used to reveal more relevant methods beyond those mentioned in the problem statement. The additionally revealed code context can thereby influence the set of retrieval APIs invoked in the *context retrieval* stage. Moreover, with testcases available, the patch generation agent can employ an additional patch validation step in its retry-loop when crafting a patch. In the remainder of this section, we discuss in greater detail the components of AutoCodeRover and their design considerations.

4.2 Context Retrieval APIs

In the *context retrieval* stage, the basic components are a set of APIs which an LLM agent can use to gather relevant code context and snippets from the codebase. For typical software project issues, we observe that users often mention some "hints" on which part of the codebase is relevant. These hints can be the names of the relevant methods, classes, or files, and sometimes also contain short code snippets. Although these hints may not directly point to the precise location for code modification, they often reveal code context in the project that is relevant to the current issue. Figure 4 illustrates two real-world issues submitted to the sympy and django projects, respectively, and the "hints" are highlighted. In sympy-12481, the class `Permutation` is mentioned twice; in django-13230, multiple hints are mentioned, such as code snippet `feed.add_item()`, package path `syndication.views`, method name `item_extra_kwargs` and etc. Based on the type of project and code related hints, we design a set of APIs for an LLM agent to retrieve code context from these hints. The current set of APIs in AutoCodeRover and their outputs are shown in Table 1. Once invoked by the LLM agent, the retrieval APIs search for classes, methods and code snippets in the code-base, and return the results back to the agent. To avoid forming very lengthy code context that creates distraction during patch generation, we return only necessary information as API outputs. For example, since the complete definition of a class can be lengthy in large projects, we only return signature of the class as output for `search_class` and `search_class_in_file`. Returning the signature to shorten cntext, is a better approach than cutting off the context at a certain bound. Upon receiving the class signature, the agent could then invoke another API to search for the relevant methods / snippets inside the class.

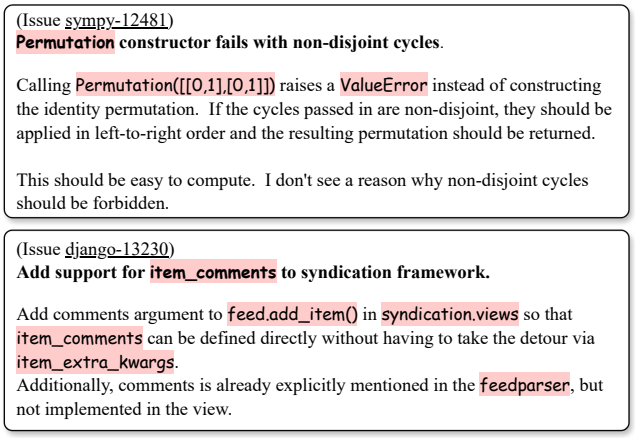

Figure 4: Issue description of sympy-12481 and django-13230. "Hints" are highlighted.

Interfacing with LLM. For the LLM agent to invoke the context retrieval APIs, the description and expected output of them are presented to it as part of prompt. When the agent decides to invoke

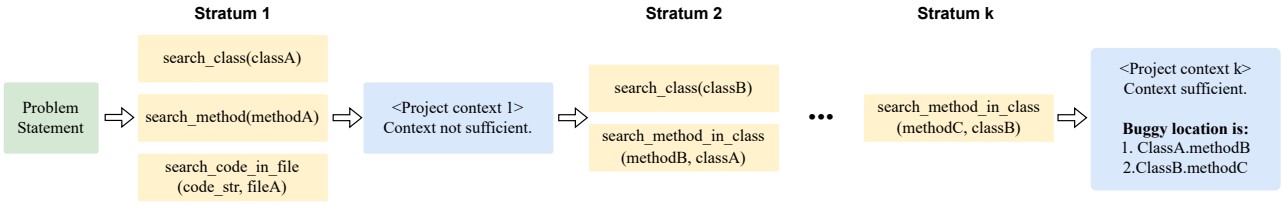

Figure 5: Stratified search with retrieval APIs for context gathering.

a set of retrieval APIs, it responds with the list of API call names and the corresponding arguments. These retrieval API requests are processed locally by parsing a local codebase of the project into AST and searching over it. Results of locally executing these APIs are returned to the agent, forming the code context.

4.3 Stratified Context Search

The set of context retrieval APIs listed in Table 1 serves as building blocks for searching relevant code context. With the context retrieval APIs and the LLM-identified keyword "hints" from the problem statement, a set of possible *API invocations* can be derived by using the identified keywords as API parameters. We discuss a few observations on using these API invocations to gather code context, and propose a *stratified* context retrieval process.

Our first observation is that the context retrieval should not be restricted to a single API invocation. For example, in the issue django-13230 mentioned in Figure 4, if the retrieval starts from the invocation `search_method("add_item")`, the implementation of method `add_item` can be considered by the LLM agent as a sufficient context, since it appears to be relevant to the problem statement. However, searching from only one method can lead to incomplete context for the agent to reason about root cause of the problem. On the other hand, if all API invocations are executed at once, a large code context can be retrieved, especially when the problem statement mentions many class and method names. This large code context can be difficult for an LLM to comprehend, or may even exceed its context window.

The second observation is that some of the API invocation results provide more elements to build new possible API invocations, which means the process of invoking retrieval APIs should be iterative. For example, the result of a `search_class` call returns the method signatures within the searched class, and an LLM can iteratively invoke method-related APIs afterwards.

With these two observations, we propose a *stratified* search process for invoking context retrieval APIs, as illustrated in Figure 5. From a problem statement, stratified search iteratively invokes retrieval APIs to gather project code context, and finally outputs a list of potentially buggy locations to be fixed. In each stratum, we prompt the LLM agent to select a set of *necessary* API invocations, based on the current context. In stratum 1, the current context only contains the problem statement; in the following strata, the context contains both the problem statement and the code searched so far. By allowing LLM to select more than one API invocations and also instructing it to only select the necessary ones, we make the best use of the context, building what we deem an *optimal* context.

After the API invocations in a stratum are executed, the newly retrieved code snippets are added to the current context. The LLM agent is then prompted to analyze whether the current context is sufficient for understanding the issue, thereby deciding whether (a) we continue the iterative search process, or (b) we decide on the buggy locations which will be considered for fixing.

4.4 Analysis-Augmented Context Retrieval

We also investigate how program debugging techniques can augment our workflow. Specifically, we integrate the *Spectrum-based Fault Localization* (SBFL) analysis into AutoCodeRover to study the effect of such a test-based dynamic analysis. We make a test-suite T available to AutoCodeRover, in addition to the problem statement P and codebase C.

Spectrum-based Fault Localization. The goal of SBFL is to identify the location of software faults [45]. Given a test-suite T containing both passing and failing tests, SBFL considers control-flow differences in the passing and failing test executions, and assigns a suspiciousness score to different program locations. This suspiciousness score can be computed with various metrics such as Tarantula [21] and Ochiai [3]. Program elements (e.g. statements/basic blocks) with the highest suspiciousness scores are identified as likely fault locations. SBFL can be performed at different granularities of program elements, such as statements or basic blocks. Since the LLM can process reasonably long code snippets, we use *method-level* SBFL in AutoCodeRover. Given an test-suite, SBFL can be used to directly output a few program locations to be repaired. However, the accuracy of SBFL highly relies on the quality of the test-suite [22] - since the SBFL results are effectively an abstraction of the differential of control flows between passing tests and failing tests. Therefore, instead of replacing the stratified context retrieval with SBFL, we use the SBFL identified methods to augment the search process. Before AutoCodeRover enters the context retrieval stage, we provide the SBFL-identified methods to the LLM agent as "results from an external analysis tool that identifies suspicous code". The main role of SBFL-identified methods is to reveal more "hints" on relevant classes and methods beyond those mentioned in the problem statement. The LLM agent can then use the context retrieval APIs to examine these methods. Since the SBFL-identified methods are presented to the agent together with the problem statements, the agent can then cross-reference between these two sources of information. For example, if one of the SBFL-identified method names is more closly related to the problem statement, the LLM is more likely to invoke the `search_method` API on this name. We will demonstrate this observation in Section 6.2.

4.5 Patch Generation

In the *patch generation* stage, AutoCodeRover employs a *patch generation agent* to use the collected code context to write a patch for the problem statement. This agent is given the problem statement, the identified buggy locations/methods, and the history of context retrieval, including the invoked APIs, the API results, as well as the previous analysis on the code context made by the context retrieval agent.

As a first step, the patch generation agent retrieves the precise code snippets at the buggy locations from the codebase. Based on the precise code snippets and other relevant code context, the agent enters a retry-loop of generating patches. If a generated patch does not follow the specified patch format or could not be applied syntactically to the original program, the agent is prompted to retry. We also employs a linter to indentify Python-specific syntax issues such as indentation errors in this retry-loop. The agent is allowed to retry up to a pre-configured attempt limits (currently set to three), after which the best patch so far is returned as output.

5 Experiment Setup

To evaluate the capabilities of AutoCodeRover in resolving real-life software issues, we answer the following research questions.

RQ1: To what extent can AutoCodeRover automate software issues like human developers?
RQ2: Can existing debugging / analysis techniques assist AutoCodeRover?
RQ3: What are the challenges for AutoCodeRover and fully automated program improvement in future?

Benchmark. We evaluate AutoCodeRover in recently proposed benchmarks SWE-bench and SWE-bench lite [17], comprising 2294 and 300 real-life GitHub issues, respectively. The only input is the natural language description in the original GitHub issue and its corresponding buggy codebase. Details of SWE-bench and SWE-bench lite appear in Section 2.2.

Baseline and Evaluation Metric. We selected two LLM-based agent systems Devin [24], Swe-agent [47] as baselines and compare their performance against AutoCodeRover. We directly compare AutoCodeRover with Swe-agent's original results as it is publicly available at GitHub[3]. In contrast to Swe-agent, we do not have access to Devin, so we take the most relevant reported result from their technical report [40]. To avoid the natural randomness of LLM, we repeat our experiments three times. We report the result with the AutoCodeRover @1 and AutoCodeRover @3 annotations (i.e. pass@1, pass@3 metrics [8] respectively). We use (1) the percentage of resolved instances, (2) average time cost, and (3) average token cost to evaluate the effectiveness of the tools. These evaluation metrics represent overall effectiveness, time efficiency, and economic efficacy in resolving real-world GitHub issues.

Implementation and Parameters. We use the state-of-the-art OpenAI GPT-4 (gpt-4-0125-preview) as foundation inference model for AutoCodeRover. In AutoCodeRover, the GPT-4 model is responsible for selecting search APIs to retrieve codebase context, refining the issue description, and writing a final patch. For parameters of GPT-4, we set a low temperature=0.2, max_tokens=1024 to produce relatively deterministic results and enable sufficient reasoning length for AutoCodeRover, and all other parameters remain as per default. AutoCodeRover terminates either a patch is generated or the context retrieval stage repeats ten times. Experiment results and artifacts for AutoCodeRover can be found at https://github.com/nus-apr/auto-code-rover.

System Environment. All experiments are conducted on an x86_64 Linux server with Ubuntu 20.04 installed. The correctness of generated patches by AutoCodeRover is evaluated on the official SWE-bench docker environment.

[3]https://github.com/princeton-nlp/SWE-agent

6 Evaluation

6.1 RQ1: Overall Effectiveness

We first measure the overall effectiveness of AutoCodeRover and baselines with the number of resolved task instances in SWE-bench. With the goal of understanding to what extent the current AI systems can automatically resolve real-life software issues, the only inputs we provided are the natural language issue description and a local code repository checked out at the erroneous version. We repeated the experiment of AutoCodeRover three times to avoid randomness and presented the results in pass@1 and pass@3 metrics [8], which are denoted as AutoCodeRover @1 and AutoCodeRover @3 respectively. When reporting time and token/-cost for AutoCodeRover @3, we report the time and cost required for running each task three times. Since Devin was evaluated on a random 25% subset of SWE-bench [24], we also report results of AutoCodeRover on this subset (we refer to this as "SWE-bench Devin subset"). Table 2 shows the overall result of AutoCodeRover in SWE-bench lite, full SWE-bench and SWE-bench Devin subset and Figure 7 shows a visual efficacy summary of AutoCodeRover's comparison with Swe-agent, Devin.

Results on SWE-bench. The results reported in Table 2 indicate that in the full SWE-bench, AutoCodeRover @1 resolved 12.42% task instances with 248 seconds per task, and AutoCodeRover @3 resolved 17.96% task instances taking a time of 701 seconds per task, which is at par with 12.47% resolved tasks by the concurrent work Swe-agent, according to their technical report [47]. We also performed another round of experiments with AutoCodeRover on SWE-bench lite (300 instances). AutoCodeRover @1 resolved 19% task instances in taking an average time of 195 seconds, whereas Swe-agent resolved 18% task instances. Moreover, we also investigated AutoCodeRover @3 in SWE-bench lite, in which the percentage of resolved task instances increased to 26%. The results of AutoCodeRover @3 in SWE-bench imply that AutoCodeRover may be complemented by running multiple times, highlighting the possibility of improving AutoCodeRover with an iterated generate-and-validate process in the future, if any program specification is available.

Comparison with Devin. To compare with Devin, we report AutoCodeRover's result on the 570 task instances (random 25% subset of SWE-bench) Devin [24] was evaluated on. The results of Devin

Table 2: Overall Result of AutoCodeRover and baselines on full SWE-bench, SWE-bench Devin subset, and SWE-bench lite. "-" indicates data not available.

Tools	Resolved Tasks	Avg Time	Avg Tokens
Reported result on SWE-bench lite (size=300)			
SWE-agent [47]	18.00% (54)	-	245k ($2.51)
AutoCodeRover @1	19.00% (57)	195	**37k ($0.43)**
AutoCodeRover @3	26.00% (78)	520	112k ($1.30)
ACR-sbfl	22.00% (66)	250	40k ($0.47)
Reported result on full SWE-bench (size=2294)			
SWE-agent [47]	12.47% (286)	-	240k ($2.46)
AutoCodeRover @1	12.42% (285)	248	**39k ($0.45)**
AutoCodeRover @3	17.96% (422)	701	120k ($1.39)
Reported result on SWE-bench Devin subset (size=570)			
SWE-agent [47]	13.51% (77)	-	234k ($2.40)
Devin [24, 40][4]	13.86% (79)	> 600	-
AutoCodeRover @1	12.63% (72)	238	**37k ($0.42)**
AutoCodeRover @3	18.77% (107)	692	117k ($1.36)

are taken from their technical report [40]. In this Devin particular subset, AutoCodeRover successfully resolved 12.63% task instances on average in pass@1, and 18.77% of the task instances in pass@3, which is higher than Devin. Besides, the time taken by AutoCodeRover is much smaller than Devin.

Detailed Comparison with SWE-agent. In the rest of RQ1, we compare AutoCodeRover with SWE-agent on SWE-bench lite using the highlighted AutoCodeRover @1 result. We further analyzed the commonly and uniquely resolved instances between AutoCodeRover and SWE-agent in Figure 6, and we found that AutoCodeRover and SWE-agent complement each other in different scenarios. AutoCodeRover uniquely resolved 26 task instances, which benefited from the fine-grained code context search at the AST level to precisely locate the bug locations (e.g., django-13401 searches three necessary methods at one time). On the other hand, the main reason that AutoCodeRover failed on the 23 unique instances resolved by SWE-agent is unimplemented search APIs (e.g., search_file invoked in django-12286). When such unimplemented APIs are invoked, AutoCodeRover returns an error message to the LLM, but the LLM may still fail to invoke valid search APIs in later attempts. This implies more robust search APIs are desired for future improvement of AutoCodeRover.

Time / Token Cost. Today, the price of invoking state-of-the-art LLMs such as GPT-4 and Claude-3-Opus are still very expensive. Hence, we are also interested in assessing the feasibility of deploying AutoCodeRover in the real world in terms of time and economic cost. Figure 7 shows the comparison between task-resolving success rate and average costs per task for all tools. On average, AutoCodeRover takes 195 seconds and 37k tokens (equivalent to 0.43 USD) to resolve one task instance in SWE-bench lite. In comparison, SWE-agent costs 245k tokens (equivalent to 2.51 USD) per task instance. The cost of Devin is empty because it is not

[4]The reported result of Devin is evaluated on a random 25% subset of full SWE-bench

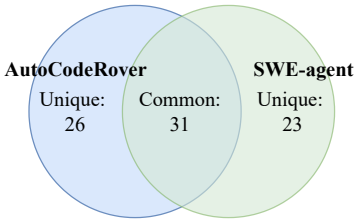

Figure 6: Venn diagrams of resolved tasks instances by AutoCodeRover and SWE-agent, on SWE-bench lite.

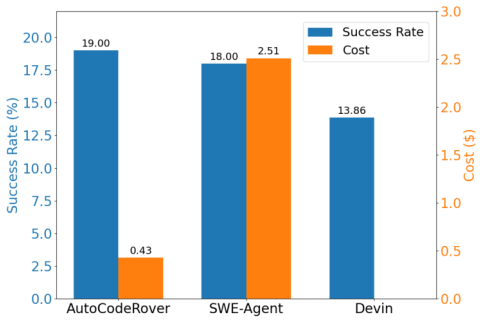

Figure 7: Task-resolving success rates (%) and average costs per task in USD of AutoCodeRover and baselines.

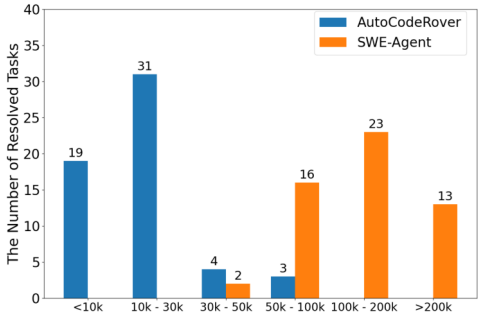

Figure 8: The number of resolved tasks and token cost distributions of AutoCodeRover and SWE-agent.

publicly available. When considering the combined three repetitions, AutoCodeRover takes 520 seconds (~8.67 minutes) per task, which is below the 30-60 minute time limit deemed acceptable by developers for automated repair tools [31]. Looking into the 78 issues resolved by AutoCodeRover @3 in SWE-bench lite, it costs on average ~2.68 days for developers to create pull requests for 66 issues, and the other 12 issues take even longer to be closed by developers (ranging from 34 - 4023 days). The short response time and significantly low success-rate/cost ratio show the potential for AutoCodeRover to act as a first affordable step in future autonomous bug fixing.

Token Distribution of Resolved Tasks. We dived into the details of token cost distribution among resolved task instances by AutoCodeRover and SWE-agent in Figure 8. The x-axis represents

token cost ranges and the y-axis represents the number of resolved tasks. Figure 8 shows that regarding the resolved tasks, AutoCodeRover is even more token-efficient. AutoCodeRover resolved 19 tasks with less than 10k tokens (~0.1 USD) and 31 tasks between 10k-30k tokens, whereas 66.7% of resolved tasks by SWE-agent at least requires more than 100k tokens. This implies the strong capability of stratified context retrieval in AutoCodeRover to fast pinpoint error locations and thus produce final patch.

Plausible / Correct patches. Overfitting is a well-known challenge in the Automated Program Repair community [12]. A program patch that passes the given test suite is said to be *plausible*. However, a plausible patch is deemed as *overfitting* if it fails to conform to the developer's intent. Otherwise, it is deemed as *correct*. To further understand the patch quality of AutoCodeRover and baselines, we manually verify the correctness of task-resolving (i.e. plausible) patches in SWE-bench lite. Since three repetitions are performed, we consider a task to have a *correct* patch if any of the three repetitions produced a *correct* patch. A plausible patch is correct if it is semantically equivalent to the developer patch. In this verification process, at least two authors of the paper cross-validated each patch, and any disagreement was resolved with another author. Overall, on SWE-bench lite, AutoCodeRover has a correctness rate of 65.4% (51 correct/78 plausible). We observed that the vast majority of AutoCodeRover's overfitting patches (all but 2 of the overfitting patches) modify the same methods as the developer patches, but the code modifications are wrong. This means that even the overfitting patches from AutoCodeRover are useful to the developer, since it helps in localization. The main causes of wrong modifications are the limits of the LLM's capability or insufficient context. Apart from these, we noticed two other interesting causes of overfitting. One cause is that the issue creator gives a preliminary patch in the description. This patch can be different from the final developer patch, misleading the LLM. The other interesting cause is that the issue creator mentioned a case that needs to be handled. The LLM only fixes this mentioned case, but the developer fixed other similar cases as well. The two causes indicate that the issue description, just like test suites, can be an incomplete specification.

6.2 RQ2: Effect of SBFL

Bug reproduction [6, 20] is a well-studied topic in the software engineering community. It aims to automatically construct input tests to reproduce bugs that are described in bug reports. Subsequently, those bug reproduction tests can be used by program analysis techniques to localize the root cause of bugs and generate patches for APR tools [30].

In this research question, we investigate whether program analysis techniques with reproducible tests can benefit the workflow of AutoCodeRover, and we use spectrum-based fault localization (SBFL) as an example. Different from RQ1, here we construct a common scenario in program repair that AutoCodeRover has access to the complete test-suite of the target task instance. We use the developer-written test cases for each task instance (provided in SWE-bench lite) as the test-suite. To understand the effect of SBFL, we additionally provide SBFL results (top-5 suspicious methods) to AutoCodeRover at the beginning of the context retrieval stage,

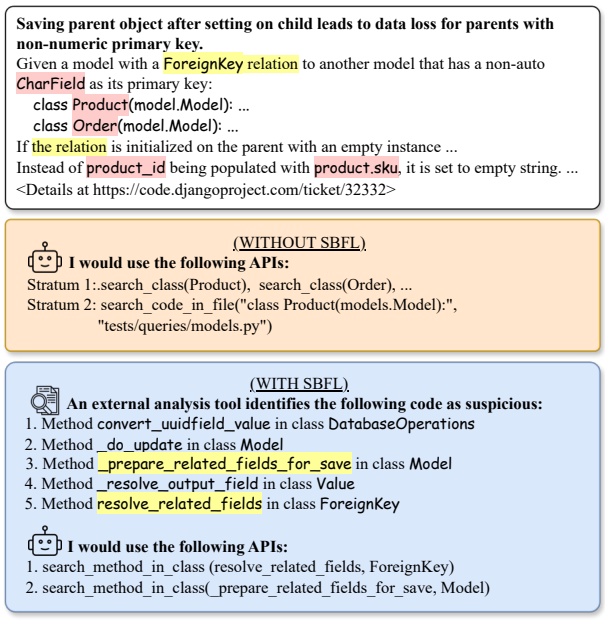

Figure 9: Issue description and AutoCodeRover's context retrieval (w. and w.o. SBFL), on django-13964.

then use the test-suite for patch validation during the patch generation retry-loop. We denote this setting as ACR-sbfl in Table 2. The patch validation is added as follows: when a patch is generated by the LLM agent, the test-suite is executed on the patched program; if the patch fails to pass the complete test-suite, AutoCodeRover re-invokes the patch generation agent to write a new patch. This validation loop is configured to run three times.

Results. Table 2 shows that, with the additional information provided by SBFL, the number of resolved tasks increased from 57 to 66 (i.e. 19% to 22% resolved rate on SWE-bench lite). Moreover, when comparing with task instances resolved in AutoCodeRover on SWE-bench lite, ACR-sbfl still uniquely resolves **7** task instances that are not resolved in any of the other runs.

Case study. We present a case study on one of the tasks uniquely resolved by ACR-sbfl. Figure 9 shows the issue description of django-13964[5] and how AutoCodeRover attempts to retrieve code context with and without SBFL. This issue reported a bug when saving django models to a database. A simplified version of the issue description is shown in the first part of Figure 9, in which important parts are highlighted. In this issue, some "hints" (highlighted in red) mentioned are actually distracting factors for a context retrieval agent. For example, the `Product` and `Order` classes describe how the bug can be reproduced, and are not classes that cause the bug. Retrieving code context from these two classes (as shown in Figure 9 part 2 "WITHOUT SBFL") does not yield useful results for forming the context and resolving the issue.

On the other hand, the SBFL component can provide extra hints for context retrieval agent (as shown in Figure 9 part 3 "WITH

[5] https://code.djangoproject.com/ticket/32332

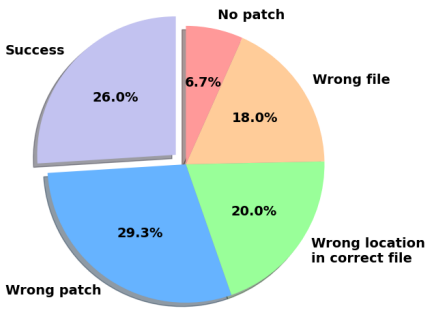

Figure 10: Taxonomy of Challenges in SWE-bench lite.

SBFL"). This is because SBFL considers test execution differences, which in this case revealed a few more methods in the codebase that are related to the issue. With these newly revealed hints, the agent decides to invoke APIs to search for the `resolve_related_fields` and `_prepare_related_fields_for_save` methods (the latter method is actually where the developer chose to fix this bug[6]). Moreover, we observe that the agent does not solely rely on the SBFL results to make API invocations. Instead of searching for methods ranked as top-1 in the SBFL results, the agent searched for the 3rd and 5th ranked methods. These methods are more related to some other hints mentioned in the issue description (highlighted in yellow), and the LLM agent is able to exploit this correlation between the natural language descriptions and the SBFL analysis results. With the correct context collected, AutoCodeRover is then able to draft a patch that resolves this issue. This suggests that an execution-based analysis can complement the agent workflow by revealing information not included in the issue description.

6.3 RQ3: Challenges on real-life tasks

In this research question, we analyze the task instances in SWE-bench lite that AutoCodeRover failed to resolve (based on the result of ACR-all in Table 2, without applying SBFL) and provide a taxonomy of the issue characteristics to highlight the practical challenges in achieving fully automated software improvement. Our taxonomy consists of challenges in the fault localization stage and patch generation stage. Specifically, for each task, we analyze the best run in the three repetitions, and classify each of the 300 tasks into one of the following:

- Success: The generated patch resolves the issue.
- Wrong patch: The generated patch modifies all methods that are modified in the developer patch. This means the patch content is wrong but the patch location(s) are correct.
- Wrong location in correct file: The generated patch that modifies the correct file but wrong location(s) in the file.
- Wrong file: The generated patch modifies the wrong file.
- No patch: No patch is generated from the retrieved context.

Figure 10 shows the distribution of the 300 tasks in SWE-bench lite. AutoCodeRover resolves 26.0% of the issues ("Success"), as mentioned in Section 6.1. The fail-to-resolve cases are included in the remaining four categories. In 29.3% of the tasks, AutoCodeRover correctly decided on all patch locations (at the method-level), but did not produce a correct patch ("Wrong patch"). More fine-grained intra-procedural analysis and specification inference techniques can play a significant role in improving these cases, by providing the patch generation agent with more method-level repair guidance. In the other three categories, the fault localization could not pinpoint all the locations to be modified. In 20.0% of the tasks, a patch is generated in the correct file, but at wrong methods / classes in the file ("Wrong location in correct file"). In some of these runs, the developer patch modifies multiple methods, but the generated patch did not modify all of them. In the other categories, a patch could not be generated at the correct file - in 18.0% of the tasks a patch is generated in wrong files, and in 6.7% of the tasks there is no applicable patch ("Wrong file" and "No patch"). We manually inspected some tasks in these two categories, and observed that their issue description mentions few methods / classes / files in the codebase. Instead, some of them contain short examples to reproduce the issue. For these tasks, one possibility is to generate a comprehensive test-suite based on the issue description, and then use execution information of the test-suite (e.g. SBFL) to reveal suspicious program locations. On the other hand, some other tasks do not contain reproducible examples and only consist of natural language descriptions. For these tasks some human involvement might be helpful. The developers could focus on these tasks.

7 Discussion on Experiments and Improvements

In this section, we discuss our position on the experiment results and a few possible directions for future improvements.

Position on Experiment Results. Building autonomous large language model agent systems for software engineering tasks is one of the fastest-growing research fields now. There were more than 17 attempts from both academia and industries on SWE-bench since April 2024. We refer the practitioners to the SWE-bench Leaderboard [16] (refer Figure 11) which maintains the latest research effort of various recent agents. The most recent resolve rate on full SWE-bench leaderboard has reached 19.27% by Factory Code Droid as of 12th July 2024. In the meantime, AutoCodeRover is also rapidly progressing towards better performance, the most recent update of AutoCodeRover (18.83% on SWE-bench on 12th July 2024) can also be found at [2]. However, despite the satisfying results of AutoCodeRover, it is useful to interpret the results in the right spirit. Apart from showing good efficacy with low cost, AutoCodeRover is significant because of the way it tries to generate program modifications. Thus, we highlight the importance of gleaning program specifications to guide the patching process by agents as a general guideline for future research.

Issue Reproducer. In Section 4, we described two scenarios that AutoCodeRover can work on, (1) issue description only, (2) aided by SBFL when the test suite is available. Although the second scenario is not always practical in real-life software development, the issue description sometimes contains a concrete reproduction script

[6] https://github.com/django/django/pull/13964/files

Leaderboard

Model	% Resolved	Date
Factory Code Droid	19.27	2024-06-17
AutoCodeRover (v20240620) + GPT 4o (2024-05-13)	18.83	2024-06-28
AppMap Navie + GPT 4o (2024-05-13)	14.60	2024-06-15
Amazon Q Developer Agent (v20240430-dev)	13.82	2024-05-09
SWE-agent + GPT 4 (1106)	12.47	2024-04-02
SWE-agent + Claude 3 Opus	10.51	2024-04-02
RAG + Claude 3 Opus	3.79	2024-04-02
RAG + Claude 2	1.96	2023-10-10
RAG + GPT 4 (1106)	1.31	2024-04-02
RAG + SWE-Llama 13B	0.70	2023-10-10
RAG + SWE-Llama 7B	0.70	2023-10-10
RAG + ChatGPT 3.5	0.17	2023-10-10

Figure 11: Snapshot of full SWE-bench leaderboard on 12th July 2024

from the user that reports the bug. In the future, it is possible to design an LLM agent specifically to generate a bug reproduction test based on the GitHub issue description. The bug reproduction test can then be used to validate the correctness of AutoCodeRover's generated patch and possibly enable a regeneration process if a patch fails to pass the reproduction test.

Semantic Artifacts. During context retrieval, AutoCodeRover effectively navigates through the codebase by visiting code entities such as classes and methods. The idea of utilizing code-specific structure for context retrieval can be taken further by considering artifacts from program semantics. For example, from an initial set of methods identified in the issue description, a static call graph analysis [36, 37] can be used to collect additional relevant methods when testcases are absent. There are also potential in integrating a language server [14] for codebase navigation, so that the context retrieval agent can perform more code-specific actions such as "jump-to-definition" from a method invocation. Finally, one can use forward data dependence analysis [19] from the methods in the issue description to search for other relevant methods.

Human Involvement. Currently, AutoCodeRover left all decision-making processes to its foundation LLM — e.g. deciding context retrieving locations and whether to terminate the retrieval stage. However, leaving it to the LLM may not always be enough. Note that it is common to have multiple rounds of discussion even between the project maintainers before a pull request is created. Hence, a flexible interface and human involvement criteria between human developers and LLM agents are desired.

8 Threats to Validity

Despite the high efficacy AutoCodeRover achieved in SWE-bench, we report a few potential threats to our approach and experiments and discuss how we addressed them. First, LLM is known for its randomness to generate different results in different runs, which may threaten the validity of AutoCodeRover's performance. We address this by repeating our main evaluation experiment three times and releasing a replication package for practitioners. Second, we consider a task instance in SWE-bench as resolved if the generated patch passes all the test cases added in the pull request for the issue. However, this patch may be overfitting. We addressed this threat by manually verifying whether the patches are semantically equivalent to developers' patches among the authors.

9 Perspectives

AI-based software engineering is currently a topic of study among researchers, innovators, and entrepreneurs. Part of the trigger for this interest has been provided by efforts like GitHub Copilot [1] in the last few years. These efforts show significant promise in the use of Large Language Models (LLMs) to automatically generate code. At the same time, code generated by LLMs may be incorrect [10] or vulnerable [34]. Thus, we need autonomous processes which allow for code improvements, such as bug fixes and feature additions.

In this paper, we suggest a LLM-based solution AutoCodeRover for autonomous code improvement. The key distinguishing feature of AutoCodeRover is its conscious attempt to inject a software engineering outlook by integrating (a) use of program representations such as ASTs instead of files, (b) iterative code search by exploiting program structure and (c) use of test-based fault localization when tests can be constructed. AutoCodeRover shows significant efficacy in terms of solving real-life GitHub issues. AutoCodeRover shows that relying only on the GitHub issue description to guide code search for patches / modifications can be misleading. From the point of view of program repair, the toughest challenge lies in inferring the intent of the developer or the specification. The outlook of AutoCodeRover is that the developer intent is gleaned from the project structure for automated program modifications.

Today, the code LLMs cannot produce safe and secure code which can be trusted enough to be integrated into real software projects. There is thus a need to autonomously improve code (both automatically generated and manually written), for which LLMs can play a role. Future work needs to focus further on the appropriate points when tools like AutoCodeRover may converse with the human programmer. Developers may need to shift to playing different roles at the same time in *future software industry* - vetting different conversations with LLM-based tools like AutoCodeRover to enable a variety of software engineering activities. This would contrast with *today's software industry* where a person has specific roles like programmer/tester/architect/requirements-engineer. Thus, apart from full-stack engineers, we may see more *full-lifecycle software engineers* in the future, who are comfortable working with the entire lifecycle of (the components of) a software system. Furthermore, the focus of LLM-oriented workflows is on *scale* today. This focus may shift to the engendering of *trust* if LLM-oriented autonomous software engineering becomes commonplace. Future software engineers may focus more on greater trust, instead of larger scale.

DATA AVAILABILITY

We share full public access to (1) AutoCodeRover's implementation, (2) all patches generated during our experiment and conversation history with LLM, (3) replication scripts for our experiments at the following website: https://github.com/nus-apr/auto-code-rover.

Acknowledgments

We thank the anonymous reviewers for their suggestions. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

References

[1] 2022. GitHub Copilot, your AI pair programmer. https://github.com/features/copilot/

[2] 2024. AutoCodeRover, Autonomous Software Engineering. Retrieved July 10, 2024 from https://autocoderover.dev/

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. IEEE, IEEE, 89–98. https://doi.org/10.1109/taic.part.2007.13

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[5] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021).

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[7] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013).

[8] 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. 2021. Evaluating Large Language Models Trained on Code. *arXiv.org* abs/2107.03374 (7 2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[9] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. Pyty: Repairing static type errors in python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[10] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models.. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, IEEE, 1469–1481. https://doi.org/10.1109/icse48619.2023.00128

[11] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947.

[12] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.

[13] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62 (11 2019), 56–65. Issue 12.

[14] Nadeeshaan Gunasinghe and Nipuna Marcus. 2021. *Language Server Protocol and Implementation*. Springer.

[15] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair., In 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023 (Melbourne, Victoria, Australia). *International Conference on Software Engineering*, 1430–1442. https://doi.org/10.1109/icse48619.2023.00125

[16] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Leaderboard results on SWE-bench. Retrieved April 8, 2024 from https://www.swebench.com/

[17] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[18] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[19] Wuxia Jin et al. 2024. PyAnalyzer: An Effective and Practical Approach for Dependency Extraction from Python Code. In *International Conference on Software Engineering (ICSE)*.

[20] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. 474–484. https://doi.org/10.1109/ICSE.2012.6227168

[21] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*. 467–477.

[22] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 114–125.

[23] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 314–325.

[24] Cognition Labs. 2024. Devin, AI software engineer. Retrieved April 12, 2024 from https://www.cognition-labs.com/introducing-devin

[25] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* abs/2203.07814, 6624 (12 2022), 1092–1097. https://doi.org/10.48550/arxiv.2203.07814

[26] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair., In ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). *International Symposium on Software Testing and Analysis*, 101–114.

[27] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis., In Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). *International Conference on Software Engineering*, 691–701.

[29] Manish Motwani and Yuriy Brun. 2023. Better automatic program repair by using bug reports and tests together. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1225–1237.

[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis., In 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013 (San Francisco, CA, USA), David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). *International Conference on Software Engineering*, 772–781. https://doi.org/10.1109/icse.2013.6606623

[31] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.

[32] Brayan Stiven Torrres Ovalle. 2023. GitHub Copilot. https://doi.org/10.26507/paper.2300

[33] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[34] H Pearce, B Tan, B Ahmad, R Karri, and B Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *IEEE Symposium on Security and Privacy (SP)*.

[35] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[36] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.

[37] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1646–1657.

[38] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair., In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). *ESEC/SIGSOFT FSE*, 532–543. http://people.cs.umass.edu/%7Ebrun/pubs/pubs/Smith15fse.pdf

[39] Shin Hwei Tan, Ziqiang Li, and Lu Yan. 2024. CrossFix: Resolution of GitHub issues via similar bugs recommendation. *Journal of Software: Evolution and Process* 36, 4 (2024), e2554.

[40] The Cognition Team. 2024. SWE-bench technical report (Devin). Retrieved April 12, 2024 from https://www.cognition-labs.com/post/swe-bench-technical-report

[41] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[42] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming., In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. *2009 IEEE 31st International Conference on Software Engineering*, 364–374. https://doi.org/10.1109/icse.2009.5070536

[43] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. 2023. User-Centric Deployment of Automated Program Repair at Bloomberg. *arXiv preprint arXiv:2311.10516* (2023).

[44] WE Wong, R Gao, Y Li, R Abreu, and F Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* (2016), 707–740. Issue 8.

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

[46] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[47] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE]

[48] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair., In ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). *ESEC/SIGSOFT FSE*, 341–353. https://arxiv.org/pdf/2106.08253

Received 2024-04-12; accepted 2024-07-03