RepairAgent: An Autonomous, LLM-Based Agent for Program Repair

Islem Bouzenia
University of Stuttgart
Germany
fi_bouzenia@esi.dz

Premkumar Devanbu
UC Davis
USA
ptdevanbu@ucdavis.edu

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

***Abstract*—Automated program repair has emerged as a powerful technique to mitigate the impact of software bugs on system reliability and user experience. This paper introduces RepairAgent, the first work to address the program repair challenge through an autonomous agent based on a large language model (LLM). Unlike existing deep learning-based approaches, which prompt a model with a fixed prompt or in a fixed feedback loop, our work treats the LLM as an agent capable of autonomously planning and executing actions to fix bugs by invoking suitable tools. RepairAgent freely interleaves gathering information about the bug, gathering repair ingredients, and validating fixes, while deciding which tools to invoke based on the gathered information and feedback from previous fix attempts. Key contributions that enable RepairAgent include a set of tools that are useful for program repair, a dynamically updated prompt format that allows the LLM to interact with these tools, and a finite state machine that guides the agent in invoking the tools. Our evaluation on the popular Defects4J dataset demonstrates RepairAgent's effectiveness in autonomously repairing 164 bugs, including 39 bugs not fixed by prior techniques. Interacting with the LLM imposes an average cost of 270k tokens per bug, which, under the current pricing of OpenAI's GPT-3.5 model, translates to 14 cents per bug. To the best of our knowledge, this work is the first to present an autonomous, LLM-based agent for program repair, paving the way for future agent-based techniques in software engineering.**

I. Introduction

Software bugs lead to system failures, security vulnerabilities, and compromised user experience. Fixing bugs is a critical task in software development, but if done manually, demands considerable time and effort. Automated program repair (APR) promises to dramatically reduce this effort by addressing the critical need for effective and efficient bug resolution in an automated manner. Researchers and practitioners have explored various approaches to address the challenge of automatically fixing bugs [1], including techniques based on manually designed [2], [3] and (semi-)automatically extracted [4], [5], [6] fix patterns, based on symbolic constraints [7], [8], [9], and various machine learning-based approaches [10], [11], [12], [13], [14], [15], [16].

The current state-of-the-art in APR predominantly revolves around large language models (LLMs). The first generation of LLM-based repair uses a one-time interaction with the model, where the model receives a prompt containing the buggy code and produces a fixed version [17], [18]. The second and current generation of LLM-based repair introduces iterative approaches, which query the LLM repeatedly based on feedback obtained from previous fix attempts [19], [20], [21].

A key limitation of current iterative, LLM-based repair techniques is that their hard-coded feedback loops do not allow the model to gather information about the bug or existing code that may provide repair ingredients. Instead, these approaches hard-code the code context that is provided in the prompt, typically to the buggy code [19], [21], and sometimes also details about the test cases that fail [20]. The feedback loop then executes the tests on different variants of the buggy code and adds any compilation errors, test failures, or other output, to the prompt of the next iteration. However, this approach fundamentally differs from the way human developers fix bugs, which typically involves a temporal interleaving of gathering information to understand the bug, searching code that may be helpful for fixing the bug, and experimenting with candidate fixes [22], [23].

This paper presents RepairAgent, the first autonomous, LLM-based agent for automated program repair. Our approach treats the LLM as an autonomous agent capable of planning and executing actions to achieve the goal of fixing a bug. We equip the LLM with a set of repair-specific tools that the models can invoke to interact with the code base in a way similar to a human developer. For example, RepairAgent has tools to extract information about the bug by reading specific lines of code, to gather repair ingredients by searching the code base, and to propose and validate fixes by applying a patch and executing test cases. Importantly, we do not hard-code how and when to use these tools, but instead let the LLM autonomously decide which tool to invoke next, based on previously gathered information and feedback from previous fix attempts.

Our approach is enabled by three key components. First, a general-purpose LLM, such as GPT-3.5, which

we query repeatedly with a dynamically updated prompt. We contribute a novel prompt format that guides the LLM through the bug repair process, and that gets updated based on the commands invoked by the LLM and the results of the previous command executions. Second, a set of tools that the LLM can invoke to interact with the code base. We present a set of 14 tools designed to cover different steps a human developer would take when fixing a bug, such as reading specific lines of code, searching the code base, and applying a patch. Third, a middleware that orchestrates the communication between the LLM and the tools. We present novel techniques for guiding tool invocations through a finite state machine and for heuristically interpreting possibly incorrect LLM outputs. The iterative loop of RepairAgent continues until the agent declares to have found a suitable fix, or until exhausting a budget of iterations.

To evaluate the effectiveness of our approach, we apply it to all 835 bugs in the Defects4J [24] dataset, a widely used benchmark for evaluating program repair techniques. RepairAgent successfully fixes 164 bugs, including 74 and 90 bugs of Defects4J v1.2 and v2.0, respectively. The correctly fixed bugs include 49 bugs that require fixing more than one line, showing that RepairAgent is capable of fixing complex bugs. Compared to state-of-the-art techniques [19], [21], RepairAgent successfully fixes 39 bugs not fixed by prior work. Measuring the costs imposed by interacting with the LLM, we find that RepairAgent imposes an average cost of 270k tokens per bug, which, under the current pricing of OpenAI's GPT-3.5 model, translates to 14 cents per bug. To account for potential data leakage from Defects4J into the LLM, an additional evaluation on the recent GitBug-Java dataset [25] shows that RepairAgent is able to achieve similar performance on single-line bugs while being slightly worse on multi-line and multi-file bugs due to a higher complexity of those bugs. Overall, our results show that our agent-based approach establishes a new state of the art in program repair.

In summary, this paper contributes the following:

- An autonomous, LLM-based agent for program repair.
- A dynamically updated prompt format that guides the LLM through the bug repair process.
- A set of tools that enable a LLM to to perform steps a human developer would take when fixing a bug.
- A middleware that guides via a finite state machine how the LLM interacts with the tools.
- Empirical evidence that RepairAgent establishes a new state of the art in program repair.
- We release our implementation as open-source: https://github.com/sola-st/RepairAgent.

To the best of our knowledge, there currently is no published work on an autonomous, LLM-based agent for any code-generation task. We envision RepairAgent to pave the way for future agent-based techniques in software engineering.

II. Background on LLM-Based, Autonomous Agents

By virtue of being trained on vast amounts of web knowledge, including natural language and source code, LLMs have demonstrated remarkable abilities in achieving human-level performance for various tasks [26]. A promising way of using these abilities are *LLM-based agents*, by which we mean LLM-based techniques with two properties: (1) The LLM autonomously plans and executes a sequence of actions to achieve a goal, as opposed to responding to a hard-coded query or being queried in a hard-coded algorithm. (2) The actions performed by the LLM include invocations of external tools that enable the LLM to interact with its environment [27], [28]. In the context of software engineering, and automated repair in particular, the tools could be tools usually used by developers, e.g., as part of an integrated development environment (IDE). The basic idea is to query the LLM with a prompt that contains the current state of the world, the goal to be achieved, and a set of actions that could be performed next. The model than decides which action to perform, and the feedback from performing the action is integrated into the next prompt. Recent surveys provide a comprehensive overview of LLM-based, autonomous agents [29] and of LLM agents equipped with tools invoked via APIs [30]. The potential of such agents for software engineering currently is not well explored, which this paper aims to address for the challenging task of automated program repair.

III. Approach

A. Overview

Figure 1 gives an overview of the approach of RepairAgent, which consists of three components: an LLM agent (left), a set of tools (right), and a middleware that orchestrates the communication between the two (middle). Given a bug to fix, the middleware initializes the LLM agent with a prompt that contains task information and instructions on how to perform it by using the provided tools (arrow 1). The LLM responds by suggesting a call to one of the available tools (arrow 2), which the middleware parses and then executes (arrow 3). The output of the tool (arrow 4) is then integrated into the prompt for the next invocation of the LLM, and the process continues iteratively until the bug is fixed or a predefined budget is exhausted.

B. Terminology

RepairAgent proceeds in multiple iterations, or cycles:

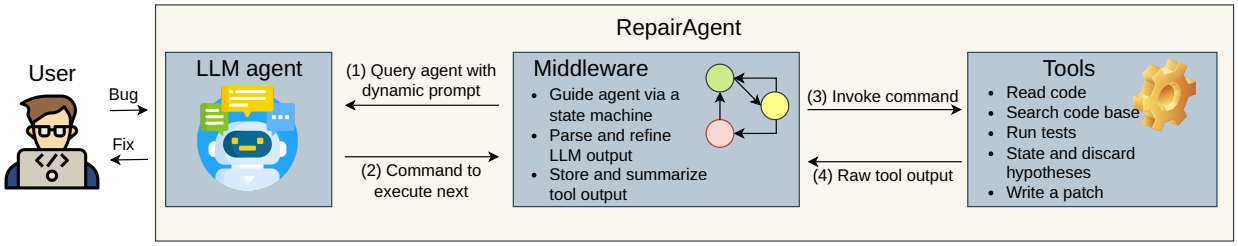


Fig. 1: Overview of RepairAgent.

TABLE I: Sections of the dynamically updated prompt.

Prompt section	Nature
Role	Static
Goals	Static
Guidelines	Static
State description	Dynamic
Available tools	Dynamic
Gathered information	Dynamic
Specification of output format	Static
Last executed command and result	Dynamic

Definition 1 (Cycle)**.** A *cycle* represents one round of interaction with the LLM agent, which consists of the following steps:

1) Query the agent
2) Post-process the response
3) Execute the command suggested by the agent
4) Update the dynamic prompt based on the command's output

In each cycle, the approach queries the LLM once. The input to the model is updated based on commands invoked by the LLM, and their results, in previous cycles. We call the model input a dynamic prompt:

Definition 2 (Dynamic prompt)**.** The *dynamic prompt* is a sequence of text sections $P = [s_0, s_1, ..., s_n]$, where each section s_i is one of the following (where $s_i(c)$ refers to a section during a cycle c):

- A *static section*, which remains the same across all cycles, i.e., $s_i(c) = s_i(c')$ for all c, c'.
- A *dynamic section*, which may differ across cycles, i.e., there may exist c, c' with $s_i(c) \neq s_i(c')$.

C. Dynamic Prompting of the RepairAgent

RepairAgent queries the LLM with a dynamic prompt that consists of a sequence of static and dynamic sections, as listed in Table I. We describe each section in detail in the following.

1) Role: This section of the prompt defines the agent's area of expertise, which is to resolve bugs in Java code, and outlines the agent's primary objective: understanding and fixing bugs. The prompt emphasizes that the agent's decision-making process is autonomous and should not rely on user assistance.

2) Goals: We define five goals for the agent to pursue, which remain the same across all cycles:

- *Locate the bug:* Execute tests and use fault localization techniques to pinpoint the bug's location. Skip this goal when fault localization information is already provided in the prompt.
- *Gather information about the bug:* Analyze the lines of code associated with the bug to understand the bug.
- *Suggest simple fixes to the bug:* Start by suggesting simple fixes.
- *Suggest complex fixes:* If simple fixes prove ineffective, explore and propose more complex ones.
- *Iterate over the previous goals:* Continue to gather information and to suggest fixes until finding a fix.

3) Guidelines: We provide a set of guidelines. First, we inform the LLM that there are diverse kinds of bugs, ranging from single-line issues to multi-line bugs that may entail changing, removing, or adding lines. Based on the observation that many bugs can be fixed by relatively simple, recurring fix patterns [31], we provide a list of recurring fix patterns. The list is based on the patterns described in prior work on single-statement bugs in Java [31]. For each pattern, we provide a short natural language description and an example of buggy and fixed code. Second, we instruct the model to insert comments above the modified code. This helps the model explain its reasoning, enhancing its abilities [32], and aids human developers in understanding the edits. Third, we instruct the model to conclude its reasoning with a clearly defined next step that can be translated into a call to a tool. Finally, we specify a limited budget of tool invocations (40 cycles by default), emphasizing the importance of efficiency.

4) State Description: To guide the LLM agent toward using the available tools in an effective and meaningful way, we define a finite state machine that constrains which tools are available at a given point in time. The motivation is that we observed, during earlier experiments without such guidance, the LLM agent to frequently get lost in aimless exploration. Figure 2 shows the finite state machine, which we design to mimic the states a human developer would go through when fixing a bug. Each state is associated with a set of

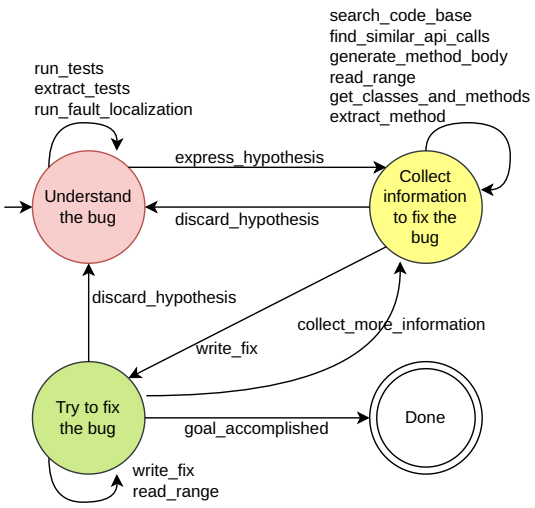


Fig. 2: State machine to guide selection of tools.

tools available to the agent, which are described in Section III-D. Importantly, the agent is free to transition between states at any point in time by using tools. That is, despite providing guidance, the state machine does not enforce a strict order of tool invocations.

The state description section of the prompt informs the agent about its current state:

- *Understand the bug*: The agent starts in this state, where it can collect information related to the failing test cases and the bug's location. Once the agent has an understanding of the bug, it formulates a hypothesis to describe the nature of the bug and the reason behind it. Throughout the repair process, the agent may refute earlier hypotheses and express new ones. After expressing a hypothesis, the agent will automatically switch to the next state.
- *Collect information to fix the bug*: In this state the agent collects information that help suggest a fix for the bug expressed by the hypothesis, e.g., by searching for specific repair ingredients or by reading possibly relevant code. Once the agent has gathered enough information to attempt a fix, it can transition to the next state.
- *Try to fix the bug*: In this state, the agent tries to fix the bug based on its current hypothesis and the collected information. Each fix attempt modifies the code base and is validated by executing the test cases. If necessarily, the agent can go back to one of the previous states to establish a new hypothesis or to gather additional information.

In addition to the three states, RepairAgent has a final state, *"Done"*, which the agent can reach by calling a command that indicates the repair is successful.

5) Available Tools: This section of the prompt describes a set of tools that the agent can call at the current state. Each tool has a name, a description, and a list of typed arguments (Section III-D).

```
interface Response {
  thoughts: string;
  command: {
    name: string;
    args: Record<string, any>;
  };
}
```

Fig. 3: JSON format of the expected response.

```
{
  "thoughts": "With the hypothesis in mind,
  it's time to gather more information to
  formulate potential fixes for the bug. I
  should focus on understanding the context
  around the condition
  'if (x < 0 && prev == '-')'
  to come up with effective solutions.",
  "command": {
    "name": "search_code_base",
    "args": {
      "key_words":
        [ "addNumber",
          "CodeConsumer",
          "if (x < 0 && prev == '-')" ]
}}}
```

Fig. 4: Example of a response from RepairAgent.

6) Gathered Information: A key ability of RepairAgent is to gather information about the bug and the code base, which serves as the basis for deciding which commands to invoke next. To make this information available to the agent, we maintain a prompt section that lists the information gathered by the different tool invocations. Intuitively, this section of the prompt serves as a memory for the agent, allowing it to recall information from previous cycles. The gathered information is structured into different subsections, where each subsection contains the outputs produced by a specific tool.

7) Specification of Output Format: Given the dynamic prompt, the LLM agent provides one response per cycle. To enable the middleware to parse the response, we specify the expected output format (Figure 3). The "thoughts" is for a textual description of the agent's reasoning when deciding about the next command. Asking the agent to express its thoughts increases the transparency and interpretability of the approach, provides a way to debug potential issues in the agent's decision-making process, and helps improve the reasoning abilities of LLMs [32]. The "command" field specifies the next command to be executed, consisting of the name and arguments of the tool to invoke.

For example, Figure 4 shows a response of the LLM agent. The model expresses the need to collect more information to understand the bug and suggests a command that searches the code base with a list of keywords.

8) Last Executed Command and Result: This section of the prompt contains the last command (tool name and arguments) that was executed (if any) and the output it produced. The rationale is to remind the agent of the last step it took, and to make it aware of any problems that occurred during the execution of the command. Furthermore, we remind the agent how many cycles have already been executed, and how many cycles are left.

D. Tools for the Agent to Use

A key novelty in our approach is to let an LLM agent autonomously decide which tools to invoke to fix a bug. The tools we provide to the agent (Table II) are inspired by the tools that developers use in their IDEs.

1) Reading and Extracting Code: A prerequisite for fixing a bug is to read and understand relevant parts of the code base. Instead of hard-coding the context provided to the LLM [19], [20], [21], we let the agent decide which parts of the code to read, based on four tools. The *read_range* tool allows the agent to extract a range of lines from a specific file, which is useful to obtain a focused view of a particular section of code. To obtain an overview of the code structure, the *get_classes_and_methods* tool retrieves all class and method names within a given file. By invoking the *extract_method* tool, the agent can retrieve the implementation(s) of methods that match a given method name within a given file. Finally, the *extract_tests* tool extracts the code of failing test cases, helping to understand details like input values and expected output.

2) Search and Generate Code: Motivated by the fact that human developers commonly search for code [33], we present tools that allow the agent to search for specific code snippets. These tools are useful for the agent to better understand the context of a bug and to gather repair ingredients, i.e., code fragments that could become part of a fix. The *search_code_base* tool enables the agent to locate instances of particular keywords within the entire code base. For example, the agent can use this tool to find occurrences of variables, methods, and classes. Given a set of keywords, the tool performs an approximate matching against all source code files in the project. Specifically, the tool splits each keyword into subtokens based on camel case, underscores, and periods, and then searches for each subtoken in the code. For example, searching for `quickSortArray` yields matches for `sortArray`, `quickSort`, `arrayQuickSort`, and other related variations. The output of the tool is a nested dictionary, organized by file names, classes, and method names, that provides the keywords that match a method's content. Another search tool, *find_similar_api_calls*, allows the agent to identify and extract usages of a method, which is useful to fix incorrect method calls. Without such a tool, LLMs tend to hallucinate method calls that do not exist in the code base [34]. Given a code snippet that contains a method call, the tool extracts the name of the called method, and then searches for calls to methods with the same name. The agent can restrict the search to a specific file or search the entire code base.

In addition to searching for existing code, RepairAgent offers a tool that generates new code by invoking another LLM. The tool is inspired by the success of LLM-based code completion tools, such as Copilot [35], which human developers increasingly use when fixing bugs. Given the code preceding a method and the signature of the method, the *generate_method_body* tool asks an LLM to generate the body of the method. The query to the code-generating LLM is independent of the dynamic prompt used by RepairAgent and may use a different model. In our evaluation, we use the same LLM for both the main agent and the code-generating LLM of this tool. The tool limits the given code context to 12k tokens and the generated code to 4k tokens.

3) Testing and Patching: The next category of tools is related to running tests and applying patches. The *run_tests* tool allows the agent to execute the test suite of the project. It produces a report that indicates whether the tests passed or failed. In case of test failures, the tool cleans the output of the test runner, e.g., by removing entries of the stack trace that are outside of the current project. The rationale is that LLMs have a limited prompt size and that irrelevant information may confuse the model. The *run_fault_localization* tool retrieves fault localization information, which is useful to understand which parts of the code are likely to contain the bug. RepairAgent offers two variants of this tool: Either, it provides perfect fault localization information or it invokes an existing fault localization tool, such as GZoltar [36], to calculate fault localization scores. In case of perfect fault localization, the tool provides all the file(s) and line(s) that need to be edited to fix the bug. As common in the field of program repair, we assume perfect fault localization as the default.

Once the agent has gathered sufficient information to fix the bug, it can apply a patch to the code base using the *write_fix* tool. RepairAgent aims at repairing arbitrarily complex bugs, including multi-line and even multi-file bugs. The *write_fix* tool expects a patch in a specific JSON format, which indicates the insertions, deletions, and modifications to be made in each file. Figure 5 shows an example of a patch in this format. Given a patch, the tool applies the changes to the code base and runs the test suite. If the tests fail, the *write_fix* reverts the changes, giving the agent a clean code base to try another fix. Motivated by the observation that some fix attempts are almost correct, the *write_fix* tool requests the LLM to sample multiple variants (default: 30) of the suggested

TABLE II: Repair-related tools invoked by RepairAgent.

Tool	Description
Read and extract code:	
read_range	Read a range of lines in a file.
get_classes_and_methods	Get the names of all classes and methods in a file.
extract_method	Given a method name, extract method implementations from a file.
extract_tests	Given the failure report from JUnit or ANT, extract the code of failing test cases.
Search and generate code:	
search_code_base	Scans all Java files within a project for a list of keywords.
find_similar_api _calls	Given a code snippet that calls an API, search for similar API calls in the project.
generate_method_body	Ask an LLM (GPT3.5 by default) to generate the body of a method based on code proceeding the method.
Testing and patching:	
run_tests	Run the test suite of a project.
run_fault_localization	Retrieve pre-existing localization information or run a fault localization tool.
write_fix	Apply a patch to the code base and execute the test suite of the project. Changes are reverted automatically if tests fail. Moves the agent into the "Try to fix the bug" state.
Control:	
express_hypothesis	Express a hypothesis about the bug. Moves the agent into the "Collect information to fix the bug" state.
collect_more_information	Move the agent back to the "Collect information to fix the bug" state.
discard_hypothesis	Discard the current hypothesis about the bug and move back to the "Understand the bug" state.
goal_accomplished	Declare that the goal has been accomplished and exiting the repair process.

```
[
  {
    "file_path": "jfree/data/time/Week.java",
    "insertions": [
      {
      "line_number": 175,
      "new_lines": [
      "// ...new lines to insert...\n",
      "// ...more new lines...\n"]
      }
    ],
    "deletions": [179, 183],
    "modifications": [
      {
      "line_number": 179,
      "modified_line": "    if (dataset == null
          ) {\n"
      }
    ]
  },
  {
    "file_path": "org/jfree/data/time/Day.java"
        ,
    "insertions": [],
    "deletions": [307],
    "modifications": []
  }
]
```

Fig. 5: Example of patch passed to the *write_fix* tool.

fix. Given the generated variants, the approach removes duplicates and launches tests for every variant.

4) Control: The final set of tools do not directly correspond to a tool a human developer may use, but rather allow the agent to move between states (Figure 2). The *express_hypothesis* tool empowers the agent to articulate a hypothesis regarding the nature of the bug and to transition to the "Collect information to fix the bug" state. Inversely, the *discard_hypothesis* tool allows the agent to discard a hypothesis that is no longer viable, which leads back to the "Understand the bug" state. Together, the two commands enforce a structured approach to hypothesis formulation, aligning with work on scientific debugging [37], [20]. In case the agent has tried multiple fixes without success, the *collect_more_information* tool allows the agent to revert to the "Collect information to fix the bug" state. Finally, once the agent has found at least one fix that passes all tests, it can invoke the *goal_accomplished* tool, which terminates RepairAgent.

E. Middleware

The middleware component plays a crucial role in RepairAgent, orchestrating the communication between the LLM agent and the tools. It performs the steps in Definition 1 as described in the following.

Parsing and Refining LLM Output: At the beginning of each cycle, the middleware queries the LLM with the current prompt. Ideally, the response adheres perfectly to the expected format (Figure 3). In practice, the LLM may produce responses that deviate from the expected format, e.g., due to hallucinations or syntactic errors. For example, the LLM may provide a "path" argument while the tool expects a "file_path" argument.

RepairAgent tries to heuristically rectify such issues by mapping the output to the expected format in three steps. First, it tries to map the tool mentioned in the response to one of the available tools. Specifically, the approach checks if the predicted tool name $n_{predicted}$ is a substring of the name of any available tool n_{actual}, or vice versa, and if yes, considers n_{actual} to be the desired tool. In case the above matching fails, the

approach checks if the Levenshtein distance between $n_{predicted}$ and any n_{actual} is below a threshold (0.1 by default). Second, the approach tries to map the argument names provided in the response to the tool's arguments, following the same logic as above. Third, the approach handles invalid argument values by heuristically mapping or replacing them, e.g., by replacing a predicted file path with a valid one. If the heuristics fail or produce multiple possible tool invocations, the middleware informs the LLM about the issue via the "Last executed command and result" prompt section and enters a new cycle.

In addition to rectifying minor mistakes in the response, the middleware also checks for repeated invocations of the same tool with the same arguments. If the agent suggests the exact same command as in a previous cycle, the middleware informs the agent about the repetition and enters a new cycle.

Calling the Tool: Given a valid command from the LLM, the middleware calls the corresponding tool. To prevent tool executions to interfere with the host environment or RepairAgent itself, the middleware executes the command in an isolated environment.

Updating the Prompt: Given the output of the tool, the middleware updates all dynamic sections of the prompt for the next cycle. In particular, it updates the state description and the available tools, appends the tool's output to the gathered information, and replaces the section that shows the last executed command.

IV. Implementation

We use Python 3.10 as our primary programming language. Docker is used to containerize and isolate command executions for enhanced reliability and reproducibility. RepairAgent is built on top of the AutoGPT framework and GPT-3.5-0125 from OpenAI. To parse and interact with Java code, we use ANTLR.

V. Evaluation

To evaluate our approach we aim to answer the following research questions:

- RQ1 How effective is RepairAgent at fixing real-world bugs?
- RQ2 What are the costs of the approach?
- RQ3 What is the influence and importance of the different components of RepairAgent?
- RQ4 How does the agent use the available tools?

A. Experimental Setup

a) Datasets: We apply RepairAgent to all bugs in the Defects4J dataset [24], which consists of 835 real-world bugs from 17 Java projects, including 395 bugs from 6 projects in Defects4Jv1.2, as well as another 440 bugs and 11 projects added in Defects4Jv2. Evaluating on the entire dataset allows us to assess the generalization capabilities of RepairAgent to different projects and bugs, without restricting the evaluation, e.g., based on the number of lines, hunks, or files that need to be fixed.

To assess the generalizability of our results and the potential influence of data leakage, we also evaluate RepairAgent on bugs from a newer dataset, GitBug-Java [25], focusing on bugs discovered and fixed after the cut-off date (January 2022) of the GPT 3.5 version we use. GitBug-Java contains 199 bugs from 55 projects. Due to budget constraints, we randomly sample 100 of these bugs, sampling at least one and at most two bugs per project. The random sample consists of 19 single-line, 64 multi-line, and 17 multi-file bugs.

b) Baselines: We compare with three existing repair techniques: ChatRepair [19], ITER [21], and SelfAPR [38]. ChatRepair and ITER are two very recent approaches and have been shown to be the current state of the art. All three baseline approaches follow an iterative approach that incorporates feedback from previous patch attempts. Unlike RepairAgent, the baselines do not use an autonomous, LLM-based agent. We compare against the baselines based on patches provided by the authors of the respective approaches.

c) Metrics: Similar to past work, we report both the number of plausible and correct patches. A fix is *plausible* if it passes all test cases, but is not necessarily correct. To determine whether a fix is correct, we automatically check whether it syntactically matches the developer-created fix. If not, we manually determine whether the RepairAgent-generated fix is semantically consistent with the developer-created fix. If and only if either of the two checks succeeds, we consider the fix to be *correct*.

B. RQ1: Effectiveness

1) Overall Results: Table III summarizes the effectiveness of RepairAgent in fixing the 835 bugs in Defects4J. The approach generates plausible fixes for 186 bugs. While not necessarily correct, plausible fixes pass all test cases and may still provide developers a hint about what should be changed. RepairAgent generates correct fixes for 164 bugs, where 116 are exactly as fixed by the developers and 48 are semantically consistent with the developer-provided patches. Being able to fix bugs from different projects shows that the approach can generalize to code bases of multiple domains. Furthermore, RepairAgent creates fixes for bugs of different levels of complexity. Specifically, as shown in Table IV, the approach fixes 115 single-line bugs, 46 multi-line (single-file) bugs, and 3 multi-file bugs.

2) Comparison with Prior Work: The right-hand side of Table III compares RepairAgent with the baseline approaches ChatRepair, ITER, and SelfAPR. Previous to this work, ChatRepair had established a new state

TABLE III: Results on Defects4J.

Project	Bugs	Plausible	Correct	ChatRepair	ITER	SelfAPR
Chart	26	14	11	15	10	7
Cli	39	9	8	5	6	8
Closure	174	27	27	37	18	20
Codec	18	10	9	8	3	8
Collections	4	1	1	0	0	1
Compress	47	10	10	2	4	7
Csv	16	6	6	3	2	1
Gson	18	3	3	3	0	1
JacksonCore	26	5	5	3	3	3
Jacksondatabind	112	18	11	9	0	8
JacksonXml	6	1	1	1	0	1
Jsoup	93	18	18	14	0	6
JxPath	22	0	0	0	0	1
Lang	63	17	17	21	0	10
Math	106	29	29	32	0	22
Mockito	38	6	6	6	0	3
Time	26	3	2	3	2	3
Defects4Jv1.2	395	96	74	114	57	64
Defects4Jv2	440	90	90	48	—	46
Total	835	186	164	162	57	110

TABLE IV: Distribution of fixes by location type

Bug type	RepairAgent	ChatRepair	ITER	SelfAPR
Single-line	115	133	36	83
Multi-line*	46	29	14	24
Multi-file	3	0	4	3

of the art in APR by fixing 162 bugs in Defects4J. RepairAgent achieves a comparable record by fixing a total of 164 bugs. Our work particularly excels in Defects4Jv2, where RepairAgent fixes 90 bugs, while ChatRepair only fixes 48 bugs. To further compare the sets of fixed bugs, Figure 6 shows the overlaps between different approaches. As often observed in the field of APR, different approaches complement each other to some extent. In particular, RepairAgent fixes 39 bugs that were not fixed by any of the three baselines. Out of these 39 bugs, 18 are single-line, 20 are multi-line, and one is a multi-file bug. Comparing the complexity of the bug fixes, as shown on the right-hand side of Table IV, RepairAgent particularly outperforms other tools for bugs that require more than a single-line fix. We attribute this result to the RepairAgent's ability to autonomously retrieve suitable repair ingredients and the ability to edit an arbitrary number of lines and files.

3) Examples: Figure 7 is a bug fixed exclusively by RepairAgent, where the agent uses the *find_similar_api_calls* tool to search for calls similar to `cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);`. It returns a call from another file, which passes `Branch.ON_EX` to the method call instead of `Branch.UNCOND`. This field name is then used as a repair ingredient by the agent. In another example fixed only by RepairAgent, Figure 8, the approach benefits from the *generate_method_body* tool to generate a missing if-statement, which led to suggesting a correct

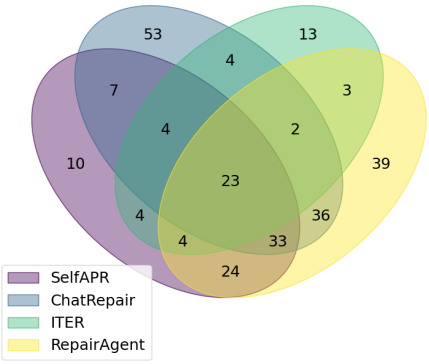


Fig. 6: Intersection of the fixes set with related work.

```
  if (cfa != null) {
    for (Node finallyNode :
        cfa.finallyMap.get(parent)) {
-     cfa.createEdge(fromNode, Branch.UNCOND,
    finallyNode);
+     cfa.createEdge(fromNode, Branch.ON_EX,
    finallyNode);
  }}
```

Fig. 7: Closure-14, bug fixed by RepairAgent.

fix afterwards. These examples illustrate the clever and proper usage of available tools by the agent. They also show these tools to be useful for finding repair ingredients that previous work fails to consider.

4) Generalization and External Validity: To assess the generalization capabilities of RepairAgent, we evaluate the approach on GitBug-Java, with results shown in Table V. Overall, RepairAgent finds 19 plausible fixes and 13 correct fixes. The table shows that the approach is particularly effective for single-line bugs, where it correctly fixes 9 out of 19 bugs. In contrast, the approach struggles with the 81 multi-line and multi-file bugs, where it finds only 4 correct fixes. This result can at least partially be attributed to the fact that the GitBug-Java dataset contains more complex bugs than Defects4J. The mean number of added and removed lines per ground truth bug fix are 2.9 and 9.3, respectively, for Defects4J, but 6.2 and 14.4 for GitBug-Java. Likewise, the mean number of modified tokens is 381 for Defects4J, but 577 for GitBug-Java. We conclude that RepairAgent generalizes well to new projects and bugs, and is not strongly affected by potential data leakage (e.g, Defects4J).

C. RQ2: Costs of the Approach

We measure three kinds of costs imposed by RepairAgent: (i) time taken to fix a bug; (ii) number of tokens consumed by queries to the LLM; (iii) monetary costs associated with the token consumption, based on OpenAI's pricing as of March 2024.

Our findings are summarized in Figure 9. The median time taken to address a bug is 920 seconds, with minimal variation between fixed and unfixed bugs. Surprisingly,

```
  Separator sep = (Separator)
      elementPairs.get(0);
+ if (sep.iAfterParser == null &&
    sep.iAfterPrinter == null) {
    PeriodFormatter f =
        toFormatter(elementPairs.subList(2,
        size), notPrinter, notParser);
    sep = sep.finish(f.getPrinter(),
        f.getParser());
    return new PeriodFormatter(sep, sep);
+ }
```

Fig. 8: Time-27, bug fixed by RepairAgent.

TABLE V: Results on GitBug-Java

Bug type	Bugs	Plausible fixes	Correct fixes
Single-line	19	11	9
Multi-line	64	8	4
Multi-file	17	0	0
Total	100	19	13

fixed bugs do not consistently exhibit lower repair times. This is due to RepairAgent's autonomous nature, where the repair process continues until the *goal_accomplished* command is invoked or the cycles budget is exhausted. The figure shows several outliers where bug fixing attempt takes multiple hours. RepairAgent spends 99% of the total time in tool executions, mostly running tests.

Analyzing the costs imposed by the LLM, we find an average consumption of approximately 270k tokens, equating to around 14 cents (US dollars). The number of tokens consumed by fixed bugs is lower than unfixed bugs. This difference is because the agent continues to extract additional information for not yet fixed bugs, saturating the prompt with operations, such as reading more lines of code or performing extensive searches.

Comparison to prior work: We compare the time and monetary costs of RepairAgent with other work based on what is reported in the respective papers. The monetary costs of ChatRepair [19] is reported as 42 cents per bug, based on the same model (GPT-3.5) as in our work. Adjusting for the change in pricing between the two evaluations, the costs of ChatRepair are about the same as for RepairAgent. The monetary costs of ITER [21] and SelfAPR [38] are not reported, as these approaches use self-trained models. However, the authors of ITER report a median bug fixing time of 4.57 hours per bug, which is much higher than the median time of 920 seconds for RepairAgent. While the comparison may be biased due to different hardware and software configurations, it suggests that RepairAgent is more efficient in terms of time costs. We mainly attribute this difference to the number of patches that need to be validated (e.g., average of 117 patches generated by RepairAgent vs. 1,000 patches generated by ITER).

TABLE VI: Different configurations of RepairAgent.

Configuration	Plausible	Correct	Cost($)	SL	ML	MF
No search tools	14	11	0.28	11	0	0
No state machine	18	14	0.31	9	5	0
Single-cycle memory	9	6	0.08	5	1	0
Realistic localization	16	16	0.29	14	2	0
RepairAgent (default)	23	21	16	16	5	0

D. RQ3: Ablation Study

To better understand the impact of different components and configurations of RepairAgent, we perform the ablation studies summarized in Table VI. Due to budget limitations, the ablations are done on a randomly selected set of 100 bugs of the entire Defects4J (same 100 for all configurations), out of which the full RepairAgent approach fixes 21 bugs. We report the number of plausible and correct patches, costs in US dollars, and a breakdown of correct fixes into single-line (SL), multi-line (ML), and multi-file (MF) bugs.

Importance of search tools: Without the search tools, RepairAgent fixes half of the bugs fixed by default. The absence of search tools also causes the agent to read long sequences of code more frequently, which saturates the prompt quickly and doubles the costs.

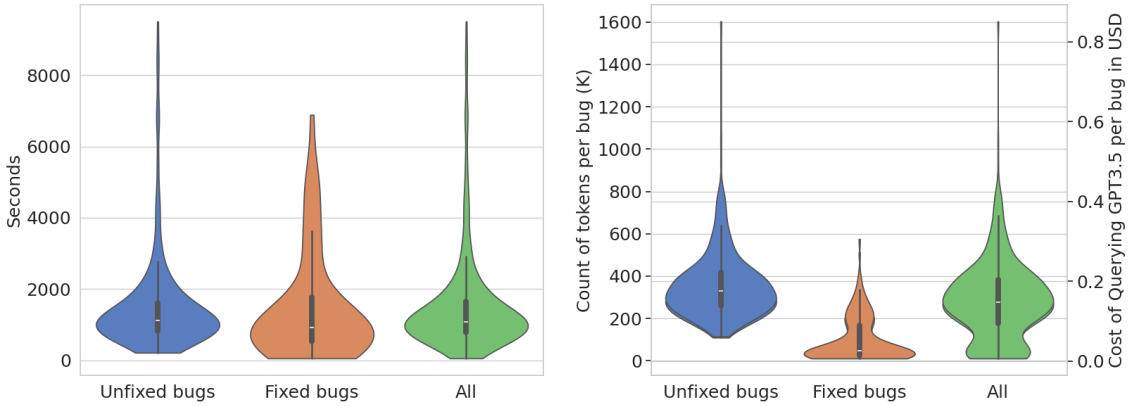
Importance of state machine: Without guidance by the state machine (Figure 2), the agent also fixes fewer bugs and has higher costs. The main reason for the reduced effectiveness is that the agent does not follow a structured approach to fixing the bug. For example, in many cases, the agent directly starts with suggesting a fix (often wrong) without collecting any information.

Importance of long-term memory: The third row of the table shows a variant of RepairAgent that keeps new information for a single cycle only, instead of accumulating all gathered information. Again, the bug fixing effectiveness suffers significantly. The reasons are that the agent repeats the same commands after a few cycles (e.g., to ask for the same information again), and it uses wrong file names and functions names. Having a long-term memory helps the agent to keep useful information for future cycles, avoiding repeated queries.

Impact of fault localization: Finally, we evaluate RepairAgent with realistic fault localization, based on the spectrum-based GZoltar technique [36]. In total, RepairAgent fixes 16 bugs for 29 dollars which is a 25% drop in fixing capability and a 81% increase in costs. These results were achieved without giving RepairAgent more cycles which would have helped otherwise since the agent spends extra time on localizing the bug.

E. RQ4: Usage of Tools by the Agent

This research question aims at better understanding the approach by analyzing how the agent uses the available tools. On average, RepairAgent makes 35 tool



(a) Time. (b) Tokens/money consumption.

Fig. 9: Distribution of cost metrics per bug (time, number of token, and monetary costs).

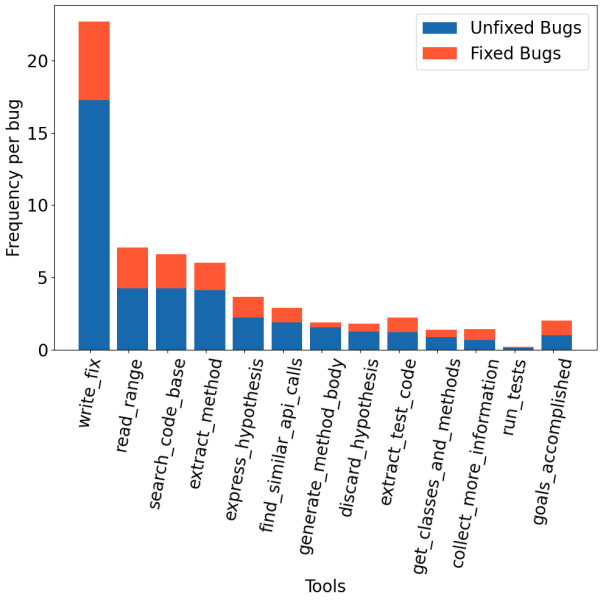


Fig. 10: Frequency of tool invocations (average per bug).

invocations per bug, which also corresponds to the number of cycles. Figure 10 shows the frequency of tool invocations, where we distinguish between fixed (i.e., "correct") and unfixed (i.e., "plausible" only or completely unfixed) bugs. The agent uses the full range of tools, with the most frequently called tool being *write_fix* (average of 6 calls for fixed bugs and 17 calls for unfixed bugs). Around 7% of *write_fix* invocations in unfixed bugs produce plausible patches, compared to 44% in fixed bugs. The least used tool is *run_tests*, which is used so infrequently because the initially provided information about the bug already provides information about any failing test cases and because the *write_fix* tool automatically invokes the test suite.

VI. Discussion

A. Qualitative Insights

The following describes qualitative insights gained from inspecting RepairAgent's logs.

Understanding the bugs: RepairAgent's ability to actively retrieve information that helps understand a bug allowed to fix a new set of bugs without higher costs. Particularly, we observe four kinds of information to be useful: (i) the code of failing test cases and the initial execution results, which we provide in the prompt of the first cycle; (ii) code snippets retrieved by searching for similar code, e.g., using the *find_similar_api_calls* tool; (iii) details about the code structure, such as the classes and methods in a file; and (iv) feedback obtained by applying a fix, which triggers the test execution and reveals any test cases that still fail.

Unfixed bugs and fix complexity: As shown in Table IV, RepairAgent clearly outperforms prior work on multi-line bugs, but fails to fix some of the simpler, single-line bugs fixed by, e.g., ChatRepair [19]. We observe that the agent sometimes suggests complex fixes for bugs that only require simple modifications. A possible remedy could be to initially limit the complexity of candidate fixes, nudging the agent toward trying simple fixes first. For multi-line, multi-file bugs, we observe that RepairAgent often edits only a subset of the required locations. Future work could explore human-in-the-loop approaches, where a partial fix found by an agent could give a developer a head start.

B. Threats to Validity and Limitations

While RepairAgent shows promising results, we acknowledge several potential threats to validity and inherent limitations: *(i) Data leakage:* GPT-3.5 may have seen parts of the Java projects we evaluate on during training. Our closest competitor, ChatRepair, also uses GPT-3.5, and thus faces the same risk. Moreover, the experiment

on GitBug-Java suggests that RepairAgent is effective also on bugs guaranteed to not be part of the training data. *(ii) Missing test cases:* Defects4J has at least one failing test case for each bug, which may not be the case for real-world usage scenarios. It will be interesting to evaluate RepairAgent on bugs with no a-priori available error-revealing test cases in future work. *(iii) Fault localization:* Inaccurate or imprecise fault localization could lead to suboptimal repair suggestions or incorrect diagnoses. *(iv) Non-deterministic output of LLMs:* The inherently non-deterministic nature of LLMs may result in different outcomes between two consecutive runs of RepairAgent. The large number of bugs we evaluate on mitigates this risk. Moreover, the logs of interactions with the LLM are available for further analysis.

VII. Related Work

a) Non-learning-based program repair: Automated program repair [1] has received significant attention. Some approaches address it as a search problem based on manually designed code mutation rules and fix patterns [2], [39], [3]. Alternatively, transformation rules can be derived (semi-)automatically from human-written patches [4], [5], [6]. Other approaches use symbolic constraints to derive fixes [7], [40], [8], [9], integrate repair into a static analysis that identifies bugs [41], [42], [43], or replace buggy code with similar code from the same project [44]. APR has been successfully deployed in industrial contexts [5], [45]. Beyond functional bugs, several techniques target other kinds of problems, such as syntax errors [46], [47], [48], performance bugs [49], vulnerabilities [50], type errors [51], common issues in deep learning code [52], and build errors [53].

b) Learning-based program repair: While early work uses machine learning to rank and select candidate fixes [10], more recent work uses machine learning to generate fixes. Approaches include neural machine translation models that map buggy code into fixed code [11], [12], [13], [14], models that predict tree transformations [15], [16], neural architectures for specific kinds of bugs [54], and repair-specific training regimes [55], [38]. We refer to a recent survey for a more comprehensive discussion [56]. Unlike the above work, RepairAgent and the work discussed below use a general-purpose LLM, instead of training a task-specific model.

LLMs have motivated researchers to apply them to program repair, e.g., in studies that explore prompts [18], [17] and in a technique that prompts the model with error messages [57]. These approaches perform a one-time interaction with the model, where the model receives a prompt with code and produces a fix. The most recent techniques introduce iterative approaches, which query the LLM repeatedly based on feedback obtained from previous fix attempts [19], [20], [21], [58]. RepairAgent also queries the model multiple times, but fundamentally differs by pursuing an agent-based approach. Section V empirically compares RepairAgent to the most closely related iterative approaches [19], [21].

c) LLMs for code generation and code editing: Beyond program repair, LLMs have been applied to a variety of other code generation and code editing tasks, including code completion [35], [59], fuzzing [60], generating and improving unit tests [61], [62], [63], [64], [65], [66], multi-step code editing [67]. Unlike our work, none of these approaches is agent-based.

d) LLM-based agents: The idea to let LLM agents autonomously plan and perform complex tasks is relatively new and has been applied to tasks outside of software engineering [29]. To the best of our knowledge, our work is the first to apply an LLM-based agent to program repair or any other code generation problem in software engineering. Copra is an agent-based approach for formal theorem proving [68]. After an initial version of this paper was made publicly available, other LLM-based agents for software engineering tasks have been proposed [69], [70], showing the potential of this kind of approach. RepairAgent is inspired by prior work [30] on augmenting LLMs with tools invoked via APIs [27], [28] and with the ability to generate and execute code [71]. Our key contribution in applying these ideas to a software engineering task is to define tools that are useful for program repair and a prompt format that allows the LLM to interact with these tools.

VIII. Conclusion

This paper presents a pioneering technique for bug repair based on an autonomous agent powered by Large Language Models (LLMs). Through extensive experimentation, we validate the effectiveness and potential of our approach. Further exploration and refinement of autonomous agent-based techniques will help generalize to more difficult and diverse types of bugs if equipped with the right tools.

Data availability

Our code and data are available at:
https://github.com/sola-st/RepairAgent
Alternatively: https://doi.org/10.5281/zenodo.14470521

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreements 851895 and 101155832), and by the German Research Foundation within the ConcSys, DeMoCo, and QPTest projects.

REFERENCES

[1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[3] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *ISSTA*. ACM, 2019, pp. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[4] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches." in *International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[5] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, 2019. [Online]. Available: https://doi.org/10.1145/3360585

[6] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: automated data-driven synthesis of repairs for static analysis violations," in *ESEC/FSE*, 2019, pp. 613–624. [Online]. Available: https://doi.org/10.1145/3338906.3338952

[7] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 772–781.

[8] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[10] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[11] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *AAAI*, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[12] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *ICSE*, 2019, pp. 25–36. [Online]. Available: https://dl.acm.org/citation.cfm?id=3339509

[13] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *ISSTA*. ACM, 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[14] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1943–1959, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2940179

[15] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *ICSE*, 2020.

[16] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/FSE*. ACM, 2021, pp. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[17] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1482–1494.

[18] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *ICSE*, 2023, pp. 1430–1442. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00125

[19] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," 2023.

[20] S. Kang, B. Chen, S. Yoo, and J. Lou, "Explainable automated debugging via large language model-driven scientific debugging," *CoRR*, vol. abs/2304.02195, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.02195

[21] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *ICSE*, 2024.

[22] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[23] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *ESEC/FSE*, 2017, pp. 117–128.

[24] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[25] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *Proceedings of the 21st International Conference on Mining Software Repositories*.

[26] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. M. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with GPT-4," *CoRR*, vol. abs/2303.12712, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.12712

[27] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *CoRR*, vol. abs/2302.04761, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.04761

[28] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *CoRR*, vol. abs/2305.15334, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.15334

[29] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J.-R. Wen, "A survey on large language model based autonomous agents," 2023.

[30] G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz, E. Grave, Y. LeCun, and T. Scialom, "Augmented language models: a survey," *CoRR*, vol. abs/2302.07842, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.07842

[31] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur?" Jun. 2020. [Online]. Available: http://dx.doi.org/10.1145/3379597.3387491

[32] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[33] L. D. Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *ACM Comput. Surv.*, vol. 55, no. 11, pp. 220:1–220:31, 2023. [Online]. Available: https://doi.org/10.1145/3565971

[34] A. Eghbali and M. Pradel, "De-hallucinator: Iterative grounding for llm-based code completion," *CoRR*, vol. abs/2401.01701, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.01701

[35] M. Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[36] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *ASE*, 2012, pp. 378–381.

[37] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[38] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution

diagnostics," in *ASE*, 2022, pp. 92:1–92:13. [Online]. Available: https://doi.org/10.1145/3551349.3556926

[39] X. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224. [Online]. Available: https://doi.org/10.1109/SANER.2016.76

[40] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *ASE*. IEEE, 2015, pp. 295–306.

[41] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *ICSE*, 2018, pp. 151–162. [Online]. Available: https://doi.org/10.1145/3180155.3180250

[42] Y. Liu, S. Mechtaev, P. Subotić, and A. Roychoudhury, "Program repair guided by datalog-defined static analysis," in *ESEC/FSE*, 2023, pp. 1216–1228.

[43] N. Jain, S. Gandhi, A. Sonwane, A. Kanade, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Staticfixer: From static analysis to static repair," 2023.

[44] D. Yang, X. Mao, L. Chen, X. Xu, Y. Lei, D. Lo, and J. He, "Transplantfix: Graph differencing-based code transplantation for automated program repair," in *ASE*, 2022, pp. 107:1–107:13. [Online]. Available: https://doi.org/10.1145/3551349.3556893

[45] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *ICSE-SEIP*, 2019.

[46] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: data-driven feedback generation for introductory programming exercises," in *PLDI*, 2018, pp. 481–495.

[47] R. Gupta, A. Kanade, and S. K. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 930–937. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.3301930

[48] G. Sakkas, M. Endres, P. J. Guo, W. Weimer, and R. Jhala, "Seq2parse: neurosymbolic parse error repair," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 1180–1206, 2022. [Online]. Available: https://doi.org/10.1145/3563330

[49] T. Yu and M. Pradel, "Pinpointing and repairing performance bottlenecks in concurrent programs," *Empirical Software Engineering (EMSE)*, pp. 1–38, 2017.

[50] J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and S. P. Chin, "Learning to repair software vulnerabilities with generative adversarial networks," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, 2018, pp. 7944–7954.

[51] Y. W. Chow, L. D. Grazia, and M. Pradel, "Pyty: Repairing static type errors in python," in *International Conference on Software Engineering (ICSE)*, 2024.

[52] X. Zhang, J. Zhai, S. Ma, and C. Shen, "AUTOTRAINER: an automatic DNN training problem detection and repair system," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 359–371. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00043

[53] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," in *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 19–20. [Online]. Available: https://doi.org/10.1145/3387940.3392181

[54] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural program repair by jointly learning to localize and repair," in *ICLR*, 2019.

[55] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *ICSE*, 2022.

[56] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.

[57] H. Joshi, J. P. C. Sánchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radicek, "Repair is nearly generation: Multilingual program repair with llms," in *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, B. Williams, Y. Chen, and J. Neville, Eds. AAAI Press, 2023, pp. 5131–5140. [Online]. Available: https://doi.org/10.1609/aaai.v37i4.25642

[58] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, "Cigar: Cost-efficient program repair with llms," *arXiv preprint arXiv:2402.06598*, 2024.

[59] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[60] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *ICSE*, 2024.

[61] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[62] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Software Eng.*, vol. 50, no. 1, pp. 85–105, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2023.3334955

[63] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage guided test generation in regression setting using llm," in *FSE*, 2024.

[64] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *FSE*, vol. abs/2402.09171, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.09171

[65] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *45th IEEE/ACM International Conference on Software Engineering, ICSE*, 2023, pp. 2312–2323. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00194

[66] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," in *ICSE*, 2024.

[67] R. Bairi, A. Sonwane, A. Kanade, V. D. C, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level coding using llms and planning," 2023.

[68] A. Thakur, G. Tsoukalas, Y. Wen, J. Xin, and S. Chaudhuri, "An in-context learning agent for formal theorem-proving," 2024. [Online]. Available: https://arxiv.org/abs/2310.04353

[69] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," 2024.

[70] J. Yang, C. E. Jimenez, K. Lieret, S. Yao, A. Wettig, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024.

[71] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig, "PAL: program-aided language models," *CoRR*, vol. abs/2211.10435, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2211.10435