Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 15, Number 1, pages 12–18, January 2026

Introspective Self-Dialogue: Iterative APR with CoT and Error Logs

1st Keisuke Kitamura
Graduate School of Science and Engineering
Doshisha University
Kyoto, Japan
ctwn0119@mail4.doshisha.ac.jp

2nd So Onishi
Graduate School of Science and Engineering
Doshisha University
Kyoto, Japan
ctwk0154@mail4.doshisha.ac.jp

3rd Akihito Kohiga
Graduate School of Science and Engineering
Doshisha University
Kyoto, Japan
akohiga@mail.doshisha.ac.jp

4th Takahiro Koita
Graduate School of Science and Engineering
Doshisha University
Kyoto, Japan
tkoita@mail.doshisha.ac.jp

***Abstract*—In recent years, Automated Program Repair (APR) using Large Language Model (LLM) has been anticipated as a technology contributing to development process. Previous researches are advancing on interactive program repairing that incorporate test result feedback. However, it has been pointed out that simply repeating iterations that only communicate the fact of failure prevents LLM from understanding the root cause of errors, causing them to repeat the same mistakes.**

This research proposes a novel correction approach called "introspective self-dialogue." This approach integrates the LLM's own "internal information" (Chain-of-Thought) with "external information" (error logs) to enable the LLM to self-analyze the cause of failure. The goal is to prompt deeper self-analysis and more accurate re-corrections by having the LLM confront its own reasoning against objective facts. Our evaluation demonstrates whether the proposed method improves bug-fixing success rates compared to conventional one-off correction techniques. Furthermore, we comprehensively verify its effectiveness and safety from more practical perspectives, such as unintended effects and quality changes introduced by the corrected code.

***Index Terms*—Automated Program Repair (APR), large language model (LLM), Self-Correction.**

I. Introduction

In recent years, Large Language Model (LLM) have experienced rapid adoption alongside significant technological advancements. Particularly since the launch of OpenAI's ChatGPT, their accessibility, requiring no specialized knowledge, has led to widespread adoption across various scenarios.

One field significantly impacted by LLM is Automated Program Repair (APR). APR has become an increasingly common approach for debugging, a notoriously time-consuming aspect of software development, and bug fix rates have improved compared to the pre-LLM era.

As an approach to further enhance LLM-APR performance, methods providing **external feedback** (e.g., test execution results or error logs) to the LLM for iterative correction are being researched. However, it has been noted that simple iterations merely conveying failure (external information) are often insufficient for the LLM to understand the error's root cause, causing the model to repeat similar mistakes. [1]

We posit that eliciting the corrective capabilities of LLM more stably requires leveraging not only such external information but also **internal information** to engage the LLM's intrinsic **Self-Correction** capabilities.

A. LLM (Large Language Model)

LLM (Large Language Model) is a type of language model that utilizes machine learning to understand and generate natural language. A language model models the probability of word occurrences in text data and is used in natural language processing tasks, such as text generation, by predicting the probability of which word will follow a given word. Among these, LLM are characterized by the massive scale of the three factors that significantly influence performance in natural language models: "compute," "data volume," and "model parameter count" [2]. This massive scaling has endowed LLM with advanced contextual understanding capabilities and general-purpose task-processing abilities.

B. Prompt Engineering

Prompt engineering refers to techniques for designing and optimizing input prompts to efficiently utilize the capabilities of LLM [3]. It is known that the output of an LLM can be significantly affected by slight differences in the input prompt, and this technology has developed as a method for designing and optimizing prompts to elicit desired responses from the LLM. Major techniques include "Zero-Shot Prompting," which gives instructions without task examples (shots), and "Few-Shot Prompting," which provides several examples before giving the task instruction.

Of particular importance to this research is **Chain-of-Thought (CoT) Prompting** [3]. CoT is a method that prompts the LLM to describe its problem-solving process step-by-step. By having the LLM verbalize its "chain of thought," similar

to how a human solves a problem, rather than just providing the final answer, it has been shown that LLM can achieve high performance on complex logical reasoning tasks.

Furthermore, the latest generation of models, such as `gpt-5` [4], are categorized as "Reasoning Models" [5]. These models are reportedly trained to internally generate a CoT to analyze and plan a task before generating a response, even without explicit prompting [5].

CoT holds an important aspect beyond mere accuracy improvement: the "visualization of the LLM's thinking." Because the logic by which the LLM reached its conclusion remains as text, even if the LLM fails a correction, analyzing that `cot` (thinking process) log can provide clues to the root cause of "why it failed."

C. APR (Automated Program Repair)

APR (Automated Program Repair) is a technology that automatically detects and corrects bugs in program code. In the software development process, debugging (identifying and fixing bugs) is one of the most time- and cost-intensive tasks, with some reports indicating that debugging accounts for the majority of software development costs [6]. APR has been gaining attention since the 2010s as a tool to reduce development costs by automating program correction.

While many APR techniques were proposed, their repair capabilities were limited. However, as LLM technology advanced, its sophisticated natural language processing capabilities were applied to APR, leading to the proposal of LLM-APR. LLM-APR has succeeded in further improving the bug fixing capabilities of APR by applying the LLM's natural language processing power to the analysis of error logs and other inputs.

II. Problem (Degradation of code quality)

While LLM-APR demonstrates high bug fixing capabilities, it has also introduced a new, significant risk: **Degradation**, also commonly referred to as "Regression."

Degradation is a phenomenon where a code change intended to fix one bug **unintentionally causes adverse effects on other functionalities**, introducing new bugs or breaking existing, correct functions.

LLM are adept at optimizing a fix based on localized information provided in the prompt (such as the buggy code or error logs). However, the LLM does not fully comprehend the global context of the entire codebase, such as the project's overall design philosophy or complex dependencies with other modules.

As a result, a patch generated by the LLM may be correct for fixing the immediate bug, but this change might break the operational assumptions of another file in the project, thereby causing a new bug.

This problem of degradation is one of the barriers to the practical implementation of APR. If automated correction introduces new bugs, developers must spend additional debugging costs to find and fix these (unexpected) secondary bugs, which directly contradicts the goal of APR (cost reduction).

Therefore, in this study, we judged it essential to introduce metrics (PMD, Checkstyle) that measure "whether code quality has deteriorated (i.e., caused degradation)" as **primary evaluation metrics**, rather than relying solely on the "bug fix success rate," to measure the effectiveness of our proposed method.

III. Research Objective

Based on the aforementioned background—namely, the limitations of iterative APR that relies solely on external feedback (error logs)—this research proposes a new correction paradigm: "**Introspective Self-Dialogue**," which aims to elicit the LLM's **Self-Correction** capabilities.

This approach is characterized by its integration of **internal information**, the "Chain-of-Thought" (CoT) generated by the LLM, and **external information**, the objective evidence of test failure from "error logs."

The objective of this method is to prompt the LLM itself to confront its own reasoning ("why it made that fix") with the objective facts ("why that fix resulted in failure"). We hypothesize that this allows the LLM to perform a deeper self-analysis of the root cause of failure, enabling higher-precision **Self-Correction** in the next iteration.

In this paper, we verify the effectiveness of this introspective self-dialogue loop. As an evaluation, we compare the bug fixing success rate of the conventional one-shot correction method (baseline) against our proposed iterative method (`Pass@k`). Furthermore, we comprehensively verify its effectiveness and safety from more practical perspectives, such as the "unintended effects" (code quality degradation) introduced by the corrected code.

IV. Related Work and Proposed Method

This research is related to two research areas: iterative Automated Program Repair (APR) using large language model (LLM), and code quality and dealing with degradation through static analysis.

A. Iterative Program Repair and Its Limitations

With the advent of LLM, research in bug fixing has shifted from one-shot repair to iterative approaches that incorporate feedback, such as test results.

Early Conversational APR, such as the work by Xia and Zhang [7], showed improved accuracy by feeding back test execution error logs to the LLM. However, Chen et al. [1] analyzed the current situation where these conversational approaches fail to fix many bugs. Their study identifies failure patterns in which the LLM, even after receiving error logs (external information), fails to identify the root cause of the bug and repeatedly generates inappropriate fixes (e.g., excessive code deletion or unrelated modifications).

In response to this challenge of "failure cause identification," more advanced feedback loops have been proposed. For example, Bouzenia et al. [8] proposed an autonomous agent called `RepairAgent`. This approach enables the LLM to autonomously plan its next actions (e.g., reading additional

file information or executing further tests) rather than relying on a fixed feedback loop.

These studies indicate that **simple iteration of external feedback (error logs) has limitations**, suggesting that a mechanism for the LLM to understand "why it failed" more deeply is necessary for higher-quality fixes.

B. Code Quality and Degradation in APR

Much of the traditional APR research has limited its evaluation to **functional correctness** (i.e., whether tests passed). However, even if a fix passes all tests, a "low-quality" fix—one that is difficult to read, introduces new bugs, or violates the project's coding standards—poses a significant practical issue.

Blyth et al. [9] discuss this issue from the perspective of going "Beyond Correctness." They point out that conventional APR benchmarks ignore code quality (security, reliability, maintainability) and propose a method that utilizes the **results of static analysis as a feedback loop** to improve the quality of LLM-generated code.

As research further developing this approach, agents like `CodeCureAgent` [10], which automatically classify and repair static analysis warnings, have also emerged.

These studies demonstrate that in LLM-APR, measuring "code quality" and "degradation" is just as important as measuring the bug fix rate. Therefore, based on the points raised by Blyth et al., this study adopts static analysis tools (PMD, Checkstyle) as **primary evaluation metrics**.

C. Proposed Method: Introspective Self-Dialogue Loop

To overcome the "limitations of simple external feedback", we propose the "**Introspective Self-Dialogue**" loop. This method focuses on maximizing the LLM's Self-Correction capabilities by integrating external information (failed test results) with the LLM's own internal information (its previous thoughts).

The overall process of our proposed method is shown in Fig. 1. We evaluate the proposed method under two experimental settings: the **Baseline mode** (used as the comparative reference) and the **Iterative mode** (proposed approach).

1) Baseline Mode ($k = 1$): The baseline corresponds to a conventional one-shot repair method.

a) The LLM receives **only the buggy code segment** as input.
b) The LLM generates a JSON object containing a `cot` (Chain-of-Thought) and a `patch` ($k = 1$).
c) The generated `patch` is applied to the target project, which is then compiled and its test suite executed.
d) If the tests pass, the bug fix is recorded as "Success ($k = 1$)".
e) If the tests fail, the bug fix is recorded as "Fail_Iter_1," and the **process terminates**.

2) Iterative Mode (Proposed Method) ($k > 1$): The proposed method is an iterative self-correction loop, with a maximum of three attempts ($k \leq 3$), which is activated only if the Baseline ($k = 1$) fails.

a) If the Baseline ($k = 1$) fails the tests, the $k = 2$ iteration phase begins.
b) The LLM receives two crucial types of feedback information as a new prompt:
   - **External Info:** The "Test Error Log" obtained when the $k = 1$ attempt failed.
   - **Internal Info:** The "Previous CoT" that the LLM itself generated in the $k = 1$ attempt, which led to the failure.
c) Based on these two pieces of information, the LLM performs a self-analysis (introspection) on "why its own reasoning (CoT) caused this error log," and generates a new `cot` and `patch` ($k = 2$).
d) The tests are executed again with this new `patch`. If successful, it is recorded as "Success ($k = 2$)". If it fails, the process proceeds to the $k = 3$ phase.
e) If the $k = 3$ attempt also fails, the bug is recorded as "Fail_Iter_3," after which the process terminates.

The key to this loop is forcing the LLM to confront the objective fact of failure (the error log) with its own subjective reasoning (the previous CoT).

V. Evaluation Experiment

To evaluate the effectiveness of the proposed method, *Introspective Self-Dialogue*, a comparative ablation study was conducted to examine the impact of incorporating internal information (CoT) into the feedback process.

A. Research Questions

Two iterative methods ($k > 1$) following the failure of the `Baseline` ($k = 1$) were compared to address the following research questions (RQs):

- **RQ1: Fixing Efficacy**
  Can the **"With CoT (Proposed Method)"** iteration rescue (`Uplift`) more bugs from the baseline (`Pass@1`) failure compared to the **"No CoT (Compared Method)"** iteration?
  - *Motivation:* To quantitatively clarify how much our research hypothesis—the integration of internal information (CoT) and external information (error logs)—improves the Self-Correction success rate compared to simple iteration using only external information ("No CoT").
- **RQ2: Safety and Quality**
  Can the **"With CoT (Proposed Method)"** fix suppress code quality degradation (deterioration) compared to the **"No CoT (Compared Method)"** fix?
  - *Motivation:* To verify the safety trade-off: whether achieving an improvement in the fix rate (RQ1) comes at the cost of worsening code quality (PMD, Checkstyle).

B. Experimental Setup

1) Dataset (Benchmark): In this study, we used `Defects4J` (version 3.0.1), a widely used dataset APR field [11]. `Defects4J` is a database of bugs collected from

real-world Java projects. Each bug is accompanied by a bug report and a comprehensive test suite, including failing tests that reproduce the bug and passing tests that verify the fix. Furthermore, `Defects4J` provides an extensive command-line interface (CLI) for tasks such as checking out buggy projects.

In this experiment, we targeted two projects included in `Defects4J`: **`Lang`** and **`Math`**. We targeted 53 source files from the `Lang` project and 94 source files from the `Math` project. This experiment was conducted on a **total of 147 bugs** (corresponding to 147 source files) from these two projects, after excluding deprecated bugs and some enormous source files.

2) LLM: We used the `gpt-5-mini` model provided by OpenAI as the LLM [4]. Initially, we planned to use `gpt-5`. However, this experiment is large-scale, involving up to three iterative attempts ($k = 3$) for numerous `Defects4J` bugs and static analysis of the entire project (for RQ2). The `gpt-5` model has high API costs and long inference times, posing challenges for executing the experimental cycle within a realistic timeframe and budget. Therefore, in this paper, we report the results of a **Preliminary Study** using `gpt-5-mini`, which offers a superior balance of cost and performance. The objective of this research is not to measure the "absolute performance of a specific model" but to verify whether the "Introspective Self-Dialogue" **approach** (`Pass@3`) is **relatively** more effective than "one-shot correction" (`Pass@1`). Thus, we determined that a comparison using `gpt-5-mini` could still provide significant insights into the effectiveness (RQ1) and safety (RQ2) of our proposed method.

C. Compared Methods and Prompt Design

To answer the RQs, this experiment compares the following three methods: `Baseline`, `Iterative / No CoT`, and `Iterative / With CoT`. The differences between these three methods are realized through their prompt designs.

1) System Prompt (Common to all methods): Figure 1 shows the 'System Prompt' template. The `system` prompt strictly defines the LLM's role and output format. It remains constant throughout the experiment.

- **Role:** You are an AI assistant specialized in fixing Java code.
- **Output Format:** The response **must** be a JSON object.
- **JSON Keys:** The JSON must include two keys: `cot` (step-by-step reasoning for the fix) and `fixed_code` (the entire fixed code).

```
Return **only** valid JSON.
Do not include explanations outside JSON.
Format: {"cot": "...", "code": "..."}

You are an AI assistant specializing in Java code
    fixes.
Your response must be a single JSON object with the
    following keys:

1. "cot": Your step-by-step reasoning (Chain-of-
    Thought) about the bug and its fix.
2. "code": The complete Java code after the fix.

### Example:

{
"cot": "The user's code has an issue X, so I will
    fix it like Y.",
"code": "public class FixedCode { ... (complete
    fixed code) ... }"
}
```

Fig. 1. Template of 'System Prompt'

2) Method 1: Baseline (Reference): Figure 2 shows the 'User Prompt' template in initial correction ($k = 1$). This serves as the common starting point ($k = 1$) for all methods and corresponds to conventional one-shot repair (`Pass@1`).

- **Prompt:** Contains only the following information:
  - Buggy Code (The body of the buggy code)
- **Behavior:** Attempts one fix. If it fails, it is recorded as a `Baseline` failure.

```
The following Java code contains a bug.
Please read the entire code carefully and fix the
    bug.
Include the entire fixed code under the    code
    key in your response (JSON).

---

Buggy Code:

```java
    {buggy_code_content}
```
```

Fig. 2. Template of 'User Prompt' in initial correction ($k = 1$)

3) Method 2: Iterative / No CoT (Compared Method): Figure 3 shows the 'User Prompt' template in providing only errors ($k > 1$). This approach iterates ($k > 1$) using **only external information** after the `Baseline` ($k = 1$) fails.

- **Prompt ($k > 1$):** Appends the following information to the `Baseline` prompt:
  - Failed Code
  - Test Error Log **(= External Info)**
- **Instruction:** "Your previous fix failed. Analyze the cause of the failure **by referring to the error log from the failed test** , then make another fix. Include the entire `fixed_code` under the "code" key in your response (JSON)."

```
Your previous fix failed.
Analyze the cause of the failure by referring to the
     error log from the failed test, then make
    another fix.
Include the entire fixed code under the    code
    key in your response (JSON).

---

Your previous fix code:

```java
    {previous_code}
```

Test error occurring with previous fix code:

```
    {error_log}
```
```

Fig. 3. Template of 'User Prompt' in introspective self-correction ($k > 1$)

```
Your previous fix failed.
Analyze the cause of the failure by referring to
    your previous thought process (CoT) and the
    error log from the failed test, then make
    another fix.
Include the entire fixed code under the    code
    key in your response (JSON).

---

Your previous thought process (CoT):

```
    {previous_cot}
```

Your previous fix code:

```java
    {previous_code}
```

Test error occurring with previous fix code:

```
    {error_log}
```
```

Fig. 4. Template of 'User Prompt' in introspective self-correction ($k > 1$)

4) Method 3: Iterative / With CoT (Proposed Method): Figure 4 shows the 'User Prompt' template in introspective self-correction ($k > 1$). This is the "Introspective Self-Dialogue" approach ($k > 1$), which iterates using **both internal and external information** after the `Baseline` ($k = 1$) fails.

- **Prompt ($k > 1$):** Appends the following information to the `Baseline` prompt:
  - **CoT (= Internal Info)**
  - Failed Code
  - Test Error Log **(= External Info)**
- **Instruction:** "Your previous fix failed. Analyze the cause of the failure **by referring to your previous thought process (CoT) and the error log from the failed test** , then make another fix. Include the entire `fixed_code` under the "code" key in your response (JSON)."

D. Evaluation Metrics

To answer the established RQs, we use the following two types of metrics for evaluation.

1) Fixing Efficacy: RQ1 is evaluated from two perspectives: `Pass@k` (cumulative success rate within k attempts) and `Uplift` (rescue rate by iteration).

- **Pass@1 (Baseline):** The success rate of Method 1 (`Baseline`). This serves as the **comparative starting point** for Methods 2 and 3.
- **Uplift ($k > 1$):** The number of bugs that failed at `Pass@1` but were **newly rescued** by Method 2 (`No CoT`) or Method 3 (`With CoT`) at $k = 2$ or $k = 3$. **This is the primary comparative metric for RQ1.**
- **Pass@3 (Final):** The final cumulative success rate for Method 2 or 3 (`Pass@1` + `Uplift`).

2) Code Safety and Quality Degradation ($\Delta_{Quality}$): To measure RQ2, we target the bugs that were **successfully fixed** by Method 2 and Method 3 and measure the change in static analysis tool warnings.

The scan target is the entire source directory of the project, not just the file being fixed.

- **PMD:** PMD is a tool that analyzes Java source code to detect serious design problems, such as "potential bugs" (e.g., potential `NullPointerException`), "inefficient code," and "overly complex code" [12]. In this study, we use a definition file that customizes the standard Java ruleset.
- **Checkstyle:** Checkstyle is a tool that verifies whether the code adheres to a specified coding standard (style guide) [13]. It primarily detects issues related to "readability" and "maintainability" (e.g., indentation, naming conventions). In this study, we use a definition file based on Google's Java style guide.

Using these tools, we calculate the difference (Δ) between the `Baseline` (pre-fix) and `Final` (post-fix) warning counts.

- PMD Degradation:
  $\Delta_{PMD} = PMD_{final} - PMD_{baseline}$
- Checkstyle Degradation:
  $\Delta_{CS} = Checkstyle_{final} - Checkstyle_{baseline}$

$\Delta_{Quality} > 0$ implies "Degradation" (worsening), while $\Delta_{Quality} < 0$ implies "Improvement."

VI. Results and Discussion

In this section, we present the quantitative analysis and discussion of our experimental results based on the two Research Questions defined in Section 3.

TABLE I
RQ1: COMPARISON OF FIXING EFFICACY

Metric	No CoT	With CoT (Proposed)
Total Bugs	147	147
Baseline Success (`Pass@1`)	38	38
Iterative Rescue (`Uplift`)	**30**	**38**
(Rescued at k=2)	*(19)*	*(24)*
(Rescued at k=3)	*(11)*	*(14)*
Total Success (`Pass@3`)	**68**	**76**
Final Success Rate	46.26%	**51.70%**
Baseline Uplift Rate	27.5%	**34.9%**

As a prerequisite for our comparison, we unified the `Baseline` (`Pass@1`) success count. Due to experimental stochasticity (API response variations), the `Pass@1` results showed slight noise (38 vs 41). We adopted the more conservative count of **38 successes** (a 25.85% success rate) as the unified `Baseline`. This analysis, therefore, compares the "rescue capability" (Uplift) of the two iterative methods on the remaining **109 bugs** (147 - 38) that failed the `Baseline`.

A. RQ1: Fixing Efficacy

RQ1: Can the "With CoT (Proposed Method)" iteration rescue (`Uplift`) more bugs from the baseline (`Pass@1`) failure compared to the "No CoT (Compared Method)" iteration?

Analysis: Yes. The With CoT (Proposed Method) successfully rescued 8 more bugs than the comparative method (No CoT), demonstrating a clear superiority in fixing efficacy.

Table I shows the fixing efficacy results for both methods applied to the 109 bugs that failed the `Baseline`.

Discussion (RQ1): As shown in Table I, the `No CoT` iteration (using only error logs) was able to rescue 30 bugs (a 27.5% uplift rate), confirming that iteration itself has a clear benefit.

However, the proposed method, `With CoT` (Introspective Self-Dialogue), rescued **38 bugs**, which is **8 more** than the `No CoT` method. This represents a **+26.7%** (8 / 30) improvement in rescue capability.

This result strongly suggests that **at least 8 bugs** were difficult to fix with external information (error logs) alone, and could only be corrected after the LLM introspected on its own "previous thoughts" (CoT = internal information). This quantitatively validates our hypothesis that CoT feedback enhances the precision of self-correction.

B. RQ2: Safety and Quality

RQ2: Can the "With CoT (Proposed Method)" fix suppress code quality degradation (worsening) compared to the "No CoT (Compared Method)" fix?

Analysis: No. In contrast to the fixing efficacy (RQ1), the "No CoT" method was superior in terms of code quality (RQ2). The With CoT (Proposed Method) achieved its higher success rate at the cost of incurring more degradation, especially in code style violations.

TABLE II
RQ2: COMPARISON OF AVERAGE CODE QUALITY DEGRADATION

Metric	No CoT	With CoT
Avg. PMD Degradation (Potential Bugs)	**+0.16**	+0.25
Avg. Checkstyle Deg. (Code Style)	**+1.53**	+6.66

(+ indicates worsening/degradation

Table II shows the average change in static analysis warnings (degradation) for the bugs that were **successfully fixed** by each method (68 bugs for `No CoT`, 76 bugs for `With CoT`).

Discussion (RQ2):

- **PMD (Potential Bugs):** Both methods slightly increased PMD warnings on average. While the degradation from `With CoT` (+0.25) was slightly worse than `No CoT` (+0.16), the difference is marginal.
- **Checkstyle (Code Style):** The difference was stark in Checkstyle. The average degradation for the `No CoT` method was minor (+1.53). In contrast, the average degradation for the `With CoT` method was **+6.66**, which is **more than 4 times worse**.

This finding suggests that by including CoT (introspection) in the feedback, the LLM is capable of performing more "complex" and "bolder" fixes (leading to the +8 successes in RQ1), but it does so while disregarding code style (Checkstyle), which was not explicitly part of its prompt.

C. Overall Discussion

The experimental results clearly demonstrate both the **efficacy** and the **challenges** of our proposed "Introspective Self-Dialogue" (`With CoT`) method.

1) **Efficacy (RQ1):** The proposed method was **26.7% superior** in bug rescue capability (`Uplift`) compared to simple iteration (`No CoT`), proving that "introspection" using CoT (internal information) is a critical component for successful self-correction.
2) **Challenge (RQ2):** This high fixing efficacy exists in a **trade-off relationship with code quality** (especially Checkstyle). The complex fixes enabled by CoT incurred **more than 4 times** the code style violations (degradation) compared to the `No CoT` method, leaving a safety challenge (risk of technical debt) for practical implementation.

VII. CONCLUSION

In this research, we focused on the challenge that Self-Correction in iterative Automated Program Repair (APR) has limitations when relying solely on simple external feedback (error logs). We proposed "Introspective Self-Dialogue" as a novel approach to overcome this challenge, integrating the LLM's own **internal information** ("Chain-of-Thought," CoT) with **external information** ("error logs").

As a result of our evaluation experiment (an ablation study) on 147 bugs from `Defects4J`, the primary contributions of this research were clarified.

First is the **efficacy of the proposed method (RQ1)**. With CoT (Proposed Method), which utilizes CoT (internal information) in its feedback, rescued **8 more bugs (+26.7%)** from the `Baseline` (`Pass@1`) failures compared to the simple iterative method (No CoT). This quantitatively demonstrates that "introspection" (referencing CoT) is a critical process for the LLM to analyze the root cause of failure and perform self-correction.

Second is the **challenge of the proposed method (RQ2)**. This high fixing efficacy was built upon a trade-off with code quality. The proposed method incurred **more than 4 times** the Checkstyle (code style) violations compared to the `No CoT` method. This indicates that the LLM optimized for "functional correctness" (passing tests) while sacrificing "quality safety" (code style) that was not specified in the prompt, highlighting the risk of technical debt for practical implementation.

Although this paper reports the results of a **Preliminary Study** using `gpt-5-mini`, it is significant in that it clearly demonstrates both the effectiveness of "introspection" in LLM-APR and the associated "quality degradation" trade-off.

Future Work includes resolving the trade-off identified in this study.

First, we can integrate the approach of using static analysis as feedback (mentioned in *Related Work*, e.g., [9]) into our introspection loop. The current iterative prompt ($k > 1$) makes the LLM introspect on "Test Error Logs" (external info) and "Previous CoT" (internal info). This could be extended by adding a third piece of information to the $k > 1$ feedback: a "**summary of PMD and Checkstyle warnings**." To keep token counts manageable, feeding a *summary* of the warnings (e.g., warning types and counts, or only warnings near the patch) rather than the full list seems practical. This would force the LLM to introspect not only on "why the test failed" (error logs) but also on "**why it degraded code quality**" (static analysis summary). We expect this multi-faceted introspection to enable a more advanced self-correction that optimizes for both "functional correctness" and "quality safety."

Second, a crucial task is to reproduce this experiment with a more capable model, such as `gpt-5`. In this study with `gpt-5-mini`, we observed a **severe trade-off** where an increase in fixing efficacy (RQ1) came at the expense of a significant degradation in code quality (RQ2). We speculate this trade-off may stem from the fundamental limitations of `gpt-5-mini`'s ability to handle **multiple constraints simultaneously** (i.e., "fix the bug" *and* "follow the coding standard"). Flagship models like `gpt-5` are expected to possess superior reasoning and the capability to perform complex, multi-objective optimization. Therefore, re-validating our "Introspective Self-Dialogue" loop with `gpt-5` to determine **the extent to which this trade-off (the conflict between RQ1 and RQ2) is resolved**—that is, whether it can achieve both high fixing efficacy and high code quality—is an essential next step in measuring the practical utility of this approach.

Third, rather than relying solely on stronger LLMs, future work will also explore prompt engineering techniques to enhance the repair capability of relatively weak models. By designing more structured prompts or introducing self-guided reasoning cues, we aim to elevate the performance of less capable LLMs without increasing model size, thereby improving the cost-effectiveness and reproducibility of LLM-based APR.

References

[1] A. Chen, H. Wu, Q. Xin, S. P. Reiss, and J. Xuan, "Studying and Understanding the Effectiveness and Failures of Conversational LLM-Based Repair," in *Proc. 2025 IEEE/ACM Int. Workshop on Automated Program Repair (APR)*, 2025.

[2] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," *arXiv preprint arXiv:2001.08361*, 2020.

[3] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[4] OpenAI, "Changelog - OpenAI API," *OpenAI Developer Platform*, Aug 7, 2025. [Online]. Available: https://platform.openai.com/docs/changelog

[5] OpenAI, "Using GPT-5 - OpenAI API," *OpenAI Developer Platform Guides*, 2025. [Online]. Available: https://platform.openai.com/docs/guides/latest-model

[6] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[7] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," *arXiv preprint arXiv:2304.00385*, 2023.

[8] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair," in *Proc. 2025 IEEE/ACM 47th Int. Conf. on Software Engineering (ICSE)*, 2025.

[9] S. Blyth, S. A. Licorish, C. Treude, and M. Wagner, "Static Analysis as a Feedback Loop: Enhancing LLM-Generated Code Beyond Correctness," in *Proc. 2025 IEEE Int. Conf. on Source Code Analysis & Manipulation (SCAM)*, 2025.

[10] [Authors of CodeCureAgent], "CodeCureAgent: Automatic Classification and Repair of Static Analysis Warnings," *arXiv preprint arXiv:2509.11787*, 2025.

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proc. 2014 Int. Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440.

[12] PMD, "PMD Source Code Analyzer," *PMD Project*, 2025. [Online]. Available: https://pmd.github.io/

[13] Checkstyle, "Checkstyle," *Checkstyle Project*, 2025. [Online]. Available: https://checkstyle.sourceforge.io/