AEGIS: An Agent-based Framework for Bug Reproduction from Issue Descriptions

Xinchen Wang[1*], Pengfei Gao[2], Xiangxin Meng[2], Ruida Hu[1*], Chao Peng[2†], Yun Lin[3], Cuiyun Gao[1†]
[1] Harbin Institute of Technology, Shenzhen, China
[2] ByteDance, Beijing, China
[3] Shanghai Jiao Tong University, Shanghai, China
200111115@stu.hit.edu.cn,gaopengfei.se@bytedance.com,mengxiangxin.1219@bytedance.com
pengchao.x@bytedance.com,200111107@stu.hit.edu.cn,lin_yun@sjtu.edu.cn,gaocuiyun@hit.edu.cn

Abstract

Automatically reproducing bugs in issue descriptions helps developers pinpoint and fix bugs timely, greatly facilitating the software development and maintenance. Built upon the powerful understanding capabilities of large language models (LLMs), agent-based approaches have achieved the state-of-the-art performance in the task. They generally leverage LLMs as the central controller to first retrieve bug-related information as context and then generate bug reproduction scripts. During the script modification process, agent-based approaches modify the script iteratively until the execution information reflects the bug accurately or the iterative turns are exhausted. Nevertheless, the agent-based approaches still face the following challenges: **(1) Lengthy retrieved bug-related information:** The retrieved bug-related information is usually long in length and contains irrelevant snippets, which is hard to be well comprehended by LLMs. **(2) Lack of guidance in bug reproduction script generation:** They generally modify bug reproduction scripts randomly and tend to generate repeated or spurious modifications, leading to bug reproduction failure.

To address the above challenges, in this paper, we propose an automated bug reproduction script generation framework named **AEGIS**. AEGIS consists of two main modules: **(1) Bug-related context summarization module**, aiming at condensing the retrieved information into structural context through further reranking and summarization. **(2) Finite state machine (FSM)-guided script generation module**, which aims at guiding the script modification process with proposed FSM which contains predefined modification rules. Extensive experiments on SWE-Bench, one public benchmark dataset, and six baseline methods show that AEGIS achieves the best performance in the task, exceeding the best baseline by 19.0% with respect to the bug reproduction rate. Besides, we deploy AEGIS in five internal repositories of ByteDance. During the three-month deployment period, it successfully reproduces 12 bugs and assists developers in implementing fixes.

[*] Work done during an internship at ByteDance.
[†] Corresponding authors.

FSE 2025, June 23–27, 2025, Trondheim, Norway

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

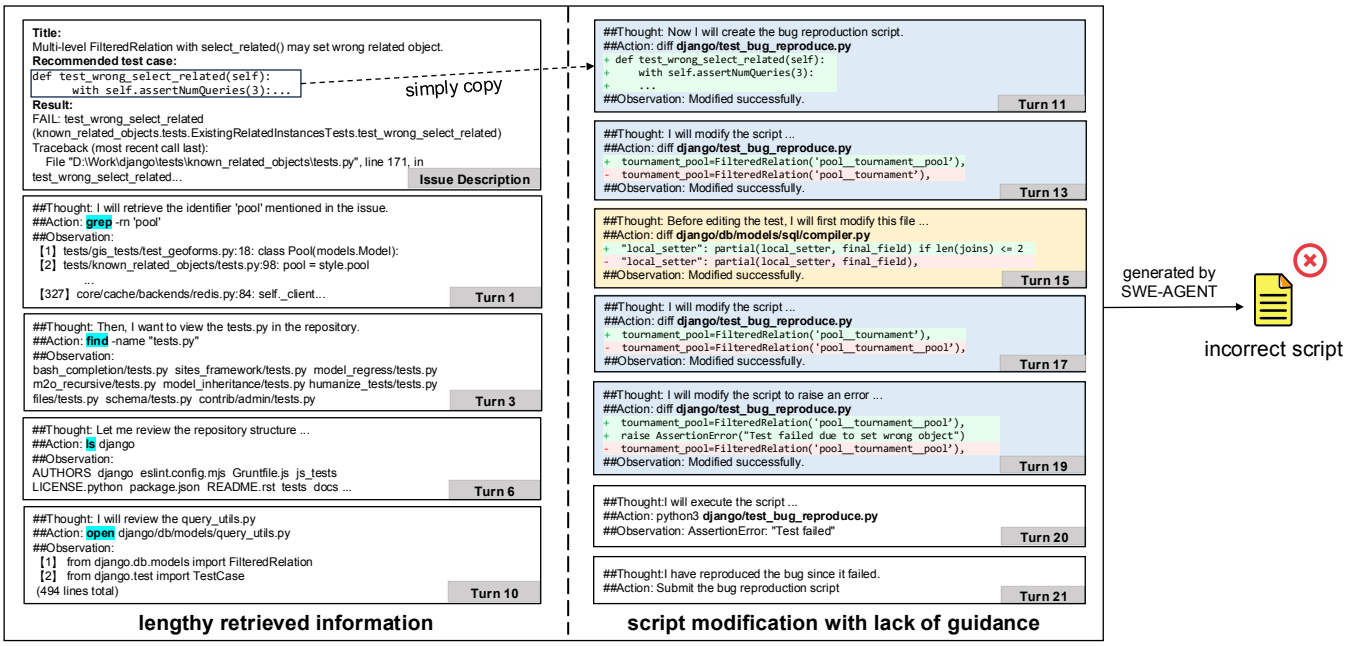
• **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
Xinchen Wang[1*], Pengfei Gao[2], Xiangxin Meng[2], Ruida Hu[1*], Chao Peng[2†], Yun Lin[3], Cuiyun Gao[1†]. 2024. AEGIS: An Agent-based Framework for Bug Reproduction from Issue Descriptions. In *Companion Proceedings of the 33rd ACM Symposium on the Foundations of Software Engineering (FSE '25), June 23–27, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

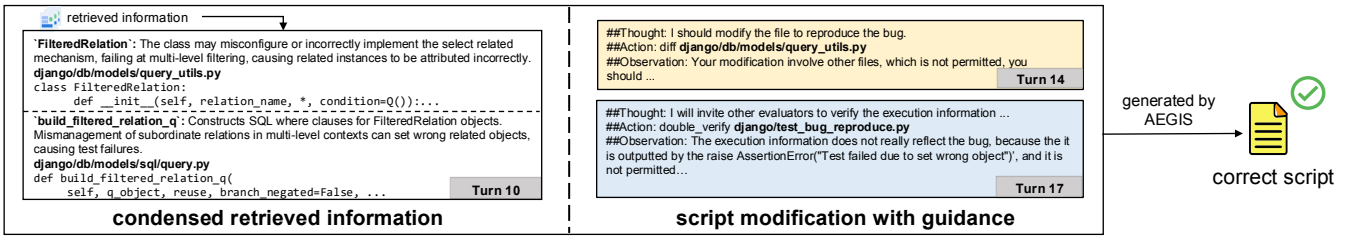
1 INTRODUCTION

Reproducing bugs in issue descriptions is essential to localize bugs and implement corresponding fixes. Existing surveys [9, 22] indicate that writing bug reproduction scripts closely aligns with developers' needs, as these scripts assist developers in comprehending the bugs and preventing their recurrence. Kang et al.[21] conduct a study on 300 popular Java projects and find that up to 28% of the test cases are related to bug reproduction. Besides, the execution information [23] obtained from running these scripts aids in more precise bug localization, thereby improving automated debugging tools' [25, 46] performance. These findings imply that writing bug reproduction scripts to verify bug fixes has become a standard workflow for developers and can greatly enhance software development and maintenance efficiency.

Manually writing bug reproduction scripts is time-consuming and automated bug reproduction approaches have garnered increasing attention. Previous bug reproduction approaches [6, 11, 13, 32, 38, 47] focus primarily on reproducing crashes, which represent only a subset of bugs, and these approaches struggle to comprehend the semantic information in issue descriptions. Given the powerful text understanding and generation capabilities of large language models (LLMs), LLM-based approaches achieve promising results in the bug reproduction task [21]. However, they can only generate scripts in a single attempt and cannot interactively modify them based on execution information, leading to limited performance [21]. In contrast, agent-based approaches [5, 28, 40, 48], which can interact with the environment by invoking external tools, have shown great potential in bug fixing. The initial and critical step in these approaches is bug reproduction. They leverage LLMs as the central controller to first retrieve bug-related information as context and then generate bug reproduction scripts. During the script modification process, agent-based approaches make iterative modifications until the execution information accurately reflects the bug or the iterative turns are exhausted. These agent-based



(a) Multi-turn interactions of SWE-AGENT during the bug reproduction process.



(b) Multi-turn interactions of AEGIS during the bug reproduction process.

Figure 1: Comparison of bug reproduction between using SWE-AGENT and AEGIS: the issue description in (a) includes the bug that needs to be reproduced and the recommended test case. The number of each turn represents the relative order of interaction turns. The actions marked in cyan are the LLM agent's interaction with the environment. Blue boxes represent the modifications of the reproduction script, while yellow boxes indicate the modifications of other functional files. The dashed arrow indicates that SWE-AGENT simply copies the code snippet in the issue description.

approaches demonstrate the state-of-the-art performance in the bug reproduction task [31].

However, existing agent-based approaches [5, 40, 48] still face the following challenges: **(1) Lengthy retrieved bug-related information:** Agents retrieve bug-related information to localize and comprehend bugs. However, the retrieved information is usually lengthy and contains numerous irrelevant snippets, increasing agents' contextual understanding burden [24]. **(2) Lack of guidance in bug reproduction script generation:** During the script modification process, agents generally modify bug reproduction scripts randomly and tend to make repeated or spurious modifications. These modifications may deviate the script reproduction process [26], leading to bug reproduction failure.

To address the above challenges, we propose an **A**gent-based fram**E**work for **G**enerating bug reproduct**I**on **S**cripts from issue descriptions, named **AEGIS**. AEGIS consists of two main modules: **(1) Bug-related context summarization module**: We first retrieve bug-related information from the code repository. Then we condense the retrieved information into the structural context through further reranking and summarization. **(2) Finite state machine (FSM)-guided script generation module**: We propose an FSM that contains predefined modification rules, aiming at well guiding the script modification process. Extensive experiments on the SWE-Bench benchmark [19], one public benchmark, and six baseline methods demonstrate the superior performance of AEGIS in the task, exceeding the best baseline by 19.0% with respect to the bug reproduction rate. In addition, AEGIS has been deployed in five internal repositories of ByteDance for three months. During this period, AEGIS generates bug reproduction scripts for emerging bugs to assist developers in debugging, further illustrating its effectiveness in practice.

In summary, the major contributions of this paper are summarized as follows:

(1) We propose AEGIS, a novel agent-based bug reproduction framework, which involves a bug-related context summarization module for constructing a condensed and structural bug-related context and an FSM-guided script generation module for guiding the script modification process under predefined modification rules utilizing the proposed FSM.
(2) We conduct extensive experiments on AEGIS and the results demonstrate the effectiveness of AEGIS in bug reproduction.
(3) We deploy AEGIS in internal repositories of ByteDance to assist developers in localizing and fixing bugs timely, demonstrating its effectiveness in practice.

The remaining sections of this paper are organized as follows. Section 2 reveals agent-based approaches' challenges in the bug reproduction task. Section 3 presents the architecture of AEGIS. Section 4 describes the experimental setup, including datasets, baselines, and experimental settings. Section 5 presents the experimental results and analysis. Section 6 further discusses AEGIS's effectiveness in bug reproduction and bug fixing, as well as the threats to validity. Section 7 introduces the background of automatic bug reproduction and LLM-based agent. Section 8 concludes the paper.

2 MOTIVATION

In this section, we explore the challenges of agent-based approaches by analyzing the failure case of SWE-AGENT [48], one advanced agent-based approach for the bug reproduction task. Agent-based approaches utilize LLMs to conduct the multi-turn interactions with the environment. During each interaction, the LLM agent responds with "thought" and "action", respectively, where "thought" represents its reasoning analysis of the environment feedback and "action" represents the command to be executed. The actions are executed within the environment, and the corresponding environment feedback is collected as input (*i.e.*, "observation") to the LLM agent in the next interaction.

Figure 1 (a) illustrates the bug reproduction process by SWE-AGENT for the issue "*django-16408*"[1]. The issue description includes the title, the recommended test case, and the result. SWE-AGENT needs to leverage this information to generate the bug reproduction script. In turns 1, 3, 6, and 10, SWE-AGENT invokes the "`grep -r`" command (returning 327 lines), calls "`find`" to search test files, uses "`ls`" to observe the repository structure, and browses bug-related files (494 lines), respectively. SWE-AGENT examines the repository structure first, looking for files related to bug localization before attempting to reproduce the bug. However, the retrieved information is often lengthy and contains irrelevant snippets, increasing the agent's contextual burden.

Although SWE-AGENT observes the repository and retrieves a great amount of information between turn 1 and turn 10, it fails to utilize this information for bug reproduction. It merely copies the recommended test case from the issue description in turn 11 and ignores the initialization of the involved classes. This indicates that the lengthy retrieved information is difficult for LLMs to comprehend and utilize effectively, leading to the previous searches being unproductive.

[1] https://code.djangoproject.com/ticket/34227

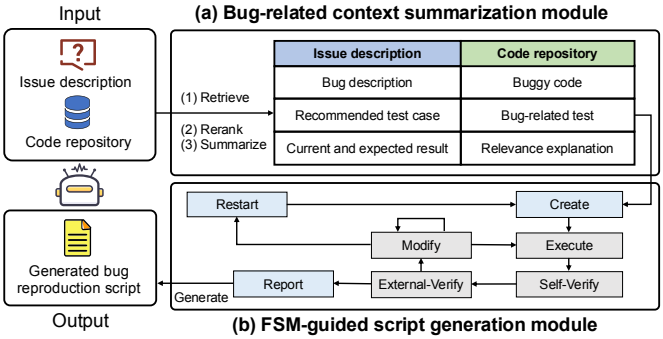


Figure 2: The architecture of AEGIS. It consists of two modules: (a) a bug-related context summarization module for condensing the retrieved bug-related information into structural context and (b) an FSM-guided script generation module for guiding the script modification process based on predefined modification rules.

From turn 13 to turn 21, SWE-AGENT randomly modifies the bug reproduction script. In turn 13 and turn 17, SWE-AGENT makes repeated modifications, resulting in unproductive iteration turns. In turn 15, SWE-AGENT modifies the functional file, which is not allowed as it violates the original functionality. In turn 19, SWE-AGENT adds the "`raise AssertionError ('Test failed due to set wrong object')`" statement in the reproduction script to spuriously "reproduce" the same bug. This indicates that SWE-AGENT struggles to modify the bug reproduction script effectively and tends to make repeated and spurious modifications, leading to bug reproduction failure.

From the above case, we demonstrate that agent-based approaches often retrieve lengthy bug-related information, which makes it difficult to comprehend and utilize the relevant details. Besides, they tend to generate repeated or spurious modifications due to the lack of guidance in bug reproduction script generation. Therefore, in this paper, we propose a novel agent-based framework to mitigate these challenges and improve the performance in the bug reproduction task.

Figure 1 (b) illustrates the bug reproduction process by AEGIS. Based on the retrieved bug-related information, the LLM agent reranks the code snippets and summarizes information according to their relevance with the bug in turn 10, producing condensed information. This condensed information displays methods, classes, and files, allowing AEGIS to avoid interference from lengthy and irrelevant information. In turn 14, the LLM agent attempts to modify functional files but is constrained by our predefined modification rules. In turn 17, we introduce external verification to offer guidance and avoid spurious modifications. Eventually, the LLM agent successfully reproduces the bug.

3 PROPOSED FRAMEWORK

In this section, we introduce the general framework of AEGIS. As shown in Figure 2, AEGIS consists of two main modules: (1) Bug-related context summarization module, which condenses the retrieved bug-related information into structural context through reranking and summarization, (2) Finite state machine (FSM)-guided

script generation module, which guides the script modification process utilizing proposed FSM and predefined modification rules. Given the issue description and code repository as inputs, the output of AEGIS is the bug reproduction script.

3.1 Bug-related Context Summarization Module

In this module, AEGIS employs an LLM agent to retrieve, rerank, and summarize bug-related context based on the issue description and code repository.

Retrieve: We employ the Abstract Syntax Tree (AST) to parse the files in the code repository and extract their code structures, including methods, classes, and identifiers. To facilitate fine-grained code retrieval, we design a set of search tools including "`search_method`", "`search_class`", and "`search_identifier`".

When the LLM agent invokes these search tools, search requests are processed locally based on the parsed files, and the results are then fed back to the LLM agent as the retrieved context. Besides these search tools, the LLM agent can also utilize bash commands like "`ls`", "`grep`", and "`find`".

Rerank: The retrieved context C_{retrieve} consists of numerous code snippets. Let $C_{\text{retrieve}} = \{c_1, c_2, \ldots, c_n\}$ be the set of retrieved code snippets. The LLM agent reranks these snippets according to their relevance to the issue description and repeats this process m times independently. This results in rankings $R_1, R_2, \ldots, R_m$. Each R_i is a vector representing the rank positions of code snippets:

$$R_i = (r_{i1}, r_{i2}, \ldots, r_{in}) \quad \text{for } i = 1, 2, \ldots, m \tag{1}$$

Here, r_{ij} denotes the rank position of the j-th code snippet in the i-th ranking, where r_{ij} is an integer from 1 to n. The average ranking R_{avg} is computed using the equation:

$$R_{\text{avg}} = \left(\frac{1}{m}\sum_{i=1}^{m} r_{i1}, \frac{1}{m}\sum_{i=1}^{m} r_{i2}, \ldots, \frac{1}{m}\sum_{i=1}^{m} r_{in} \right) \tag{2}$$

Then, we obtain the reranked context C_{rerank} by sorting the code snippets according to R_{avg}:

$$C_{\text{rerank}} = \text{Sort}(C_{\text{retrieve}}, R_{\text{avg}}) \tag{3}$$

Summarize: Based on the reranked context C_{rerank}, the LLM agent filters out lengthy and irrelevant code snippets and condenses the remaining snippets along with the issue description to obtain the following summarized bug-related context:

(1) From the issue description: ① Bug description: A clear and detailed description of the bug, which helps the LLM agent comprehend the issue. ② Recommended test case: If the issue description already contains the recommended test case, the LLM agent can leverage this information to help generate the reproduction script. ③ Current result and expected result: Analysis of the current result caused by the bug and the expected result after fixing the bug, which helps verify whether the execution information reflects the bug accurately.

(2) From the code repository: ① Buggy code: Code snippets that are likely to cause the bug. ② Bug-related test: Test cases that can be reused directly or are relevant for reproducing the bug. ③ Relevance explanation: The detailed explanation of the relevance between the buggy code, bug-related test, and the bug. Such explanations help LLM agents better leverage these code snippets for bug reproduction.

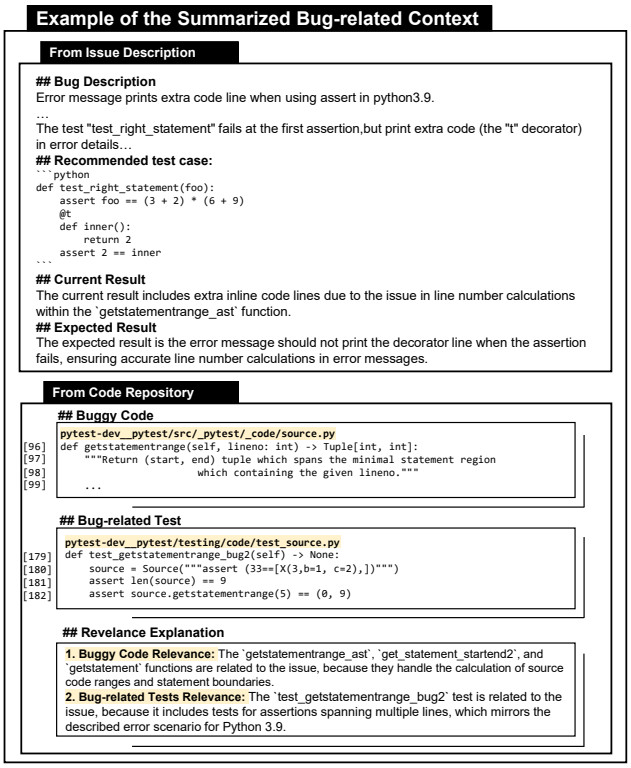


Figure 3: The example of the summarized bug-related context, including the bug-related information from the issue description and the code repository.

Figure 3 shows an example of the summarized bug-related context after retrieving, reranking, and summarizing. This structural context serves as the input for the next module.

3.2 FSM-guided Script Generation Module

In this module, we propose a finite state machine (FSM) to guide LLM agents in generating bug reproduction scripts. We first introduce the complete formulation of our proposed FSM, then describe the process of FSM-guided script modification.

3.2.1 Formulation of the proposed FSM. FSM [3] is a computational model representing a process with a finite number of states and the transitions between these states. It can guide and constrain the actions of the LLM agent, helping to prevent the LLM agent from generating repeated or spurious modifications. As illustrated in Figure 4, our proposed FSM is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, q_4)$, comprising the following components:

(1) State Set $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ denotes the different states of the LLM agent during the script modification process, which are *Create*, *Execute*, *Self-Verify*, *External-Verify*, *Report*, *Modify*, and *Restart*, respectively.

(2) Input Condition $\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l\}$ represents the conditions that the LLM agent satisfies. Specifically, once the LLM agent takes an action, one certain condition will be satisfied.

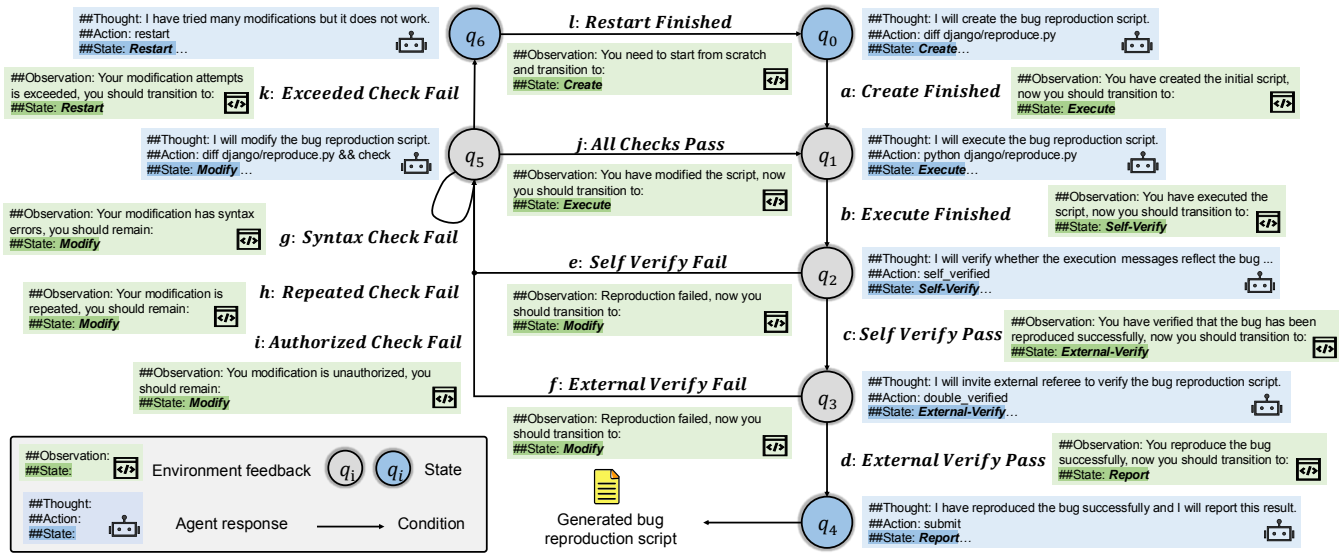


Figure 4: The proposed finite state machine (FSM). Each circle represents a state, with the blue circle q_0 indicating the initial state, q_4 indicating the final state, and q_6 indicating the restart state. Each green box and blue box represents environment feedback and agent response, respectively, with The dark background text indicating the current state. Specifically, in the state q_5, when any of the conditions g, h, or i is satisfied, the LLM agent will stay in the current state.

The conditions we define include *Create Finished*, *Execute Finished*, *Self Verify Pass*, etc.

(3) Initial State q_0 (*Create*) represents the starting point where the LLM agent begins to create the reproduction script based on the summarized bug-related context.

(4) Accepting State q_4 (*Report*) indicates the point at which the LLM agent successfully reproduces the bug and reports success. It is the final state of the FSM.

(5) Transition function $\delta = Q \times \Sigma \rightarrow Q$ describes how the LLM agent transitions between different states when certain conditions are satisfied. The detailed transition function of the FSM is as follows:

At state q_0 (*Create*)

$\delta(q_0, a) = q_1$: In state q_0, the LLM agent creates the initial script. Once the script is created, the condition a (*Create Finished*) is satisfied, and the LLM agent transitions its state to q_1 (*Execute*).

At state q_1 (*Execute*)

$\delta(q_1, b) = q_2$: In state q_1, the LLM agent executes the script and obtains the execution information. After that, the condition b (*Execute Finished*) is satisfied and the LLM agent transitions its state to q_2 (*Self-Verify*).

At state q_2 (*Self-Verify*)

In state q_2, the LLM agent verifies the execution information from q_2 to determine whether it accurately reflects the bug.

$\delta(q_2, c) = q_3$: If the bug is accurately reflected, the condition c (*Self Verify Pass*) is satisfied, and the LLM agent transitions its state to q_3 (*External-Verify*).

$\delta(q_2, e) = q_5$: Otherwise, the LLM agent should explain the reasons for bug reproduction failure. In this case, the condition e (*Self Verify Fail*) is satisfied and the LLM agent transitions its state to q_5 (*Modify*).

At state q_3 (*External-Verify*)

In state q_3, we leverage another independent LLM agent to act as an external judger. Given the current bug reproduction script from q_0, the corresponding execution information from q_1, and the summarized bug-related context from 3.1, this external judger determines whether the execution information reflects the bug accurately.

$\delta(q_3, d) = q_4$: If the external judger believes that the reproduction script accurately reflects the bug, the condition d (*External Verify Pass*) is satisfied, and the LLM agent transitions its state to the accepting state q_4 (*Report*).

$\delta(q_3, f) = q_5$: Otherwise, the external judger should explain the reasons for bug reproduction failure. In this case, the condition g (*External Verify Fail*) is satisfied, and the LLM agent transitions its state to q_5 (*Modify*).

At state q_4 (*Report*)

In state q_4, the LLM agent submits the generated bug reproduction script and reports success to indicate that it successfully reproduces the bug.

At state q_5 (*Modify*)

In state q_5, the LLM agent analyses the explanations for bug reproduction failure from q_2 or q_3, or the checking reports from q_5 itself, and provides the proposed modification. Before applying the modification, we perform the exceeded,

syntax, repeated, and authorized checks. If all of them pass, the modification is applied.

$\delta(q_5, k) = q_6$: For the exceeded check, we count the number of applied modifications in the modification history. If the number reaches the predefined limit, the condition k (*Exceed Check Fail*) is satisfied and the LLM agent transitions its state to q_6 (*Restart*).

$\delta(q_5, g) = q_5$: For the syntax check, we leverage syntax-checking tools to verify the correctness of the proposed modification's syntax. If it contains syntax errors, the condition i (*Syntax Check Fail*) is satisfied. Thus, the LLM agent remains in its state at q_5. The tools also provide a detailed checking report, including the error code and the types of errors.

$\delta(q_5, h) = q_5$: For the repeated check, we evaluate whether the proposed modification results in the same script as any previous version in the modification history. If so, the condition h (*Repeated Check Fail*) is satisfied and the LLM agent remains in its state at q_5. We further provide the LLM agent with a checking report, which clarifies the reasons for considering the modification as repeated. The report also includes a consolidated context of the modification history.

$\delta(q_5, i) = q_5$: For the authorized check, we evaluate whether the LLM agent attempts to modify files in the original code repository instead of the generated script. If so, the condition i (*Authorized Check Fail*) is satisfied and the LLM agent remains in its state at q_5. We further provide the LLM agent with a checking report, which clarifies the reasons for considering the modification as unauthorized.

$\delta(q_5, j) = q_1$: If the proposed modification passes all four checks above, we apply it to the current script. Hence, the condition j (*All Checks Pass*) is satisfied and the LLM agent transitions its state to q_1 (*Execute*).

At state q_6 (*Restart*)

$\delta(q_6, l) = q_0$: In state q_6, the LLM agent summarizes the explanation for the bug reproduction failure from states q_2 and q_3. Besides, we clear the current script and its modification history, preparing the LLM agent to restart and create a new script. In this case, the condition l (*Restart Finished*) is satisfied. Therefore, the LLM agent transitions its state to q_0 (*Create*).

3.2.2 FSM-guided script modification process. Figure 4 illustrates the script modification process guided and constrained by our proposed FSM. In each interaction with the environment, the LLM agent's response includes thoughts, actions, and the current state. We extract actions and the current state from LLM agent's response and then execute actions in the environment. Based on the current state and conditions satisfied by executing the actions, we guide the LLM agent using environment feedback.

This feedback includes the "observation" to explain satisfied conditions and provide additional information, such as script execution details, explanations for bug reproduction failure, and checking reports on failed modification checks. It also includes the "state", which details the next state and required actions in the next state.

Based on environment feedback, the LLM agent transitions between different states and iteratively modifies the script. To prevent the script modification process from becoming an endless loop, we limit the maximum number of restarts. If the number of restarts reaches this limit, the LLM agent stops and outputs the current bug reproduction script.

4 EXPERIMENTAL SETUP

In this section, we evaluate AEGIS and aim to answer the following research questions (RQs):

- **RQ1:** How does AEGIS perform in the bug reproduction task compared with different methods?
- **RQ2:** What is the impact of different modules on the performance of AEGIS?
- **RQ3:** How do the different hyper-parameters impact the performance of AEGIS?

4.1 Dataset

To answer the questions above, we utilize the popular SWE-Bench [19] dataset, which is designed to assess the capability of addressing software engineering bugs. For faster and more cost-effective evaluation, we focus on a refined subset called SWE-Bench Lite. This subset comprises 300 instances from SWE-Bench that have been sampled to be more self-contained, ensuring a comparable range and distribution of projects as the full dataset. Each instance in SWE-Bench Lite includes an issue description of a bug, the corresponding code repository containing the bug, and the patch to fix the bug. Our goal is to generate bug reproduction scripts for each instance in SWE-Bench Lite.

4.2 Comparative Methods

To evaluate the performance of our framework, we compare two types of bug reproduction methods: LLM-based and agent-based.

4.2.1 LLM-based methods.

- **ZEROSHOT** [31] prompts the LLM with the issue description, bug-related context retrieved using BM25 [37], and instructions to generate scripts in unified diff format.
- **ZEROSHOTPLUS** [31] is similar to ZEROSHOT but leverages adjusted diff format, which allows entire functions or classes to be inserted, replaced, or deleted.
- **LIBRO** [21] generates multiple candidate scripts based on the issue description. It then executes all the generated scripts and selects the one whose execution information most accurately reflects the bug.

4.2.2 Agent-based methods.

- **SWE-AGENT** [48] comprises several principal components, including search, file viewer, file editor, and context management. It is employed to fix bugs, and bug reproduction is one of its stages.
- **AUTOCODEROVER** [51] consists of two distinct stages. In the first stage, it is tasked with retrieving bug-related code snippets. Then AUTOCODEROVER generates patches based on the issue description and the retrieved context, retrying until the patch is successfully applied.

Table 1: Comparison results between AEGIS, the LLM-based methods, and the agent-based methods.

	Method	$F \rightarrow \times$	$P \rightarrow P$	$F \rightarrow P$
LLM-based	ZEROSHOT	38.8	3.6	5.8
	ZEROSHOTPLUS	55.4	7.2	10.1
	LIBRO	60.1	7.2	15.2
Agent-based	AUTOCODEROVER	43.8	7.6	9.1
	AIDER	57.6	8.7	17.0
	SWE-AGENT	48.2	9.8	16.7
	AEGIS	90.0	9.0	**36.0**

- **AIDER** [1] includes a repository indexing step to guide file selection and proposes modifications in an edit block format. Before a modification is applied, it undergoes validation via syntax-checking tools.

4.3 Implementation Details

AEGIS is provided access to GPT4o-2024-0513 [33] with the sampling temperature set to 0.7. We limit the maximum number of applied modifications per restart to 5 and allow up to 5 restarts during the bug reproduction script generation process. Besides, we construct environment-completed Docker images for each instance in the SWE-bench Lite. For the comparison methods, we use their default settings [31].

4.4 Evaluation Metrics

We leverage the bug reproduction rate to measure AEGIS's performance.

Bug Reproduction Rate ($F \rightarrow P$): Following the prior study [31], we consider a script to successfully reproduce the bug described in the issue if it fails on the original code repository (*i.e.*, before the patch is applied) but passes on the patched repository (*i.e.*, after the patch is applied). We call this a fail-to-pass script. Hence, the bug reproduction rate ($F \rightarrow P$) measures the portion of instances where the generated script is a fail-to-pass script. Besides, we consider a script as a fail-to-any script if it fails on the original repository, and as a pass-to-pass script if it passes on both the original and patched repositories. We further measure the fail-to-any rate ($F \rightarrow \times$) and the pass-to-pass rate ($P \rightarrow P$) for a more comprehensive analysis.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness of AEGIS in Bug Reproduction

To answer RQ1, we conduct a comprehensive comparative analysis against three LLM-based methods and three agent-based methods. The experimental results are shown in Table 1.

AEGIS exhibits superior performance compared with the baseline methods. As shown in Table 1, AEGIS outperforms all the baseline methods in terms of bug reproduction rate ($F \rightarrow P$). Specifically, AEGIS achieves an absolute improvement of 19.0% over the best baseline method. Compared to the average performance of compared methods, AEGIS demonstrates an absolute improvement of 23.7%. This is due to the ability of AEGIS to better leverage bug-related information and guide the script modification process. Overall, AEGIS can reproduce more bugs successfully, showcasing its effectiveness.

Table 2: Impact of the bug-related context summarization module (*i.e.* BCS) and the FSM-guided script generation module (*i.e.* FSG) on the performance of AEGIS.

Module	$F \rightarrow \times$	$P \rightarrow P$	$F \rightarrow P$
w/o BCS	90.7	8.0	31.7 ↓ 4.3
w/o FSG	56.7	27.0	12.3 ↓ 23.7
w/o Modify-in-FSG	84.7	14.0	26.0 ↓ 10.0
w/o Restart-in-FSG	92.0	7.7	33.3 ↓ 2.7
w/o External-Verify-in-FSG	92.7	6.7	35.0 ↓ 1.0
AEGIS	90.0	9.0	36.0

Fail-to-any scripts matter for reproducing more bugs. When considering the best-performing methods in the LLM-based and agent-based baselines, LIBRO and AIDER, we find that these two methods achieve the highest $F \rightarrow \times$ metrics, 60.1% and 57.6% respectively. Additionally, we observe that AEGIS achieves a much higher rate in the $F \rightarrow \times$ metric, reaching 90.0%, whereas the best baseline method only reaches 57.6%. Under the constraints of our proposed FSM and modification rules, the LLM agent aims to make the bug reproduction script fail in the original code repository. Although fail-to-any scripts are not always desirable, they are essential for generating fail-to-pass scripts, as pass-to-any scripts can never reproduce the bug. When considering the $P \rightarrow P$ metric, AEGIS and agent-based methods show little difference, indicating that some bugs are difficult to reproduce solely based on the issue description, resulting in unhelpful scripts (*i.e.*, pass-to-pass scripts).

> **Answer to RQ1:** AEGIS achieves the best performance in the bug reproduction task, exceeding the best baseline by 19.0% on the bug reproduction rate.

5.2 RQ2: Effectiveness of Different Modules in AEGIS

To answer RQ2, we explore the effectiveness of different modules on the performance of AEGIS. Specifically, we study the two involved modules: the bug-related context summarization module (BCS) and the FSM-guided script generation module (FSG).

5.2.1 Bug-related Context Summarization Module. To understand the impact of this module, we deploy a variant of AEGIS without the bug-related context summarization module (*i.e.*, w/o BCS). This variant generates the bug reproduction script based on the issue description and the retrieved context without reranking and summarizing. Table 2 shows the performance of this variant. Adding

the bug-related context summarization module yields a 4.3% enhancement in the bug reproduction rate ($F \rightarrow P$). Overall, the results indicate that the bug-related context summarization module enables the LLM agent to focus on a condensed context, thereby improving its comprehension of the bug and better leveraging the retrieved information.

5.2.2 FSM-guided Script Generation Module. To explore the contribution of this module, we also construct a variant of AEGIS without the FSM-guided script generation module (*i.e.*, w/o FSG). This variant modifies the bug reproduction script without the guidance of the proposed FSM. As illustrated in Table 2, adding the FSM-guided script generation module yields a 23.7% enhancement in the bug reproduction rate ($F \rightarrow P$). This enhancement can be attributed to the module's capability to guide the LLM agent in avoiding repeated and spurious modifications, thus leading to a constrained script modification process. We also observe that this variant shows a great decrease in the $F \rightarrow \times$ metric to 56.7% and a notable increase in the $P \rightarrow P$ metric to 27.0%. This indicates that without guidance, the LLM agent is more likely to generate unhelpful scripts.

5.2.3 Different States in the Proposed FSM. To further investigate the influence of different states within the proposed FSM, we design three variants: one without Modify-in-FSG, one without Restart-in-FSG, and one without External-Verify-in-FSG. These variants represent generating scripts in a single attempt without modification, continuously modifying scripts until reporting success or exceeding the maximum iterations, and verifying scripts' execution information without external verification, respectively. Table 2 shows the performances of three variants. The states of *Modify*, *Restart*, and *External-Verify* contribute to enhancements in the bug reproduction rate ($F \rightarrow P$) by 10.0%, 2.7%, and 1.0%, respectively. The *Modify* state shows a notable improvement, suggesting that AEGIS can iteratively optimize the reproduction scripts based on execution information. The *Restart* state improves AEGIS's performance by allowing the exploration of diverse reproduction paths and avoiding endless modifications. The *External-Verify* state enhances AEGIS's performance by providing external oversight, which helps prevent the LLM agent from being misled by spurious modifications.

Answer to RQ2: Both BSG and FSG modules can improve the performance of AEGIS. The BSG module boosts the bug reproduction rate of 4.3% and the FSG module enhances AEGIS by 23.7%. Besides, different states within the proposed FSM are essential to the bug reproduction.

5.3 RQ3: Influence of Hyper-parameters on the Performance of AEGIS

To answer RQ3, we explore the impact of different hyper-parameters, including the maximum number of restarts and the maximum number of applied modifications per restart during the script modification process.

5.3.1 Maximum number of restarts. Figure 5 shows the performance of AEGIS with different maximum numbers of restarts. As the maximum number of restarts increases from 1 to 5, the bug reproduction rate ($F \rightarrow P$) improves from 30.7% to 33.7%. This suggests that more restarts allow LLM agents to explore diverse bug reproduction paths and accumulate more failure experiences. However, as the maximum number of restarts increases from 7 to 9, the bug reproduction rate stabilizes, indicating that further increases have a limited impact on performance. This is likely because the LLM agents have already tried all possible reproduction paths and cannot derive additional insights from the failure experiences. Considering the balance between resource consumption and performance, we choose 5 as the optimal number of restarts.

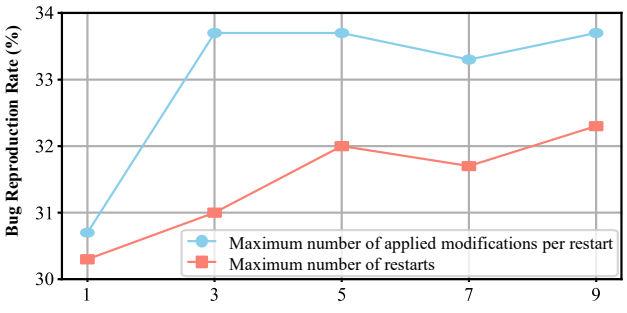


Figure 5: The influence of the maximum number of applied modifications per restart and maximum number of restarts for AEGIS. The horizontal axis represents the number of modifications per restart or the number of restarts.

5.3.2 Maximum number of applied modifications per restart. Similarly, as the maximum number of applied modifications per restart increases from 1 to 5, the bug reproduction rate ($F \rightarrow P$) rises from 30.0% to 32.0%. This demonstrates that LLM agents can effectively leverage guidance from the proposed FSM to refine the reproduction script. However, as the maximum number of modifications further increases from 5 to 9, the bug reproduction rate tends to stabilize, indicating that further modifications become increasingly challenging. This may be due to the involvement of multiple methods and classes in reproducing bugs. To balance resource consumption and performance, we select 5 as the optimal number of applied modifications per restart.

Answer to RQ3: The performance of AEGIS is influenced by the maximum number of restarts and the maximum number of applied modifications per restart. Our default settings yield optimal results.

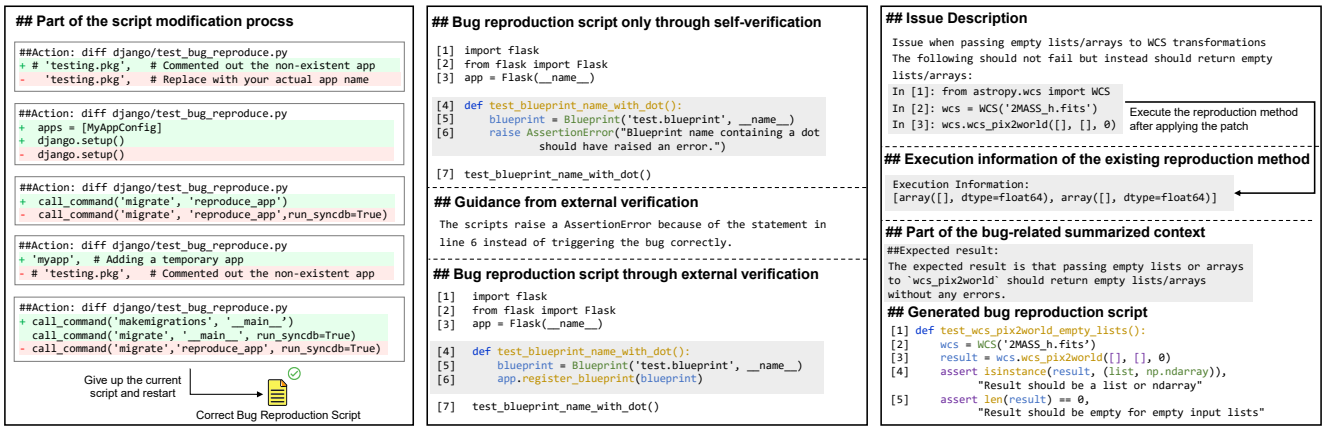
6 DISCUSSION

6.1 Case Study

To further demonstrate the effectiveness of AEGIS in the bug reproduction task, we analyse three cases from SWE-Bench. The results are shown in Figure 6.

Figure 6 (a) shows the bug reproduction process for the issue *"django-11964"* [2]. The LLM agent encounters difficulties in the script

[2] https://code.djangoproject.com/ticket/30902



(a) An example from django-11964, illustrating avoiding endless modifications through restart.

(b) An example from flask-4045, illustrating correcting spurious modifications through external verification.

(c) An example from astropy-7746, illustrating generating script based on the bug-related summarized context.

Figure 6: Examples for illustrating the effectiveness of AEGIS.

configuration and becomes stuck in endless modifications. Our proposed FSM limits the number of modifications per restart, allowing the LLM agent to restart and recreate the reproduction script. After restarting, the LLM agent adopts an alternative configuration method and successfully generates the correct bug reproduction script. This example demonstrates that the restart helps avoid endless modifications and enables the LLM agent to explore diverse bug reproduction paths.

Figure 6 (b) shows the bug reproduction process for the issue *"flask-4045"* [3]. The script verified only by the LLM agent fails to reproduce the bug because it directly uses the "`raise AssertionError`" statement to output the bug described in the issue. In our designed FSM, the bug reproduction script and its execution information undergo additional external verification. This external verification provides guidance, indicating that "the scripts raise an AssertionError because of the statement in line 6 instead of triggering the bug correctly". Based on the guidance from external verification, the LLM agent modifies the script by employing the "`app.register_blueprint(blueprint)`" statement to trigger the bug. This example shows that the external verification in our proposed FSM can prevent the LLM agent from being misled by spurious modifications.

Figure 6 (c) shows the bug reproduction process for the issue *"astropy-7746"* [4]. The issue description provides a bug reproduction method. Based on the constructed bug-related summarized context, the LLM agent expects the "`wcs.wcs_pix2world`" method to return empty lists or arrays when given empty inputs. It uses "`assert isinstance()`" and "`assert len()`" to validate the type and length of the method's return values. However, this bug reproduction script triggers assert errors in both the original and patched code repositories. When we execute the bug reproduction method provided by the issue description in the patched code repository, we find that the execution information consists of "`[array([], dtype=float64), array([], dtype=float64)]`" instead of empty lists or arrays. This indicates that the issue creator does not accurately describe the expected behaviour of the reproduction method. Although AEGIS can meticulously generate bug reproduction script, such inappropriate issue descriptions are beyond its understanding capabilities.

[3] https://github.com/pallets/flask/issues/4041
[4] https://github.com/astropy/astropy/issues/7389

Table 3: Effectiveness of AEGIS on bug fixing, including the fix rates and the number of fixed bugs.

Method	Fix Rate (%)	# of Fixed Bugs
Agentless	27.3	82
+ w/ LLM Voting	23.3 ↓ 4.0	70 ↓ 12
+ w/ AEGIS	30.7 ↑ 3.4	92 ↑ 10

6.2 AEGIS's Effect on Bug Fixing

We explore the impact of AEGIS in the entire bug fixing pipeline. Specifically, we apply AEGIS to the advanced bug fixing approach, Agentless [44], and compare the performance before and after applying AEGIS. Besides, we deploy AEGIS in internal repositories of ByteDance and help developers in localizing and fixing bugs.

For the Agentless approach, we leverage the bug reproduction scripts to filter its generated patches. We prioritize patches that cause the bug reproduction script to fail before applying the patch and pass afterwards, as such patches are likely to fix the bug. If such patches do not exist, we then select those that produce different execution information before and after applying the patch, as such patches may alter the execution path related to the bug. Besides, we explore the effectiveness of leveraging LLM to select the most suitable patch by providing the issue description and generated patches. This approach is referred to as LLM Voting. As shown in Table 3, leveraging the bug reproduction scripts generated by AEGIS improves the performance of Agentless. Specifically, the number of fixed bugs increases from 82 to 92, with the fix rate rising from 27.3% to 30.7%, showing a 12.5% relative improvement. This result demonstrates the effectiveness of AEGIS in bug fixing. However, LLM Voting results in a 4.0% decrease, as it is challenging for the LLM to select the correct patch without execution information.

Table 4: Impact of different search tools to AEGIS.

Method	$F \rightarrow \times$	$P \rightarrow P$	$F \rightarrow P$
AEGIS	90.0	9.0	36.0
AEGIS-BM25	93.0	6.0	34.7 ↓ 1.3
AEGIS-BashCommand	91.0	8.3	34.0 ↓ 2.0

We deploy AEGIS in five internal repositories of ByteDance, covering multiple programming languages and involving over 200 developers. Other staff of ByteDance report issues in the issue tracking system when they encounter bugs in these repositories, and ask developers for solutions. During the three-month deployment period, AEGIS generates bug reproduction scripts for 25 bugs. Each bug reproduction script is evaluated by three developers of its corresponding repository, and 12 of these scripts accurately reproduce the bugs in the staff's issue descriptions. The execution information from these scripts helps developers localize the bugs and implement fixes. Overall, AEGIS proves to be effective in facilitating bug fixing in industrial practice.

6.3 Impact of Searching Tools on the Performance of AEGIS

In this section, we design two variants to explore the impact of different searching tools on AEGIS's performance: AEGIS-BM25 and AEGIS-BashCommand. The AEGIS-BM25 variant leverages BM25 [37] to obtain bug-related information from the code repository based on the issue description, and the AEGIS-BashCommand variant can only retrieve context through invoking bash commands.

As illustrated in Table 4, AEGIS achieves a slight enhancement of 1.3% and 2.0% in the bug reproduction rate ($F \rightarrow P$) compared to AEGIS-BM25 and AEGIS-BugCommand, respectively. Both variants show notable improvements over state-of-the-art bug reproduction methods. This result indicates that AEGIS exhibits a weak dependency on the capabilities of the searching tools, demonstrating robustness and stability.

6.4 Threats and Limitations

Dataset Validity Concerns: SWE-Bench Lite's scope is limited because it is derived from only 12 popular repositories, and may not cover all issue types and testing scenarios. Hence, the experimental results based on this dataset may not be fully generalizable. In future research, we intend to extend our investigations to more repositories.

Result Variability: Due to AEGIS relying on LLM agents for context retrieval and bug reproduction script generation, the experimental results exhibit variability. We therefore conduct multiple trials and average the results to obtain a more stable measure.

7 RELATED WORK

7.1 Automatic Bug Reproduction

Automatically reproducing bugs from issue descriptions helps developers localize and fix bugs in a timely manner, greatly enhancing software development efficiency. Prior bug reproduction approaches [6, 32, 38, 47] have explored reproducing specific types of bugs, such as those in Android applications [14, 17, 41], configuration-triggered bugs [15], and program crashes [11, 13, 52]. However, these bugs represent only a small subset of all bugs.

Given the strong understanding and generation capabilities of LLMs, LIBRO [21] utilizes LLMs to reproduce bugs from issue descriptions. It leverages LLMs to generate bug reproduction scripts and employs post-processing to select promising scripts. However, LIBRO cannot dynamically modify reproduction scripts based on execution information, resulting in scripts that often lack dependency statements and include incorrect assertions, thereby limiting its performance. Recently, agent-based approaches [5, 28, 48] have shown great potential in bug reproduction. These approaches first retrieve bug-related information as context and then generate bug reproduction scripts through iterative modifications until the execution information accurately reflects the bug.

7.2 LLM-based Agent

AI agents are artificial entities capable of autonomously perceiving the environment and taking action to achieve specific goals [27, 43]. The rapid advancements in LLMs have greatly increased researchers' interest in LLM-based agents [7, 34]. These agents enhance LLMs by integrating external resources and tools, thereby enabling them to address more complex real-world challenges. LLM-based agents specifically designed for software engineering have demonstrated substantial potential across a variety of software development and maintenance tasks, including requirements engineering [2, 20], code generation [16, 18], static bug detection [12, 30], code review [36, 39], unit testing [8, 50], system testing [10, 49], fault localization [35, 42], program repair [4, 45], end-to-end software development [48], and end-to-end software maintenance [29].

Recently, LLM-based agents such as CodeR [5], MASAI [40], and SWE-AGENT [48] have shown great potential in the bug fixing task. These agents consider bug reproduction as part of the overall pipeline. However, they still face challenges such as handling lengthy retrieved bug-related information and lacking guidance in bug reproduction script generation. In this paper, we introduce AEGIS, an agent-based framework for bug reproduction from issue descriptions, aimed at addressing these challenges and improving the bug reproduction rate.

8 CONCLUSION

This paper focuses on the bug reproduction task and proposes a novel agent-based framework, named AEGIS. AEGIS consists of a bug-related context summarization module for condensing the retrieved information into structural context through reranking and summarization and an FSM-guided script generation module for guiding the script modification process with the proposed FSM which contains predefined modification rules. Compared with the state-of-the-art methods, the experimental results validate the effectiveness of AEGIS. Besides, we deploy AEGIS in five internal repositories of ByteDance. During the three-month deployment period, it successfully reproduces 12 bugs and helps developers implement corresponding fixes. In the future, we intend to further evaluate AEGIS on a broader range of datasets for bug reproduction.

References

[1] Aider-AI. 2024. "aider". https://github.com/Aider-AI/aider.

[2] Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2024. Advancing requirements engineering through generative ai: Assessing the role of llms. In *Generative AI for Effective Software Development*. Springer, 129–148.

[3] Mordechai Ben-Ari, Francesco Mondada, Mordechai Ben-Ari, and Francesco Mondada. 2018. Finite state machines. *Elements of Robotics* (2018), 55–61.

[4] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *CoRR* abs/2403.17134 (2024). https://doi.org/10.48550/ARXIV.2403.17134 arXiv:2403.17134

[5] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *CoRR* abs/2406.01304 (2024). https://doi.org/10.48550/ARXIV.2406.01304 arXiv:2406.01304

[6] Ning Chen and Sunghun Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Trans. Software Eng.* 41, 2 (2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469

[7] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=EHg5GDnyq1

[8] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. 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*, Marcelo d'Amorim (Ed.). ACM, 572–576. https://doi.org/10.1145/3663529.3663801

[9] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. IEEE Computer Society, 201–211. https://doi.org/10.1109/ISSRE.2014.11

[10] Gelei Deng, Yi Liu, Victor Mayoral Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2023. PentestGPT: An LLM-empowered Automatic Penetration Testing Tool. *CoRR* abs/2308.06782 (2023). https://doi.org/10.48550/ARXIV.2308.06782 arXiv:2308.06782

[11] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2020. Botsing, a Search-based Crash Reproduction Framework for Java. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1278–1282. https://doi.org/10.1145/3324884.3415299

[12] Gang Fan, Xiaoheng Xie, Xunjin Zheng, Yinan Liang, and Peng Di. 2023. Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents. *CoRR* abs/2310.08837 (2023). https://doi.org/10.48550/ARXIV.2310.08837 arXiv:2310.08837

[13] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 141–152. https://doi.org/10.1145/3213846.3213869

[14] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 67:1–67:13. https://doi.org/10.1145/3597503.3608137

[15] Ying Fu, Teng Wang, Shanshan Li, Jinyan Ding, Shulin Zhou, Zhouyang Jia, Wang Li, Yu Jiang, and Xiangke Liao. 2024. MissConf: LLM-Enhanced Reproduction of Configuration-Triggered Bugs. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 484–495. https://doi.org/10.1145/3639478.3647635

[16] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. *CoRR* abs/2312.13010 (2023). https://doi.org/10.48550/ARXIV.2312.13010 arXiv:2312.13010

[17] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. CrashTranslator: Automatically Reproducing Mobile Application Crashes Directly from Stack Trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 18:1–18:13. https://doi.org/10.1145/3597503.3623298

[18] Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization. *CoRR* abs/2404.02183 (2024). https://doi.org/10.48550/ARXIV.2404.02183 arXiv:2404.02183

[19] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=VTF8yNQM66

[20] Dongming Jin, Zhi Jin, Xiaohong Chen, and Chunhui Wang. 2024. MARE: Multi-Agents Collaboration Framework for Requirements Engineering. *CoRR* abs/2405.03256 (2024). https://doi.org/10.48550/ARXIV.2405.03256 arXiv:2405.03256

[21] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[22] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 61–70. https://doi.org/10.1109/ICSE-SEIP.2019.00015

[23] Tien-Duy B. Le, Richard Jayadi Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 579–590. https://doi.org/10.1145/2786805.2786880

[24] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context LLMs Struggle with Long In-context Learning. arXiv:2404.02060 [cs.CL] https://arxiv.org/abs/2404.02060

[25] 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*, Dongmei Zhang and Anders Møller (Eds.). ACM, 169–180. https://doi.org/10.1145/3293882.3330574

[26] Zelong Li, Wenyue Hua, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Formal-LLM: Integrating Formal Language and Natural Language for Controllable LLM-based Agents. arXiv:2402.00798 [cs.LG] https://arxiv.org/abs/2402.00798

[27] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *CoRR* abs/2409.02977 (2024). https://doi.org/10.48550/ARXIV.2409.02977 arXiv:2409.02977

[28] Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024. MarsCode Agent: AI-native Automated Bug Fixing. arXiv:2409.00899 [cs.SE] https://arxiv.org/abs/2409.00899

[29] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? *CoRR* abs/2406.01422 (2024). https://doi.org/10.48550/ARXIV.2406.01422 arXiv:2406.01422

[30] Zhenyu Mao, Jialong Li, Dongming Jin, Munan Li, and Kenji Tei. 2024. Multi-Role Consensus Through LLMs Discussions for Vulnerability Detection. In *24th IEEE International Conference on Software Quality, Reliability, and Security, QRS - Companion, Cambridge, United Kingdom, July 1-5, 2024*. IEEE, 1318–1319. https://doi.org/10.1109/QRS-C63300.2024.00173

[31] Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. 2024. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=9Y8zUO11EQ

[32] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. 2015. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (Eds.). IEEE Computer Society, 101–110. https://doi.org/10.1109/SANER.2015.7081820

[33] OpenAI. 2024. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/.

[34] Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Eleanor Jiang, Chengfei Lv, and Huajun Chen. 2024. AutoAct: Automatic Agent Learning from Scratch for QA via Self-Planning. 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*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 3003–3021. https://aclanthology.org/2024.acl-long.165

[35] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. AgentFL: Scaling LLM-based Fault Localization to Project-Level Context. *CoRR* abs/2403.16362 (2024). https://doi.org/10.48550/ARXIV.2403.16362 arXiv:2403.16362

[36] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. AI-powered Code Review with LLMs: Early Results. *CoRR* abs/2404.18496 (2024). https://doi.org/10.48550/ARXIV.2404.18496 arXiv:2404.18496

[37] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (April 2009), 333–389.

https://doi.org/10.1561/1500000019

[38] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. 2018. Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 325–340. https://doi.org/10.1007/978-3-319-99241-9_18

[39] Daniel Tang, Zhenghan Chen, Kisub Kim, Yewei Song, Haoye Tian, Saad Ezzini, Yongfeng Huang, Jacques Klein, and Tegawendé F. Bissyandé. 2024. CodeAgent: Collaborative Agents for Software Engineering. *CoRR* abs/2402.02172 (2024). https://doi.org/10.48550/ARXIV.2402.02172 arXiv:2402.02172

[40] Nalin Wadhwa, Atharv Sonwane, Daman Arora, Abhav Mehrotra, Saiteja Utpala, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. [n. d.]. MASAI: Modular Architecture for Software-engineering AI Agents. In *NeurIPS 2024 Workshop on Open-World Agents*.

[41] Dingbang Wang, Yu Zhao, Sidong Feng, Zhaoxu Zhang, William G. J. Halfond, Chunyang Chen, Xiaoxia Sun, Jiangfan Shi, and Tingting Yu. 2024. Feedback-Driven Automated Whole Bug Report Reproduction for Android Apps. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1048–1060. https://doi.org/10.1145/3650212.3680341

[42] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management, CIKM 2024, Boise, ID, USA, October 21-25, 2024*, Edoardo Serra and Francesca Spezzano (Eds.). ACM, 4966–4974. https://doi.org/10.1145/3627673.3680016

[43] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. 2023. The Rise and Potential of Large Language Model Based Agents: A Survey. *CoRR* abs/2309.07864 (2023). https://doi.org/10.48550/ARXIV.2309.07864 arXiv:2309.07864

[44] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *CoRR* abs/2407.01489 (2024). https://doi.org/10.48550/ARXIV.2407.01489 arXiv:2407.01489

[45] 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* (Vienna, Austria) *(ISSTA 2024)*. ACM, New York, NY, USA, 819–831. https://doi.org/10.1145/3650212.3680323

[46] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 416–426. https://doi.org/10.1109/ICSE.2017.45

[47] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 910–913. https://doi.org/10.1145/2786805.2803206

[48] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *CoRR* abs/2405.15793 (2024). https://doi.org/10.48550/ARXIV.2405.15793 arXiv:2405.15793

[49] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. *CoRR* abs/2311.08649 (2023). https://doi.org/10.48550/ARXIV.2311.08649 arXiv:2311.08649

[50] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1703–1726. https://doi.org/10.1145/3660783

[51] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1592–1604. https://doi.org/10.1145/3650212.3680384

[52] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 128–139. https://doi.org/10.1109/ICSE.2019.00030