# Repair of Thought: Advancing Function-Level Automated Program Repair through a Dual-Model Reasoning Framework

Satwik Pandey
Independent Researcher
New Delhi, India
psatwik2711@gmail.com

Suresh Raghu
Independent Researcher
New Delhi, India
sureshraghu0706@gmail.com

Bhawna Saxena
Dept. of CSE & IT
JIIT, Noida, India
bhawna.saxena@jiit.ac.in

Sandeep Kumar Singh
Dept. of Computer Applications
JIIT, Noida, India
sandeep.singh@jiit.ac.in

*Abstract*—**Automated Program Repair (APR) constitutes a fundamental component of modern software engineering, aimed at mitigating the expensive, time-consuming process of manual debugging. Although Large Language Models (LLMs) have significantly advanced the state-of-the-art, prevailing techniques are still bounded by critical limitations, including reliance on proprietary models, lack of transparency in the repair process, and reliance on monolithic architectures that conflate bug understanding with code generation. The primary objective of this work is to design a transparent, cost-efficient APR framework that can achieve high performance without such limitations. We present Repair of Thought (ROT), a novel framework that explicitly decouples the repair process into two distinct phases: reasoning and action. ROT relies on a dual-model architecture solely based on open-source LLMs, where a reasoning component performs deep Chain-of-Thought analysis by multi-step prompting of non-reasoning models to synthesize structured repair strategies, which are then translated into precise code patches by a dedicated synthesis component. Operating at the function level, the framework avoids brittle statement-level fault localization. To address the systemic bottleneck of manual patch inspection, we introduce an automated validation pipeline that synthesizes structural, symbolic, and semantic analyses into a unified classification verdict. Achieving 95.7% agreement with human annotations on a verified dataset, this integrated methodology enables the scalable, objective assessment of semantic correctness that has historically required labor-intensive manual effort. We introduce two main evaluation metrics: the Correct Repair Rate (CRR), defined as the proportion of bugs for which semantically equivalent patches are generated, and the Plausible Repair Rate (PRR), capturing patches passing all the tests independently of semantic equivalence. When evaluated on the Defects4J benchmark, ROT reaches a CRR of 41.2%, repairing 215 out of 522 bugs correctly, while achieving a PRR of 83.1% validated by our comprehensive pipeline. These results suggest that transparent, reasoning-driven, and open-source approaches can achieve state-of-the-art performance, setting a new bar for accessible APR that researchers and practitioners can easily adopt and extend.**

## I. Introduction

The continued need for reliable software combined with rapid development cycles has made Automated Program Repair (APR) an important field in software engineering [1]–[3]. The main objective of APR is to automate the process of bug identification and fixing, thus reducing the significant time developers spend on debugging and performing maintenance. Traditionally, the domain moved forward from more classic approaches, like the ”Generate and Validate” (G&V) paradigm that utilizes genetic programming to evolve patches. Although these were pioneering methods, they often suffered from limited patch variability and could not easily overcome difficult or new types of bugs [4].

The recent prominence of Large Language Models (LLMs) has advanced a paradigm shift in APR. Based on the Transformer architecture [5], these models have transformed the field from early Neural Machine Translation (NMT) techniques [6] and edit-time code completion [7] toward more robust generative repairs. Trained on vast code corpora, state-of-the-art LLMs demonstrate an unparalleled ability to understand programming semantics [8] and to synthesize code indistinguishable from that written by humans, going beyond the patch-diversity limitations of earlier approaches [1], [2], [9]. This has recently led to the emergence of a new breed of LLM-based APR tools that greatly broaden the panorama of automated bug fixing. Despite these advances, several fundamental limitations inherent in the current state-of-the-art LLM-based APR impede broad adoption and hinder scientific reproducibility. A large fraction of the top research depends on proprietary, closed-source models, such as the GPT-series used by the ChatRepair [9] framework. This dependence entails significant operational costs; adds barriers to entry, complicating the replication of experimental results; and greatly diminishes the potential for scientific progress. Furthermore, many systems act as ”black boxes,” offering little insight into their rationale [10]. In the absence of automated mechanisms to articulate the intent behind a patch [11], developers may find it hard to trust or verify generated code. Architecturally, most frameworks either employ a monolithic model or depend on brittle workflows that conflate fault localization with code generation [12]. This conflation of responsibilities frequently presents a trade-off: while a single model may serve well as a reasoner or as a coder, it seldom performs both tasks well simultaneously, potentially constraining the quality of the final patch [13]. Finally, the field has long struggled with the need for painstaking manual validation to ensure that the

Fig. 1: Anatomy of a Defects4J Bug Entry (Chart-1).

**1. Buggy Function**
(`AbstractCategoryItemRenderer.java`)

```
public LegendItemCollection getLegendItems() {
    ...
    CategoryDataset dataset = this.plot.getDataset(
    index);
    if (dataset != null) { // BUG IS HERE
        return result;
    }
    int seriesCount = dataset.getRowCount(); //
    Throws NPE
    ...
}
```

**2. Triggering Test Case**
(`...Tests.java`)

```
public void test2947660() {
    ...
    DefaultCategoryDataset dataset = new
    DefaultCategoryDataset();
    CategoryPlot plot = new CategoryPlot();
    plot.setDataset(dataset);
    plot.setRenderer(r);
    ...
    dataset.addValue(1.0, "S1", "C1");
    LegendItemCollection lic = r.getLegendItems();
    assertEquals(1, lic.getItemCount());
}
```

**3. Error Message**

```
junit.framework.AssertionFailedError:
    expected:<1> but was:<0>
    at ...AbstractCategoryItemRendererTests.java:409
```

**4. Ground Truth Fix**

The logical error is corrected:

```
// Before (Buggy)
if (dataset != null)

// After (Fixed)
if (dataset == null)
```

generated patches are correct, a bottleneck that has always been characterized by being slow, expensive, and intrinsically subjective. To put these challenges into context, consider the structure of a typical defect in the popular Defects4J benchmark, illustrated in Figure 1. A bug function, along with a failing test case and the respective error log resulting from the execution of the test, serves as an input for an APR system. The system tries to synthesize not only a correct patch but also a logically consistent one, which again emphasizes the necessity for a clear and reasoning-aware approach.

This paper presents Repair of Thought (ROT), a novel function-level APR framework designed to directly address these limitations. At the core of ROT is its **Thinker–Action dual-model architecture**, a design that explicitly separates the cognitive process of bug analysis from the mechanical process of patch synthesis. In such a setting, each model is specialized for its respective task, enabling more accurate and transparent repairs. The "Thinker" model conducts deep, structured reasoning to diagnose the root cause of a bug and to propose high-level repair strategies. These strategies are afterwards implemented by the "Action" model, synthesizing a precise and minimal code patch.

In an effort to deliberately democratize state-of-the-art APR, ROT is designed and built using only powerful, open-source LLMs, specifically LLaMA 3.3 and DeepSeek-R1. This commitment inherently provides no API-related costs, and opens the way for community-driven innovation and reproducibility. Following a function-level repair approach, ROT eliminates the need for exact statement-level fault localization, a notoriously hard and computationally expensive prerequisite for many state-of-the-art APR techniques. This is also in line with the evidence provided by D4C [12], which has shown that allowing models to refine whole programs end-to-end better utilizes their pre-trained next-token prediction capabilities. We also argue that our metric Plausible Repair Rate (PRR) is a more practically meaningful measure of effectiveness than strict ground truth equivalence alone, given that multiple semantically valid repairs can exist. Using our automated semantic validation pipeline we achieve 83.14% PRR (434/522) on Defects4J-SF, outperforming current state-of-the-art benchmarks under aligned definitions.
The key contributions of this work are:

1) A novel dual-model ”Thinker–Action” framework that decouples structured reasoning from patch generation, hence enhancing both the accuracy and transparency of the automated repair process.
2) A very efficient and effective function-level APR system that is exclusively based on open-source LLMs, achieving state-of-the-art performance without using any proprietary APIs.
3) A scalable, automated, reasoning-based approach to validation methodology that addresses the critical bottleneck of manual patch evaluation in APR research, yielding 95.7% agreement with human annotations on a verified dataset.

## II. Background and Related Work

In order to provide context to the contribution of the ROT approach, it is important to establish the historical perspective of APR, tracing back from the heuristic origins to the present-day LLM approaches.

### A. The Trajectory of Automated Program Repair

APR started off with methodologies that relied upon manipulating syntactical features of programs using heuristics or predefined templates. The G&V class of approaches was most prevalent, where systems used Genetic Programming to evolve a mutated form of preexisting code to meet a given test suite, though frequently limited by the quality of the test suite and a high rate of incorrect plausible patches [4]. Around the same time, template learning started to be used, where experts predefined a set of cached patterns that functioned as tools for a set of predefined types of defects [1]. However, these methods had a fundamental drawback in that they could not generalize to other types of bugs beyond their predefined sets using heuristics or knowledge represented in their templates.

The rise of deep learning brought about the onset of a fresh paradigm for APR. Scientists reformulated the program repair task as an NMT task [2], [6], emulating developer behavior by learning from vast datasets of pull requests [6]. These models, such as those proposed by Tufano et al. [6], learned from vast datasets of pull requests to emulate developer behavior. Early iterations of this logic were also applied to edit-time code completion, where Transformer-based models began predicting sequences of code tokens to assist developers in real-time [7].

### B. Granularity in Program Repair

APR studies involve a very important distinction between statement-level and function-level repairs. Statement-level approaches focus on pinpointing and repairing individual statements or lines of code, usually involving accurate fault localization. Although very helpful for simple problems, such approaches may not be very efficient at repairing complex problems involving several statements. Function-level approaches, on the other hand, like ROT and SRepair [13], involve examining complete functions. Function-level techniques allow a broader comprehension of program executions, allowing the solution of problems involving several statements, thus overcoming the need for fine-grained fault localization, a process that significantly improved the quality of patches [13].

### C. Evolution of LLMs in APR

LLMs have substantially advanced the state-of-the-art in APR, and a number of different APR repair paradigms have emerged to use the massive, implicit code knowledge present in these models. Currently, there are debates on the need to fine-tune models vs. using the models in a zero-shot fashion. Research within the overall Impact of Code Language Models on APR studies has shown that fine-tuning models on APR-centric datasets can result in significant performance gains [2]. Nonetheless, Full Model Fine-Tuning (FMFT) of larger models is computationally infeasible. This led to the need to investigate the application of Parameter Efficient Fine-Tuning (PEFT) methods that require the tuning of only a small part of the model's parameters to achieve significant performance gains [14]. The inherent trade-offs within this domain makes ROT's zero-shot, open-source approach a conscious and strong choice, which focuses on accessibility and ready-to-go performance without the need to train models of significant size and complexity.

### D. Multi-modal APR Techniques

A lot of research into APR still depends on unimodal datasets, such as source code, text, or test result logs. A new, active area of research explores multi-modal information, which seeks to include data other than the textual representation of the program.

**Zero-Shot and Infilling Techniques:** One of the key developments in this area has been the establishment of the ability of LLMs to apply APR in the zero-shot paradigm. The work conducted in **AlphaRepair** is one of the first of its kind in this domain and has repurposed APR as an "infilling" problem rather than considering it an equivalent neural machine translation exercise [15]. The work uses an LLM where the faulty span of code is directly predicted on the basis of the context obtained.

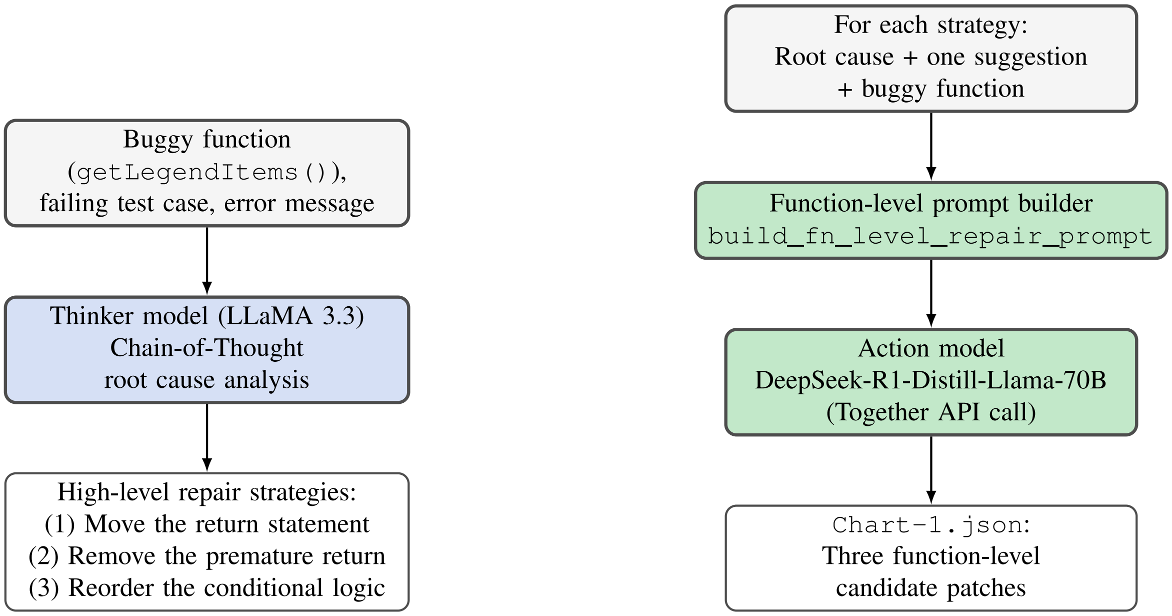
**Conversational and Iterative Frameworks:** Realizing that sometimes the process of fixing bugs takes place through iteration, more recent works have explored interaction-based approaches. An example is **ChatRepair**, which specifically harnesses the conversational elements that exist in models such as ChatGPT [9]. They make use of the output received through failing test cases in order to communicate the weakness in the prior repair solution, eventually triggering the generation of an improved one within the next move. Iteration has proven very effective, especially for more intricate bugs [9], [12].

**Hybrid and Analysis-Augmented Methods:** Some of the most influential recent approaches combine the generative power of LLMs with traditional program analysis techniques. **GiantRepair** proposed a novel two-step hybrid approach to fixing bugs in software systems [16]. In the first step, an LLM is used to produce a variety of candidate patches. The candidates are then represented as "patch skeletons," which represent the fundamental changes embodied by each candidate patch [16].

**Reasoning-Augmented Function-Level Repair:** The most closely related approaches to ROT are those frameworks that function at a function level and involve a reasoning step as an integral part of their process. **SRepair** is an example of a state-of-the-art system that demonstrates the effectiveness of a dual LLM approach to function-level repairing. There, the first LLM takes the faulty function as input to produce a natural language "repair suggestion," which a second LLM then uses to synthesize the final code patch; the effectiveness of this approach showed the strength of the function-level approach [13].

### E. Benchmark Datasets: Defects4J

One of the most important aspects of APR studies on a real-world scale is standardized benchmarking. Benefiting all APR


Buggy function
(getLegendItems()),
failing test case, error message
Thinker model (LLaMA 3.3)
Chain-of-Thought
root cause analysis
High-level repair strategies:
(1) Move the return statement
(2) Remove the premature return
(3) Reorder the conditional logic
For each strategy:
Root cause + one suggestion
+ buggy function
Function-level prompt builder
build_fn_level_repair_prompt
Action model
DeepSeek-R1-Distill-Llama-70B
(Together API call)
Chart-1.json:
Three function-level
candidate patches

(a) Thinker model flow.

(b) Action model flow.

Fig. 2: The ROT end-to-end Thinker–Action pipeline applied to the Chart-1 bug.

studies, **Defects4J** is considered a gold standard. As shown in Figure 1 (Chart-1 bug, with an example of a Defects4J bug), each bug report contains necessary artifacts that help in bug fixing, which include a buggy function where the bug is marked, a particular failing test case, an exception message, and a fix.

**Defects4J** consists of a carefully chosen set of 835 real-world, reproduceable Java bugs from prominent open-source projects like Apache Commons, JFreeChart, and Joda-Time. It comes with the faulty and correct codes, a thorough test program written by a developer, and scripts to reproduce a bug programmatically. It provides a comprehensive infrastructure for both statement- and function-level evaluation, thereby making it a perfect tool for comparing the performance of various APR approaches [13].

## III. The Repair of Thought (ROT) Framework

The ROT framework has been created as a remedy for the deficiencies that exist in opaque and proprietary APR systems. This approach was built on a number of key principles.

### *A. Framework Design and Architecture*

At the design level, the ROT approach fundamentally relies on the idea of separating the understanding of bugs from the generation of patches. This approach is inspired by the cognitive processes involved in software design by human programmers, who spend considerable time conceptualizing issues before actually applying them to coding. Consequently, the goal of ROT is to develop patches that are not merely syntactically valid but, more fundamentally, are grounded in a semantic, causal understanding of the bug.

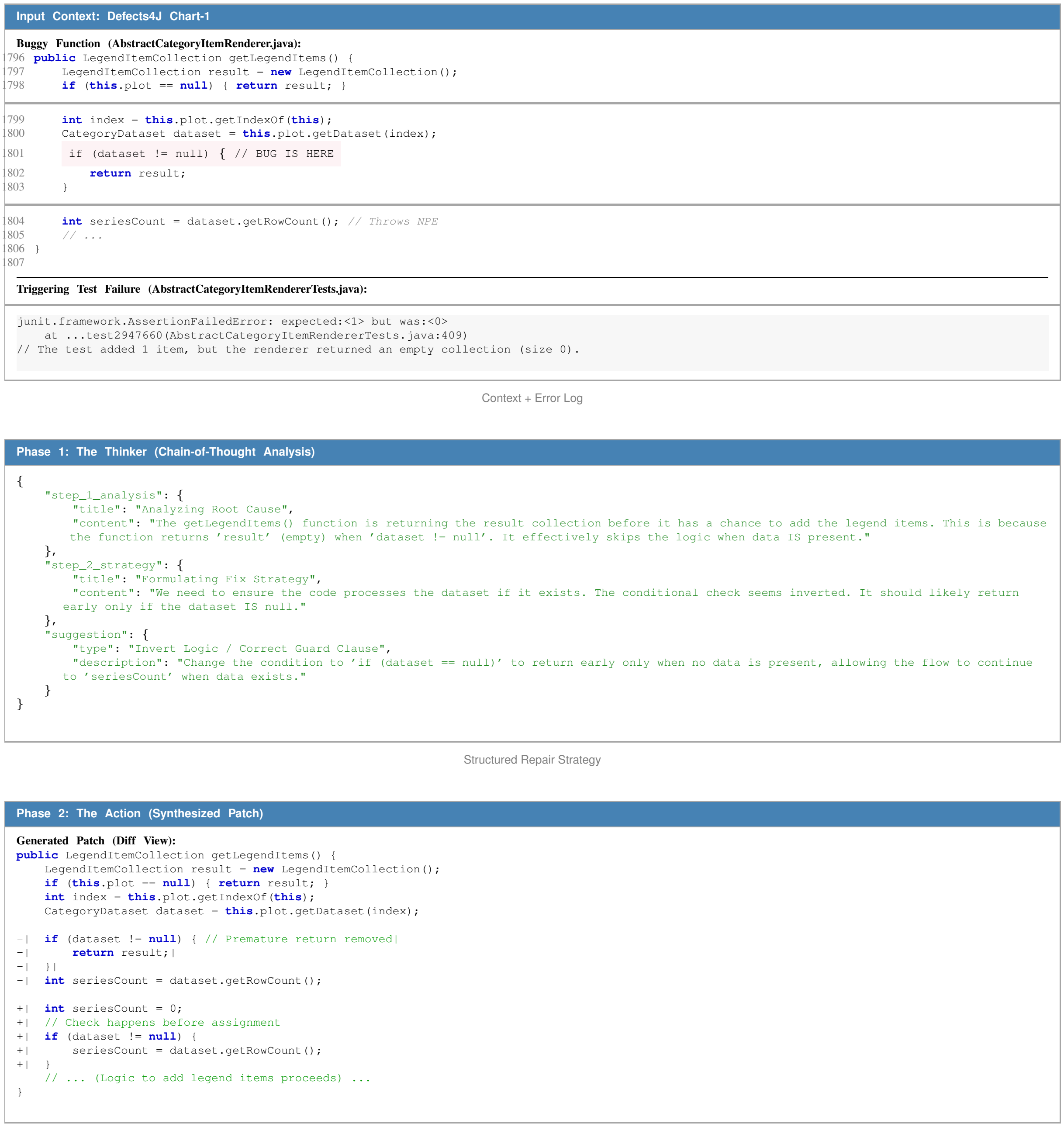
### *B. The Thinker Model: Structured Bug Diagnosis via Chain-of-Thought (CoT)*

The Thinker component acts as a self-directed reasoning engine, building upon recent CoT repair techniques [17] to understand the semantics of the buggy function. As depicted in **Figure 2a**, it takes the raw bug artifacts and produces a structured representation of the problem by analyzing the buggy code, the test that failed, and the error message using LLaMA 3.3 (70B) to make a diagnostic judgment.

Unlike methods based on a single input, Thinker performs a CoT. This is illustrated by the flow of Figure 2a, where from raw input, the model moves to a high-level output with varied repair strategies like “Move return statement” without directly generating code.

- **Step 1: Root Cause Analysis.** The Thinker model begins with the defective function, test case, and error message as inputs in order to conduct the root cause analysis and generate the primary hypothesis in the structured JSON format.
- **Step 2: Diverse Strategy Generation.** The JSON output from Step 1 is then given back to the model in another input query, this time asking for three different high-level strategies to resolve the problem.
- **Step 3: Detailed Strategy Refinement.** Each of the high-level strategies is followed by a third prompt, wherein the model is asked to elaborate on specific implementation details and provide a list of the exact code changes required.

By chaining these recursive inferences, the Thinker can build its repair plan incrementally. Although the overall workflow from Figure 2 shows the Thinker, the more detailed interaction of the ROT pipeline with the defect in the Chart-1



Input Context: Defects4J Chart-1

**Buggy Function (AbstractCategoryItemRenderer.java):**

```java
public LegendItemCollection getLegendItems() {
    LegendItemCollection result = new LegendItemCollection();
    if (this.plot == null) { return result; }

    int index = this.plot.getIndexOf(this);
    CategoryDataset dataset = this.plot.getDataset(index);
    if (dataset != null) { // BUG IS HERE
        return result;
    }

    int seriesCount = dataset.getRowCount(); // Throws NPE
    // ...
}
```

**Triggering Test Failure (AbstractCategoryItemRendererTests.java):**

```
junit.framework.AssertionFailedError: expected:<1> but was:<0>
    at ...test2947660(AbstractCategoryItemRendererTests.java:409)
// The test added 1 item, but the renderer returned an empty collection (size 0).
```

Context + Error Log

Phase 1: The Thinker (Chain-of-Thought Analysis)

```
{
    "step_1_analysis": {
        "title": "Analyzing Root Cause",
        "content": "The getLegendItems() function is returning the result collection before it has a chance to add the legend items. This is because
       the function returns 'result' (empty) when 'dataset != null'. It effectively skips the logic when data IS present."
    },
    "step_2_strategy": {
        "title": "Formulating Fix Strategy",
        "content": "We need to ensure the code processes the dataset if it exists. The conditional check seems inverted. It should likely return
      early only if the dataset IS null."
    },
    "suggestion": {
        "type": "Invert Logic / Correct Guard Clause",
        "description": "Change the condition to 'if (dataset == null)' to return early only when no data is present, allowing the flow to continue
      to 'seriesCount' when data exists."
    }
}
```

Structured Repair Strategy

Phase 2: The Action (Synthesized Patch)

**Generated Patch (Diff View):**

```
public LegendItemCollection getLegendItems() {
    LegendItemCollection result = new LegendItemCollection();
    if (this.plot == null) { return result; }
    int index = this.plot.getIndexOf(this);
    CategoryDataset dataset = this.plot.getDataset(index);

-|  if (dataset != null) { // Premature return removed|
-|      return result;|
-|  }|
-|  int seriesCount = dataset.getRowCount();

+|  int seriesCount = 0;
+|  // Check happens before assignment
+|  if (dataset != null) {
+|      seriesCount = dataset.getRowCount();
+|  }
    // ... (Logic to add legend items proceeds) ...
}
```

Fig. 3: Information flow from input artifacts to synthesized patch for the Chart-1 bug.

program, which transforms the raw Defects4J output into the generated patch, is shown in Figure 3.

### C. The Action Model: High-Fidelity Patch Synthesis

After arriving at the set of overall repair strategies produced by the Thinker, the process moves forward to the step illustrated in Figure 2b. In this step, the "Action" model is engaged in the high-fidelity conversion process of the natural-language plans into code.

As shown in the figure, a prompt builder creates a specific context based on each strategy. In particular, a root cause and a recommendation from the Thinker are combined with a defective function to come up with a final input that is then used by the Action model (DeepSeek-R1-Distill-Llama-70B) to generate a possible fixed function (such as `Chart-1.json`) based on a carefully constructed prompt that follows strict guidelines.

The prompt structure of the Action model, shown in Table I, is designed to enable high-fidelity patch generation. The dynamic parts of the prompt provide the necessary context, which is extracted using the Thinker model (the root cause, the suggestion, and the buggy function). These parts are framed by

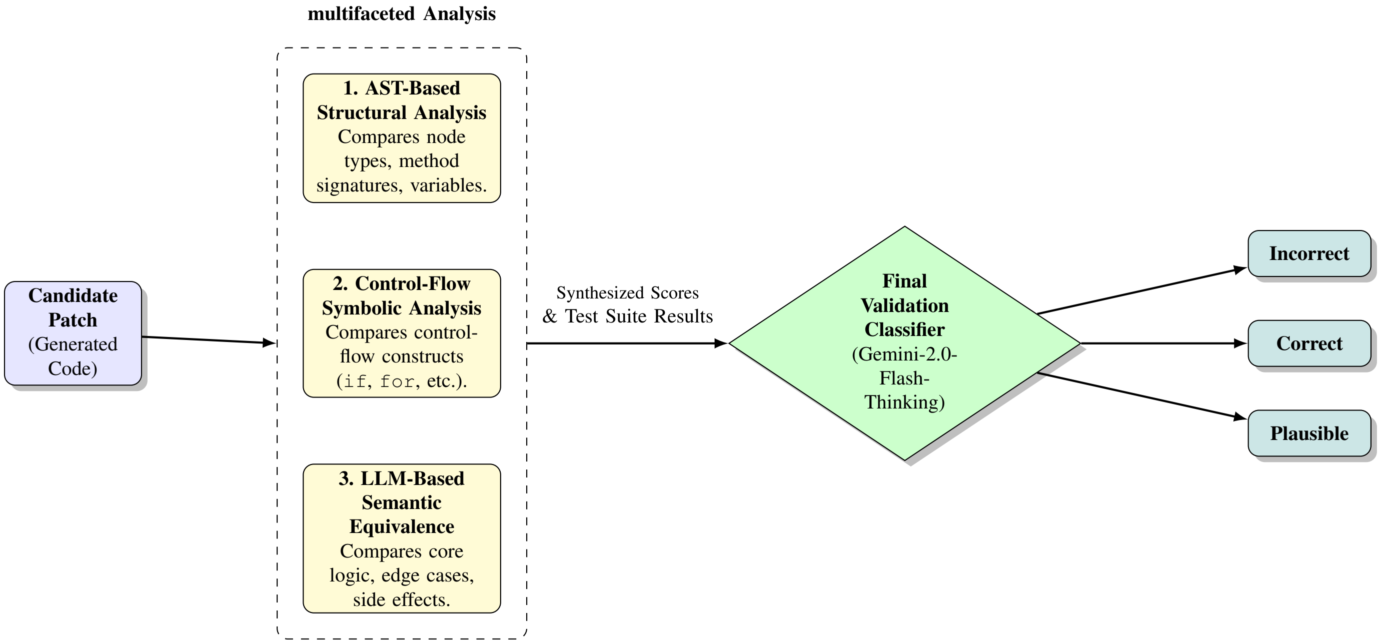

multifaceted Analysis
1. AST-Based Structural Analysis
Compares node types, method signatures, variables.
2. Control-Flow Symbolic Analysis
Compares control-flow constructs (if, for, etc.).
3. LLM-Based Semantic Equivalence
Compares core logic, edge cases, side effects.
Candidate Patch
(Generated Code)
Synthesized Scores
& Test Suite Results
Final Validation Classifier
(Gemini-2.0-Flash-Thinking)
Incorrect
Correct
Plausible

Fig. 4: Architecture of the ROT multifaceted validation pipeline.

static labels, which serve a very important purpose: to force the model to provide a minimal patch while maintaining the logic and function signature of the original function and making the output machine-parseable for validation.

TABLE I: Components of the Action Model Prompt

| **Prompt section** | **Nature** |
|---|---|
| `// Provide a fix for...` label (`apr_label`) | Static |
| `Root cause:` {*`root_cause`*} | Dynamic |
| `Suggestion:` {*`suggestion`*} | Dynamic |
| `// Buggy Function` label (`buggy_label`) | Static |
| Full buggy function code (`buggy_function`) | Dynamic |
| `// Fixed Function` label (`fixed_label`) | Static |

## IV. Validation Methodology and Evaluation Metrics

The most dominant bottleneck for current APR research is reliance on manual validation for the assurance of patch correctness, which is costly and non-scalable. To address this, ROT performs comprehensive automated validation. Distinct from the majority of state-of-the-art LLM-driven APR pipelines, which essentially report test-suite adequacy, and then rely on manual inspection to make semantic correctness judgments, our evaluation is fully end-to-end automated: ROT makes a Correct, Plausible or Incorrect verdict on each candidate patch in an automated fashion, without any need for human intervention at the evaluation stage of the process. This provides the means for scalable, reproducible assessment of semantic repair beyond pass/fail results.

We present our validation methodology, define the evaluation metrics, and provide the experimental results in this section.

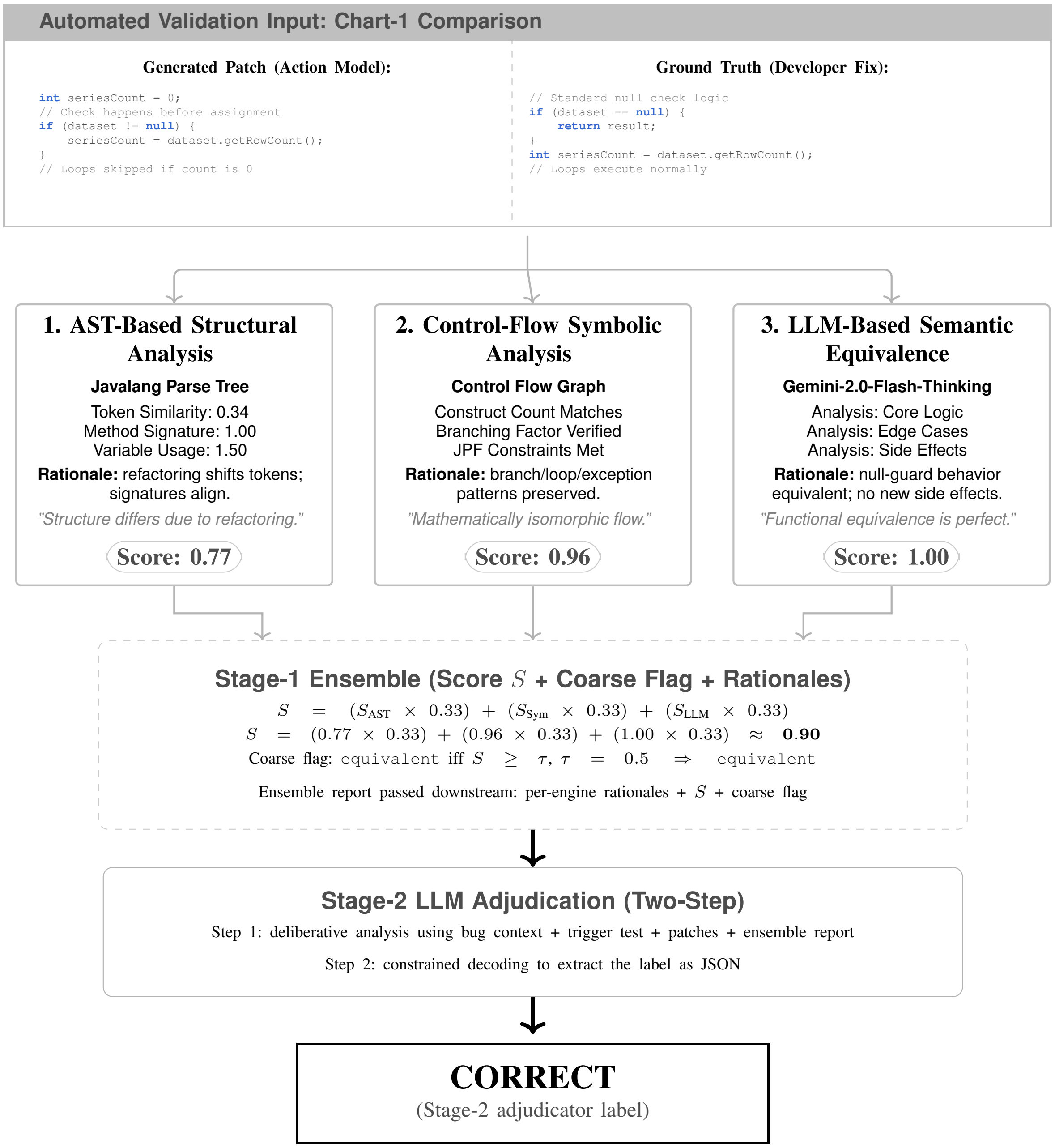
### A. Validation Methodology

The validation pipeline serves as an automated adjudicator of patch quality by subjecting candidate patches to a hierarchical filtration sequence. Importantly, this is an automated adjudicator: once a candidate patch has been synthesized, the pipeline itself performs all subsequent filtering, analysis, and labeling, with no subjective manual post hoc judgment in the benchmarking.

The candidate patches are evaluated based on the diverse process shown in Figure 4.

The core methodological framework of this chapter is predicated on a threefold approach to guarantee correctness in both structural and semantic aspects. Each validator produces (i) a normalized equivalence score in the range $[0, 1]$ and (ii) a short machine-generated explanation that summarizes the evidence behind the score:

1) **AST-Based Structural Analysis:** The produced patch and the ground truth are expressed as Abstract Syntax Trees (ASTs). A node comparison metric evaluates and measures similarity on the basis of node types, method declarations, and variable declarations, together with an explanation outlining the most important structural differences and similarities.
2) **Control-Flow Symbolic Analysis:** The framework builds an approximation of behavioral equivalence using the comparison of Control Flow Graphs. The process examines the control flow patterns and the structures of exception handling to arrive at a control-flow similarity degree and an explanation of the main differences.
3) **LLM-Based Semantic Equivalence:** Aware that structural equivalence is not always synonymous with semantically equal concepts, the Gemini-2.0-Flash-Thinking


Automated Validation Input: Chart-1 Comparison
Generated Patch (Action Model):
Ground Truth (Developer Fix):

```
int seriesCount = 0;
// Check happens before assignment
if (dataset != null) {
    seriesCount = dataset.getRowCount();
}
// Loops skipped if count is 0
```

```
// Standard null check logic
if (dataset == null) {
    return result;
}
int seriesCount = dataset.getRowCount();
// Loops execute normally
```

1. AST-Based Structural Analysis
Javalang Parse Tree
Token Similarity: 0.34
Method Signature: 1.00
Variable Usage: 1.50
Rationale: refactoring shifts tokens; signatures align.
"Structure differs due to refactoring."
Score: 0.77
2. Control-Flow Symbolic Analysis
Control Flow Graph
Construct Count Matches
Branching Factor Verified
JPF Constraints Met
Rationale: branch/loop/exception patterns preserved.
"Mathematically isomorphic flow."
Score: 0.96
3. LLM-Based Semantic Equivalence
Gemini-2.0-Flash-Thinking
Analysis: Core Logic
Analysis: Edge Cases
Analysis: Side Effects
Rationale: null-guard behavior equivalent; no new side effects.
"Functional equivalence is perfect."
Score: 1.00
Stage-1 Ensemble (Score $S$ + Coarse Flag + Rationales)
$S = (S_{\text{AST}} \times 0.33) + (S_{\text{Sym}} \times 0.33) + (S_{\text{LLM}} \times 0.33)$
$S = (0.77 \times 0.33) + (0.96 \times 0.33) + (1.00 \times 0.33) \approx \mathbf{0.90}$
Coarse flag: equivalent iff $S \geq \tau$, $\tau = 0.5 \Rightarrow$ equivalent
Ensemble report passed downstream: per-engine rationales + $S$ + coarse flag
Stage-2 LLM Adjudication (Two-Step)
Step 1: deliberative analysis using bug context + trigger test + patches + ensemble report
Step 2: constrained decoding to extract the label as JSON
CORRECT
(Stage-2 adjudicator label)

Fig. 5: Walkthrough of the automated validation pipeline applied to the Chart-1 bug.

system operates as a standalone evaluation tool that rates the functional logic, edge scenarios, and possible side effects of the generated semantically equivalent concepts.

**Stage-1 Ensemble (Equivalence Report):** The three validators produce normalized equivalence scores $s_i \in [0, 1]$. These are combined into an aggregate equivalence metric S defined as

$$S = w_{\text{AST}}\, s_{\text{AST}} + w_{\text{Sym}}\, s_{\text{Sym}} + w_{\text{LLM}}\, s_{\text{LLM}}, \tag{1}$$

where the weights are uniform $w_{\text{AST}} = w_{\text{Sym}} = w_{\text{LLM}} = 0.33$. A hard threshold $\tau = 0.5$ is introduced only to generate a deterministic flag for *coarse equivalence* (the two sequences are considered equivalent if $S \geq \tau$). This flag is a summary signal. The object that passes on to the next stages is an Ensemble report containing the rationales for the individual engines, together with S and the flag. The internal confidence scores are evaluated and recorded for diagnostic purposes but do not affect decision-making.

**Stage-2 Final Adjudication (Correct/Plausible/Incorrect):** The final decision is generated with the help of an automated adjudicator based on an LLM that is conditioned on: (i) the buggy function and issue description, (ii) the trigger test context, (iii) the ground-truth patch, (iv) the generated patch, and (v) the Stage-1 Ensemble report. The adjudicator uses a fixed rubric: *Correct* is assigned to a verdict in case the generated patch is semantically equivalent to the ground-truth fix; *Plausible* is assigned if the patch passes the test filter but is behaviorally different from the ground truth; and *Incorrect* is assigned in all other cases. We realized the adjudication process as a two-step procedure: first, a deliberative analysis pass and second, a constrained decoding pass that extracts only the final label within a fixed JSON schema. All adjudicator calls make use of deterministic decoding with fixed seeds (T = 0), so as to minimize run-to-run variance. In practice, we treat the Stage-1 report as evidence; an adjudicator may overwrite a coarse flag when bug context and patch semantics suggest disagreement.

Figure 5 gives a concrete example of this process for the Chart-1 bug. Despite the large structural edits in the generated patch (AST Score: 0.77), the Stage-1 Ensemble combines the control-flow evidence at 0.96 with the semantic assessment from LLM at 1.00. Based on the report and the bug/test context, the Stage-2 human adjudicator assigns the final outcome of *Correct* to indicate semantic equivalence to the developer patch.

### B. Evaluation Metrics

To formally quantify the performance of the proposed framework, two standard metrics have been used. Though pass/fail rates are informative, the present definition enables discrimination between superficial test passage (which may reflect overfitting) and genuine semantic repair.

Let $B$ be the set of all bugs analyzed. For each bug $b \in B$, let $P_b$ be the set of candidate patches. Let $V(p, b_{gt})$ be the validation function that maps a patch $p$ and ground truth $b_{gt}$ to a label drawn from $\{$Correct, Plausible, Incorrect$\}$. In our formulation, $V(p, b_{gt})$ is computed by our automated two-stage validator (see Fig. 4–5): a multi-engine equivalence report followed by an LLM adjudicator that assigns $\{$Correct, Plausible, Incorrect$\}$ under a fixed rubric, enabling scalable quantification without manual post hoc judgment.

The **Correct Repair Rate (CRR)** captures the system's ability to produce a patch which is functional, trustworthy, and maintainable. It measures the share of bugs for which ROT produced at least one semantically equivalent patch.

$$\mathrm{CRR} = \frac{\left|\left\{b \in B \;\middle|\; \exists\, p \in P_b :\; V(p, b_{gt}) = \text{Correct}\right\}\right|}{|B|} \times 100\% \tag{2}$$

The **Plausible Repair Rate (PRR)** measures the system's exploratory competence, which is defined by the percentage of bugs where at least one generated patch passed all test cases, regardless of whether the patch reflected the exact logic the developer intended. A high PRR indicates significant potential for "human-in-the-loop" workflows where the developer may embrace a plausible alternative.

$$\mathrm{PRR} = \frac{\left|\left\{b \in B \;\middle|\; \exists\, p \in P_b :\; V(p, b_{gt}) \in \{\text{Correct, Plausible}\}\right\}\right|}{|B|} \times 100\% \tag{3}$$

*Metric Alignment:* Correct, Plausible-only, and Incorrect are treated as disjoint labels for granular analysis. However, the **PRR** reported herein is computed as $(\text{Correct} \cup \text{Plausible-only})/\text{Total}$ to maintain consistency with prior work, in which "Plausible" generally represents all test-suite-adequate patches.

### C. Experimental Setup and Results

**Setup:** This work utilizes the Defects4J benchmark, focusing on the subset containing 522 single-function bugs. The Thinker model, LLaMA 3.3 70B, and the Action model, DeepSeek-R1-Distill-Llama-70B, were deployed using Groq and Together AI inference APIs, respectively, and each bug was processed at an average of less than 60 seconds.

**Results:** The empirical results are given in Table II. ROT yielded a **CRR of 41.2%**, correctly repairing 215 of the 522 bugs. Following standard APR reporting conventions, (where 'Plausible' includes 'Correct'), **ROT achieves a PRR of 83.1%** which significantly improves SRepair's [13] PRR of 68.4% on the same subset. In order to further validate the reliability of these metrics, we tested our automated validator on 300 human-verified patches from SRepair [13], achieving a 95.7% classification agreement.

TABLE II: Overall Performance of ROT on Defects4J

| Metric | Count | Percentage |
|---|---|---|
| Total Bugs Analyzed | 522 | 100% |
| Bugs with Correct Patches | 215 | 41.2% |
| Bugs with Plausible Patches (Only) | 219 | 42.0% |
| Bugs with Incorrect Patches (Only) | 88 | 16.8% |

A detailed project-by-project breakdown, comparing ROT directly against the SRepair framework, is provided in Table III.

## V. Discussion

The empirical results reported in Section IV demonstrate the utility of ROT in the state-of-the-art automatic program repair setting using LLMs. This section begins with understanding the place of these empirical results in the context of an in-depth comparison, followed by analyzing discrepancies as well as points of convergence with previous work, exploring relationships among the empirical points, understanding implications for researchers, and noting limitations to this approach.

### A. Comparative Performance and State-of-the-Art Assessment

The results, summarized in Table III, position ROT at the cutting edge of state-of-the-art technology for function-level repair. For the sake of rigorous comparison, it is necessary to consider the underlying repair performance measures, since presentation is varied within the literature. In related literature,

TABLE III: Head-to-Head on Defects4J (SRepair vs. ROT): Aligned Plausible and Correct Repair Rates

| | | **SRepair** | | **ROT (Ours)** | |
|---|---|---|---|---|---|
| **Project** | **Bugs** | **Plausible**† (PRR) | **Correct** (CRR) | **Plausible**† (PRR) | **Correct** (CRR) |
| **Defects4J 1.2** | | | | | |
| Chart | 16 | 14 | 13 | 15 | 12 |
| Closure | 105 | 56 | 47 | 89 | 26 |
| Lang | 42 | 32 | 26 | 37 | 28 |
| Math | 74 | 55 | 42 | 64 | 35 |
| Mockito | 24 | 12 | 11 | 20 | 9 |
| Time | 16 | 7 | 7 | 13 | 7 |
| **Subtotal D4J 1.2** | **277** | **176** | **146** | **238** | **117** |
| **Defects4J 2.0** | | | | | |
| Cli | 28 | 19 | 18 | 23 | 13 |
| Codec | 11 | 11 | 11 | 10 | 7 |
| Collections | 1 | 1 | 1 | 1 | 1 |
| Compress | 36 | 28 | 21 | 32 | 14 |
| Csv | 12 | 11 | 11 | 9 | 5 |
| Gson | 9 | 9 | 8 | 9 | 5 |
| JacksonCore | 13 | 10 | 10 | 10 | 5 |
| JacksonDatabind | 67 | 45 | 33 | 47 | 22 |
| JacksonXml | 5 | 3 | 2 | 4 | 2 |
| Jsoup | 53 | 39 | 35 | 46 | 20 |
| JxPath | 10 | 5 | 4 | 5 | 4 |
| **Subtotal D4J 2.0** | **245** | **181** | **154** | **196** | **98** |
| **Overall Counts** | **522** | **357** | **300** | **434** | **215** |
| **Overall PRR / CRR (%)** | **100%** | **68.4%** | **57.5%** | **83.1%** | **41.2%** |

†Plausible denotes test-suite-adequate patches: **Correct** ∪ **PlausibleOnly**. Disjoint PlausibleOnly counts are 57 for SRepair (357–300) and 219 for ROT (434–215).

such as SRepair [13], "Correct" patches are discovered from the pool of test-suite-adequate patches; hence, the "Plausible" metric reported is non-disjoint (includes "Correct" patches). To enable a fair comparison, we follow this conventional definition presented in both frameworks in Table III, where "Plausible" columns indicate the union of all test-suite-adequate patches, $Correct \cup PlausibleOnly$, marked with †.

With this aligned definition in mind, ROT is able to attain a Plausible Repair Rate (PRR) of 83.1% (434/522 bugs), a significant improvement over the 68.4% (357/522) rate attained by SRepair. We argue that in the autonomous software engineering domain, PRR is the metric that has the most operational meaning for declaring state-of-the-art performance. Though the Correct Repair Rate (CRR) is measured based on strict semantical equivalence to a particular past ground truth, our Plausible Repair Rate is more aligned towards test-suite adequacy—fixes that satisfy the constraints available in the test suit and go through the validation gates. As pointed out above for real-world deployments (Section IV.A), our validation procedure is a multifaceted state validation procedure equipped for dealing with overfitting from various semantical perspectives; hence, the high Plausible Repair Rate is an indicator of its actual operational validity under these constraints.

### B. Divergences from Prior Research

ROT's performance profile helps identify major deviations from existing approaches, which makes it possible to define architectural differences of the new framework.

*1) Reasoning Depth vs. Suggestion Generation:* The biggest changes over existing work relate to the depth of the reasoning phase. Although SRepair [13] refined the dual-LLM framework by distinguishing "repair suggestions" and patch generation, its suggestion model is a single-step process that pinpoints causes and makes recommendations within a single pass. In comparison, ROT's Thinker module demonstrates recursive CoTs over three different phases: preliminary root cause identification, generating varied strategies, and refining strategies and tactics down to their final form. Over multiple passes, this enables ROT to recommend multiple strategies per problem, providing its Action model with more informed direction and overcoming the inherent constraints of statically defined fix patterns mined by humans that have traditionally characterized APR systems [3].

*2) Independence from Natural Language Bug Descriptions:* A key operational aspect of ROT's design is its independence from natural language bug descriptions. This is because other techniques have already been using developer-provided comments whenever available, which can prove to be a rich

TABLE IV: Comparison of Modern LLM-based APR Frameworks

| Framework | Granularity | Core Technique | Model Type | Key Contribution |
|---|---|---|---|---|
| **AlphaRepair** [15] | Statement/Hunk | Zero-Shot Infilling | Open-Source (CodeBERT) | Reframed APR as a cloze-style task, removing the need for bug-fix datasets. |
| **ChatRepair** [9] | Statement/Hunk | Conversational Refinement | Closed (ChatGPT) | Introduced an iterative feedback loop using test failures to guide patch generation. |
| **GiantRepair** [16] | Statement/Hunk | Hybrid (LLM + Analysis) | User-Defined | Used LLM-generated ”patch skeletons” to constrain the search space for higher-quality patches. |
| **ThinkRepair** [17] | Function-Level | Self-Directed CoT (Collection + Fixing Phases) | User-Defined | Uses a two-phase self-directed framework that first collects reusable chains-of-thought and then applies few-shot CoT reasoning with optional test-feedback to fix functions, outperforming prior APRs on Defects4J and QuixBugs. |
| **SRepair** [13] | Function-Level | Dual-Model (Suggestion + Patch) | User-Defined | Pioneered a dual-LLM approach for function-level repair, separating reasoning from generation. |
| **Repair of Thought (ROT)** | Function-Level | Dual-Model (CoT Reasoning + Action) | Open-Source | Decouples deep CoT reasoning from precise patch synthesis using an entirely open-source stack, and introduces a multifaceted automated validator for scalable semantic patch evaluation. |

source of context information for intended behavior. What is significant here is that although operational purely on execution artifacts such as code, test cases, and error messages, structured reasoning is capable of making up for lacking descriptions which is a point of practical relevance for real-life deployment environments where such information may not be available.

*3) Open-Source Model Performance with Cost and Transparency Advantages:* Our findings challenge the assumption in current APR research, which holds that state-of-the-art performance is not possible without proprietary models. ChatRepair obtained 162 correct repairs in Defects4J [9] for GPT-3.5-turbo, whereas other current advances use GPT-4. For comparison, ROT scored 215 correct repairs with open-source models (LLaMA 3.3 70B, DeepSeek-R1-Distill-Llama-70B) only, showing how architectural novelties, in particular the separation of Thinker & Action models, successfully close the capability gap generally associated within APR systems. This approach has very important implications in practice, going beyond the issue of reproducibility. Those organizations that possess existing GPU infrastructure can run ROT without any additional cost, as they can choose to self-host open-source models without paying API costs on a per-token basis. Even in cases using external providers like TogetherAI and Groq, costs are significantly lower in comparison to others, as self-hosted open-source models incur costs 10 to 20 times lower than GPT-4 models. Additionally, self-hosted models remove all concerns of potential data leaks altogether, as code does not travel out of their infrastructure. This feature is pertinent in regulated settings (healthcare, finance, and government) that may not fully comply if they share their code on external APIs. The open nature of model weights also allows them to audit models' behavior and customize models within their domains, none of which are possible in traditional APIs.

### C. Convergent Findings with Prior Research

Despite these differences, there are a number of results that support and expand current knowledge in the APR community.

*1) Validation of Function-Level Repair Benefits:* Our results strengthen the central thesis of function-level APR, namely, that removing precise statement-level fault localization (traditionally brittle and expensive) does not weaken repair quality, but clearly establish that with proper test failure hints, a PRR of 83.1% is sufficient to identify and fix faulty code within whole functions using an LLM with a framework such as ROT. This is fully consistent with other findings from other frameworks such as **SRepair** [13] and **ThinkRepair** [17], which together confirm that function-level functionality is a sound approach for LLM-based repair. Furthermore, evidence that iterative/feedback-based repair is effective at both statement and function levels, as represented by systems such as **ChatRepair** [9] and **AlphaRepair** [15], strongly argues for multi-step reasoning, regardless of whether conversational or CoT, to be considerably more effective than stand-alone generation.

*2) The Dual-Model Architectural Advantage:* The success of ROT confirms the hypothesis that reasoning and generation should be decoupled. A monolithic design, which seeks to tackle bug understanding and correct code synthesis with the same model, faces the challenge that improving code synthesis quality could hurt bug understanding, and so forth. Our results add evidence that specialization, wherein distinct system parts are each optimized for logical reasoning or syntactic detail, among others, yields superior outcomes. This pattern appears across various forms in the APR literature: SRepair's suggestion-generation split [13], GiantRepair's [16] LLM-plus-program-analysis combination, and multi-agent frameworks like AgentVerse [18] that decompose complex problem-solving into specialized roles.

*3) Bug Type Success Patterns:* As reported by the findings from AlphaRepair [15], ChatRepair [9], GiantRepair [16], and ThinkRepair [17], ROT demonstrates strong performance on:

- **Logical Errors**: Reversals in Boolean expressions, off-by-one errors, or wrong operators for comparators in which corrections require minimal code-level modifications but require understanding of program behavior.
- **Null Handling**: Inadequate null checking or improper null handling logic, where diagnostic information can be extracted from test failure messages (NullPointerException).
- **API Misuse**: Incorrect API calls or usage patterns, where knowledge about correct usage patterns is implicitly present in the training data of the LLM.

These patterns exist across individual models and imply some inherent benefits of LLM-based APR that have been captured by current design choices.

### *D. Analysis of Result Correlations*

Besides the metrics, analyzing correlations within our results reveals some insights into ROT's operational features.

*1) PRR-CRR Gap Analysis:* ROT achieves a PRR of 83.1% but a CRR of 41.2%, leaving a difference of 41.9%. This difference can be attributed to bugs for which ROT produces convincing but incorrect patches. The production of convincing but incorrect patches has been widely cited in APR as a challenge; patches that pass all test cases but possess semantically different behaviors from the expected solution. This "overfitting" problem, identified in seminal APR studies, most notably by Qi et al. [4], which demonstrated that patches passing a test suite often fail to resolve the actual underlying defect. Our work aims to address this problem using our multimodal validation methodology containing abstract syntax tree (AST) analysis, heuristic from symbolic execution, and semantic analysis through LLMs which are all techniques designed to go further than test pass behavior and analyze semantic equivalence.

*2) Error Message Informativeness:* We note that bugs with more specific exception traces (e.g., `ArrayIndexOutOfBoundsException` with index values, `NullPointerException` with specific stack traces) seem to lead to more successful repair tasks compared to bugs with more general assertion violations (`expected:<X> but was:<Y>`). While expected, our findings offer further insight into the debugging capacity benefits brought to the Thinker by more informed execution feedback.

### *E. Implications for Researchers*

Our work outlines some new directions for future research and makes methodological contributions for the APR research community.

*1) The Thinker-Action Architecture as a Research Template:* The explicit distinction between reasoning and generation provides a modular framework for subsequent research. Researchers can optimize each part in isolation:

- **Thinker Variations**: Exploring the effects of differing levels of reasoning (2-step, 3-step, or N-step reasoning) or differing methods of prompting (Self-Consistency, or Tree of Thoughts), or specialized reasoning models.
- **Action Model Selection**: Analyzing the best model properties for performing code synthesis based on structured repair plans, including comparing general-purpose models like LLMs against models designed specifically for code.
- **Interface Design**: Analysis of which information needs to be passed from Thinker to Action. Currently, we are passing natural language strategies; other representation schemes, such as pseudo-code skeleton transformations, could be more effective.

*2) Automated Validation as a Research Enabler:* The proposed ensemble validation approach, which attained a parity level of 95.7% to the human validation, targets a significant gap in APR studies, which have been hindered for quite some time due to the dependency on manual validation of the patch for assessment purposes. The study proves that the synergistic effect of the AST-based structural approach, control flow, and the LLM semantic approach can come very close to human validation, providing an actionable approach that can significantly expedite the assessment of future APR studies but must be noted to be for *evaluation purposes* only, not for deployment purposes, where the validation based on the test suite is supreme.

*3) Benchmark Considerations:* The concentration of APR research on Defects4J, while enabling comparability, raises questions about generalization. At a time when a number of frameworks are succeeding at a high level on Defects4J's single function dataset, new tasks must be introduced that cover different languages than Java, different functionalities, or that include more contemporary efforts. This would help, as a whole, the field better gauge progress.

### *F. Limitations and Threats to Validity*

We acknowledge several limitations and threats that contextualize our findings.

*1) Threats to Internal Validity:* **Evolving Model Landscape.** A significant consideration is that LLM capabilities represent a rapidly evolving landscape. The models used for ROT (LaMA 3.3 70B for reasoning, DeepSeek-R1-Distill-Llama-70B for code writing) represented what was current at their point of release; other models can have different performance. It is, however, highly flexible to switch models whenever needed based on progress made: as better models become open source, it is possible to easily adapt to them without changing system architecture. The value is not in what is created here.

**The Role of Proprietary Models in Validation.** While ROT's repair loop is entirely open-source, our validation pipeline employs Google's Gemini-2.0-Flash-Thinking model for semantic equivalence assessment. This design decision is worth discussing. Our decision to limit repair model generation to only open-source models was to prove that good APR results can be achieved independent of proprietary models. The validation stage is a separate concern and is to provide a truth baseline for evaluation purposes only. In a practical setting, a test suite validation determines if a patch is correct. Our use of Gemini models in evaluation for research purposes does not invalidate our claim that high-quality program repair could be performed using only open-source models. In a practical setting, adoption of ROT would utilize a test suite and not our evaluation method. We reached 95.7% agreement with human evaluations, proving our system to be reliable for evaluation purposes and accepting its limitation as a subjective system using an automated system.

*2) Threats to External Validity:* **Language and Benchmark Scope.** The evaluation of ROT has been conducted exclusively on the Java language on the Defects4J dataset. While the underlying models support multiple languages, we cannot claim generalization to Python, JavaScript, or other ecosystems without additional empirical validation. Additionally, Defects4J, while widely used, may not represent the full spectrum of real-world bugs.

**Single-Function Subset.** The evaluation focuses on the 522 single-function bugs in the Defects4J dataset. This excludes multi-function bugs that may require cross-file reasoning, potentially overstating ROT's practical applicability to certain bug categories.

## VI. Conclusion

This paper presents Repair of Thought (ROT), a novel function-level Automated Program Repair (APR) framework that addresses some limitations observed in modern Large Language Model (LLM)-based APR systems. By explicitly decoupling bug comprehension from patch synthesis through a dual-model "Thinker-Action" architecture, ROT shows that transparent, reasoning-driven approaches can achieve state-of-the-art performance while maintaining exclusive reliance on open-source models. Empirical evaluation on the Defects4J benchmark substantiates the effectiveness of ROT's methodology and establishes it as a new state-of-the-art in function-level repair. When employing aligned metric definitions where "plausible" encompasses all test-suite-adequate patches, ROT achieves a **Plausible Repair Rate (PRR) of 83.1%** (434/522 bugs), which considerably outperforms previously reported benchmarks on the same dataset. Although we do report a strict Correct Repair Rate (CRR) of 41.2% (215 bugs) for historical comparability, we argue that the significantly higher PRR—validated by our automated multi-engine validation pipeline—constitutes the more meaningful indicator of ROT's utility as an autonomous agent capable of resolving active software defects. This performance is all the more notable given ROT's limitations: it operates exclusively with open-source LLMs (LLaMA 3.3 and DeepSeek-R1-Distill-Llama-70B) and requires only direct execution artifacts without dependence on natural language bug descriptions or proprietary APIs.

The main contributions of this paper go beyond standard performance measures. First, the structured Chain-of-Thought reasoning process of ROT brings transparency to the repair logic; it thus avoids the ”black box” issue that has prevented industrial acceptance of automated repair tools. The function-level approach of the framework avoids the computational overhead and brittleness that come with statement-level fault localization. The automated validation pipeline based on LLM classification using reasoning reaches 95.7% agreement with human annotations (287/300 correct classifications) during patch evaluation. In this context, ROT plays the dual role of a repair framework and an automated evaluation harness that allows for large-scale semantic patch assessment and reduces the need for expensive human validation when benchmarking APR tools.

Success in ROT on various bug types (especially logical mistakes, API misuse, and data handling issues) demonstrates the strength of its reasoning-based approach. While challenges remain concerning more sophisticated structural changes or project-specific implementations, the framework opens new perspectives for a comprehensively available, transparent, and affordable APR.

The impact of this work goes beyond technical contributions. By showing that state-of-the-art APR performance is achievable without proprietary models, ROT democratizes access to sophisticated repair capabilities to researchers and practitioners worldwide. Because of this accessibility, coupled with the framework's transparency, ROT can serve as a basis for community-driven innovation in the area of APR.

### *Ethics Statement*

The work proposed here is intended to increase the transparency and applicability of APR and thus promote software reliability and security. This study only uses open-source models LLaMA 3.3, DeepSeek-R1, and public benchmarks Defects4J, so full reproducibility and broad accessibility to the research community are obtained, hence democratizing software engineering research. The proposed technique leverages a dual-model framework that operates over public code repositories and does not process Personally Identifiable Information, and hence the approach is friendly to privacy with no associated privacy risks. The separation of the reasoning mechanism and generation of code phases further enables clear auditing of automated repairs, addressing the typical ”black-box” concern with conventional LLM-based APR, but also enhances environmental considerations due to reduced reliance on repeated model retraining in cost-effective inference.

### *AI Assistance Disclosure*

LLMs have been utilized in helping write parts of this paper and in related research activities. Let it be clear that none of these LLMs, being used to aid in writing, are related or owned by the open-sourced models LLaMA 3.3 and DeepSeek-R1 reviewed under the ROT framework. All research contributions of general applicability, including but not limited to two-model architectures, implementations of research using Defects4J, designs of automated validation, and subsequent analyses, are original contributions of the authors. LLM use has been confined to writing refinement, literature searching, or formatting, and authors maintained full control over all research outputs to ensure accuracy.

## VII. Future Work

Although ROT performs well in APR, there are still several avenues for future investigations.

**Enhanced Explainability:** Although the Chain-of-Thought reasoning of ROT provides insight into the repair process, further work could explore ways to make patches more interpretable. This might include methods for making the translation from reasoning to patch more accessible to developers, which would potentially increase developer trust and adoption by incorporating techniques from explainable AI [10].

**Validation Refinement:** While our multifaceted validation pipeline achieves a 95.7% agreement with human annotation, further validation dimensions might be explored in order to increase accuracy, such as the use of more advanced program analysis techniques [16] or leveraging code-understanding model improvements [2].

**Broader Applicability:** Currently, ROT works on Java programs from the Defects4J benchmark. A natural next step involves modifying the framework to accept multiple programming languages and a variety of bug categories. Performance studies on industrial codebases would also be much welcomed, as this would provide insights into real-world applicability, extending recent work on scaling APR systems to production settings [9].

**Architectural Enhancements:** The dual-model architecture investigated in this work could be complemented by alternative model configurations or also by feedback mechanisms between models. The exploration of optimal model selection strategies and prompt engineering techniques [14] can also lead to performance improvements.

**Scalability Considerations:** As emergent LLMs remain under active development, determining how best ROT can be adapted to take advantage of next-generation models is an important challenge with regard to maintaining open-source principles. This includes investigating trade-offs involving model size, inference speed, and quality of repair [13].

Overall, these directions suggest that although ROT offers a good starting point for transparent, open-source APR, considerable scope exists to improve the state-of-the-art in developing automated debugging tools fit for practical deployment.

## References

[1] C. S. Xia, Y. Wei, and L. Zhang, ”Automated Program Repair in the Era of Large Pre-trained Language Models,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[2] N. Jiang, K. Liu, T. Lutellier, and L. Tan, ”Impact of Code Language Models on Automated Program Repair,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[3] D. Kim, J. Nam, J. Song, and S. Kim, ”Automatic patch generation learned from human-written patches,” in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[4] Z. Qi, F. Long, S. Achour, and M. Rinard, ”An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[5] A. Vaswani et al., ”Attention is all you need,” in *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[6] M. Tufano et al., ”On Learning Meaningful Code Changes via Neural Machine Translation,” *arXiv preprint arXiv:1901.09102v1*, 2019.

[7] S. N. Svyatkovskiy et al., ”IntelliCode Compose: Code Generation Using Transformer,” in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[8] X. Hu et al., ”Deep Code Comment Generation,” in *Proceedings of the 26th International Conference on Program Comprehension (ICPC)*, 2018.

[9] C. S. Xia and L. Zhang, ”Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

[10] M. T. Ribeiro, S. Singh, and C. Guestrin, ”“Why should I trust you?”: Explaining the predictions of any classifier,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[11] T. T. Vu et al., ”Automated Description Generation for Software Patches,” *arXiv preprint arXiv:2402.03805v2*, 2024.

[12] J. Xu, Y. Fu, S. H. Tan, and P. He, ”Aligning LLMs for FL-free Program Repair,” *arXiv preprint arXiv:2404.08877v1*, 2024.

[13] J. Xiang et al., ”How Far Can We Go with Practical Function-Level Program Repair?,” *arXiv preprint arXiv:2404.12833v2*, 2024.

[14] G. Li et al., ”Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024.

[15] C. S. Xia and L. Zhang, ”Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[16] F. Li, J. Jiang, J. Sun, and H. Zhang, ”Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis,” *arXiv preprint arXiv:2406.00992v2*, 2024.

[17] X. Yin et al., ”ThinkRepair: Self-Directed Automated Program Repair,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

[18] W. Chen et al., ”AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors,” *arXiv preprint arXiv:2308.10848v3*, 2023.