Evaluator Bias in LLM-Based Program Repair: A Disagreement Analysis on Android Resource Leaks

Andrezza Bonfim
Centro de Informática,
Universidade Federal de Pernambuco
Recife, PE, Brasil
amb8@cin.ufpe.br

Leopoldo Teixeira
Centro de Informática,
Universidade Federal de Pernambuco
Recife, PE, Brasil
lmt@cin.ufpe.br

Abstract

Proper management of system resources in Android applications remains a challenge due to the need for explicit release calls during component lifecycles. While Large Language Models (LLMs) are increasingly deployed to automate resource leak repairs, evaluating these repairs introduces substantial metric-dependent biases. Rather than measuring absolute model capabilities, this study diagnoses validator disagreement and evaluator bias across traditional and modern verification paradigms. Using 300 patches generated by GPT-5.4, Claude Opus 4.6, and Gemini 3.1 as a diagnostic vehicle, we evaluate three distinct verification layers: lexical analysis (TF-IDF), structural verification (AST nodes), and neural semantic extraction (GraphCodeBERT). Our results show that the choice of evaluation method drastically alters the reported effectiveness of automated repairs. Specifically, string-matching metrics such as TF-IDF introduce severe false-negative biases, underestimating modern repair patterns (such as try-with-resources) by up to 20 percentage points due to a historical mismatch with legacy ground-truth datasets.

Keywords

Evaluator Bias, Program Repair, Large Language Models, Resource Leaks, Android

1 Introduction

Resource management in Android applications, such as database cursors and network connections, requires explicit release by the developer during the component's lifecycle. When neglected, this step results in resource leaks, causing systemic instability and performance degradation, a problem widely documented in the *dataset* DroidLeaks [22], one of the primary reference databases for Android failures and a key reference for this study.

While the use of Large Language Models (LLMs) has gained popularity in Automated Program Repair (APR) to fix these flaws [42], the literature currently faces a dual challenge: using generative AIs to resolve defects and, simultaneously, establishing evaluation methods capable of measuring the performance of these corrections in a fair and automated manner.

The apparent effectiveness of LLMs in code repair has traditionally been evaluated via predefined code attributes and manual similarity rules [35]. The use of TF-IDF, although common for generating vector representations, is based on lexical frequency and is unable to capture semantic relationships [16]. This approach is limited to seeking exact character matches, ignoring the syntactic evolution of languages. In preliminary tests, it was observed that technically superior fixes, such as the use of *try-with-resources*, are systematically classified as incorrect because they fail to replicate `finally` blocks or explicit calls to the `.close()` method present in legacy *datasets*. Since the DroidLeaks *dataset* reflects development practices from earlier Java versions, there is a mismatch between the historical *ground truth* and the modern solutions generated by contemporary LLMs.

This paper addresses this problem by proposing a comparative diagnosis of how evaluation paradigms distort the reported performance of automated repairs. Using 300 resource leak patches generated by GPT-5.4, Claude Opus 4.6, and Gemini 3.1 models as an experimental baseline, we isolate the validator's impact. Rather than proposing a new absolute oracle, this study uses three distinct verification layers (Lexical, Structural, and Semantic) to analyze and map systematic metric disagreement on identical repair suggestions.

To guide this investigation, we establish a progressive rationale across three research questions. First, we establish a baseline of LLM repair attempts under traditional evaluation paradigms. Second, because a model's apparent success might be an artifact of the metric rather than code quality, we isolate the evaluator's impact by comparing alternative structural and semantic validation paradigms on identical patches. Finally, we investigate the root cause of these metric discrepancies, specifically testing whether modern, clean syntax triggers false negatives in legacy-trained lexical checkers. We express these objectives through the following research questions:

- **RQ1:** What is the baseline acceptance rate of LLM models (GPT-5.4, Claude Opus 4.6, and Gemini 3.1) in repairing resource leaks in Android under traditional metrics?
- **RQ2:** To what extent does the choice of evaluation method (Lexical, Structural, or Semantic) alter the reported metrics of repair effectiveness?
- **RQ3:** How does the adoption of modern syntactic patterns impact the false-negative bias in traditional, string-matching validators?

In addressing these questions, our results demonstrate that structural and semantic validation substantially mitigates textual bias. When transitioning from a lexical evaluation (under which Claude Opus 4.6 received a 64% acceptance rate) to a semantic validator, Claude's acceptance rate rose to 72.67%, with analogous gains for GPT-5.4 and Gemini 3.1. These results indicate that string-matching-based checks are insufficient for evaluating AI-generated code, necessitating the adoption of methods that consider the structure and intent of the adjustment. The contributions of this paper are:

- **Empirical disagreement benchmark:** A multi-model dataset mapping the acceptance criteria fluctuations of GPT-5.4,

Claude Opus 4.6, and Gemini 3.1 across 300 Android resource leak patches.
- **Bias analysis in lexical metrics:** Demonstration that TF-IDF-based validators underestimate correct repairs by ignoring logical equivalences.
- **Diagnostic measurement of evaluator bias:** A side-by-side comparison showing that, on the same 300 patches, Gemini 3.1's apparent acceptance rate moves from 24.33% (lexical) to 45.00% (semantic), a 20-point swing attributable strictly to the validator selection rather than patch quality.

Following this introduction, Section 2 reviews background concepts, Section 3 reviews related work in APR and resource leaks, Section 4 details the triangulation methodology, Section 5 presents the results, Section 6 discusses their implications, Section 7 covers threats to validity, and Section 8 concludes the work and points toward future directions.

2 Background

The effectiveness of vulnerability detection and similarity analysis depends directly on how source code is converted into processable representations. This section details the methods adopted in this research, categorized by their level of abstraction. The analysis begins with statistical vectorization using TF-IDF, progresses to hierarchical representation via AST, and concludes with the use of GraphCodeBERT, which integrates data flow to achieve deep semantic understanding. This progression reflects the evolution of the field: moving from superficial lexical analysis to the interpretation of structural software logic.

Term Frequency-Inverse Document Frequency (TF-IDF): Transforms unstructured textual data into numerical vector representations [25, 30], balancing term frequency with its rarity. In vulnerability detection, it converts programming tokens into vectors capturing the relative importance of instructions within a project [31, 37].

Abstract Syntax Tree (AST): A hierarchical representation capturing program structure while abstracting away constructs lacking semantic value [2]. Ensuring tree generation accuracy is critical for preserving intended semantics [4]. ASTs are the standard representation for static analysis and a crucial feature source for machine learning tasks.

GraphCodeBERT: Extends CodeBERT by incorporating the Data Flow Graph (DFG) to capture semantic relationships between variables unobservable in token sequences [13, 38]. While GraphCodeBERT excels at capturing data dependencies, Explainable AI techniques reveal it primarily targets critical tokens defining program behavior, aiding fault diagnosis when models deviate from training patterns [24, 27].

3 Related Work

To establish the methodological foundation proposed in this study, this section reviews traditional lexical and structural analysis methods, advances in semantic understanding via GraphCodeBERT, and the state-of-the-art in LLM-driven APR for the Android ecosystem.

3.1 TF-IDF

Recent literature demonstrates that TF-IDF remains a powerful tool, being applied both in isolation and in conjunction with frontier Artificial Intelligence models.

3.1.1 Security, Vulnerability Detection, and Code Clones. In the field of Vulnerability Type Identification (VTI), Vo and Nguyen [37] demonstrated that traditional feature extraction based on TF-IDF, when coupled with lightweight models, can outperform complex Deep Learning models. In the context of software maintenance, TF-IDF has also been integrated into security workflows to reduce noise in large databases. Chen et al. [6] used it for the rapid filtering of vulnerable libraries, while Eom et al. [7] proposed CovRL, using TF-IDF-weighted coverage maps to guide JavaScript fault discovery through reinforcement learning.

3.1.2 Algorithm Evolution and Hybrid Models. A clear trend in recent work is the modification of the classic TF-IDF calculation to improve accuracy in classification tasks. Jain et al. [14] proposed a new parameter to express in-class features, correcting the tendency of the original TF-IDF to ignore terms that, while frequent, are highly descriptive of a specific category. Similarly, Lan [16] followed a similar direction by proposing a Term Similarity Weighting Tree (TSWT) that joins semantic information with TF-IDF, circumventing the sparsity problems typical of high-dimensional vectors.

Integration with neural networks has also evolved. In addition to Liang and Niu's [19] work with bidirectional LSTMs, Zhou [44] investigated the combination of TF-IDF with a CNN-LSTM architecture. It reinforces that by using TF-IDF as an input layer, the model can simultaneously capture global text features (via LSTM) and local relevance patterns (via CNN), outperforming purely statistical models.

3.2 AST

The use of ASTs has evolved from traditional static analysis to approaches based on Deep Learning and Large Language Models (LLMs).

3.2.1 Code Understanding, Classification, and Similarity. One of the primary research fronts seeks to improve structural representation for classification and security tasks. Wang et al. [40] proposed the Unified Abstract Syntax Tree (UAST) to enable language-independent program classification. In the domain of security and academic integrity, tools such as SourcePlag [32] use the AST in conjunction with distance metrics (such as Levenshtein distance) to capture the structural essence and robustly detect code plagiarism. Furthermore, in the field of automatic summarization, Lin et al. [20] introduced the BASTS method, which partitions the AST based on the control flow graph and uses Tree-LSTMs to generate more precise descriptions.

3.2.2 AST Integrated with LLMs and Code Generation. The native integration between Neural Networks and ASTs represents the state-of-the-art in intelligent software engineering. Rabinovich et al. [28] introduced Abstract Syntax Networks, a modular architecture coupled to the decoder that ensures code generated by machine learning models is syntactically well-formed. Advancing this line, Jiang et al. [15] proposed TreeBERT, a pre-trained model

that incorporates tree structure directly into learning through Tree Masked Language Modeling (TMLM), outperforming purely textual models in code generation and documentation tasks.

3.2.3 Practical Applications. Beyond understanding, the AST is applied in code correction and maintenance. The VerilogCoder system [9] uses AST-based tracking to debug functional errors in hardware. Similarly, Abdelmalak et al. [1] applied AST embeddings and RAG for the synthesis of SVRF code in semiconductors. Finally, Alikhanifard and Tsantalis [3] developed an AST diffing tool capable of identifying semantic changes and refactorings with much higher precision than plain-text-based tools.

3.3 GraphCodeBERT

The literature has evolved to integrate GraphCodeBERT into complex security and optimization pipelines.

3.3.1 Robustness and Data Adaptation. To improve generalization, Wang et al. [39] proposed semantic-preserving transformations for training examples, enriching data diversity during fine-tuning and helping the model learn features invariant to syntactic alterations. In the field of adversarial robustness, ContraBERT uses contrastive learning to mitigate variable renaming attacks [21], while GraphCodeAttack explores structural graph patterns to generate adversarial examples that evaluate model boundaries. More recently, supervised contrastive learning approaches such as SCL-CVD [41] have combined R-Drop with GraphCodeBERT to produce more resilient code embeddings, allowing the model to adapt effectively to vulnerability detection tasks even with reduced labeled datasets.

3.3.2 Vulnerability Detection. Vulnerability detection concentrates a substantial portion of recent research on code models. In addition to ensemble frameworks such as EnStack [29], more recent works combine GraphCodeBERT with Code Property Graphs (CPG) and adaptive Graph Neural Networks (GNNs), seeking structural representations that capture semantic relationships beyond sequential code flow [18]. The EFVD framework extends this direction by integrating the sequential view of CodeBERT/GraphCodeBERT with edge-based attention mechanisms, allowing the model to simultaneously consider local syntactic dependencies and broader-scope control flows [36].

In parallel, research into unconventional architectures investigated the use of Quantum Convolutional Neural Networks (QCNN) associated with GraphCodeBERT representations for Java vulnerability analysis [12]. Another line of work focuses on the embedding process: hybrid loss functions have been employed to reduce the impact of the severe imbalance characteristic of vulnerability datasets, where vulnerable examples constitute a small minority of samples [45].

3.3.3 Similarity and Model Evolution. In clone analysis, Martinez-Gil [26] investigated the incorporation of dynamic execution signals and the production of interpretable explanations as ways to make models more transparent and auditable.

3.4 LLMs for Automated Program Repair (APR)

The adoption of LLMs has reshaped APR [43]. Empirical studies demonstrate that fine-tuning Large Language Models for Code (LLMCs) allows for effective multi-line bug repairs across various languages [10, 11, 33, 42], often using Parameter-Efficient Fine-Tuning (PEFT) to reduce computational costs [34]. To address the lack of precise semantic knowledge in LLMs, hybrid frameworks combine LLM-generated skeletons with formal program analysis and test-suite feedback to ensure valid corrections [8, 17].

Within the Android ecosystem, recent works evaluate generative models on security best-practice violations [5] and compatibility bugs in XML configurations [23]. However, the methods surveyed above use TF-IDF, AST, and GraphCodeBERT primarily as detectors or rankers. The closest prior work to ours is Tian et al. [35], who combine learned embeddings with engineered features to predict patch correctness. Their setting is patch ranking, whereas ours is evaluator diagnosis. We do not propose a new patch ranker or code embedding. Instead, we repurpose these three evaluators as parallel measurement instruments applied to the same set of LLM-generated patches. The contribution is methodological: we show that the choice of evaluator is a substantial confound in published APR effectiveness numbers, demonstrate the magnitude of this confound on a 300-patch test set, and argue that any single-evaluator success rate for an LLM should be reported alongside the evaluator that produced it.

4 Methodology

To evaluate the effectiveness of Large Language Models (LLMs) in repairing Android resource leaks and to examine the limitations of traditional evaluation methods, this work proposes a validation framework based on methodological triangulation. The experimental design is structured into three stages: (1) Dataset Systematization, (2) Construction of Multilayer Validation Methods, and (3) Execution and Results Evaluation.

4.1 Dataset Systematization and Data Augmentation

We extracted the initial database from *DroidLeaks* and isolated 97 real cases of resource leaks. Each sample consists of a pair of Java *snippets* containing the defective code (*Bug*) and the fix applied by developers (*Fix*), covering sensitive Android classes such as `Cursor`, `Camera`, and `MediaPlayer`.

To enable predictive methods, which require a larger volume of data, we adopted a *Data Augmentation pipeline* using the gpt-5.4-mini model, following the *Few-Shot Prompting* technique: We provided real pairs as references and instructed the model to produce 10 synthetic variations per case, diversifying application contexts to reduce the risk of *overfitting* while preserving the logic of the original fault.

We controlled the collection process with an automation script featuring three mechanisms:

- **Structural Consensus:** The `json_object` parameter in the API restricted outputs to a structured format, eliminating natural language text and ensuring that only code was processed.
- **Failure Recovery (*Auto-Retry*):** A triple-retry system with exponential backoff was implemented to handle network instabilities and *rate limits*.

- **Checkpoints (*Auto-Resume*):** Progress was saved after each completed case, preventing reprocessing and instance duplication in the event of an interruption.

The synthetic expansion stage resulted in 1,007 new scenarios. After structuring these scenarios into Bug/Fix pairs, a final corpus of 2,014 unique code samples (representing exactly 1,007 pairs) was obtained, ensuring a training base with high syntactic variability and no overlap with the original DroidLeaks data. The augmentation preserves the bug-class distribution of the 97 seed cases and introduces variation in application context, variable naming, and control-flow structure, without introducing novel defect categories. The resulting corpus therefore explores syntactic and contextual diversity within the DroidLeaks defect taxonomy rather than expanding it. We used this corpus differently across our validation streams, depending on each method's syntax tolerance constraints.

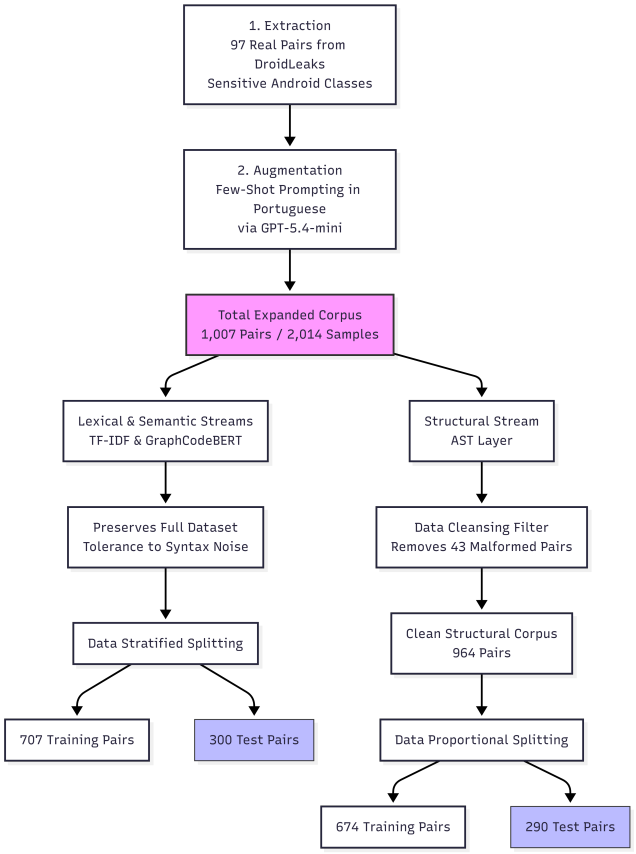

Figure 1: Data processing and validation pipeline schema.

For the Lexical (TF-IDF) and Semantic (GraphCodeBERT) layers (which process code as flat token sequences or continuous attention vectors) the full corpus of 1,007 pairs was preserved due to their natural resilience to syntax noise. This dataset was split into a training set of 707 pairs and a validation/test suite of 300 pairs.

Regarding prompt engineering, we selected Portuguese as the instruction language based on an exploratory design reflecting our primary development environment. While this setup allows us to evaluate the multi-lingual instruction-following robustness of state-of-the-art models when interacting with English codebases (like DroidLeaks), we acknowledge that the choice of language was not a controlled experimental variable. The potential prompt-language-dependent variations in code generation quality are thoroughly discussed as a threat to construct validity in Section 7.

For the Structural (AST) layer, generating a valid Abstract Syntax Tree requires code that complies structurally. Therefore, a data cleansing filter was applied to the entire corpus prior to splitting, identifying and removing 43 syntactically malformed pairs generated by the probabilistic nature of the models (such as unclosed braces). This filtering step reduced the total clean dataset available for the AST layer to exactly 964 pairs. Maintaining a proportional distribution, this clean structural corpus was split into a training set of 674 pairs and a validation/test suite of exactly 290 pairs.

The complete architectural progression of this data lifecycle—encompassing extraction, synthetic augmentation, syntactic filtering, and the stream-specific dataset splits—is visually schematized in Figure 1.

4.2 Multilayer Validation Methods

The proposed methodology evaluates the code across three complementary layers—lexical, structural, and semantic—aiming to circumvent a known limitation of classical validators: the tendency to penalize semantically correct solutions that diverge from the vocabulary or syntactic structure seen during training.

4.2.1 Lexical Method (TF-IDF). The first method adopts a classic NLP approach and serves as the study's *baseline*. The code is treated as an unstructured sequence of tokens (*Bag of Words*), and the TF-IDF algorithm tracks the presence of terms associated with resource release, such as `close` and `finally`. This approach is robust to syntactic errors—processing malformed code without interruption—but is restricted to the vocabulary seen during training, leading to the rejection of valid solutions that use alternative syntaxes.

4.2.2 Structural Method (AST-based Abstraction). The second method analyzes the code's structure rather than its vocabulary. The snippets are converted into Abstract Syntax Trees (AST), and variable identifiers are discarded, preserving only the function's logical skeleton. Different forms of resource release are normalized into a single structural token, producing a representation termed *Structural Bag-of-Nodes*.

Specifically, the feature engineering process extracts a highly specialized, low-dimensional feature vector (9 dimensions) that captures the exact syntactic signatures of resource management. The `javalang` parser isolates relevant control-flow nodes, specifically `TryStatement`, `CatchClause`, `HasFinallyBlock`, and implicit closures (`HasResources_ImplicitClose`). Furthermore, explicit method invocations are mapped to standardized tokens representing resource termination (e.g., `MethodCall_close`, `MethodCall_release`, `MethodCall_recycle`, `MethodCall_disconnect`, `MethodCall_free`, and `MethodCall_destroy`). A *Random Forest* model trained on this 9-dimensional vector evaluates the control flow independently of the syntax used, classifying the generated output based on structural patterns rather than certifying absolute semantic execution. This allows the framework to recognize an explicit *try-finally* block

and the *try-with-resources* construct as equivalent structural solutions (Class 0).

4.2.3 Semantic Method (GraphCodeBERT). The third method employs GraphCodeBERT to evaluate correctness through the code's data flow. The *Zero-Shot Feature Extraction* technique processes the generated code through the model's 12 attention layers and isolates the hidden state of the `[CLS]` token, producing a 768-dimensional *embedding* that represents the *patch*'s semantic behavior. This vector is fed as input to a *Random Forest* classifier, which predicts whether the patch aligns with verified human repair data flow behaviors. The primary distinction of this approach is its tolerance to syntactic noise: The model is capable of attesting to the semantic correctness of a response that would be rejected by the AST *parser* due to minor grammatical errors.

4.3 Inference Configuration and Execution Environment

To ensure the reproducibility of the experiments, the evaluated models (GPT-5.4, Claude Opus 4.6, and Gemini 3.1) were subjected to identical inference parameters. Queries were conducted via the official API of each provider with the *temperature* hyperparameter set to 0.1 and *top_p* set to 1.0, ensuring highly deterministic model behavior without completely eliminating syntactic variability in code generation. Crucially, all models were evaluated under standard single-turn text generation configurations. No advanced multi-step reasoning planning, internal chain-of-thought budgets, or reasoning effort parameters were enabled.

Regarding the context window, the models were fed isolated code snippets encompassing the specific method or localized block where the resource was allocated and used, rather than full class architectures or comprehensive framework lifecycle methods. This forced the models to strictly infer and manage the local scope lifecycle.

A *Zero-Shot Prompting* strategy with *persona* injection was adopted: The models were instructed to act as software engineers specialized in Java and received only the defective *snippet* as input. To enable extraction automation and prevent failures in subsequent *parsing* stages, the prompt imposed formatting constraints: the model was to return exclusively the corrected code, without explanations, greetings, or Markdown formatting. We orchestrated execution with an automation *script* featuring fault tolerance and state control mechanisms (*checkpoints*) to handle API *rate limits*.

We implemented the validation environment in Python 3.10. The lexical *pipeline* and *Random Forest* classifiers were developed using the `scikit-learn` library; structural processing used the `javalang` *parser* for AST extraction; and the semantic method employed Hugging Face's `transformers` library to obtain *embeddings* from the pre-trained GraphCodeBERT.

4.4 Repair Lifecycle and Prompt Template

To address the transition lifecycle from a defective snippet to a validated fix, we first define the exact prompt template used during the evaluation. To prevent failures in subsequent parsing stages, the prompt imposed strict formatting constraints, instructing the model to return exclusively the corrected code:

```
Você é um engenheiro de software especialista em Java.
Corrija o Resource Leak no código abaixo.
REGRA: Retorne APENAS o código corrigido.
Sem explicações, sem markdown (```java), sem saudações.
```

In a practical scenario, the input *Bug* (inserted into the placeholder above) often exhibits complex leak patterns, such as the reassignment leak shown in Listing 1. In this fitness tracking module snippet, the original `Cursor c` is overwritten with a new query result before the first resource is released. The reference to the first cursor is lost in memory, and the block lacks any terminal cleanup routine.

Listing 1: Defective Snippet (Bug): Cursor reassignment without closure in a fitness tracking module.

```
Cursor c = null;
try {
    c = fitnessDb.query("workouts", new String[]{"duration", "
          calories"}, "user_id=?", new String[]{String.valueOf(
          userId)}, null, null, "date_DESC");
    if (c.moveToFirst()) {
        lastDuration = c.getLong(0);
        lastCalories = c.getInt(1);
    }
    // DEFECT: The first cursor is reassigned and lost before
          being closed
    c = fitnessDb.query("goals", new String[]{"steps_target"}, "
          user_id=?", new String[]{String.valueOf(userId)}, null,
          null, null);
    if (c.moveToFirst()) {
        targetSteps = c.getInt(0);
    }
} catch (RuntimeException e) {
    analytics.trackError(e);
}
```

When the AST validator parses this defective structure, it detects the `TryStatement` and `CatchClause` nodes, but registers the complete absence of a `HasFinallyBlock` and `MethodCall_close`. Consequently, the structural feature vector is classified as an active leak (Class 1).

Upon processing the Portuguese prompt template, the LLM (in this specific instance, Claude Opus 4.6) generates the structurally transformed *Fix* presented in Listing 2. The model correctly identifies the reassignment flaw, splits the resource into two distinct variables (`c` and `c2`), and implements a comprehensive explicit closure strategy.

The AST validator then re-evaluates the output's structural signature. It now detects the newly added `HasFinallyBlock` along with multiple `MethodCall_close` nodes associated with the local identifiers. This effectively alters the 9-dimensional vector, allowing the Random Forest model to correctly classify the generated output as a verified structural repair pattern (Class 0).

It is important to note that because this specific repair generated by Claude uses legacy syntactic patterns (explicit `finally` and `.close()` tokens), it is successfully recognized by the traditional Lexical Layer (TF-IDF) as well. However, when contemporary models generate functionally equivalent repairs using modern paradigms like *try-with-resources* (which lack these explicit string tokens), the lexical validator fails to recognize them. This structural flexibility highlights the exact diagnostic mismatch investigated throughout this paper.

Listing 2: LLM Generated Fix (Claude Opus 4.6): Resolution of reassignment leak via variable splitting and legacy explicit closure.

```
Cursor c = null;
Cursor c2 = null;
try {
    c = fitnessDb.query("workouts", new String[]{"duration", "
          calories"}, "user_id=?", new String[]{String.valueOf(
          userId)}, null, null, "date DESC");
    if (c.moveToFirst()) {
        lastDuration = c.getLong(0);
        lastCalories = c.getInt(1);
    }
    c2 = fitnessDb.query("goals", new String[]{"steps_target"}, "
          user_id=?", new String[]{String.valueOf(userId)}, null,
          null, null);
    if (c2.moveToFirst()) {
        targetSteps = c2.getInt(0);
    }
} catch (RuntimeException e) {
    analytics.trackError(e);
} finally {
    if (c != null) {
        c.close();
    }
    if (c2 != null) {
        c2.close();
    }
}
```

4.5 Evaluation Metrics

The responses generated by the evaluated models (GPT-5.4, Claude Opus 4.6, and Gemini 3.1) for the 300 test instances were assessed under three complementary metrics, detailed in Table 1.

LFV is the central metric of this study, as the variation in the approval rate of the same *patch* when moving across Lexical, Structural, and Semantic methods quantitatively exposes the limitations of criteria in the current literature.

5 Results

To answer the proposed research questions, the experiment was conducted in two stages. First, the predictive capacity of the three validation methods was evaluated through *holdout* tests. Then, the multilayer *framework* was applied to the 300 test instances submitted to the models, allowing for an analysis of the evaluation method's impact on the perceived repair effectiveness.

5.1 Internal Validation of the Predictive Methods

Each method was evaluated on a rigorously isolated set of instances (*holdout set*) from the original *dataset* before being applied to the code generated by the models.

The lexical classifier (TF-IDF *baseline*) was initially evaluated on an internal *holdout* set. Following standard machine learning practices, 20% of the training corpus (1,414 distinct snippets) was reserved for this validation, resulting in exactly 283 test instances. On this internal suite, the method achieved an overall accuracy of 90.81%. As detailed in Figure 2, the model obtained an F1-Score of 0.91 for both classes. The Confusion Matrix (Figure 3) recorded 14 False Positives and 12 False Negatives, confirming the method's adequacy for tracking explicit resource release terms before being exposed to the 300 novel instances of the LLM tournament.

The structural method (AST) was similarly evaluated on its respective internal *holdout* set. Applying the same 20% split ratio to the clean structural training corpus (1,348 valid instances), an internal validation set of exactly 270 instances was established. On this set, the classifier achieved an accuracy of 88.89% and a macro

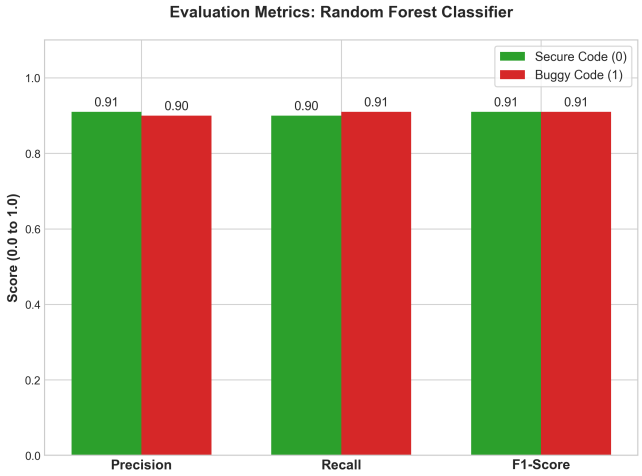

Figure 2: Baseline Classifier Evaluation Metrics (Precision, Recall, and F1-Score). Source: Prepared by the author (2026).

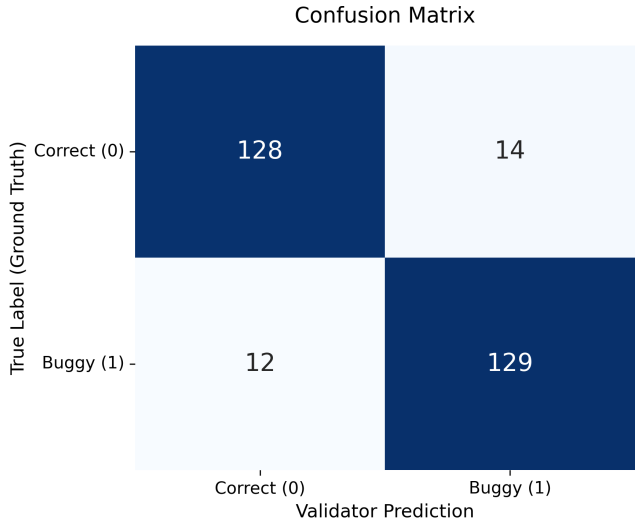

Figure 3: Confusion Matrix of the Random Forest predictive model (TF-IDF). Source: Prepared by the author (2026).

F1-Score of 0.89 (Figure 4). The internal Confusion Matrix (Figure 5) registered 18 False Positives and 12 False Negatives.

It is worth noting that this performance was obtained from only 9 fundamental logical paths extracted by the structural abstraction, in contrast to the extensive vocabulary used by the lexical *baseline*. This result highlights the effectiveness of dimensionality reduction through signature normalization.

The semantic validator (GraphCodeBERT) operated on a slightly different internal validation paradigm to prevent data leakage between code representations. Using a Group-Shuffle Split, 30% of

Table 1: Experimental Evaluation Metrics of the Framework.

Metric	Definition	Justification
Compilation Rate (CR)	Percentage of generated suggestions that compile without syntax errors.	Evaluates the LLM's capacity to produce functional code adherent to Android APIs.
Repair Accuracy (RA)	Algorithmic similarity index with the human reference *patch*.	Quantifies the degree of alignment between the generated solution and the patch produced by developers, allowing for the assessment of whether the model reproduces the repair strategies adopted in practice.
Leak-Free Verifiability (LFV)	*Patch* approval rate across the methods' triangulation.	Quantifies how often the same patch is classified as "fixed" by validators operating at different levels of abstraction; divergence across layers exposes the bias each validator introduces.

the pairs (Bug and corresponding Fix) were isolated for the *hold-out* set, ensuring that variants of the same snippet did not cross-contaminate the training phase. This partition resulted in an evaluation suite of exactly 426 instances. On this internal suite, the semantic model reached an accuracy of 82.39% and F1-Scores between 0.81 and 0.85 (Figure 6).

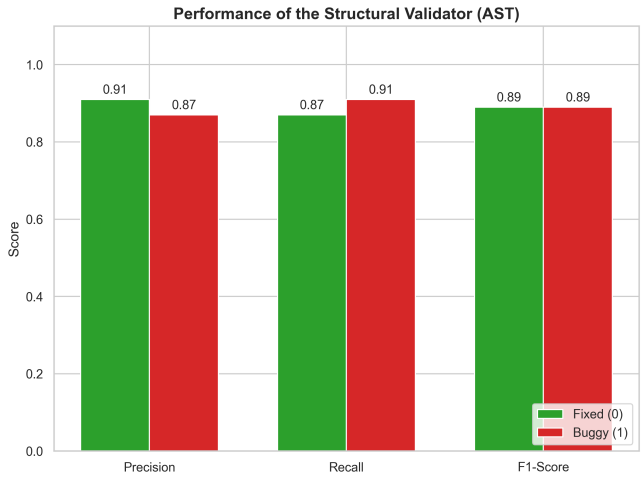

Figure 4: AST Structural Validator Evaluation Metrics (Precision, Recall, and F1-Score). Source: Prepared by the author (2026).

The corresponding Confusion Matrix (Figure 7) indicated 32 False Negatives and 43 False Positives. Although it presents a lower accuracy than the structural method during internal validation, its inclusion in the *framework* is justified by its tolerance to morphological deviations: the neural model is capable of correctly classifying valid resource release logic even in code that presents superficial syntactic errors.

5.2 LLM Evaluation and the Impact of Method Transition

Following internal validation and the retraining of the classifiers using the complete historical database, the *framework* was applied

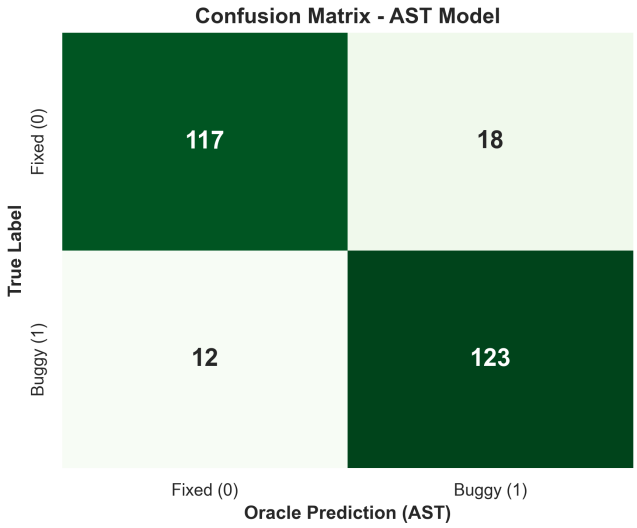

Figure 5: Confusion Matrix of the structural predictive model (AST). Source: Prepared by the author (2026).

to the solutions generated by the Claude Opus 4.6, GPT-5.4, and Gemini 3.1 models for the 300 test instances.

In the first layer, the lexical method (TF-IDF) positioned Claude Opus 4.6 with a 64.00% approval rate, GPT-5.4 with 50.00%, and Gemini 3.1 with 24.33% (Figure 8). Inspection of the rejected responses revealed the cause of Gemini's low score: while Claude and GPT adopted explicit corrections using `finally` and `.close()`, Gemini prioritized the `try-with-resources` construct—a modern and semantically equivalent solution, but whose vocabulary does not match the terms tracked by TF-IDF. This result empirically illustrates the lexical limitation discussed in Section 4: correct solutions with alternative syntax are systematically penalized by the *baseline*.

The second layer, based on the AST, produced a substantial redistribution of the results (Figure 9). Gemini 3.1's approval rate rose from 24.33% to 39.31%, confirming that its responses were structurally correct. Claude Opus 4.6 and GPT-5.4 also showed gains, reaching 70.69% and 50.34%, respectively.

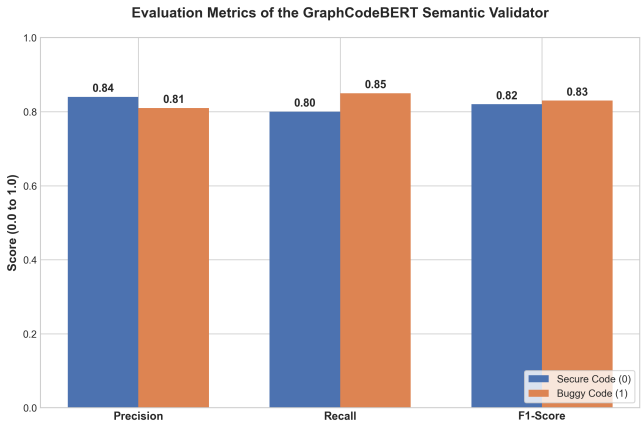

Figure 6: GraphCodeBERT Semantic Validator Evaluation Metrics (Precision, Recall, and F1-Score). Source: Prepared by the author (2026).

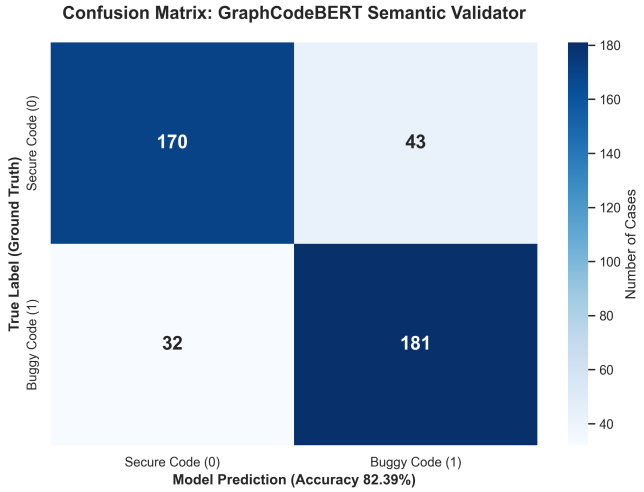

Figure 7: Confusion Matrix of the neural predictive model (GraphCodeBERT). Source: Prepared by the author (2026).

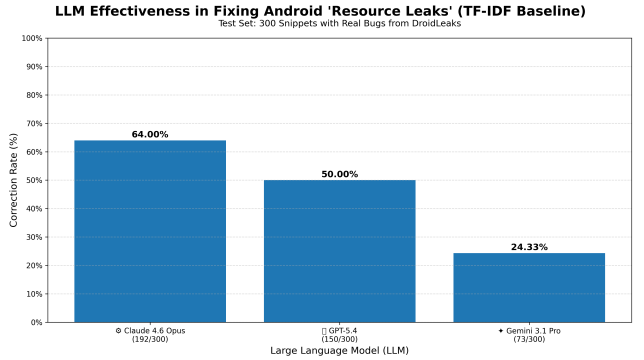

Figure 8: Initial Repair Tournament Scoreboard (Overall Effectiveness vs. TF-IDF Baseline Validator). Source: Prepared by the author (2026).

In the third layer, GraphCodeBERT corrected false negatives introduced by minor syntactic errors (such as unclosed braces) that the AST *parser* did not tolerate. The final rates were: Claude Opus 4.6 at 72.67%, GPT-5.4 at 60.00%, and Gemini 3.1 at 45.00% (Figure 10).

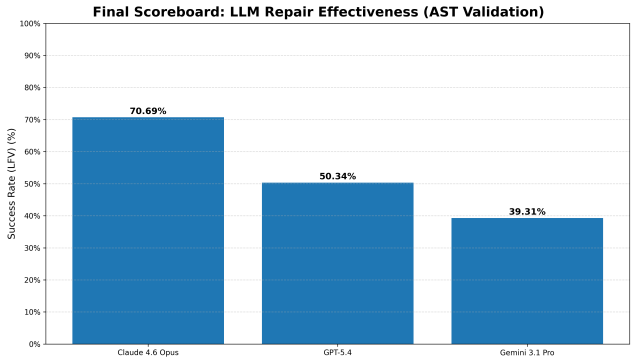

Figure 9: Intermediate Tournament Scoreboard (AST Structural Validator). Source: Prepared by the author (2026).

The progression across layers directly addresses the study's hypothesis: Lexical validators systematically reject patches whose vocabulary differs from the legacy patterns in the augmentation corpus, even when the patches are structurally and semantically equivalent. Structure- and semantics-aware validators reduce this rejection rate; whether they fully eliminate it cannot be established without compiler- or test-based ground truth.

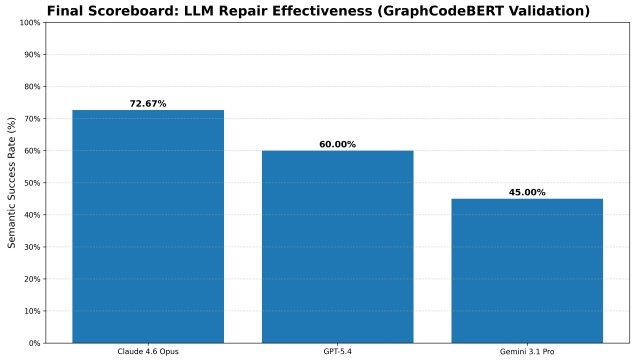

Figure 10: Final Scoreboard: LLM Effectiveness in Semantic Repair (GraphCodeBERT Validation). Source: Prepared by the author (2026).

5.3 Answers to the Research Questions

RQ1: What is the baseline acceptance rate of LLM models (GPT-5.4, Claude Opus 4.6, and Gemini 3.1) in repairing resource leaks in Android under traditional metrics?
Evaluated under the traditional lexical *baseline* (TF-IDF), which simulates the string-matching criteria prevalent in legacy literature, the models achieved baseline acceptance rates of 64.00% for Claude

Opus 4.6, 50.00% for GPT-5.4, and 24.33% for Gemini 3.1. These rates measure strict vocabulary alignment with explicit release patterns (such as the presence of `.close()` and `finally`). As is standard for automated metrics, these numbers reflect agreement with the trained classifier rather than a compiled guarantee of correctness, establishing the baseline upon which the impact of evaluation bias is subsequently measured.

RQ2: To what extent does the choice of evaluation method (Lexical, Structural, or Semantic) alter the reported metrics of repair effectiveness?
The choice of validator alters the reported metrics substantially, exposing a considerable evaluation bias. Gemini 3.1, for instance, received a 24.33% lexical acceptance rate, 39.31% under the AST structural validator, and 45.00% under GraphCodeBERT (a 20.67 point swing for the exact same set of patches). Direct examination of the rejected patches indicates that this cross-validator swing is a measurable artifact of the validators' varying tolerance to syntactic diversity, not random noise. This demonstrates that any single-metric effectiveness score reported for an LLM is heavily dependent on the chosen evaluation layer and should be reported alongside the validator that produced it.

RQ3: How does the adoption of modern syntactic patterns impact the false-negative bias in traditional, string-matching validators?
The adoption of modern paradigms generates a substantial increase in false negatives within traditional, string-matching validators. The experiment showed that semantically equivalent refactorings using the `try-with-resources` construct (which implicitly manages resource release) are systematically rejected by the TF-IDF *baseline*, as it rigidly requires the explicit presence of specific *tokens*. This establishes that traditional metrics introduce a systematic bias, penalizing syntactically diverse, modern solutions in favor of the fixed textual patterns present in legacy training data.

6 Discussion

The primary empirical finding of this study is that the central limitation of Automated Program Repair (APR) lies in the criteria used to evaluate LLMs, not in their capabilities. Reference databases, such as DroidLeaks, incorporate a temporal bias: As traditional validators were trained on patches produced in earlier periods, they reflect the predominant syntax of that era, such as the explicit use of `try-finally` blocks. When more recent models solve the same problem using modern constructs, such as `try-with-resources`, methods based on textual similarity or rigid syntactic structures fail to recognize the functional equivalence, generating systematic false negatives.

If continuous integration (CI/CD) pipelines base their decisions exclusively on textual validations, there is an indirect incentive for models to produce code that closely mimics historical patterns solely to satisfy acceptance criteria. In this scenario, the evaluation process would restrict the quality of the solutions rather than promote it.

The incorporation of a semantic method based on neural models, such as GraphCodeBERT, reduces this reliance on historical syntactic patterns. By prioritizing the intent of the correction and the behavior of the code, this approach yields an evaluation more aligned with the functional outcome. The experiments indicate that Claude Opus 4.6 and Gemini 3.1 frequently produce modern resource-management constructs whose acceptance is gated by the validator: A contemporary LLM can be made to look weak or strong on this benchmark depending on which evaluator one chooses, and a fair comparison must therefore report all three.

A potential limitation in the data augmentation strategy was identified: the risk of stylistic circularity. Since GPT-5.4-mini was used to generate the training set for the validators, the semantic classifier could have become oversensitized to the stylistic patterns typical of the GPT family, such as specific indentation styles and variable naming conventions. Theoretically, this might confer an unfair advantage to GPT-5.4 compared to Claude or Gemini. We observed consistent approval gains for all three models when transitioning from lexical to semantic evaluation, which is consistent with (but does not conclusively demonstrate) the validators generalizing beyond stylistic patterns of the augmentation model.

7 Threats to Validity

In this section, we discuss the potential threats to the validity of our study, categorized into external, construct, and internal validity, along with the mitigation strategies and open limitations.

7.1 External Validity

The primary threat to external validity concerns the generalizability of our findings to other modern programming languages. Currently, our evaluation pipeline is strictly Java-specific, relying on the `javalang` parser and classifiers trained exclusively on Java resource leaks. Adapting this framework to languages such as Kotlin represents an engineering and data bottleneck due to the absence of a curated, human-validated leak corpus equivalent to DroidLeaks. Furthermore, we hypothesize that the false-negative bias documented in RQ3 could intensify in Kotlin when using scope functions like `.use()`, which handles resource release implicitly without explicit `.close()` tokens. We frame the construction of a multi-lingual benchmark as immediate future work.

7.2 Construct Validity

Construct validity threats relate to whether our experimental setup accurately measures evaluator bias. First, the repair prompts submitted to the LLMs were written in Portuguese. Because literature indicates that prompt language can introduce generation quality variations, we cannot definitively rule out that our reported acceptance rates are partially influenced by the language barrier rather than strictly by metric discrepancies. Second, our framework evaluates patch correctness through statistical classifier agreement rather than dynamic verification (e.g., compilation rates or execution-based test suites). Thus, our metrics represent a baseline acceptance rate under specific abstraction layers rather than a certified guarantee of absolute resource leak elimination. Furthermore, our experimental design is restricted to a single-turn, zero-shot generation setting. We did not investigate whether providing iterative feedback—such as compilation errors or structural rejections from the AST parser—back to the LLM would allow the models to correct

their initial mistakes, which restricts our findings to the models' zero-shot capabilities.

7.3 Internal Validity

Internal validity concerns internal factors and risks of bias within our data pipeline. One prominent threat is data contamination, as we did not apply strict deduplication filters against the models' pre-training corpora. However, because our test instances consist of newly generated patches rather than exact replicas of DroidLeaks historical fixes, this risk is partially minimized. Additionally, a circularity or same-family bias might exist, particularly favoring GPT-5.4, since its smaller family variant (`gpt-5.4-mini`) was responsible for generating the synthetic data expansion. We present these factors as open limitations of our current diagnostic design.

8 Conclusion and Future Work

This study demonstrated that evaluation criteria built on legacy syntactic patterns can systematically underestimate modern, semantically equivalent repairs. The case of Gemini 3.1 is illustrative: Its `try-with-resources` patches were rejected by the lexical validator even when they were accepted by both structure- and semantics-aware validators, with no change to the patch itself.

The implementation of the multilayer validation *framework* proved to be an effective technical alternative to this bias. The structural layer (AST) normalized different programming paradigms and evaluated the control flow independently of the vocabulary used. The semantic layer (GraphCodeBERT) allowed the correction to be classified by the code's intent, tolerating minor syntactic errors that would compromise static analysis. Under the semantic validator, Claude Opus 4.6 received the highest patch-acceptance rate (72.67%), followed by GPT-5.4 (60.00%) and Gemini 3.1 Pro (45.00%). These are validator-relative measurements, not verified repair rates.

Despite the success in isolating and circumventing the methodological bias, this research opens new avenues of investigation for consolidating this *framework*. We envision future work directed towards six areas. First, **modernization of Reference Benchmarks:** This study highlights a temporal gap in current reference databases. *DroidLeaks* consists of human-made fixes that reflect legacy Java practices, which may be considered suboptimal or overly verbose by modern standards. Future efforts should focus on Ground Truth Augmentation, creating hybrid benchmarks that combine historical bug reports with modern, linter-verified solutions. This would prevent supervised classifiers from learning a bias that equates correctness with legacy syntactic patterns. Second, **expansion of Languages and Vulnerabilities:** The current model was limited to the scope of resource leaks in Android devices. It is necessary to test the structural and semantic architecture in languages critical for low-level memory management (such as C, C++, and Rust) and broaden the spectrum of vulnerabilities to include concurrency flaws and dependency injection. Third, **evaluation of Open-Source Models:** This experiment evaluated exclusively commercial models. The next step is to apply the *framework* to open-weights models tailored for code, such as the Llama, Mistral, and StarCoder families, analyzing the cost-benefit of autonomous repairs in local environments. Fourth, **integration into Development Environments:** From an applied software engineering perspective, the AST and *Deep Learning*-based methods developed in this study can be integrated as IDE *plugins* or incorporated into CI/CD *pipelines*. This would allow for the real-time interception and predictive validation of snippets generated by assistants like GitHub Copilot, actively blocking the introduction of resource leaks before the developer's final *commit*. Fifth, **human-in-the-loop Correlation:** While this study bridges the gap between automated metrics and semantic intent, it is vital to assess whether human developers perceive modern `try-with-resources` patches as more maintainable than legacy `try-finally` blocks. A user study comparing automated scores with developer ratings would help calibrate the weights of the structural and semantic layers. Finally, **autonomous Self-Healing Pipelines**: The current study evaluates repairs in a static, single-turn diagnostic setting. Future work will investigate closing the loop by feeding the rejection signals from the AST and GraphCodeBERT validators back to the LLMs as iterative prompts. This transition from a diagnostic metric to an active, self-healing framework for Android resource leaks will explore whether LLMs can autonomously resolve structural anomalies when provided with targeted validator feedback.

Artifact Availability

All artifacts related to this study, including datasets, scripts, and additional resources, are publicly accessible online.[1] For more information or further assistance regarding the artifacts, please contact the first author directly.

Acknowledgements

We would like to state that generative Artificial Intelligence tools were used to support the development of this work. Specifically, the Gemini model was used to assist in the generation and structuring of the Table 1. Additionally, ChatGPT was used to generate the code responsible for creating Figures. The terms of use for both tools were verified and respected in the preparation of this material. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence),[2] CNPq grant 408817/2024-0. Leopoldo is partially supported by CNPq grant 310405/2025-4. We thank the anonymous SBES 2026 reviewers for their constructive feedback.

References

[1] Abanoub E Abdelmalak, Mohamed A Elsayed, Ilhami Torunoglu, and David Abercrombie. 2025. An AST-guided LLM Approach for SVRF Code Synthesis. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 68–76.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[3] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–63.

[4] Clement Blaudeau and Natarajan Shankar. 2020. A verified packrat parser interpreter for parsing expression grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 3–17.

[5] Elisa Braconaro and Eleonora Losiouk. 2024. A dataset for evaluating LLMs vulnerability repair performance in Android applications: Data/toolset paper. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy*. 353–358.

[1] Available at: https://github.com/amb8-cin/Diagnosing-Evaluator-Bias-in-Automated-Program-Repair

[2] https://www.ines.org.br

[6] Tianyu Chen, Lin Li, Bingjie Shan, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. 2025. Identifying Affected Third-Party Java Libraries from Textual Descriptions of Vulnerabilities and Libraries. *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–27.

[7] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1656–1668.

[8] Anmin Fu, Pengyu Xu, Jichunyang Li, Boyu Kuang, and Yansong Gao. 2025. InstructRepair: Instruct Large Language Models With Rich Bug Information for Automated Program Repair. *IEEE Transactions on Information Forensics and Security* (2025).

[9] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 300–307.

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

[11] Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu. 2025. Comprehensive fine-tuning large language models of code for automated program repair. *IEEE Transactions on Software Engineering* 51, 4 (2025), 904–928.

[12] Shumaila Hussain, Muhammad Nadeem, Junaid Baber, Mohammed Hamdi, Adel Rajab, Mana Saleh Al Reshan, and Asadullah Shaikh. 2024. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. *Scientific Reports* 14, 1 (2024), 7406.

[13] Büşra İçöz and Göksel Biricik. 2025. Automated Code Review Using Large Language Models with Symbolic Reasoning. In *2025 9th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*. IEEE, 1–5.

[14] Shitanshu Jain, Sapan Kumar Jain, and Somil Vasal. 2024. An effective TF-IDF model to improve the text classification performance. In *2024 IEEE 13th International Conference on Communication Systems and Network Technologies (CSNT)*. IEEE, 1–4.

[15] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*. PMLR, 54–63.

[16] Fei Lan. 2022. Research on Text Similarity Measurement Hybrid Algorithm with Term Semantic Information and TF-IDF Method. *Advances in Multimedia* 2022, 1 (2022), 7923262.

[17] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–28.

[18] Chen Liang, Qiang Wei, Zirui Jiang, Yisen Wang, and Jiang Du. 2024. A source code vulnerability detection method based on adaptive graph neural networks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 187–196.

[19] Minhui Liang and Tiansen Niu. 2022. Research on text classification techniques based on improved TF-IDF algorithm and LSTM inputs. *Procedia Computer Science* 208 (2022), 460–470.

[20] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 184–195.

[21] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487.

[22] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. 2019. DroidLeaks: a comprehensive database of resource leaks in Android apps. *Empirical Software Engineering* 24, 6 (2019), 3435–3483.

[23] Zhijie Liu, Yutian Tang, Meiyun Li, Xin Jin, Yunfei Long, Liang Feng Zhang, and Xiapu Luo. 2026. Llm-compdroid: Repairing configuration compatibility bugs in android apps with pre-trained large language models. *ACM Transactions on Software Engineering and Methodology* 35, 3 (2026), 1–32.

[24] Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhang Wang, Qiang Hu, Jie Zhang, and Yang Liu. 2024. Unveiling Code Pre-Trained Models: Investigating Syntax and Semantics Capacities. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 169 (Aug. 2024), 29 pages. doi:10.1145/3664606

[25] Christopher D Manning. 2008. *Introduction to information retrieval*. Syngress Publishing,.

[26] Jorge Martinez-Gil. 2025. Augmenting the interpretability of graphcodebert for code similarity tasks. *International Journal of Software Engineering and Knowledge Engineering* 35, 05 (2025), 657–678.

[27] Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmatif. 2023. Explaining transformer-based code models: What do they learn? when they do not work?. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 96–106.

[28] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1139–1149.

[29] Shahriyar Zaman Ridoy, Md Shazzad Hossain Shaon, Alfredo Cuzzocrea, and Mst Shapna Akter. 2024. Enstack: An ensemble stacking framework of large language models for enhanced vulnerability detection in source code. In *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 6356–6364.

[30] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[31] Janaka Senanayake, Harsha Kalutarage, Mhd Omar Al-Kadri, Andrei Petrovski, and Luca Piras. 2023. Android source code vulnerability detection: a systematic literature review. *Comput. Surveys* 55, 9 (2023), 1–37.

[32] Nakul Sharma, Siddharth Shinde, Swarup Bhosale, and Suyog Patil. 2024. SourcePlag-Source Code Plagiarism Detection Based on Abstract Syntax Trees. In *2024 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS)*. IEEE, 1–5.

[33] Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Pavel Zadorozhny, Anton Cheshkov, Rustam Mussabayev, Alymzhan Toleu, Gulmira Tolegen, and Alexander Krassovitskiy. 2025. Finetuning large language models for vulnerability detection. *IEEE Access* (2025).

[34] André Silva, Sen Fang, and Martin Monperrus. 2025. Repairllama: Efficient representations and fine-tuned adapters for program repair. *IEEE Transactions on Software Engineering* (2025).

[35] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2023. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–34.

[36] Lei Tian and Cheng Zhang. 2025. EFVD: A framework of source code vulnerability detection via fusion of enhanced graph representation learning and pre-trained transformer-based model. In *Proceedings of the 2025 5th International Conference on Computer Network Security and Software Engineering*. 316–320.

[37] Hieu Dinh Vo and Son Nguyen. 2023. Can an old fashioned feature extraction and a light-weight model improve vulnerability type identification performance? *Information and Software Technology* 164 (2023), 107304.

[38] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th international conference on software engineering*. 2377–2388.

[39] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th international conference on software engineering*. 287–298.

[40] Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022. Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification. In *30th IEEE/ACM International Conference on Program Comprehension*.

[41] Rongcun Wang, Senlei Xu, Yuan Tian, Xingyu Ji, Xiaobing Sun, and Shujuang Jiang. 2024. SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT. *Computers & Security* 145 (2024), 103994.

[42] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[43] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2026. A Systematic Literature Review on Large Language Models for Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* (March 2026). doi:10.1145/3799693

[44] Hai Zhou. 2022. Research of Text Classification Based on TF-IDF and CNN-LSTM. In *Journal of Physics: Conference Series*, Vol. 2171. IOP Publishing, 012021.

[45] Zhengbin Zou, Tao Jiang, Yizheng Wang, Tiancheng Xue, Nan Zhang, and Jie Luan. 2025. Code vulnerability detection based on augmented program dependency graph and optimized CodeBERT. *Scientific Reports* 15, 1 (2025), 39301.