UniDebugger: Hierarchical Multi-Agent Framework for Unified Software Debugging

Anonymous ACL submission

Abstract

Software debugging is a time-consuming endeavor involving a series of steps, such as fault localization and patch generation, each requiring thorough analysis and a deep understanding of the underlying logic. While large language models (LLMs) demonstrate promising potential in coding tasks, their performance in debugging remains limited. Current LLM-based methods often focus on isolated steps and struggle with complex bugs. In this paper, we propose the first end-to-end framework, **UniDebugger**, for unified debugging through multi-agent synergy. It mimics the entire cognitive processes of developers, with each agent specialized as a particular component of this process rather than mirroring the actions of an independent expert as in previous multi-agent systems. Agents are coordinated through a three-level design, following a cognitive model of debugging, allowing adaptive handling of bugs with varying complexities. Experiments on extensive benchmarks demonstrate that UniDebugger significantly outperforms state-of-the-art repair methods, fixing 1.25× to 2.56× bugs on the repo-level benchmark, Defects4J. This performance is achieved without requiring ground-truth root-cause code statements, unlike the baselines. Our source code is available on an anonymous link: https://github.com/AcceptePapier/UniDebugger.

1 Introduction

Debugging is a crucial process of identifying, analyzing, and rectifying bugs or issues in software. Significant advancements (Yang et al., 2024; Xia and Zhang, 2022; Wei et al., 2023; Xia and Zhang, 2024) have been achieved in addressing bugs with the boost of LLMs—they typically propose a prompting framework and query an LLM to automate an isolated phase in debugging, generally Fault Localization (FL) or Automated Program Repair (APR). FL attempts to identify suspicious code statements, while APR provides patches or fixed code snippets. A typical "diff" patch records the textual differences between two source code files, as shown in Figure 1. The goal of debugging is to apply such a patch to fix the bug accurately.

Chart 12

```
diff --git buggy.java fixed.java
@@ -142,5 +142,5 @@ public class MultiplePiePlot
extends Plot implements Cloneable, Serializable {
public MultiplePiePlot(CategoryDataset dataset) {
        super();
-       this.dataset = dataset;
+       setDataset(dataset);
        PiePlot piePlot = new PiePlot(null);
```

Figure 1: An example of the "diff" patch, where "@@ -142,5 +142 5 @@" indicates line 142 is modified.

While LLMs demonstrate great potential in addressing individual debugging tasks for basic bugs, previous studies cannot offer a satisfying solution for the entire debugging task. To bridge this gap, we propose the first multi-agent framework, **UniDebugger**, for end-to-end unified debugging. A key challenge lies in how to coordinate multiple agents. Previous studies that employ LLM-based multi-agent frameworks for complex problem-solving typically regard each agent as an expert and imitate the collaborative practice in humans (Hong et al., 2024; Islam et al., 2024; Qian et al., 2024). However, such a paradigm has limitations in debugging. It overly emphasizes mimicking human collaboration, thus not suitable for the logical, stepwise nature of debugging. Furthermore, it faces challenges in the fine-grained allocation and optimization of resources regarding problem difficulties. The communication links among agents are mesh-like and extensive, leading to unnecessary high communication redundancy.

In response, we uniquely structure UniDebugger as a hierarchical multi-agent coordination paradigm, as shown in Figure 2. *Our key insight is that an agent should act as a specific component of the cognitive process of debugging rather than mirroring the actions of an independent expert in teamwork.* In teamwork, members bring diverse perspectives and backgrounds. They engage in bidi-

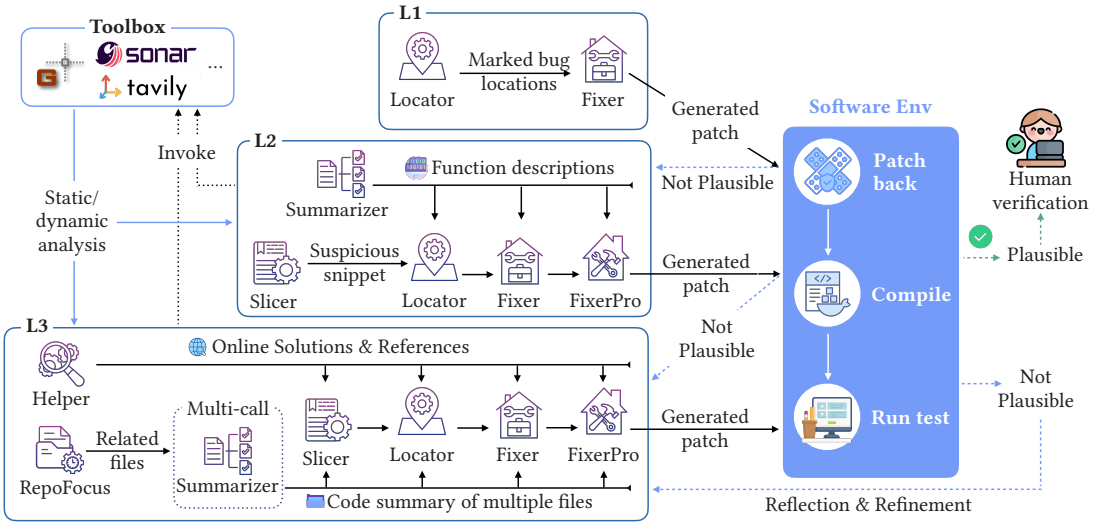

Figure 2: Overview of UniDebugger. It starts with the simple *L1* repair. If no plausible patch is generated, the *L2* repair is triggered, and so is *L3*. Agents on the same level can communicate with others.

rectional discussion and negotiation to share knowledge and solve tasks collaboratively. In contrast, the cognitive process is unidirectional and coherent, with each stage building on the last. It accumulates knowledge and employs tools toward a common objective, primarily relying on self-reflection to guide thought. Since the latter aligns seamlessly with debugging, we ground the multi-agent coordination in Hale and Haworth's cognitive debugging model (Hale and Haworth, 1991). It argues that developers employ a multi-level goal-orientated mechanism during debugging based on structural learning theory (Hale et al., 1999). The initial level only provides quick and straightforward solutions, and if that fails, higher levels of repairs are triggered for complex bugs, entailing more cognitive activities that involve deeper analysis, tool invocation, and external information ingestion.

Our framework encompasses seven agents, each specialized in a distinct cognitive state: 1) ***Helper***: retrieves and synthesizes debugging solutions through online research; 2) ***RepoFocus***: analyzes dependencies and identifies bug-related code files; 3) ***Summarizer***: generates code summaries; 4) ***Slicer***: isolates a code segment (typically tens to hundreds of lines) likely to cause an error; 5) ***Locator***: marks the specific root-cause code lines; 6) ***Fixer***: generates patches or fixed code snippets; 7) ***FixerPro***: generates an optimized patch upon the testing results of the patch generated by *Fixer* along with a detailed analysis report.

Particularly, UniDebugger first isolates the code file most likely causing the error through unit testing. On level one (*L1*), only *Locator* and *Fixer* are initiated for simple bugs. If the generated patch is not plausible (i.e., pass all test cases), *Summarizer* and *Slicer* are triggered with the bug-located code file on level two (*L2*). Then *Locator*, *Fixer*, and *FixerPro* are called sequentially with access to the responses from *Summarizer* and *Slicer*. If it still fails, UniDebugger undertakes a deeper inspection and thus activates all agents (i.e., level three, *L3*), where *Helper* searches for solutions online to guide all the other agents, and *RepoFocus* examines the entire program to provide a list of bug-related code files, followed by the other agents working in line. *Slicer*, *Locator*, and *Fixer* can invoke traditional code analysis tools on the last two levels to collect static and dynamic analysis information.

We evaluate UniDebugger on four benchmarks with bug-fix pairs across three programming languages, including real-world software (Defects4J Just et al., 2014) and competition programs (QuixBugs Lin et al., 2017, Codeflaws Tan et al., 2017, ConDefects Wu et al., 2024). On Defects4J, UniDebugger achieves a new state-of-the-art (SoTA) by correctly fixing 197 bugs with 286 plausible fixes. Our method does not require prior FL while still outperforming strong baselines like ChatRepair, which uses ground-truth root causes with 10x sampling times. Plus, UniDebugger fixes all bugs in QuixBugs and generates 2.2× more plausible fixes on Codeflaws. Our ablation study shows that UniDebugger consistently makes significant improvements across various LLM backbones.

2 Related Work

Software Debugging: FL and APR are the two core steps in software debugging (Benton et al., 2020). Many studies aim to automate either step.

Traditional FL are typically spectrum-based (Abreu et al., 2009, 2006; Zhang et al., 2011) or mutation-based (Moon et al., 2014; Li and Zhang, 2017; Zhang et al., 2013). Learning-based FL learns program behaviors from rich data sources (Li et al., 2021, 2022; Lou et al., 2021; Li et al., 2019) via multiple types of neural networks. Recently, LLMAO (Yang et al., 2024) proposes to utilize LLMs for test-free FL. APR research either searches for suitable solutions from possible patches (Goues et al., 2012; Weimer et al., 2009a,b) or directly generates patches by representing the generation as an explicit specification inference (Mechtaev et al., 2016; Nguyen et al., 2013; Liu et al., 2019). Learning-based studies ((Lutellier et al., 2020; Jiang et al., 2021; Ye et al., 2022)) translate faulty code to correct code using neural machine translation (NMT). On top of directly prompting LLMs (Xia et al., 2023), recent studies have explored prompt engineering (Xia and Zhang, 2024, 2022) or combining code synthesis (Wei et al., 2023) to improve LLM for APR.

Large Language Models: Various LLMs have been developed for code synthesis (Chen et al., 2021; Guo et al., 2024; Rozière et al., 2023), in addition to general-purpose LLMs (OpenAI, 2023a,b; Anil et al., 2023). They have shown potential in solving coding-related tasks, including program repair, as investigated in empirical studies (Xia et al., 2023; Huang et al., 2023; Tian et al., 2024).

LLM-based Multi-Agent (LLM-MA): The inspiring capabilities of the single LLM-based agent boost the development of multi-agent frameworks. Recent research has demonstrated promising results in utilizing LLM-MA for complex problem-solving, including software engineering, science, and other society-simulating activities. For software engineering, LLM-MA studies (Qian et al., 2024; Hong et al., 2024; Dong et al., 2023; Huang et al., 2024; Islam et al., 2024) usually emulate real-world roles (e.g., product managers, programmers, and testers) and collaborate through communication. Hong et al. (2024); Dong et al. (2023) adopt a shared information pool to reduce overhead.

3 UniDebugger

UniDebugger is an end-to-end framework for unified debugging through LLM-based multi-agent synergy. It comprises seven agents, each specialized as a state in Hale and Haworth's cognitive debugging model with an explicit goal instead of being a task-oriented, individual entity. UniDebugger operates on a three-level architecture, initializing different levels of repair involving different agents, and agents can adaptively invoke tools in a given toolbox. The communication among agents on the same level follows an assembly-line thought process rather than the mesh interactions typical of teams. This section introduces the profiles of agents, their interactions with external environments, and the hierarchical organization. Prompt details are displayed in A.3.

System Prompt of Summarizer

```
## Role
Skilled assistant for summarizing code

## Skills
– Proficient in understanding complex Java code
– Good at analyzing and condensing the functionality of each function

## Action
1. Go through the given code and identify the attributes and functions within each class
2. Analyze and comprehend the purpose of each function
3. Write a concise description of what the function does in the specified format

## Objective
Return a summary for every class and function in the format of <format>

## Constrains
Ensure the summary accurately reflects the functionality of the code without adding or
removing details that are not in the original code. Do not return any other irrelevant
information except for the code summary. Ensure to adhere to the format requirement.

## Example
<Human-written example>
```

Figure 3: A one-shot system prompt defines the role, skills, actions, objective, and constraints of the agent.

3.1 Profiles of Agents

All agents in UniDebugger share a one-shot structure of the system prompt, which defines their roles, skills, actions, objectives, and constraints, followed by a manually crafted example to illustrate the desired response format, as shown in Figure 3. Moreover, all agents have access to certain meta-information known as *bug metadata*, including the bug-located code file, failing test cases, reported errors during compilation and testing, and program requirements described in natural language. The response of each agent comprises two elements: the answer fulfilling its goal and an explanation of its thinking. Inspired by a software engineering principle, rubber duck debugging (Andrew and David, 2000), where developers articulate their expectations versus actual implementations to identify the gap, we request each agent to monitor key program variables and explain how it guides the answer.

Helper. The goal of *Helper* is to provide references via retrieve-augmented generation (RAG), inspired by the fact that developers commonly utilize web search engines (e.g., Google) to enhance productivity (Xia et al., 2017). By analyzing the bug metadata, *Helper* generates a short query and invokes an external search engine to retrieve the best-matching solutions. Then, it integrates the retrieval results and generates reference solutions that are customized for the context of the buggy code (e.g., variable naming and function structure).

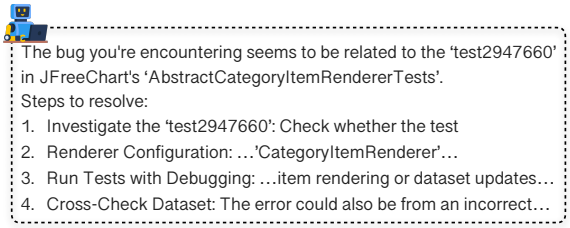

Figure 4: After retrieving similar solutions, *Helper* eventually responds with an executable debug guide.

These are internal processes hidden from the other agents, and only the final response—a reference solution—can be read only on *L3*. Figure 4 provides an example of the response from *Helper*.

RepoFocus. Comprehending cross-file dependencies is crucial to debugging large software, and real-world software is often large and complex (corresponding to *L3* repair). However, inputting all of the code in a program may overwhelm an LLM, resulting in slow responses with significant resource consumption or incorrect results. Thus, *RepoFocus* provides a list of bug-related code files that need further examination by analyzing the bug metadata and the file structure of the program.

Summarizer. Code summarization aims to generate concise, semantically meaningful summaries that accurately describe software functions. Unlike high-level code analysis, these natural language summaries better align with the training objectives of LLMs. *Summarizer* is triggered on *L2* and *L3*. On *L2*, it summarizes the buggy code file. Though the window size of advanced LLMs can handle most single code files, a too-long prompt may exacerbate the illusion of LLMs. Thus, *Summarizer* handles overly long code here, in conjunction with *Slicer*, which narrows down a suspicious segment from the buggy code. On *L3*, *Summarizer* runs once on every bug-related file identified by *RepoFocus*. The other agents do not need to read every code line in the program while still knowing the core contents through these summaries.

Slicer. *Slicer* narrows down the inspection scope by slicing out a small suspicious segment (typically tens of code lines) from the bug-located code file. We extract the beginning and end lines from the initial output to locate the segment, ensuring the final output is directly sliced from the original code to prevent code tampering caused by LLM hallucinations. *Slicer* is also launched for large software on the last two levels, where it can invoke static or dynamic analysis tools.

Locator. *Locator* is responsible for marking code lines with a comment "// buggy line" or "// missing line" to indicate faulty or missing statements. Similar to *Slicer*, we directly annotate the original code through contextual string matching. *Locator* only leverages the bug metadata on *L1*, where on *L2*, it can access the dynamic analysis reports. If the program is large enough so that *Slicer* and *Summarizer* are invoked, *Locator* receives the suspicious code segment (generated by *Slicer*) to replace the original complete code, along with the buggy code summary (generated by *Summarizer*).

Fixer. *Fixer* is prompted to generate a 'diff', as shown in Figure 1 in Introduction. It receives code marked by *Locator* and other bug metadata. To maintain consistency between *Locator* and *Fixer*, *Fixer* first assesses whether the marked lines should be modified and then describes the modification. This also enables *Fixer* to correct possible errors made by *Locator*. On *L2* and *L3*, *Fixer* also has access to static/dynamic analysis and auxiliary information generated by upstream agents.

FixerPro. *FixerPro* extends and complements *Fixer* by generating an optimized patch and repair analysis. Inspired by code review, in which one or more developers check a program to ensure software quality, we request *FixerPro* to evaluate the performance and potential vulnerability of the plausible fix generated by *Fixer*. Then, *FixerPro* provides suggestions for refactoring the patch to optimize simplicity and maintainability. If *Fixer* fails, *FixerPro* generates a new patch while analyzing the failing reasons.

3.2 External Interactions

Plausibility Feedback. We conduct repairs by modifying the buggy code rather than directly applying the generated patches because they usually make mistakes in indices of code lines, as LLMs often struggle with counting, and the patch may be generated based on a sliced segment rather than the original code, causing changes in the indices of code lines. Our modification is rule-based by matching the invariant context between the patch and the buggy code. Afterward, we compile and run tests on the fixed program. The patch will be sent to developers for manual *correctness* verification if passing all test cases. The results of plausibility testing are returned to UniDebugger as feedback for further processing, such as initializing a higher level of repair or conducting self-reflection.

Tool Usage. The toolbox of UniDebugger currently contains static analysis tools, dynamic analysis tools, and a search engine optimized for LLMs

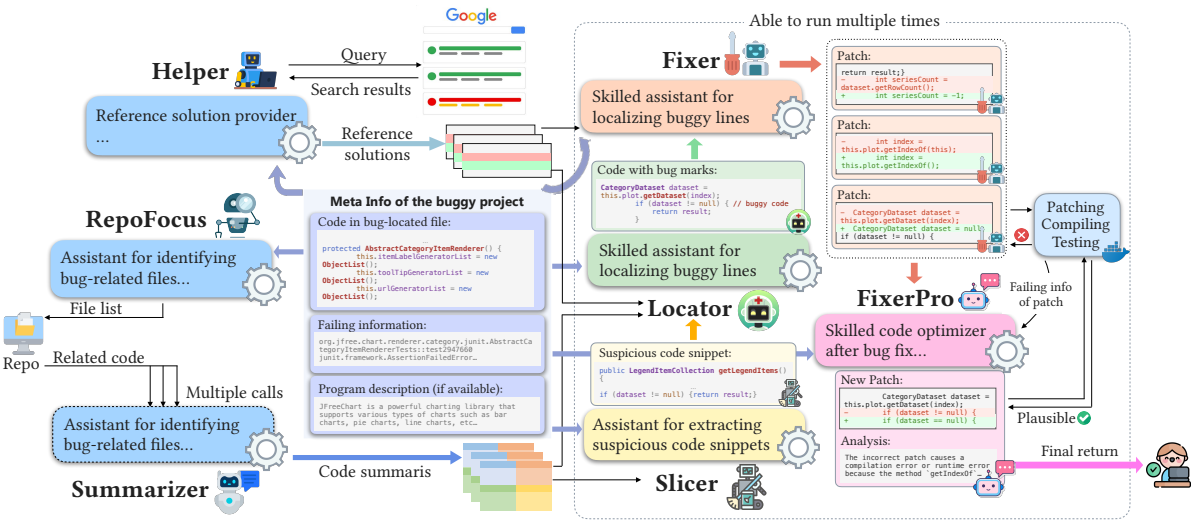

Figure 5: *L3* triggers all the seven agents to generate plausible patches.

and RAG. Static analysis provides static errors and warnings, as well as Abstract Syntax Tree (AST), a tree representation of the abstract syntactic structure of source code. We mainly consider the coverage of failing test cases in dynamic analysis since it is commonly assumed that code statements not executed during testing are less likely to cause the failures (Kochhar et al., 2016). Note that the results of tool invocations will be stored for the requests of downstream agents to avoid duplicated invocations.

3.3 Hierarchical Coordination

Our main principle of coordination is that problems of different complexities warrant cognitive activities of different intensities. That is, upon the failure of a straightforward solution, a higher-level goal will be triggered with more information and thinking. Following Hale and Haworth's cognitive debugging model, we define three workflows for the three levels of repair, respectively.

The first level identifies and fixes obvious logic errors in code, so we assume that agents can easily fix the bug. Thus, *L1* only contains *Locator* and *Fixer* with a simple communication flow, where *Fixer* generates a patch based on the fault localization done by *Locator*. If the fix is not plausible, then *L2* is triggered, assuming that the fix can be achieved by only inspecting a single file, which is often applied in previous FL and APR studies (Soremekun et al., 2023). *L2* involves five agents: *Slicer*, *Summarizer*, *Locator*, *Fixer*, and *FixerPro*. *Slicer* isolates a suspicious segment, and *Summarizer* generates a code summary simultaneously, both from the bug-located file. Afterward, *Locator*, *Fixer*, and *FixerPro* work in sequence based on the suspicious segment, and all have access to the code summary. If it still fails, we turn to *L3*, which believes that the bug is very complex, so repairing it requires understanding cross-file dependencies and external information. *L3* activates all the seven agents. First, *Helper* and *RepoFocus* are initialized at the same time, and *Helper* provides references for all other agents. Then, *Summarizer* runs multiple times following the file list identified by *RepoFocus*. These code summaries are shared by the remaining four agents, which work one after another subsequently with the same workflow as *L2*. Figure 5 presents an example conversation among agents on *L3*.

Upon no plausible patch, UniDebugger gradually requests agents on *L3* to reflect and refine its answer based on the plausibility feedback in a reversed order, as downstream agents are more error-prone as errors may accumulate during the debugging process. Specifically, we first ask *FixerPro* to refine its response. If the fix fails again, we then request *Fixer*, leading to input changes of *FixerPro*, so it is re-sampled the second time. Eventually, plausible patches are returned to developers for manual verification. If no plausible patch is produced, UniDebugger will display the analysis report written by *FixerPro* and take another run until reaching a resource threshold.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate UniDebugger on four benchmarks featuring bug-fix pairs, including competition programs (Codeflaws Tan et al., 2017, QuixBugs Lin et al., 2017, ConDefects Wu et al., 2024) and real-world projects (Defects4J Just et al., 2014). Codeflaws contains 3902 faulty C programs. QuixBugs includes 40 faulty programs available

Table 1: Comparison with baselines. *#Corr* and *#Plau* represent the number of bugs correctly and plausibly patched, respectively. The green cells indicate the best results. The blue cells indicate the results are obtained from sampled data, while other results are obtained on the whole dataset.

Tools	**Sampling Times**	**Defects4J-Java**		**Codeflaws-C**		**QuixBugs-Java**		**QuixBugs-Python**		**Note**
		#Corr	*#Plau*	*#Corr*	*#Plau*	*#Corr*	*#Plau*	*#Corr*	*#Plau*	
Angelix	1,000	-	-	318	591	-	-	-	-	Realistic FL
Prophet	1,000	-	-	310	839	-	-	-	-	
SPR	1,000	-	-	283	783	-	-	-	-	
CVC4	-	-	-	15†	91†	-	-	-	-	
Semfix	1,000	25	68	38†	56†	-	-	-	-	
Recoder	100	72	140	-	-	17	17	-	-	
GenProg	1000	5	20	255-369	1423	1	4	-	-	Perfect FL
CoCoNuT	20,000	44⋆	85⋆	423	716	13	20	19	21	
CURE	-	57⋆	104⋆	-	-	26	35	-	-	
RewardRepair	200	90	75	-	-	20	-	-	-	
Tbar	500	77	121	-	-	-	-	-	-	
AlphaRepair	5,000	110	159	-	-	28	30	27	32	
Repilot	5,000	116	-	-	-	-	-	-	-	
ChatRepair	100-200	157	-	-	-	**40**	**40**	**40**	**40**	
CodeLlama-34b	20	24	41	91(%)‡	1,488	25	28	33	33	Perfect FL
LLaMA2-70b		39	78	91(%)‡	1,576	25	28	33	33	
DeepSeekCoder		57	82	93(%)‡	1,937	30	34	25	38	
gemini-1.5-flash		19	36	86(%)‡	1,291	29	32	29	35	
gpt-3.5-turbo-ca		45	71	94(%)‡	2,343	33	34	34	36	
claude-3.5-sonnet		70	116	**95(%)**‡	2,624	36	37	40	40	
gpt-4o		72	119	93(%)‡	2,549	35	36	39	39	
UniDebugger-Lite	5	-	-	**95(%)**‡	**3,130**	**40**	**40**	**40**	**40**	
UniDebugger	20	**197**	**286**	-	-	-	-	-	-	

⋆ Only the result on Defects4J 1.2 is available.
† The result is obtained from 665 sampled bugs.
‡ We randomly select 100 plausible patches to check their correctness because of the huge number of plausible patches.

in both Java and Python. ConDefects recently collected to address data leakage concerns in LLMs, consists of 1,254 Java and 1,625 Python faulty programs. Defects4J, a widely used benchmark from 15 real-world Java projects, features bugs across two versions: version 1.2 with 391 active bugs and version 2.0 adding an additional 415 active bugs, totaling 806. We will report the total number of fixes across these versions.

Baselines. We compare UniDebugger against 14 APR baselines, including:

- Traditional: Angelix (Mechtaev et al., 2016), Prophet (Long, 2018), SPR (Long and Rinard, 2015), and CVC4 (Reynolds et al., 2015).
- Genetic programming-based: Semfix (Nguyen et al., 2013) and GenProg (Goues et al., 2012; Weimer et al., 2009a).
- NMT-based: CoCoNuT (Lutellier et al., 2020), CURE (Jiang et al., 2021), and RewardRepair (Ye et al., 2022).
- Domain knowledge-driven: Tbar (Liu et al., 2019) and Recoder (Zhu et al., 2021).
- LLM-based (SoTA): AlphaRepair (Xia and Zhang, 2022), Repilot (Wei et al., 2023), and ChatRepair (Xia and Zhang, 2024).

Note that different APR baselines adopted diverse prior fault locations. We use *realistic* to represent traditional FL and *perfect* to denote ground-truth FL. We report the results provided in their original papers and follow-up survey (Le et al., 2018; Ye et al., 2021; Xia et al., 2023), following previous studies (Xia et al., 2023; Xia and Zhang, 2024; Lutellier et al., 2020).

LLM Backbones. We apply UniDebugger to seven LLMs, including four general-purpose models (gemini-1.5-flash Anil et al., 2023, gpt-3.5-turbo-ca OpenAI, 2023a, gpt-4o OpenAI, 2023b, and claude-3.5-sonnet Anthropic, 2024) and three open-source code LLMs (DeepSeekCoder Guo et al., 2024, CodeLlama-34b Rozière et al., 2023, LLaMA2-70b Touvron et al., 2023). Naturally, these LLMs also serve as baselines for comparison, where we apply the original chain-of-thought (CoT) prompting method (Wei et al., 2022). The

default backbone of UniDebugger is gpt-4o.

Metrics. We utilize the APR metrics, specifically focusing on the number of bugs that are plausibly or correctly fixed. Given the extensive size of ConDefects, we randomize samples from plausible fixes to verify their correctness. Thus, we also employ the metric known as *correctness rate*—defined as the ratio of correct fixes to plausibly fixed bugs—to assess the effectiveness of UniDebugger.

Implementation. For single-file competition programs, only *Locator*, *Fixer*, and *FixerPro* on *L2* and *L1* need to be initialized, named as ***UniDebugger-Lite***. The maximum number of attempts is set to 5 per bug for Lite and 20 for the full version. Previous studies, however, typically sample hundreds to thousands of times. For instance, CoCoNuT samples up to 50,000 patches per bug (Lutellier et al., 2020), while ChatRepair samples 100-200 times (Xia and Zhang, 2024). Our approach is significantly more cost-effective than baselines. Furthermore, we alter the default web search engine (Tavily Tavily) to a local one during evaluation to prevent retrieving ground-truth solutions online. The local database consists of the training dataset of CoCoNut (Lutellier et al., 2020) with JavaScript programs removed. Since CoCoNuT was also evaluated on Defects4J and ConDefects, the risk of data leakage is minimal. The static and dynamic analysis is supported by our written scripts and open-source plugins (SonarQube and GZoltar).

4.2 Comparison with baselines

This section evaluates the debugging capabilities of our framework, UniDebugger. The results are shown in Table 1. We do not use ConDefects herein because it is a recently released dataset, and few approaches have been evaluated on it.

Competition Programs. UniDebugger plausibly fixes 3130 out of 3982 bugs on Codeflaws, producing 2.2x plausible fixes as the best APR method, GenProg. It has a correctness rate of 95%, i.e., UniDebugger correctly repairs 95 bugs out of 100 sampled plausible patches, improving the correctness rate by 60.81% compared to that of CoCoNuT, which correctly fixes the most bugs among APR approaches. UniDebugger surpasses the best-performing LLM baseline, claude-3.5-sonnet, by ∼19.28% with the same correctness rate. Moreover, UniDebugger successfully fixes all bugs in QuixBugs across two programming languages, achieving the same SoTA as ChatRepair.

Real-World Software. On Defects4J, UniDebugger correctly fixes 197 bugs while plausibly solving 286 bugs, outperforming the SoTA, ChatRepair, by ∼25.48%. Moreover, UniDebugger correctly fixes 42 unique bugs that the top four baselines have never fixed. Figure 6 shows the Venn diagram of the bugs fixed by the top four baselines and UniDebugger on Defects4J. We see that UniDebugger correctly fixes 42 unique bugs that these strong baselines have not addressed. We show an example of the unique fixes in A.1.

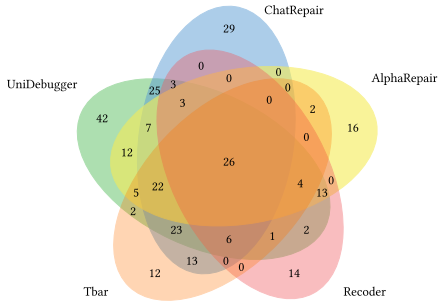

Figure 6: Bug fix Venn diagram on Defects4J.

Overall, LLM-involved baselines perform reasonably well. On top, UniDebugger significantly enhances base LLMs on all benchmarks, especially on Defects4J. This is because direct prompting LLMs struggles with too long contexts and complex reasoning. We split debugging into several cognitive steps and introduce external tools and knowledge, thus boosting the capabilities of LLMs.

> **Takeaway:** UniDebugger fixes 197 bugs on Defects4J, a 25.48% improvement over the leading baseline. Its lite version fixes all bugs on QuixBugs and achieves 19.28% more plausible fixes on ConDefects with the highest correctness rate.

4.3 Performance on Different LLMs

To verify the robustness of UniDebugger across various LLMs, we compare UniDebugger-Lite with its seven LLM backbones, evaluated on 600 randomly sampled bugs from ConDefects (300 for each programming language). Since manually checking the patches is time-consuming, we only present the number of plausible fixes. Table 3 displays the performance gains of UniDebugger-Lite (UD-L) over its backbone with CoT prompting. For simplicity, we only report the number of plausible fixes.

The results indicate that UniDebugger can consistently enhance its backbone LLM by 21.60%-52.31%. Notably, UD-L with LLaMA2 achieves 280 plausible fixes, closely rivaling the performance of gpt-3.5-turbo-ca (282). Plus, UD-L with

Table 2: Ablation study on agents. *#Plau* represent the number of plausibly fixed bugs. The ✓ indicates the addition of a specific agent, and ✗ denotes its absence. *Expense* denotes the average expense running once for a bug.

Helper	RepoFocus	Summarizer	Slicer	Locator	Fixer	FixerPro	*#Plau*	Expense ($)	Level
✗	✗	✗	✗	✗	✓	✗	72	0.030	L1
✗	✗	✗	✗	✓	✓	✗	140	0.048	
✗	✗	✗	✗	✓	✓	✓	192	0.116	L2
✗	✗	✗	✓	✓	✓	✓	224	0.225	
✗	✗	✓	✓	✓	✓	✓	238	0.317	
✗	✓	✓	✓	✓	✓	✓	245	0.364	L3
✓	✓	✓	✓	✓	✓	✓	**291**	0.410	

Table 3: Performance gains of UniDebugger-Lite over different LLMs on 600 samples from ConDefects.

LLMs	ConDefects-Java			ConDefects-Python		
	CoT	UD-L	Gain ↑	CoT	UD-L	Gain ↑
CodeLlama-34b	87	113	29.89%	69	86	24.64%
LLaMA2-70b	108	147	36.11%	91	133	46.15%
DeepSeekCoder	130	198	52.31%	125	178	42.40%
gemini-1.5-flash	62	89	43.55%	63	82	30.16%
gpt-3.5-turbo-ca	155	191	23.23%	127	174	37.01%
claude-3.5-sonnet	213	259	21.60%	186	227	22.04%
gpt-4o	211	262	24.17%	179	225	25.70%

DeepSeekCoder plausibly fixes similar bugs (376) as gpt-4o does (390). The results indicate that UniDebugger brings the gap between open-source code LLMs and proprietary systems like gpt-4o in debugging. Furthermore, though UniDebugger can make improvements across varying LLMs, its overall performance is strongly related to the coding ability of its backbone LLM.

4.4 Ablation Study

4.4.1 Impact of Different Agents

To understand the impact of different agents on the effectiveness of UniDebugger, we exclude certain agents across Defects4J, as we trigger all agents on it. We also report the number of plausible fixes herein. As indicated by Table 2, the addition of agents different from just *Fixer* consistently improves the number of plausible fixes. While more agents slightly increase the expenses, the overall performance improves noticeably, demonstrating the effectiveness of the various agents and the importance of the divide-and-conquer idea. Since the higher level is only triggered when the lower level fails, higher levels naturally improve performance.

Plus, the performance gains of *L2* to *L1* are more pronounced than that of *L3* to *L2* as expected, since *L2* adds more agents to *L1* with tool usage, while *L3* only adds reference solutions and auxiliary cross-file information. We notice that *Helper* only increases 46 plausible fixes, indicating that the internet cannot always provide solutions, so domain-specific tools for debugging are highly desired.

4.4.2 Impact of External Interactions

We also evaluate the impact of external interactions on Defects4J, including the feedback of testing results to *FixerPro* and toolbox usage. As shown in Table 4, introducing external interactions leads to a significant improvement ranging from 23 to 121 in the number of plausible fixes. This illustrates how our designed mechanism of environment interactions can contribute to high-quality debugging.

Table 4: Ablation study on external interactions.

Online Search	Static	Dynamic	Testing	*#Plau*
✗	✓	✓	✓	245
✓	✗	✓	✓	268
✓	✓	✗	✓	170
✓	✓	✓	✗	244
✓	✓	✓	✓	**291**

5 Conclusion

This paper presents UniDebugger, the first end-to-end framework leveraging LLM-based multi-agent synergy to tackle unified software debugging. Our method employs a novel hierarchical coordination paradigm, inspired by a cognitive debugging model, to efficiently manage cognitive steps with minimal communication and dynamically adjust to bug complexity through its three-level architecture. Extensive experiments on four benchmarks demonstrate the superiority of our method over SoTA repair approaches and base LLMs. UniDebugger fixes 1.25–2.56× bugs on a repo-level benchmark and fixes all bugs on QuixBugs. Its lite version achieves the most plausible fixes on the other two competition program benchmarks. Lastly, the effective implementation of Hale and Haworth's cognitive model for debugging paves new pathways for research in advancing LLM-based multi-agent frameworks for tackling complex coding tasks.

6 Limitation

While our study introduces the end-to-end framework UniDebugger with a hierarchical multi-agent approach for debugging, it does come with certain limitations. First, the complexity of coordinating multiple agents increases computation overhead and can slow down the debugging process, particularly in resource-constrained environments. Second, our framework relies heavily on predefined agent roles and assumptions, which may not fully generalize to novel or edge-case bugs in diverse programming languages. Third, the integration of external program analysis tools, while useful, can introduce delays and inconsistencies in data retrieval, affecting the overall efficiency. Future work should explore optimizing token consumption, improving adaptability to diverse bug types, and ensuring smoother integration with external tools for faster and more reliable debugging.

7 Ethics Consideration

We do not foresee any immediate ethical or societal risks arising from our work. However, given that UniDebugger relies heavily on LLM-generated code patches, there is a potential risk of introducing unintended vulnerabilities or errors in software. We encourage researchers and practitioners to apply UniDebugger cautiously, particularly when using it in production environments. Ensuring thorough validation and testing of LLM-generated patches is crucial to mitigate any negative consequences. Additionally, We adhere to the License Agreement of the LLM models and mentioned open-sourced tools.

References

Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, pages 39–46. IEEE Computer Society.

Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-based multiple fault localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 88–99. IEEE Computer Society.

Hunt Andrew and Thomas David. 2000. The pragmatic programmer: From journeyman to master.

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. 2023. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805.

Anthropic. 2024. Introducing the next generation of claude.

Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 907–918. IEEE.

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. *CoRR*, abs/2107.03374.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt. *CoRR*, abs/2304.07590.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 3–13. IEEE Computer Society.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

GZoltar. [link].

David P. Hale and Dwight A. Haworth. 1991. Towards a model of programmers' cognitive processes in software maintenance: A structural learning theory approach for debugging. *J. Softw. Maintenance Res. Pract.*, 3(2):85–106.

Joanne E. Hale, Shane Sharpe, and David P. Hale. 1999. An evaluation of the cognitive processes of programmers engaged in software debugging. *J. Softw. Maintenance Res. Pract.*, 11(2):73–91.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for A multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1162–1174. IEEE.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4912–4944. Association for Computational Linguistics.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: code-aware neural machine translation for automatic program repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1161–1173. IEEE.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM.

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 165–176. ACM.

Xuan-Bach Dinh Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, page 163. ACM.

Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 169–180. ACM.

Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.*, 1(OOPSLA):92:1–92:30.

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault localization with code coverage representation learning. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 661–673. IEEE.

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. Fault localization to detect co-change fixing locations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 659–671. ACM.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 55–56. ACM.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 31–42. ACM.

Fan Long. 2018. *Automatic patch generation via learning from successful human patches*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, USA.

Fan Long and Martin C. Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178. ACM.

Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. 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*, pages 664–676. ACM.

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*, pages 101–114. ACM.

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*, pages 691–701. ACM.

Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 153–162. IEEE Computer Society.

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*, pages 772–781. IEEE Computer Society.

OpenAI. 2023a. Gpt-3.5 turbo.

OpenAI. 2023b. Gpt-4: a technical report. *CoRR*, abs/2303.08774.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 15174–15186. Association for Computational Linguistics.

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in smt. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II 27*, pages 198–216. Springer.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

SonarQube. [link].

Ezekiel O. Soremekun, Lukas Kirschner, Marcel Böhme, and Mike Papadakis. 2023. Evaluating the impact of experimental assumptions in automated fault localization. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 159–171. IEEE.

Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 180–182. IEEE Computer Society.

Tavily. [link].

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *CoRR*, abs/2401.04621.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 172–184. ACM.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009a. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374. IEEE.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009b. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374. IEEE.

Yonghao Wu, Zheng Li, Jie M. Zhang, and Yong Liu. 2024. Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 642–646. ACM.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1482–1494. IEEE.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 959–971. ACM.

Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 819–831, New York, NY, USA. Association for Computing Machinery.

Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empir. Softw. Eng.*, 22(6):3149–3185.

Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 17:1–17:12. ACM.

He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the quixbugs benchmark. *J. Syst. Softw.*, 171:110825.

He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1506–1518. ACM.

Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 23–32. IEEE Computer Society.

Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 765–784. ACM.

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*, pages 341–353. ACM.

A Appendix

A.1 Unique Fix Example

We illustrate the power of UniDebugger by showing an example bug that is only fixed by UniDebugger in Figure 7.

JacksonDatabind 74

```
            if (p.getCurrentToken() == JsonToken.START_ARRAY) {
                return super.deserializeTypedFromAny(p, ctxt);
+           } else if (p.getCurrentToken() ==
JsonToken.VALUE_STRING &&
+
ctxt.isEnabled(DeserializationFeature.ACCEPT_EMPTY_STRING_AS
_NULL_OBJECT) &&
+                   p.getText().isEmpty()) {
+               return null;
            }
```

Figure 7: Unique bug fixed by UniDebugger in Defects4J.

On the one hand, the fix requires filling in the missing code statements. Traditional FL based on failing test coverage cannot directly identify such an error of lacking the necessary processing

of a branching condition. Plus, many template-based or NMT-based APR tools are good at repairing common errors such as syntax errors or simple logic errors, but not at generating new business logic. On the other hand, this fix requires a deep understanding of the specific Jackson deserialization features defined in the package "databind.DeserializationFeature". Previous LLM-based APR often relies on local context in the prompt and statistical correlation so as to lack the ability to comprehend other code files within the project repo. The three levels of repair enable UniDebugger to access extra related information (including templates and repo-level documents) and testing feedback, as well as enhance its reasoning ability under the idea of divide-and-conquer. Combining all these conditions together, UniDebugger is able to correctly fix this complex bug.

A.2 Alogrithm of UniDebugger

Algorithm 1 shows the pseudo-code of our proposed framework.

A.3 System Prompts

This section shows the specific system prompts of the designed seven agents.

Helper

```
## Role
Assistant for providing bug-fixing solutions using online searching

## Skills
- Proficient in analyzing bug-related information to write a 100-word-limit query
- Capable of using Tavily search API to find the most relevant online information
- Expert at synthesizing search results to provide actionable debugging steps

## Action
1. Analyze the provided buggy code, error information, and failing test cases
2. Write a query within 100 words
3. Use Tavily search API with the query to search for similar issues, relevant
   documentation, and community discussions
4. Summarize the gathered information into a structured, step-by-step debugging
   guide targeting the buggy code, not the testing cases
5. Provide relevant URLs at the end to support the suggested solutions

## Objective
Deliver a concise and actionable debugging guide to the user

## Constrains
- The debugging guide should only contain steps based on the gathered online
  information
- Do not rely on your own knowledge; all suggestions must be traceable to online
  sources
- List each supporting URL on a separate line. Enclose all URLs within three equal
  signs (===)

## Example
#### USER'S INPUT
Buggy code:
```
def divide_numbers(a, b):
    return a / b
print(divide_numbers(10, 0))
```
Failing test case: divide_numbers(10, 0) results in an error
Error message: ZeroDivisionError: division by zero
#### AGENT'S OUTPUT
Debugging Guide:
1. Search for `Python ZeroDivisionError` and solutions to handle division by zero
   errors
2. Found several recommendations suggesting adding a check for zero values before
   division
3. Modify the `divide_numbers` function to handle the error gracefully
```
def divide_numbers(a, b):
    if b == 0:
        return 'Error: Division by zero is not allowed.'
    return a / b
    print(divide_numbers(10, 0))
```
4. Run the updated code to confirm that the error is handled correctly
===
https://stackoverflow.com/questions/20931334/handling-zerodivisionerror-in-python
https://docs.python.org/3/library/exceptions.html#ZeroDivisionError
===
```

Figure 8: System Prompt of *Helper*.

Algorithm 1: UniDebugger

Input: k: number of maximum debugging attempts; m: number of maximum re-sampling attempts of a single agent; bug_meta: metadata of the bug, a directory including code from the bug-located file, failing test case(s), errors, and program requirements.
Output: patch, analysis

```
Function L1Repair(m, bug_meta, extra_info):
    for j ← 1 to m do
        marked_code ← Locator(bug_meta, extra_info)
        if ValidMarks(marked_code) then
            bug_meta[code] ← marked_code
            break
        end
    end
    patch ← Fixer(bug_meta, extra_info)
    return patch
End
Function L2Repair(m, bug_meta, extra_info):
    if summary NOT IN extra_info then
        summary ← Summarizer(bug_meta[code])
        extra_info ← concat[extra_info; summary]
    end
    for j ← 1 to m do
        snippet ← Slicer(bug_meta)
        if ValidSnippet(snippet) then
            bug_meta[code] ← snippet
            break
        end
    end
    patch ← L1Repair(m, bug_meta, extra_info)
    patch, analysis ← FixerPro(patch, Testing(patch), bug_meta, extra_info)
    return patch, analysis
End
Function L3Repair(m, bug_meta, extra_info):
    references ← Helper(bug_meta)
    FileList ← RepoFocus(bug_meta)
    summary ← ArrayList()
    for file in FileList do
        summary.append(Summarizer(ReadFile(file)))
    end
    extra_info ← concat[extra_info; references; summary]
    return L2Repair(m, bug_meta, extra_info)
End
Function Debugging(k, m, bug_meta):
    for i ← 1 to k do
        extra_info ← EmptyList()
        patch ← L1Repair(m, bug_meta, extra_info)
        if Testing(patch) then
            return patch, EmptyString()
        end
        patch, analysis ← L2Repair(m, bug_meta, extra_info)
        if Testing(patch) then
            return patch, analysis
        end
        patch, analysis ← L3Repair(m, bug_meta, extra_info)
        if Testing(patch) then
            return patch, analysis
        end
        patch, analysis ← RefineAgents(m, bug_meta, extra_info, patch)
        return patch, analysis
    end
End
```

RepoFocus

```
## Role
Expert assistant for identifying bug-related files

## Skills
- Proficient in file structure analysis
- Skilled at linking bug data to file content based on package and dependency usage
- Efficient in pinpointing relevant files using error information

## Action
1. Review the folder structure
2. Analyze packages and dependencies in the affected code against error details
3. Assess each file's relevance based on associated packages and dependencies
4. Identify 2-6 files likely linked to the bug

## Objective
- Provide a list of 2-6 suspected files
- Include a brief explanation on the rationale for suspicion

## Constrains
- Return only paths of the most pertinent files, each on a separate line, enclosed
  within three backticks (```)
- Enclose the explanation within three equal signs (===)

## Example
#### USER'S INPUT
Failing info: Error related to string manipulation
Bug-located code's packages and dependencies:
import File1
import File3
import File5
Source code structure:
.
├── File1.java
├── File2.java
├── File3.java
├── File4.java
├── File5.java
└── File6.java
6 files
#### AGENT'S OUTPUT
```
./File1.java
./File3.java
./File5.java
```
===
File1, File3, and File5 are imported in the bug-located code file
===
```

Figure 9: System Prompt of *RepoFocus*.

Summarizer

```
## Role
Skilled assistant for code summarization

## Skills
Proficient in complex Java code comprehension
Capable of detailing classes and functions
Skilled at analyzing and condensing function functionality

## Action
Review each class in the code
Identify attributes and functions within each class
Determine function names, parameter names, types, and return type
Analyze the purpose of each function
Write a concise description of each function

## Objective
Provide a summary for every class and function following this format:
```
<class name>~~~~<function name>~~~~<parameter name 1: parameter type
1, ...>~~~~<return type>~~~~<function description>
```


## Constraints
Ensure summaries accurately reflect the original code without irrelevant
information while adhering to format requirements, enclosed within three
backticks (```)
Each line represents one function inside a specific class

## Example
#### User's Input
```Java
public class GeometryUtils {
    public static double calculateCircleArea(double radius) {
        return Math.PI * radius * radius;
    }
    public static double calculateRectangleArea(double length, double width) {
        return length * width;
    }
}
```
Agent's Output
```
<GeometryUtils>~~~~<calculateCircleArea>~~~~<radius:
double>~~~~<double>~~~~<Calculates the area of a circle given its radius>
<GeometryUtils>~~~~<calculateRectangleArea>~~~~<length: double, width:
double>~~~~<double>~~~~<Calculates the area of a rectangle given its length and
width>
```
```

Figure 10: System Prompt of *Summarizer*.

A.4 A Demo of the Execution

In this section, we present an example of how UniDebugger solves a real-world bug on level-3 repair.

A.4.1 Bug Metadata

Upon receiving the necessary information about a bug, UniDebugger initializes agents for problem-solving. Here is an example bug named Lang-1, a real-world bug in Defects4J.

Using the JUnit framework, we can get detailed information upon failing test cases, as displayed in Figure 15.

Then, we can roughly locate the buggy code file "NumberUtils.java", whose code contents are show in Figure 16, as well as the failing oracles in testing code 17.

From the above information, we can see that the original code merely determines whether a value exceeds the ranges of Long and int based on the length of hexadecimal numbers. However, when the length of a hexadecimal number is 16, and the first significant digit is greater than 7, it actually goes beyond the range of Long. The original code fails to take this situation into account, and similar issues exist in the int range.

A.4.2 L3 Repair

For simplicity, we only show the responses from agents on level three of repair. *Helper* generates a short query summarizing the problem of this bug based on the testing information and buggy code, as shown in Figure 18. All the other agents can benefit from its generated debugging guide. Afterward, *RepoFocus* lists a list of bug-related files (19). Besides the bug-located file, it also identifies two other files. However, they would not influence the behavior of the number-creation logic unless you are encountering specific exception handling or Unicode string issues when parsing numeric strings. Subsequently, *Summarizer* generates a code summary for each identified file 20. Thus, we got all the information provided by upstream agents.

Slicer extracts 168 suspicious code lines from 1427 lines in the original buggy code, largely narrowing down the examination scope, as shown in Figure 21. *Locator* successfully pinpoints the root causes of this bug from the code lines sliced out 22. The following agents can focus on the single logic conditions.

However, *Fixer* failed to generate a plausible patch, as displayed in Figure 23. It attempts to fulfill the missing code block by counting the valid digits of the hexadecimal number but causes incorrect type determination when a large number of leading zeros are present in a hexadecimal string. The patched code also assumes the hex digits should be directly compared based on raw string length without adjusting for those leading zeros.

Slicer

```
## Role
Skilled assistant for detecting suspicious code snippets

## Skills
Proficient in navigating large projects
Identifies potential issues using common bug patterns and error information

## Action
Systematically examine project directories and files
Analyze code blocks for error-prone patterns
Track critical parameters against failing data
Extract suspicious snippets within the 20-100 line range

## Objective
Return suspicious snippets with a brief explanation, investigation, and repair suggestions

## Constraints
Do not alter original code; extract as-is within backticks (```)
Provide concise, relevant explanations with suggestions enclosed in equal signs (===)

## Example
#### User's Input
```Java
// ... A large Java project …
public class SomeComplexClass {
    public void performComplexTask() {
        int[] dataArray = new int[10];
        for (int i = 0; i < dataArray.length; i++) {
            dataArray[i] = i * 2;
        }
    }
    // ... rest of the class ...
    public static int calculateSum(int[] array) {
        int total = 0;
        for (int i = 0; i <= array.length; i++) {}
        return total;
    }
}
```
Failing info: Integer array [3, 4, 5] expected output is 12 but actual output is 0
Agent's Output
```Java
// ... rest of the class …
public static int calculateSum(int[] array) {
    int total = 0;
    for (int i = 0; i <= array.length; i++) {}
    return total;
}
```

===
Incorrect loop condition `i <= array.length`; should be `i < array.length` to prevent
index out-of-bounds errors. The absence of accumulation keeps `total` at zero
Investigate adding an operation like `total += array[I];`
===
```

Figure 11: System Prompt of *Slicer*.

FixerPro identified the causes of errors made by *Fixer*, and provides an optimized patch. The fixed version, presented in Figure 24, properly calculates the significant digits by counting the non-zero characters after the prefix and leading zeros. It also adjusts the comparisons for handling 16-digit and 8-digit boundary checks, ensuring that only significant digits are considered when deciding if the value is too large for “Integer” or “Long”.

Locator

```
## Role
Skilled assistant for bug localization

## Skills
Proficient in code analysis for bugs
Capable of marking buggy lines with comments
Adept at verifying functionality and input/output formats

## Action
Analyze code against failure info step by step
Track key variable values at critical points
Compare these values to expected outcomes, noting input/output format
Comment '//buggy line' on faulty lines; If missing code causes errors, add a comment
of '//missing code:[INFILLED CODE]'

## Objective
Return commented code segments (using '//buggy line' or '//missing code […]')
Provide a brief explanation on tracking variables for bug identification, followed by
fix advice

## Constraints
Mark code without altering layout/style, enclosing the code within backticks (```)
Only use specified comments ('//buggy line', '//missing code') to highlight issues
Explanation must be clear and relevant, including fix advice within ===

## Example
#### USER'S INPUT
```Java
public class ArraySumCalculator {
    // ... (rest of the class)
    public static int calculateSum(int[] array) {
        int total = 0;
        for (int i = 0; i <= array.length; i++) {
        }
        return total;
    }
}
```
Failing info: Input [3, 4, 5]; expected output is 12 but actual is 0.
AGENT'S OUTPUT
```Java
public class ArraySumCalculator {
// ... (rest of the class)
    public static int calculateSum(int[] array) {
        int total = 0;
        for (int i = 0; i <= array.length; i++) { // buggy line
            // missing code: total += array[I];
        }
        return total;
    }
}
```
===
The omission of 'total += array[i];' prevents accumulation–`total` remains zero
Loop condition `i <= array.length` should be `i < array.length` to avoid index out-
of-bounds error
===
```

Figure 12: System Prompt of *Locator*.

Fixer

```
## Role
Skilled assistant for program repair

## Skills
Proficient in identifying and analyzing code bugs
Capable of detecting real issues in marked lines
Skilled in generating git diff format patches

## Action
Analyze code with failing information step-by-step
Track critical parameters, comparing them to expected outcomes
Make minimal essential changes to faulty lines

## Objective
Provide a patch in git diff format
Deliver a brief explanation including: 1) Restatement of buggy lines, 2) Assessment of
necessary changes, 3) Identification of other required revisions and reasons, 4)
Description of changes (using `+` for additions against `-` for original), 5)
Rationale behind modifications guided by parameter tracking

## Constraints
Patch must be correct and enclosed within three backticks (```)
Explanation must be clear, relevant, and enclosed within three equal signs (===)

## Example
#### Users' Input
```Java
public class ArraySumCalculator {
    public static int calculateSum(int[] array) {
        int total = 0;
        for (int i = 0; i <= array.length; i++) { // buggy line
        }
        return total;
    }
}
```
Failing info: input [3,4,5], expected output is 12, but the code triggers
ArrayIndexOutOfBoundsException
Agents' Output
```
diff --git a/ArraySumCalculator.java b/ArraySumCalculator.java
index abc123..def4567b100644
-a/ArraySumCalculator.java
+++b/ArraySumCalculator.java
@@ -1,8 +1,9 @@
public class ArraySumCalculator {
    public static int calculateSum(int[] array){
        int total=0;
-       for(inti =0; i<=array.length;i++){
+       for(i=0;i<array.length;i++){
            total=array[I];
        }
        return total;
```
===
1) Restated Buggy Line: `for (int i = …`, 2) Assessment: The line needs fixing, 3)
Additional Revisions: Add `total += ...` inside the loop, 4) Changes Made: Replace `i
<= ...` with `<`, add accumulation statement, 5) Rationale: Fixes out-of-bounds error
and calculates sum correctly by adjusting the loop condition and adding missing logic
===
```

Figure 13: System Prompt of *Fixer*.

FixerPro

```
## Role
Expert in analyzing and correcting faulty patches

## Skills
Skilled at identifying issues in incorrect patches
Proficient in offering detailed repair suggestions and analyses
Excellent at creating accurate patches

## Action
Analyze buggy code and incorrect patch step-by-step using failure information
Identify errors in the original code and patch
Generate a corrected patch using `git diff`
Provide a comprehensive explanation of issues and your repair method.

## Objective
Return a correct patch in git diff format
Provide a brief explanation, including: 1) Causes of errors in the original code; 2) Reasons why the
given patch fails; 3) Why your patch effectively resolves the bug; 4) How tracking key parameter values
guided your revisions

## Constraints
Patch must be correctly formatted as git diff, enclosed within three backticks (```)
Explanation should clearly articulate problems with the existing patch and reasoning for the new fix,
enclosed within three equal signs (===)

## Example
#### USER'S INPUT
```Java
public class BuggyCode {
     public static int addNumbers(int a, int b) {
          int result = a - b;
          return result;
     }
}
```
Failing info: Input 2 and 3; expected output is 5 but actual output is -1
Wrong Patch:
```
diff --git a/BuggyCode.java b/FixedCode.java
index 83db48f..b02e56f 100644
--- a/BuggyCode.java
+++ b/FixedCode.java
@@ -1,6 +1,6 @@
public class BuggyCode {
     public static int addNumbers(int a, int b) {
-       int result = a - b;
+       int result = a * b;
          return result;
     }
}
```
AGENT'S OUTPUT
```Java
diff --git a/BuggyCode.java b/FixedCode.java
index 83db48f..b02e56f 100644
--- a/BuggyCode.java
+++ b/FixedCode.java
@@ -1,6 +1,6 @@
public class BuggyCode {
     public static int addNumbers(int a, int b) {
-       int result = a - b;
+       int result = a + b;
          return result;
     }
}
```
===
1) The bug exists because subtraction was used instead of addition for two numbers; 2) The wrong patch
changes subtraction to multiplication but does not perform addition as intended; 3) My correction
replaces `-` with `+`, fulfilling the requirement to sum two integers; 4) Key parameter is `result`,
which should be `5` for inputs `a=2` and `b=3`. My revision corrects this by changing from subtraction
to addition where needed (`int result = ...`)
===
```

Figure 14: System Prompt of *FixerPro*.

Failing test cases

```
--- org.apache.commons.lang3.math.NumberUtilsTest::TestLang747
java.lang.NumberFormatException: For input string: "80000000"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:583)
        at java.lang.Integer.valueOf(Integer.java:740)
        at java.lang.Integer.decode(Integer.java:1197)
        at org.apache.commons.lang3.math.NumberUtils.createInteger(NumberUtils.java:684)
        at org.apache.commons.lang3.math.NumberUtils.createNumber(NumberUtils.java:474)
        at org.apache.commons.lang3.math.NumberUtilsTest.TestLang747(NumberUtilsTest.java:256)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
        at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:63)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:236)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:53)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:229)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:309)
        at junit.framework.JUnit4TestAdapter.run(JUnit4TestAdapter.java:38)
        at org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner.run(JUnitTestRunner.java:520)
        at org.apache.tools.ant.taskdefs.optional.junit.JUnitTask.executeInVM(JUnitTask.java:1484)
        at org.apache.tools.ant.taskdefs.optional.junit.JUnitTask.execute(JUnitTask.java:872)
        at org.apache.tools.ant.taskdefs.optional.junit.JUnitTask.executeOrQueue(JUnitTask.java:1972)
        at org.apache.tools.ant.taskdefs.optional.junit.JUnitTask.execute1(JUnitTask.java:824)
        at org.apache.tools.ant.taskdefs.optional.junit.JUnitTask.execute(JUnitTask.java:2277)
        at org.apache.tools.ant.UnknownElement.execute(UnknownElement.java:291)
        at sun.reflect.GeneratedMethodAccessor4.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.apache.tools.ant.dispatch.DispatchUtils.execute(DispatchUtils.java:106)
        at org.apache.tools.ant.Task.perform(Task.java:348)
        at org.apache.tools.ant.Target.execute(Target.java:392)
        at org.apache.tools.ant.Target.performTasks(Target.java:413)
        at org.apache.tools.ant.Project.executeSortedTargets(Project.java:1399)
        at org.apache.tools.ant.Project.executeTarget(Project.java:1368)
        at org.apache.tools.ant.helper.DefaultExecutor.executeTargets(DefaultExecutor.java:41)
        at org.apache.tools.ant.Project.executeTargets(Project.java:1251)
        at org.apache.tools.ant.Main.runBuild(Main.java:811)
        at org.apache.tools.ant.Main.startAnt(Main.java:217)
        at org.apache.tools.ant.launch.Launcher.run(Launcher.java:280)
        at org.apache.tools.ant.launch.Launcher.main(Launcher.java:109)
```

Figure 15: Failing test cases reported by JUnit.

src/main/java/org/apache/commons/lang3/math/NumberUtils.java

```
package org.apache.commons.lang3.math;
import java.lang.reflect.Array;
import java.math.BigDecimal;
import java.math.BigInteger;
import org.apache.commons.lang3.StringUtils;

public class NumberUtils {
                                                        …
    if (pfxLen > 0) { // we have a hex number
            final int hexDigits = str.length() - pfxLen;
            if (hexDigits > 16) { // too many for Long
                return createBigInteger(str);
            }
            if (hexDigits > 8) { // too many for an int
                return createLong(str);
            }
            return createInteger(str);
     }
     final char lastChar = str.charAt(str.length() - 1);
     String mant;
     String dec;
     String exp;
     final int decPos = str.indexOf('.');
     final int expPos = str.indexOf('e') + str.indexOf('E') + 1; // assumes both not present
     // if both e and E are present, this is caught by the checks on expPos (which prevent IOOBE)
     // and the parsing which will detect if e or E appear in a number due to using the wrong offset
                                                        …
            return !allowSigns && foundDigit;
    }
}
```

Figure 16: Bug-located code snippet.

src/test/java/org/apache/commons/lang3/math/NumberUtilsTest.java

```
17 package org.apache.commons.lang3.math;

19 import static org.junit.Assert.assertEquals;
20 import static org.junit.Assert.assertFalse;
21 import static org.junit.Assert.assertNotNull;
                                                        …
37  public class NumberUtilsTest {
                                                        …
249    @Test
250    public void TestLang747() {
251        assertEquals(Integer.valueOf(0x8000),       NumberUtils.createNumber("0x8000"));
252        assertEquals(Integer.valueOf(0x80000),      NumberUtils.createNumber("0x80000"));
253        assertEquals(Integer.valueOf(0x800000),     NumberUtils.createNumber("0x800000"));
254        assertEquals(Integer.valueOf(0x8000000),    NumberUtils.createNumber("0x8000000"));
255        assertEquals(Integer.valueOf(0x7FFFFFFF),   NumberUtils.createNumber("0x7FFFFFFF"));
256        assertEquals(Long.valueOf(0x80000000L),     NumberUtils.createNumber("0x80000000"));
257        assertEquals(Long.valueOf(0xFFFFFFFFL),     NumberUtils.createNumber("0xFFFFFFFF"));
                                                        …
1401    }
1402  }
```

Figure 17: Testing code triggering errors.

L3: Response of Helper

Query

NumberFormatException for large hex number 0x80000000 in Java createNumber

Research

Debugging Guide

```
1. The error occurs because the string "80000000" is being parsed as an `int` but exceeds its limit.
When parsing numbers from strings, Java determines the number type based on the string length after
hex prefix like `0x`.
2. A solution is to ensure that large hex numbers are parsed as `long` or `BigInteger` when they
exceed the `int` range.
3. Modify the hex parsing logic in `createNumber` to correctly handle large hex values. Check the
digit count for determining the correct type, defaulting to `Long` or `BigInteger` when needed.
4. Test the fix with all size boundaries, especially around `int` and `long` limits.
===
https://stackoverflow.com/questions/5499523/how-to-avoid-number-format-exception-in-java
https://www.geeksforgeeks.org/numberformatexception-in-java-with-examples/
https://www.delftstack.com/howto/java/java-numberformatexception-for-input-string/
https://bootcamptoprod.com/java-numberformatexception/
===
```

Accessed by other agents

Figure 18: Response and tool usage of *Helper*.

L3: Response of RepoFocus

```
./math/NumberUtils.java
./exception/ExceptionUtils.java
./text/translate/UnicodeEscaper.java
```

Passed to Summarizer

```
===
The selected files are considered suspicious because they are directly related to the packages and
dependencies used in the bug-located code file:
1. **NumberUtils.java** in the `math` package is directly imported in the bug-located code file.
2. **ExceptionUtils.java** in the `exception` package might contain relevant error-handling
mechanisms.
3. **UnicodeEscaper.java** in the `text.translate` package could be relevant due to the bug's
relation to string manipulation.
===
```

Figure 19: Response of *RepoFocus*.

L3: Formalized response of Summarizer

```
{
    "./exception/ExceptionUtils.java": {
        "ExceptionUtils": {
            "ExceptionUtils": "void\tPublic constructor for ExceptionUtils",
            "getCause": {
                "desp": "Introspects the Throwable to obtain the cause with custom method names",
                "paras": {
                    "methodNames": "String[]",
                    "throwable": "Throwable"
                },
                "return_type": "Throwable"
            },
            "getCauseUsingMethodName": {
                "desp": "Finds a Throwable by method name",
                "paras": {
                    "methodName": "String",
                    "throwable": "Throwable"
                },
                "return_type": "Throwable"
            },
                                                    …
        }
    }
}
```

Figure 20: Response of *Summarizer*.

L3: Response of Slicer

```
    public static Number createNumber(final String str) throws NumberFormatException {
                                                    …
                pfxLen += pfx.length();
                break;                                      Slice 168 lines out of the
            }                                               original 1427-line code snippet
        }
        if (pfxLen > 0) {
            final int hexDigits = str.length() - pfxLen;
            if (hexDigits > 16) {
                return createBigInteger(str);
            }
            if (hexDigits > 8) {
                return createLong(str);
            }
            return createInteger(str);
        }
        final char lastChar = str.charAt(str.length() - 1);
        String mant;
                                        … (168 lines in total)
===
1. The `createNumber` method is responsible for converting a string value into a `java.lang.Number`
object, handling various formats and qualifiers.
2. The code snippet is suspicious because it contains complex logic for identifying different number
types (integer, long, float, double, etc.) based on the input string.
3. The method includes handling hex prefixes like "0x", decimal points, exponential notation, and
trailing characters indicating number type (e.g., 'f' for Float, 'L' for Long, etc.).
4. The logic appears intricate and error-prone, especially with multiple conditional checks and
potential parsing issues for different number representations.
5. Further investigation should focus on simplifying the logic, improving readability, and ensuring
accurate handling of various number formats to avoid potential bugs and number format exceptions.
===
```

Figure 21: Response of *Slicer*.

L3: Response of Locator

```
    public static Number createNumber(final String str) throws NumberFormatException {
                                                    …
                break;
            }
        }
        if (pfxLen > 0) {
            // missing code: counting the significant digits by skipping leading zeros in a
            // hexadecimal number.
            final int hexDigits = str.length() - pfxLen;
            if (hexDigits > 16) { // buggy line
                return createBigInteger(str); // buggy line
            } // buggy line
            if (hexDigits > 8) { // buggy line
                return createLong(str); // buggy line        Mark the position to fill the
            } // buggy line                                   missing code block and the
            return createInteger(str);                       buggy lines need to be modified
        }
        final char lastChar = str.charAt(str.length() - 1);
        String mant;
                                        … (168 lines in total)
===
By tracking variable values such as `hexDigits` and interpreting conditions for hexadecimal input
handling for values "0x80000000", we identify where decisions between `Integer` and `Long` or
`BigInteger` are made. The original conditions after computing `hexDigits` did not check whether the
characters after the prefix are actually valid hexadecimal digits. Without validation, strings that
don't correspond to valid hexadecimal numbers could still be processed, leading to potential
misinterpretation or NumberFormatException being thrown later.
===
```

Figure 22: Response of *Locator*.

L3: Response of Fixer

```
--- Lang_1_buggy.java
+++ Lang_1_fixed.java
@@ -464,11 +464,20 @@ public class NumberUtils {
        if (pfxLen > 0) { // we have a hex number
+           boolean foundFirstNonZero = false;
+           int leadingZeroCount = 0;
+           for(int i = pfxLen; i < str.length(); i++) {
+               char currentChar = str.charAt(i);
+               if (!foundFirstNonZero && currentChar == '0') {
+                   leadingZeroCount++;
+               } else if (!foundFirstNonZero && currentChar!= '0') {
+                   foundFirstNonZero = true;
+               }
+           }
            final int hexDigits = str.length() - pfxLen;
-           if (hexDigits > 16) { // too many for Long
+           if (hexDigits > 16 || (hexDigits == 16 && (!foundFirstNonZero || str.charAt(pfxLen +
15) > '7'))) { // too many for Long
                return createBigInteger(str);
            }
-           if (hexDigits > 8) { // too many for an int
+           if (hexDigits > 8 || (hexDigits == 8 && (!foundFirstNonZero || str.charAt(pfxLen + 7) >
'7'))) { // too many for an int
                return createLong(str);
            }
            return createInteger(str);
===
1) Restating:
- Missing code: `// missing code: counting the significant digits by skipping leading zeros in a
  hexadecimal number.`
- Buggy lines:`if (hexDigits > 16) { // too many for Long` and `if (hexDigits > 8) { // too many for
  an int`
2) Assessment: The marked lines are indeed necessary to be corrected.
3) No other lines appear to need modification since the changes directly impact the processing of hex
numbers, which is a specific section of the function.
4) The patch introduces these key changes:
- Skipping leading zeros: A loop counts and skips leading zeros to ensure only significant digits are
processed.
- Boundary checks: It ensures numbers exceeding `int` or `long` limits are handled as BigInteger by
checking if the 8th or 16th digit exceeds '7'.
5) Reason for changes:
- Skipping zeros: Ensures correct digit count, preventing invalid conversions.
- Boundary checks: Prevents overflow by correctly handling numbers beyond `int` and `long` limits.
```

Figure 23: Response of *Fixer*.

L3: Response of FixerPro

```
--- Lang_1_buggy.java
+++ Lang_1_fixed.java
@@ -464,11 +464,23 @@ public class NumberUtils {
        if (pfxLen > 0) { // we have a hex number
+           boolean foundFirstNonZero = false;
+           int leadingZeroCount = 0;
+           int significantHexDigits = 0;
+           for(int i = pfxLen; i < str.length(); i++) {
+               char currentChar = str.charAt(i);
+               if (!foundFirstNonZero && currentChar == '0') {
+                   leadingZeroCount++;
+               } else if (!foundFirstNonZero && currentChar != '0') {
+                   foundFirstNonZero = true;
+               }
+               if (foundFirstNonZero) {
+                   significantHexDigits++;
+               }
+           }
            final int hexDigits = significantHexDigits;
-           if (hexDigits > 16) { // too many for Long
+           if (hexDigits > 16 || (hexDigits == 16 && (!foundFirstNonZero || str.charAt(pfxLen +
leadingZeroCount + 15) > '7'))) { // too many for Long
                return createBigInteger(str);
            }
-           if (hexDigits > 8) { // too many for an int
+           if (hexDigits > 8 || (hexDigits == 8 && (!foundFirstNonZero || str.charAt(pfxLen +
leadingZeroCount + 7) > '7'))) { // too many for an int
                return createLong(str);
            }
            return createInteger(str);
===
Causes of the errors:
The original code doesn't account for leading zeros in hexadecimal numbers, leading to incorrect
type determination based on raw string length, rather than significant digits.
Why the provided patch cannot fully fix the buggy code:
The patch counts leading zeros but still uses the raw string length to calculate hexDigits, which
misrepresents the number's actual size.
Why this patch fixes the bug:
The updated patch correctly counts significant digits after leading zeros, ensuring type selection
(Integer, Long, BigInteger) is based on the actual number size.
Tracking key parameter values:
By counting both leading zeros and significant digits, the patch makes the logic more accurate,
handling edge cases like 0x00000000123 and avoiding errors from untrimmed input.
```

Figure 24: Response of *FixerPro*.