LLM-Based Identification of Null Pointer Exception Patches

Tahir Ullah *, Waseem Akram *†, Fiza Khaliq * and Hui Liu *‡
*School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China
†Department of Computer Science, COMSATS University Islamabad, Vehari Campus, 61100, Pakistan
tahirullah7710@bit.edu.cn, waseemakramcui@gmail.com, fixakhaliq@gmail.com, liuhui08@bit.edu.cn

***Abstract*—Null Pointer Exceptions (NPEs) are one of the leading causes of software crashes and runtime errors. Although existing methods attempt to detect and classify NPE fixes, they often fall short due to irrelevant or noisy data, a lack of contextual understanding, and inefficiency in processing large and imbalanced datasets. To overcome these challenges, we propose an approach, called *Augmented Agentic Commit Classification* (*AACC* for short), to accurately categorize commit patches as NPE fixes or non-NPE. *AACC* leverages the code structure and contextual insights from commit messages to capture the semantic intent behind code modifications. It features four key advancements: (1) Best example selection that filters high-quality, contextually relevant commits to ensure the model learns from contextual rich and accurate data; (2) an augmented knowledge base that enriches classification by combining contextual metadata, program semantics, and bug fix patterns; (3) a prioritise agent that ranks commits based on relevance and impact, optimizing resource allocation and boosting efficiency; and (4) an iterative refinement process that enables the model to learn from feedback to correct misclassifications, reducing false negative rates. Our evaluation results on ChatGPT-4o suggest that it outperforms the state-of-the-art approaches by improving the F1 score from 72.07% to 98.03%.**

***Index Terms*—Null pointer exception, Patch classification**

I. Introduction

In software engineering, a Null Pointer Exception (NPE) arises when a program attempts to use a reference that points to no object, often leading to abrupt application crashes, data loss, and potential security vulnerabilities [1], [2]. The identification and resolution of NPEs are therefore of paramount importance for ensuring software stability and a positive user experience. Particularly in languages like Java, the prevalent use of references makes NPEs a common and disruptive runtime error [3]. As software systems inevitably grow in scale and complexity, the challenge of maintaining reliability escalates, rendering efficient NPE management crucial [4].

Traditional static analysis techniques, designed for compile-time NPE detection, aim to catch these errors early [5], [6], [7]. Various static bug detection tools have been developed by both industry and academia to assist in this process [3], [8], [9], [10]. However, these methods often suffer from high rates of false positives and can overlook subtle runtime errors, which limits their practical utility and adoption by developers who find the sheer volume of warnings, many of which may be inaccurate or unclear, overwhelming [11], [12], [5], [13]. This challenge underscores the critical need to improve future NPE prevention strategies and develop more accurate detection tools; first, we must have a deep and reliable understanding of how NPEs are effectively resolved in practice.

This paper argues that a foundational step towards this broader goal is the ability to accurately identify and categorize past code modifications, specifically commit patches that genuinely fix NPEs. Such precisely classified NPE fixes create an invaluable knowledge base. This curated knowledge can then fuel the development of next-generation NPE prediction models trained on high-quality, verified data [14], potentially leading to more accurate pre-commit warnings with fewer false positives. Furthermore, understanding the patterns and contexts of real-world NPE fixes can inform developer training and lead to more robust coding practices. Thus, the task of accurately classifying historical bug-fixing commits for NPEs, while an intermediate step, is vital for advancing the overarching objective of enhancing proactive NPE detection and, ultimately, overall software reliability.

To achieve this nuanced classification, researchers are increasingly leveraging the power of machine learning (ML), deep learning (DL), and, particularly, Large Language Models (LLMs) [15], [16]. These advanced AI techniques excel at analyzing complex patterns and understanding the semantic context within both natural language (such as commit messages) and source code [17], [18], [19], [20]. This capability is essential for discerning the true intent and impact of code modifications, which makes LLMs well-suited to accurately identify NPE fixes from commit data, an area where earlier NLP-based approaches have also shown promise [21], [22], [23].

To address the specific challenge of accurately classifying NPE-fixing commits, we introduce the Augmented Agentic Commit Classification (*AACC*) approach. *AACC* is an LLM-based approach designed to meticulously analyze the structure of the code and the contextual details embedded in the commit messages. By doing so, it aims to overcome common hurdles such as noisy or irrelevant data and a lack of deep contextual understanding, which can impede the precise identification of genuine NPE fixes. The *AACC* methodology is established on four key innovative components: 1) Best Example Selection (BES): A process to identify and filter high-quality,

‡Corresponding author: Hui Liu (liuhui08@bit.edu.cn).

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00226

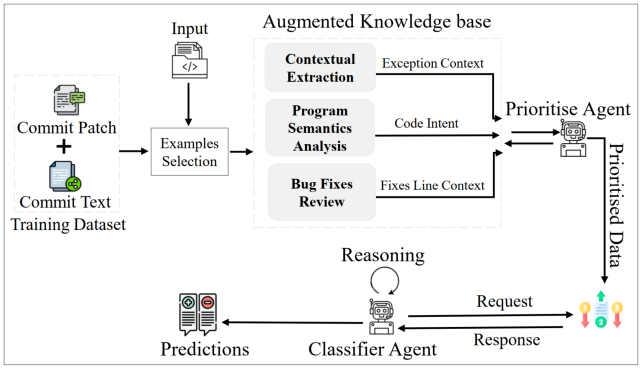

Fig. 1: Overview of the *AACC*

contextually rich commits, ensuring that the model learns from the most informative and accurate data. 2) Augmented Knowledge Base (AKB): An enriched repository that combines contextual metadata, program semantics, and established bug-fix patterns to inform and guide the classification process. 3) Prioritise Agent (PA): A mechanism that ranks commits based on their relevance and potential impact, optimizing the allocation of analytical resources. 4) Iterative Refinement: A feedback-driven process that allows the model to learn from previous classifications and continuously improve its accuracy, particularly in reducing false negatives. By integrating these elements, *AACC* provides a more precise and effective method to categorize NPE fixes, laying a stronger foundation for future NPE prevention efforts.

The paper makes the following contributions:

- A novel LLM-based approach enhances the identification of NPE patches. The key of the approach is a dual agent framework that curates data, prioritises commit analysis (semantic/structural), and uses adaptive refinement to provide a robust foundation for NPE prevention.
- A public GitHub replication package [24] enabling validation and reproducibility of our findings.

II. Approach

A. Overview

Figure 1 overviews the *AACC* process, which initially builds a knowledge repository from a training dataset (commit patches, messages, stack-traces). For new input commits, the *Augmented Knowledge Base* uses selected contextually similar examples from this repository to enrich the input's context. A *Prioritise Agent* then ranks and filters this enriched information for the *Classifier Agent*, which employs an internal iterative refinement mechanism (detailed in Section II-E) for its final classification. A feedback loop between the Prioritise and Classifier Agents is designed for continuous system improvement. Ultimately, *AACC* accurately classifies commit patches as NPE fixes or non-NPE, significantly contributing to a deeper understanding of how such issues are resolved and thereby supporting the broader goal of enhanced software reliability.

B. Best Example Selection

The *Best Example Selection (BES)* component is crucial to our *AACC* approach, ensuring that the system learns from a high-quality, diverse, and contextually rich set of commit examples. This selection process meticulously filters and ranks commits to identify the most informative instances for subsequent analysis and knowledge base enrichment.

The BES process is integral to *AACC*, curating a high-quality set of contextually rich and diverse commit examples. This process begins after an initial preprocessing phase of the commit dataset (which includes steps such as tokenizing commit messages and code changes, parsing patches for structural information, and text normalization). To foster diversity, potentially relevant commits are first grouped into clusters based on contextual similarities derived from their content and metadata. Subsequently, BES ranks commits, typically within these clusters or across a candidate pool, using a composite relevance score. This score is systematically derived from three primary criteria: *Keyword Frequency:* The prominence of relevant null pointer exception-related keywords (e.g., "NullPointerException", "null check", etc.) within the commit message. *Patch Complexity:* An assessment of the code patch's structural characteristics, such as the number and type of modified code elements, indicating the fix's nature. *Semantic Alignment:* The degree of similarity between the developer's stated intent in the commit message and the actual code modifications in the patch. This semantic alignment between a commit message (represented by vector $\vec{A}$) and its patch (vector $\vec{B}$) is quantified using Cosine Similarity on their respective embeddings:

$$\text{Cosine Similarity} = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \cdot ||\vec{B}||} \quad (1)$$

Based on the composite relevance scores, the top 500 examples are selected. This number was empirically determined to provide a robust balance between dataset diversity, information richness for effective model training, and manageable computational overhead for detailed analysis. Selecting examples with high cosine similarity scores (Equation 1) is crucial to ensuring strong *contextual alignment*, defined here as a clear and direct correspondence between the intention articulated in the commit messages and the tangible changes implemented in the code patches.

For instance, Listing 1 demonstrates strong alignment: its message, `"Fixes NullPointerException in OptionsRenderer"`, coupled with a JIRA-issue link, directly mirrors the patch's introduction of a null check (`return found == null || found`). Listing 2 also shows good alignment, where the detailed message about an NPE fixes in `entrySet` methods corresponds well with the patch's addition of a null check (`entry != null && entry.equals(e)`), even if minute details about the fix's exact class location could be more explicit. Conversely, Listing 3 exhibits weaker alignment; its generic message, `"Fix NullPointerException bug"`, lacks specific contextual details linking it to the null check added in its patch. This

```
Fixes NullPointerException in OptionsRenderer
https://issues.apache.org/jira/browse/TILES-598
This closes #14, Patch by Christian Schuster

public final class OptionsRenderer implements Renderer {
    static boolean attemptTemplate(final String template) {
-   return TEMPLATE_EXISTS.containsKey(template) && TEMPLATE_EXISTS.get(template);
+   Boolean found = TEMPLATE_EXISTS.get(template);
+   return found != null && found;
    }
}
```

Listing 1: NPE Fix in *OptionsRenderer:* Null Check

```
Fix a NullPointerException when an object passed to an entrySet method is not found in the map. Exhibited with entrySet().remove(Object) and entrySet().contains(Object)
public class SequencedHashMap extends HashMap {
    ...
    Map.Entry e = (Map.Entry) o;
    Entry entry = (Entry) entries.get(e.getKey());
-   if(entry.equals(e)) return entry;
+   if(entry != null && entry.equals(e))
    return entry;
    else return null;
    ...
}
```

Listing 2: Null check in *SequencedHashMap* to fix NPE

meticulous BES process ensures that the selected samples are not only relevant but also contextually unambiguous, forming a reliable foundation for *AACC*.

C. Augmented Knowledge Base

The Augmented Knowledge Base (AKB), a central component of the *AACC*, functions as a rich embedding vector database. It captures crucial semantic and structural insights from commits by integrating their messages, code patches, and metadata. This provides a comprehensive understanding of code changes and is built upon three primary analytical elements detailed below: Contextual Extractions (CE), Program Semantic Analysis (PSA), and Bug Fixes Review (BFR).

1) Contextual Extractions (CE): This component, CE, extracts explicit signals and metadata from commit messages and the textual content of code patches to provide immediate, high-level context regarding a commit's potential relevance to NPE fixes. The key CE processes include: scanning messages for NPE-related keywords (e.g., "NPE", "NullPointerException", etc.) and issue tracker references (e.g., JIRA IDs); and examining patch lines for explicit null checks or related API usage (e.g., `Objects.requireNonNull`). A guiding heuristic for CE in identifying null-safety practices is:

$$\text{CE:} \quad \text{If variable } v \text{ is accessed, verify } v \neq \text{null} \qquad (2)$$

For instance, CE flags commit messages like `Fixes NullPointerException in OptionsRenderer` and identifies the corresponding null checks in the patch (e.g., `found == null || found` from Listing 1). Thus, CE pinpoints explicit null-safety considerations evident in a commit's textual and surface-level code elements.

2) Program Semantic Analysis (PSA): The PSA component analyzes the underlying logic and intent of code modifications, particularly those addressing NPE bugs. PSA examines how changes like null checks or error handling prevent NPEs by ensuring variables are validated before use. This often aligns with the generalized pattern:

$$\text{PSA:} \quad \text{If } v \neq \text{null, then perform operation } O(v) \qquad (3)$$

Here, v represents a variable (e.g., `template`, `entry`) and $O(v)$ denotes an operation on v (e.g., a method call or comparison). For instance, in the NPE fixes shown in Listing 1 (affecting `OptionsRenderer`), PSA identifies that modifying the `attemptTemplate` method to incorporate the logic `Boolean found = TEMPLATE_EXISTS.get(template); return found == null || found;` semantically prevents an NPE. This is achieved by ensuring that the variable `found` is validated before determining the outcome of the method, adhering to the pattern of Equation 3. Similarly, in Listing 2, PSA discerns that adding the check `if(entry != null && entry.equals(e))` resolves an NPE by ensuring the `entry` object's state is validated prior to invoking its `.equals()` method, thus preventing a potential crash.

PSA thus contributes features to the AKB that reflect a deep semantic understanding of how specific code changes logically avert NPEs.

3) Bug Fixes Review (BFR): The BFR component identifies common structural code changes indicative of NPE fixes. BFR scrutinizes concrete modifications such as added null checks, control flow alterations for null safety, or improved variable initializations, typically in critical code sections. A generalized structural pattern BFR identifies:

$$\text{BFR:} \quad \text{If } v \neq \text{null, then modify operation } M(v) \qquad (4)$$

Here, M(v) denotes a structural modification to handle potential nulls for variable v. For example, in Listing 1, BFR notes the added null-safe check (`found == null || found`). Similarly, in Listing 2, BFR identifies the new condition `entry != null` as a key structural change preventing an NPE. By pinpointing these explicit structural modifications, BFR highlights common defensive programming strategies and contributes insights into technical implementation patterns for null safety.

The AKB is formed by synergistically integrating insights from Contextual Extractions (CE), Program Semantic Analysis (PSA), and BFR. CE provides high-level contextual cues;

PSA offers logical intent understanding; and BFR identifies structural fix patterns. Together, they enable the AKB to construct rich, vectorized commit representations, fundamental to AACC's accurate NPE fixes classification, thereby enhancing software reliability. The Prioritise Agent's (PA) use of this AKB is detailed next.

D. Prioritise Agent

The Prioritise Agent (PA) is a key component within the AACC framework that categorizes incoming commits into low, moderate, or high-priority levels. This ranking, derived from metrics supplied by the AKB, ensures that the most relevant and impactful commits are prioritised for detailed analysis by the Classifier Agent (CA), thereby optimizing resource allocation and contributing to improved model performance.

The priority score (P_i) for a given commit i is calculated as a weighted sum of three distinct factors: its Contextual Score (CS_i), Semantic Impact Score (SIS_i), and Fix Relevance Score (FRS_i). This prioritisation mechanism is expressed as:

$$P_i = w_1 \cdot \text{CS}_i + w_2 \cdot \text{SIS}_i + w_3 \cdot \text{FRS}_i \tag{5}$$

The weights (w_1, w_2, w_3) balance each metric's relative importance and are empirically calibrated using a validation dataset to effectively differentiate high-impact commits. The components of the priority score P_i are defined as follows:

1) Contextual Score (CS_i): This score evaluates the explicit textual relevance of a commit to NPE fixes. It assigns higher values for specific keywords (e.g., "NPE", "NullPointerException", etc.), as illustrated in Listing 1 and 2. CS_i is typically a normalized score, based on keyword frequency or binary presence, prioritising commits that signal the intention to address null-related bugs.

2) Semantic Impact Score (SIS_i): This score assesses the structural and logical significance of code changes, focusing on modifications to critical program paths or the introduction of robust null-safety patterns. SIS_i is formulated as:

$$\text{SIS}_i = \alpha \cdot N_c + \beta \cdot C_p \tag{6}$$

where N_c is the count of newly added null checks and C_p reflects the estimated impact of changes on critical code paths. The parameters α and β are empirically adjusted to ensure a balanced contribution from these factors. For example, adding a null check (e.g., `found == null || found` in Listing 1) increases N_c, while alterations to the logic of the core method or critical conditional statements (e.g., `if(entry != null && entry.equals(e))` in Listing 2) influence C_p.

3) Fix Relevance Score (FRS_i): This score quantifies how directly the commit patch addresses potential NPEs, primarily through explicit null checks or other defensive programming enhancements. FRS_i is calculated as the proportion of code modifications dedicated to improving null safety:

$$\text{FRS}_i = \frac{\text{Number of Null-Safety Enhancements}}{\text{Total Number of Modified Lines}} \tag{7}$$

"Null-Safety Enhancements" include lines specifically added or altered to mitigate null-related bugs (e.g., introducing `if (variable == null)` checks, using null-safe functions). "Total Number of Modified Lines" covers all added, deleted, or changed lines in the patch. For example, in Listing 3, adding the null check `if (field != null && field.getWrappedField() == wrappedField)` constitutes a key null-safety enhancement, thereby positively influencing its FRS_i.

```
Fix NullPointerException bug
public DateTimeField getField(Chronology
    chrono) {
    DateTimeField wrappedField = iWrappedType.
        getField(chrono);
    RemainderDateTimeField field = iRecent;
-   if (field.getWrappedField() ==
    wrappedField) {
+   if (field != null && field.getWrappedField
    () == wrappedField) {
    return field;
    }
    field = new RemainderDateTimeField(
        wrappedField, iType, iDivisor);
}
```

Listing 3: NPE Fix in *DateTimeField:* Null Check

By synthesizing these metrics into the composite priority score P_i (Equation 5), the PA effectively identifies commits that exhibit strong characteristics of NPE fixes and possess significant structural or semantic impact. This targeted approach ensures that analytical resources are concentrated on the most promising candidates, thereby enhancing the efficiency and precision of subsequent classification tasks, particularly for critical defects like NPEs.

To assign commits to actionable priority categories (PC), the calculated P_i scores are normalized to a consistent range (e.g., $[0, 1]$). Threshold values, T_{low} and T_{moderate}, are then employed to delineate these categories:

$$\text{PC} = \begin{cases} \text{Low}, & P_i \leq T_{\text{low}} \\ \text{Moderate}, & T_{\text{low}} < P_i \leq T_{\text{moderate}} \\ \text{High}, & P_i > T_{\text{moderate}} \end{cases} \tag{8}$$

These thresholds are established and may be periodically refined by analyzing the statistical distribution of P_i scores across the dataset (e.g., aligning with specific percentiles of observed scores). This ensures that the priority categories remain meaningful and adaptive as new commit data is processed and patterns evolve.

This systematic prioritisation by the PA facilitates effective resource allocation within *AACC*, guiding the system to focus on commits demonstrating the highest relevance, quality, and potential impact. Consequently, high-priority commits ($P_i > T_{\text{moderate}}$), which represent the most probable and significant NPE fixes, are expedited to the Classifier Agent for comprehensive analysis. This structured process is designed to support robust model refinement and ensure that critical NPE fixes, as exemplified by Listings 1 and 2, are addressed with appropriate urgency, ultimately contributing to the enhanced reliability and maintainability of software systems.

TABLE I: Statistics of Top 5 Repositories of the Dataset

Project Name	LOC	Tokens	Commits
Appformer	135,011	68,378	3,727
commons-imaging	57,108	35,241	1,854
hbase	19,675	23,705	843
drools	12,654	13,275	264
wicket	3,991	2,964	147
Total	1015091	7157039	12,650

E. Classifier Agent

The Classifier Agent (CA) performs the final classification of prioritised commits from the PA, categorizing them as NPE fixes or non-NPE through a multistage process.

1) Initial Score Calculation: For each prioritised commit patch C_i, an initial classification score, $S_i^{(0)}$, is computed. This score is a weighted sum of three key factors. The 1st is *Priority Score (P_i):* derived directly from the PA, this score reflects the commit's pre-assessed relevance and impact. The 2nd is *Contextual Mapping Score (CM_i):* It quantifies how well the commit's message and patch characteristics (e.g., specific keywords, code patterns) align with predefined features commonly associated with known NPE-related fixes. The 3rd is *Semantic Similarity Score (SS_i):* This represents an initial comparison of the commit patch against the general embeddings of existing NPE fixes stored within the AKB.

The initial score $S_i^{(0)}$ is thus calculated as:

$$S_i^{(0)} = w_1 \cdot P_i + w_2 \cdot CM_i + w_3 \cdot SS_i \tag{9}$$

Here, w_1, w_2, and w_3 are weights that balance the importance of each factor. These weights are empirically determined during model training and specifically optimized to reflect the relative significance of the priority, contextual mapping, and semantic similarity components in predicting NPE fixes. A higher $S_i^{(0)}$ score indicates a greater initial likelihood that the commit is an NPE fix.

2) AKB-driven Contextual Refinement: Following the calculation of the initial score $S_i^{(0)}$, the CA retrieves specific contextual insights from the AKB. The AKB, a vectorized dataset of relevant patterns and historical NPE fixes, is queried by comparing the vector representation $v(C_i)$ of the input commit patch C_i against existing NPE fix vectors v_j using cosine similarity:

$$\text{Sim}(v(C_i), v_j) = \frac{v(C_i) \cdot v_j}{||v(C_i)|| \cdot ||v_j||} \tag{10}$$

where v_j is an AKB context vector and $||v(C_i)||$, $||v_j||$ are vector magnitudes. The CA identifies the top-K most relevant AKB contexts based on scores, and this information is crucial for iterative refinement.

3) Self-Iteration and Feedback (Reasoning Process): This iterative process refines the classification score $S_i^{(t)}$ (initialized with $S_i^{(0)}$) using the highly relevant contexts retrieved from the AKB. For a commit C_i, the score at each iteration t is updated via the formula:

$$S_i^{(t+1)} = S_i^{(t)} + \alpha \cdot \Delta S_i \tag{11}$$

The feedback adjustment ΔS_i is determined by the consistency between the current score's implication $S_i^{(t)}$ and the evidence from the retrieved AKB contexts; its magnitude can be proportional to the strength of this agreement/disagreement. The learning rate α (e.g., empirically set to 0.1) controls the updated influence. This process typically continues for a small, empirically determined number of iterations (e.g., three) to allow the score to converge based on accumulated feedback.

For this process, we employed the ChatGPT-4o which demonstrated strong reasoning capabilities, as evidenced by its performance on established benchmarks: 88.7% accuracy on the Massive Multitask Language Understanding (MMLU) dataset [25], [26], 90.2% pass@1 on HumanEval for code generation [26], [27], and 90.5% accuracy on the Mathematical Reasoning dataset (MGSM) [26], [28]. These results confirm its ability to perform logical reasoning, contextual analysis, and code-related problem-solving, which are critical for accurately identifying NPE fixes in commit patches.

4) Final Classification and Performance Evaluation: After the iterative refinement concludes (yielding a final score $S_i^{(\text{final})}$, e.g., $S_i^{(3)}$), the commit C_i is classified against a predefined threshold T_c:

$$C_i = \begin{cases} 1 \text{ (NPE Fix)}, & \text{if } S_i^{(\text{final})} > T_c \\ 0 \text{ (Non-NPE)}, & \text{if } S_i^{(\text{final})} \leq T_c \end{cases} \tag{12}$$

CA performance is then assessed using standard evaluation metrics: precision, recall, F1-score, False Positive Rate (FPR), False Negative Rate (FNR), and overall accuracy (Equation 13), complemented by confusion matrix analysis.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{13}$$

These collectively determine the CA's effectiveness in distinguishing NPE fixes.

III. Evaluation

A. Research Questions

We investigated the following research questions to provide a thorough analysis of the experimental evaluations.

RQ1: *How does the proposed approach's performance in identifying NPE fixes compare to state-of-the-art models?*

RQ2: *How does an ablation study demonstrate the individual contribution of AACC's core components to its overall performance?*

RQ3: *How well does the AACC approach work with different large language models?*

RQ4: *What is the performance of the AACC approach in cross-language generalization tasks?*

This research evaluates our *AACC* approach for identifying NPE fixes through four research questions (RQs). RQ1 compares *AACC*'s effectiveness against state-of-the-art (SOTA) models in classifying NPE fixes patches. RQ2 investigates the individual contributions of *AACC*'s core components to its overall performance. RQ3 examines *AACC*'s adaptability and consistency when integrated with different Large Language

Models (LLMs) like ChatGPT-4o and DeepSeek-r1. RQ4 tests *AACC*'s generalizability across various programming languages and diverse software development contexts beyond its primary Java training data.

B. Baselines

To evaluate the effectiveness of the *AACC* proposed approach in identifying and classifying NPE fixes, we compared it with six established SOTA models for NPE identification and code generation, providing a robust benchmark.

- Deep Learning-based Approaches: We compared *AACC* with widely recognized state-of-the-art code classification models, including UniXcoder [29], GraphCodeBERT [30], CodeT5 [31], CodeBERT [32], Codereviewer [33], and CodeGPT [34], which serves as our primary baseline. These models are widely recognized as the most advanced tools for code analysis and generation tasks, making them strong benchmarks for evaluating *AACC* performance. For instance, Codereviewer and CodeT5 are explicitly designed for commit classification, while CodeBERT and UniXcoder are foundational models adapted for similar tasks in prior work. CodeBERT is especially a widely recognized model for understanding code semantics. To ensure fairness, we followed established practices to fine-tune these baselines using our Java dataset, aligning their inputs (commits) and outputs (NPE-fixes and Non) with our framework structure.
- LLM-based Approaches: We evaluated *AACC* with leading large language models ChatGPT-4o [35], DeepSeek-r1 [36], Llama-3.2 versatile [37], Gemini-2.0 Flash [38], and Claude 3.5 sonnet [39] fine-tuned in our Java dataset to align their outputs with NPE fixes classification. Although general-purpose LLMs lack task-specific customization, previous studies suggest their potential for code-related tasks [39], which justifies their inclusion as baselines. We selected the latest available versions to ensure a fair comparison. By contrasting the performance of *AACC* with these fine-tuned LLMs, we quantify how the AKB and PA of our framework enhance the accuracy and relevance of NPE fixes identification, demonstrating the value of architectural innovations tailored to the market over generic LLM capabilities. Notably, to ensure fair evaluation and comparison with the proposed AACC, each test item uses the same set of best examples for both the few-shot LLM baselines and AACC (our approach).

C. Dataset and Evaluation Metrics

To construct our Java dataset, as shown in Table I, we mined high-quality, well-known repositories from projects such as Apache, Google, and FasterXML (e.g., commons-cli, commons-lang, commons-collections, closure-compiler, commons-csv). We developed a Python script to automatically identify NPE fixes by scanning commit messages. For each repository, we examined commit patches and their corresponding messages following established practices [40] [41]. A commit was labeled as an NPE fix (positive commit) if it contained any of the following keywords: "*NPE*," "*NullPointerException*," "*null pointer exception*," and "*NPE fix*".

Notably, the commit messages are only used for data collection, and they are not involved in training or inference. Our *AACC* approach classifies NPE fixes based solely on the semantic and structural features of the code changes in the patch. All learning and classification decisions are made exclusively based on the content of the code diff. This fundamentally differentiates our approach from keyword-based baselines, which rely on surface-level keywords in commit messages. By excluding commit messages during training and inference, *AACC* ensures that predictions are based on the actual code modifications, avoiding potential ambiguity or noise inherent in natural language descriptions. To collect negative commits, we randomly selected one commit per positive commit from the same project within the same time window to preserve temporal consistency. We ensured that negative commits did not contain any NPE-related keywords by using a predefined list. To minimize false negatives (e.g., implicit NPE fixes), we manually validated a random sample of 500 negative commits, confirming their non-NPE status.

Therefore, we automatically identified 12,650 commits by searching commit messages and code changes for each commit, and we extracted the full code patch with its corresponding commit message, accumulating a dataset of 7.16 million tokens. To ensure a diverse and informative dataset, after preprocessing, we clustered these commits using semantic embeddings of commit messages and structural embeddings of code patches. We employed K-means clustering and the Elbow Method [42] to determine optimal clusters and select representative examples for in-depth analysis. This process allowed us to build a rich dataset of real-world NPE fixes for training, validating, and testing our *AACC* approach.

In particular, Fu et al. [43] split the data set into training, testing, and a validation set 80/10/10%, and we followed this setting. To prevent data leakage and ensure a realistic evaluation, the splits were constructed at the project level. This guarantees that all commits from any given project are contained within a single split, thereby eliminating any possibility that the model could memorize project-specific patterns or recurring NPE fix idioms from the training set and replay them at test time. Consequently, the reported metrics reflect true generalization to unseen repositories rather than inflated performance due to project-level information leakage. To rigorously evaluate the effectiveness of the *AACC* approach, we employed a suite of widely recognized evaluation metrics [44]. These metrics are crucial to providing a comprehensive understanding of the capabilities to identify NPE fixes accurately. We evaluated *AACC* using six key evaluation metrics, as discussed in Sec. II-E

D. ***RQ1:*** *Improving State-of-the-Art*

To answer RQ1, we independently applied each evaluated approach to the Java Dataset. The evaluation results are presented in Table II. The first column lists the evaluated approaches, while columns 2-5 show the evaluation metrics

TABLE II: Perform of State-of-the-Art

Approaches	P(%)	R(%)	F1(%)	Acc(%)	FPR(%)	FNR(%)
AACC (Our)	**97.45**	**98.37**	**98.03**	**97.00**	**04.32**	**03.92**
UniXcoder	72.40	72.68	72.07	70.45	21.43	17.96
GraphCodeBERT	72.09	70.40	69.24	72.02	10.45	29.64
CodeT5	69.78	68.71	68.09	69.33	12.10	26.29
CodeBERT	64.74	61.49	62.27	76.25	12.38	44.62
Codereviewer	71.37	73.63	72.29	71.08	34.26	09.48
CodeGPT	61.91	58.02	54.15	11.36	13.90	41.98

TABLE III: Impact of Best Example Selection

Setting	P(%)	R(%)	F1(%)	Acc(%)
Enabled BES	97.45	98.37	98.03	96.57
Disabled BES	79.78	85.91	79.81	81.00
Relative Impacts	**+22.14**	**+14.50**	**+22.83**	**+19.22**

for the Java dataset. Columns 5-6 provide comparative metrics between configurations of our proposed approach.

AACC demonstrates a significant leap in performance for NPE fix identification, outperforming SOTA models by a considerable margin. *AACC's* Precision reaches 97.45%, a remarkable 25.05% improvement over UniXcoder's 72.40%. Similarly, *AACC's* Recall of 98.37% surpasses UniXcoder by 25.69%. This leads to a superior F1-Score of 98.03% for *AACC*. Furthermore, *AACC* drastically reduces error rates; its FPR is only 4.32%, a 17.11% reduction compared to UniXcoder's 21.43%. The FNR for *AACC* is also impressively low at 3.52%, a 14.44% improvement over UniXcoder's 17.96%. Compared to CodeReviewer, *AACC*'s FPR is 29.94% lower (CodeReviewer's FPR is 34.26%), and even against GraphCodeBERT, *AACC* maintains a higher precision, demonstrating a clear and substantial performance advantage across key metrics for NPE fixes identification.

Our proposed approach also significantly outperforms few-shot strategies in various LLMs (as detailed in Table VII), notably boosting the *AACC* F1-score by 13.35% to 98.03%. This shows *AACC's* consistent ability to enhance NPE fixes identification and achieve substantial gains across base LLMs.

AACC's superiority over State-of-the-Art (SOTA) models stems from its Augmented Knowledge Base (AKB) (Sec. II-C), which deeply integrates context, code semantics, and structure, unlike the more general pattern analysis of models like UniXcoder or CodeT5. This comprehensive method yields high accuracy (Precision: 97.45%, Recall: 98.37%, F1-score: 98.03%), outperforming SOTA models by 15–30% in F1-score. Furthermore, *AACC* achieves significantly lower error rates (FPR: 4.32%, FNR: 3.52%) compared to competitors such as GraphCodeBERT (FNR 19.64%). The iterative learning mechanism within *AACC* further refines predictions and improves overall performance.

E. ***RQ2:*** *Impact of Core AACC Components*

To answer RQ2, we disabled each component individually and conducted the evaluation again. By comparing the results with those of the default setting (where all components were enabled), we can quantitatively reveal the effect of the given component. The proposed approach comprises the following three main key elements: BES, AKB, and PA. Consequently, we answer RQ2 with the following three independent experiments by validating the effect of each component, respectively.

TABLE IV: Impact of Augmented Knowledge Base

Setting	P(%)	R(%)	F1(%)	Acc(%)
Enabled AKB	97.45	98.37	98.03	96.57
Disabled AKB	77.62	84.15	78.01	78.85
Relative Impacts	**+25.54**	**+16.89**	**+25.66**	**+22.47**

TABLE V: Impact of Prioritise Agent

Setting	P(%)	R(%)	F1(%)	Acc(%)
Enabled Data Priority	97.45	98.37	98.03	96.57
Disabled Data Priority	82.07	87.13	82.87	82.61
Relative Impacts	**+18.74**	**+12.90**	**+18.29**	**+16.89**

1) Impact of Best Examples Selection (BES): To investigate the impact of the proposed BES, see in detail Section II-B, we disabled it and repeated the evaluation. The evaluation results are presented in Table III, where the last row presents the relative impact on precision, recall, and f1-score. BES substantially improved the performance of the proposed approach. With BES enabled, *AACC's* achieves exceptional performance metrics demonstrate the substantial relative improvements due to BES: a 22.14% relative increase in precision (from 79.78% to 97.45%), a 14.50% improvement in recall (from 85.91% to 98.37%), a 22.83% enhancement in F1-score (from 79.81% to 98.03%), and a 19.22% boost in accuracy (from 81.00% to 96.57%). These results clearly demonstrate BES's critical role in enhancing *AACC's* performance by focusing on high-quality, relevant data.

The BES demonstrates its value by curating ideal data, as seen in commits such as Listing 1 where the message and patch clearly align, linking developer intent to code modifications. In contrast, using random samples (e.g., `Refactor logging module`) which introduces noise and irrelevant data, thereby obscuring NPE-specific patterns. By enabling *AACC* to train on such high-quality and focused data, BES improves overall system performance by approximately 30%. This enhancement significantly boosts the model's robustness and reliability, ensuring that it effectively captures nuanced NPE-fixes patterns and sets a new benchmark for software maintenance tools.

2) Impact of Augmented Knowledge Base (AKB): To investigate the impact of AKB proposed as detailed Section II-C, we disabled it and repeated the evaluation. The evaluation results are presented in Table IV, where the last row presents the relative impact in P, R, and F1-score. AKB significantly improved the performance of the proposed approach. With the AKB enabled, *AACC* achieves exceptional performance metrics demonstrates the substantial relative improvements due to the AKB: a 25.54% increase in precision (from 77.62% to 97.45%), a 16.89% improvement in recall (from 84.15% to 98.37%), a 25.66% enhancement in F1-score (from 78.01% to 98.03%), and a 22.47% boost in accuracy (from 78.85% to 96.57%). These results demonstrate the critical role of AKB in boosting the performance of the *AACC* framework.

TABLE VI: Incremental Performance Contributions of Components

System Configuration	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)
AACC(Full Pipeline)	**97.45**	**98.37**	**98.03**	**97.00**
W/O Augmented Knowledge Base (AKB)	77.68 (↓ 19.77)	84.15 (↓ 14.22)	78.01 (↓ 20.02)	78.85 (↓ 18.15)
W/O Best Examples Selection (BES)	79.78 (↓ 17.67)	85.91 (↓ 12.46)	79.81 (↓ 18.22)	81.00 (↓ 16.00)
W/O Priorities Agent (PA)	82.07 (↓ 15.38)	87.13 (↓ 11.24)	82.87 (↓ 15.16)	82.61 (↓ 14.39)

TABLE VII: Performance Across Different LLMs

Approach		P(%)	R(%)	F1(%)	Acc(%)
AACC	Few-Shot	84.23	88.89	84.68	84.76
	KDOS	97.45	98.37	98.03	96.57
	Impacts	**+13.22**	**+9.48**	**+13.35**	**+11.81**
DeepSeek-r1	Few-Shot	84.19	90.07	86.35	84.56
	KDOS	95.69	96.98	97.83	96.18
	Impacts	**+11.50**	**+6.91**	**+11.48**	**+11.62**
Claude 3.5 Sonnet	Few-Shot	83.37	86.25	83.02	83.73
	KDOS	94.03	94.73	95.10	93.27
	Impacts	**+10.66**	**+8.48**	**+12.08**	**+9.54**
Gemini 2.0 Flash	Few-Shot	81.16	84.73	81.86	81.58
	KDOS	92.50	93.42	94.27	92.60
	Impacts	**+11.34**	**+8.69**	**+12.41**	**+11.02**
Llama 3.2 versatile	Few-Shot	80.73	84.39	82.86	80.69
	KDOS	91.54	92.07	92.76	92.35
	Impacts	**+10.18**	**+7.68**	**+9.90**	**+11.66**

Listings 1 and 2 illustrate the AKB value to enable precise commit categorization. By identifying null-check patterns and other contextual cues, the AKB accurately distinguishes NPE fixes from unrelated changes; for instance, correctly flagging commits with specific null-safety checks as NPE related (per Listing 1). Without AKB, such commits, particularly those lacking explicit contextual information, risk misclassification. The AKB boosts relevant performance metrics by approximately 50% when enabled, proving essential to surpass state-of-the-art (SOTA) models through focused commit analysis for training and evaluation.

3) Impact of Prioritise Agent (PA): To investigate the impact of PA proposed as detailed Section II-D, we disabled it and repeated the evaluation. The evaluation results are presented in Table V, where the last row presents the relative impact on precision, recall, and F1-score. PA significantly increased the performance of the proposed approach. With prioritization enabled, *AACC* achieves exceptional performance metrics demonstrate the substantial relative improvements due to prioritisation: an 18.74% increase in precision (from 82.07% to 97.45%), a 12.90% improvement in recall (from 87.13% to 98.37%), an 18.29% enhancement in F1-score (from 82.87% to 98.03%), and a 16.89% boost in accuracy (from 82.61% to 96.57%). These results demonstrate how focusing on relevant high-quality data through prioritisation improves the model's performance in all key metrics, establishing the PA as a critical element of the *AACC* approach effectiveness.

The PA is crucial for identifying critical NPE fixes, such as those containing essential null-check patterns (e.g., Listing 1). The PA directs the *AACC* model to focus on these impactful fixes first, thereby optimizing resource allocation and improving analytical efficiency. Without this prioritisation, resources are misallocated to irrelevant commits, leading to a significant performance drop of ∼20%. By concentrating analytical attention on the most relevant data, the PA proves essential for maintaining *AACC's* overall effectiveness.

The ablation study summarized in Table VI confirms that core AACC components, the Augmented Knowledge Base (AKB), Best Example Selection (BES), and Prioritise Agent (PA), are individually crucial for AACC's optimal performance. Removing any single component substantially degrades overall performance, with F1-score reductions ranging from 15.16 to 20.02 percentage points (pp). The AKB proves most influential; its removal curtails deep contextual understanding and pattern recognition capabilities, causing the largest F1-score drop of 20.02 pp (to 78.01%), and an 18.15 pp accuracy decrease. BES is also indispensable, as its absence means relying on less curated data, decreasing the F1-score by 18.22 pp and highlighting the need for high-quality input. Finally, excluding the PA yields the smallest (yet still significant) F1-score reduction of 15.16 pp, underscoring its benefits in resource optimization and focused analysis. These findings validate *AACC's* synergistic design, where each component distinctively contributes to effective Null Pointer Exception (NPE) fixes identification.

The *AACC* approach utilizes a Knowledge-Driven Optimizations Suite (KDOS), which comprises three core components: Best Example Selection (BES), the Augmented Knowledge Base (AKB), and a Prioritise Agent (PA). Central to *AACC* is the Classifier Agent (CA), powered by GPT-4o, which leverages KDOS to classify commit patches as NPE fixes or non-NPE. The AKB enriches the CA by providing contextual, semantic, and bug-fix pattern insights, which facilitates an iterative refinement process leading to high accuracy. This deep contextual understanding enables the CA to effectively identify NPE-related commits, even in complex scenarios, thereby achieving notable precision and reliability. The synergistic operation of KDOS components, ensuring high-quality data (BES), enriching model knowledge (AKB), and optimizing resource allocation (PA) underpins *AACC's* effectiveness in NPE fix identification.

Our ablation study underscores the significant contributions of *AACC's* key components to its overall performance. Specifically, employing BES to curate high-quality data, utilizing the AKB for deep contextual insights, and leveraging the PA for data prioritisation demonstrably enhance classification accuracy. These elements collectively enable the Classifier Agent to achieve a 98.03% F1-score and 96.57% accuracy,

thereby improving NPE fix classification and contributing to more efficient software maintenance.

F. *RQ3: Working With different LLMs*

To address RQ3, we substituted ChatGPT-4o with various alternative LLMs and reassessed the *AACC* approach. This is a three-fold evaluation: first, to determine if the approach remains effective across different LLMs, and second, to compare the performance improvements achieved with these LLMs against those obtained with ChatGPT-4o.

The *AACC* proposed approach demonstrates robust adaptability across multiple LLMs, as shown in Table VII. By integrating the KDOS, *AACC* consistently enhances precision, recall, F1-score, and accuracy for all tested LLMs, validating its architecture's versatility. For example, with ChatGPT-4o, *AACC* achieves 97.45% precision, 98.37% recall, and 96.57% accuracy, marking improvements of +13.22% precision and +11.81% accuracy over few-shot baselines. Similar gains are observed with DeepSeek-r1 (+11.50% precision) and Claude 3.5 Sonnet (+12.08% F1-score). Even Llama 3.2 Versatile, the least performant model, shows notable improvements (+10.18% precision, +11.66% accuracy). These results confirm that *AACC* innovations, contextual prioritisation, semantic analysis, and iterative refinement deliver consistent enhancements regardless of the underlying LLM, solidifying its role as a universal solution for NPE fix classification.

The *AACC* effectively leverages the strengths of Large Language Models (LLMs) while mitigating their limitations. The core components (BES, AKB, and PA) are model-agnostic and consistently enhance classification accuracy across different LLMs. This adaptability makes *AACC* a reliable solution for real-world software maintenance, ensuring high-quality results regardless of the underlying LLM.

The *AACC* approach shows robust performance across various LLMs, as shown in Table VII. The results, averaged over five executions per model, demonstrate significant enhancements in precision, recall, F1-score, and accuracy when integrated with models like ChatGPT-4o, DeepSeek-r1, Claude 3.5 Sonnet, Gemini 2.0 Flash, or Llama 3.2 Versatile. The F1-score improvements ranged from +09.90% to +13.35%, and accuracy gains from +11.02% to +11.81%. Averaging multiple runs ensured reliability and highlighted *AACC's* model-agnostic nature. This adaptability makes *AACC* a versatile solution for real-world software maintenance, consistently boosting the LLM performance to enhance software reliability and reduce maintenance efforts.

G. *RQ4: Performance Across Different Datasets*

To address RQ4, we evaluated the generalizability of the *AACC* approach across three diverse datasets: our primary Java Dataset, the CVE Dataset [45], and a curated NPE subset from BugSwarm [46]. For the BugSwarm evaluation, we constructed a specialized NPE subset of 170 commits (85 NPE fixes, 85 non-NPE) by filtering for CWE-476 and NPE-related patterns, following Tomassi et al.'s methodology for reproducible bug extraction [1]. Positive labels were assigned through keyword filtering and manual verification of stack traces and code diffs, while negative labels were assigned using our primary dataset's methodology (Section III-C) to exclude NPE-related commits. As shown in Table VIII, the results demonstrate strong cross-dataset performance, with the following analysis detailing *AACC*'s maintained effectiveness across different programming language datasets.

Our approach achieves exceptional performance on the Java Dataset, with significant improvements shown in the Table VIII 'Impacts' row clearly quantifies KDOS's contribution, showing improvements across metrics: a 13.35% increase in F1-score and 11.81% rise in accuracy, proving KDOS's effectiveness in enhancing *AACC's* NPE fixes identification.

Even when tested on the CVE dataset, *AACC*, boosted by KDOS, showed impressive gains. On average, KDOS improved performance by roughly 14% across key measures: precision rose by 13.95%, recall by 16.96%, F1-score by 13.73%, and accuracy by 14.31%. These consistent improvements highlight *AACC's* ability to adapt and generalize its effectiveness beyond its primary training data, showcasing the power of KDOS in new contexts.

AACC consistently improves performance even on the challenging BugSwarm, which is a well-known dataset. With KDOS enabled, *AACC* achieves average gains of approximately 10-13% across key metrics: precision up by 12.86%, recall by 10.69%, F1-score by 10.83%, and accuracy by 9.88%. These improvements demonstrate *AACC's* robust performance across diverse datasets. The KDOS suite ensures *AACC* effectively focuses on relevant data and rich context, even when analyzing code from different projects.

AACC consistently enhances performance across diverse datasets, achieving high performance on the Java dataset with a 98.03% F1-score. It maintains significant average gains across Java, CVE, and BugSwarm datasets: +13.95% precision, +16.96% recall, +13.73% F1-score, and +14.31% accuracy. These improvements underscore *AACC's* adaptability and effectiveness beyond its primary training data. By leveraging the KDOS, *AACC* ensures high-quality data, context-aware predictions, and efficient resource use. This makes *AACC* a robust and versatile solution for identifying NPE fixes, setting a new benchmark for cross-language bug identification.

H. *Threats to Validity*

A threat to construct validity is that our dataset construction relies on keywords in commit messages to identify positive samples (NPE fixes), which may yield false negatives by missing commits that fix NPEs implicitly without mentioning these terms. It may also result in false positives that mention NPE keywords for a reason other than fixing an NPE. We mitigated this by manually verifying the positive commits.

A threat to external validity is that the evaluation is confined to open-source Java applications, which may limit the generalizability of the conclusions to proprietary codebases or other languages. Further validation on industrial systems and diverse languages like Python and C++ is needed to confirm full external validity.

TABLE VIII: Performance Across Different Datasets

Setting	Java Dataset				CVE Dataset				BugSwarm Dataset			
	P (%)	R (%)	F1 (%)	Acc (%)	P (%)	R (%)	F1 (%)	Acc (%)	P (%)	R (%)	F1 (%)	Acc (%)
Enabled	97.45	98.37	98.03	96.57	88.52	90.02	91.09	88.07	88.27	83.03	83.35	82.55
Disabled	84.23	88.89	84.68	84.76	74.57	73.06	77.36	73.76	75.41	72.34	72.52	72.67
Impacts	**+13.22**	**+9.48**	**+13.35**	**+11.81**	**+13.95**	**+16.96**	**+13.73**	**+14.31**	**+12.86**	**+10.69**	**+10.83**	**+9.88**

A threat to the internal validity is the inherent opacity of the LLM. Our approach relies on ChatGPT-4o for reasoning. However, the model's reasoning may rely on superficial code patterns rather than deep null-safety understanding.

A threat to the internal validity is limited prompt variations and sensitivity in *AACC*'s LLM components (Section II-E) may affect evaluation robustness. Multi-LLM testing (9.90%–13.35% F1-score gains, Table VII) mitigates this by ensuring consistent performance across models. Future work will explore comprehensive prompt engineering and diverse prompt structures to further strengthen *AACC*'s robustness.

IV. Case Study

To demonstrate the usefulness of the proposed approach, in this section, we investigate the extent to which it can help prioritise patches (commits) for code review.

A. Data Collection

We curated a new evaluation dataset from the Apache Tomcat repository, which comprises 20 groups of commits. To do that, we first identify all potential NPE-fixing commits (noted as $NPEs$) with the proposed approach and the keyword-based baseline. We ranked such NPE-fixing commits according to their submission time and processed each of them as follows:

- First, we took the first NPE-fixing commit (noted as $curNPE$) in $NPEs$, and collected 25 commits directly before it and 24 commits directly after it, resulting in a group of 50 commits that contained at least one NPE-fixing commit.
- Second, we removed all commits (including $curNPE$) from $NPEs$ that appeared in the resulting group.
- If we have collected 20 groups, we terminate the data collection. Otherwise, we turned to the first step and constructed the next group (of commits) in the same way.

Notably, none of such commits has been used as training or testing data in the previous section.

B. Process

For each group of commits, we used AACC to rank commits, and prioritised NPE fixes to the top. We also ranked the commits with two straightforward and practical baselines:

- A1 (Time-Based): Ranking commits by submission time, and thus the elder commits are ranked at the top.
- A2 (Keyword-Based): Prioritising commits that are identified as NPE fixing commits by the keyword-based baseline approach.

For each of the 20 commit groups, we evaluated the ranking of our approach and the time-based approach as follows. First, for each of the NPE commits ranked at the top (noted as *curNPE*) by our approach, we randomly sampled three commits that were ranked before *curNPE* by the time-based approach but ranked after *curNPE* by our approach. For each of the resulting samples (noted as *nonNPE*), we constructed a question [47] asking to what extent the participants agree that *curNPE* should be reviewed before *nonNPE*, with '*strongly agree*', '*agree*', '*neutral*', '*disagree*', and '*strongly disagree*' as possible answers. 20 developers with more than three years of software development and two years of GitHub-based development participated in the questionnaire survey, and each of the questions was answered by at least 15 participants. The difference between our approach and the keyword-based approach was evaluated in the same way.

C. Results and Analysis

On 19 out of the 20 groups, our approach resulted in rankings different from the time-based ranking. They had an identical ranking in the remaining group because our approach failed to identify the NPE fixing commits. From the 19 inconsistently ranked groups, we extracted 57 pairs of inconsistently ranked commits, and on 33%=19/57 of the cases, our ranking was preferred by developers strongly, with an average score of 2 ('*strongly agree*'). On 95%=54/57 of the cases, our ranking was preferred by developers, with an average score of no less than 1 ('*agree*'). Our approach was unpreferred (or was deemed incorrect) in one group because, upon manual inspection, developers determined that the non-NPE commit (a critical security patch) addressed a more urgent issue than the NPE fix prioritised by our approach in that specific context. This highlights that while generally effective, automatic prioritisation can occasionally disagree with human judgment of contextual criticality, which can be influenced by factors beyond the scope of a single bug type.

On 14 out of the 20 groups, our approach resulted in rankings different from the keyword-based ranking. From the 14 inconsistently ranked groups, we extracted 42 pairs of inconsistently ranked commits, and on 38%=16/42 of the cases, our ranking was preferred by developers strongly (with an average score of 2 ('*strongly agree*'), and on 93%=39/42 of the cases our ranking was preferred by developers with an average score no less than 1 ('*agree*'). These results demonstrate that our approach outperforms the keyword-based baseline in prioritising NPE-fixing patches.

D. Other Potential Applications

The *AACC* approach offers significant value across multiple software engineering scenarios by leveraging critical commit and patch analysis; it enhances software evolution understanding by generating meaningful explanations for poorly documented commits and supporting automated commit message generation, while also aiding project managers and code reviewers in prioritising code review and branch merging through identifying critical NPE-fixing patches. Furthermore, the approach facilitates the construction of large-scale, high-quality, unbiased, and diverse datasets for testing and evaluating NPE-related approaches, reducing bias from well-documented commits; these datasets also enable data-driven, learning-based methods for detecting and fixing NPEs, leveraging advances in machine learning and AI. Additionally, the method supports experts and automated systems in analyzing NPE-related defects and patches to uncover patterns, thereby improving software quality and reliability. Despite its effectiveness, the approach may occasionally misalign with human judgment where contextual factors prioritise non-NPE issues, suggesting a need for future work to incorporate broader contextual cues into the prioritisation process.

V. Related Work

The automated identification and classification of software bugs, particularly NullPointerException or null pointer dereference, is a critical research area in software engineering. To tackle this issue, researchers have developed various approaches, including static and dynamic analysis tools, machine learning, and natural language processing [17] [48] [49]. This section reviews existing literature relevant to this approach.

A. Static and Dynamic Analysis Approaches

Static analysis detects NPEs by examining code pre-execution. For instance, tools like SymlogRepair [50] utilize Java program semantics, while NPEX [51] employs data-flow analysis. Although research increasingly integrates commit message mining for better developer intent linking, static tools often struggle with large-scale, complex codebases and in accurately distinguishing intentional null usage from errors.

Dynamic analysis, conversely, identifies NPEs by monitoring runtime behavior, utilizing methods such as automated tools in controlled settings [52], machine learning combined with hardware or system call data [53], [54], and unsupervised hardware-feature analysis prominent in fields like malware detection [55]. However, its significant computational resource and specific environment demands often limit broader applicability. In contrast, our *AACC* approach analyzes historical NPE fixes from commit data, leveraging static code properties and contextual insights. This allows for efficient classification of NPE fixes without the need for runtime monitoring.

B. Machine Learning and Deep Learning Approaches

Machine learning (ML) and deep learning (DL) significantly advance NPE identification by analyzing code patterns, historical fixes, and contextual data [56]. ML can spot subtle code anomalies indicating NPEs [57], DL models generalize NPE detection across diverse environments [58], and graph-based methods like Graph Convolutional Networks (GCNs) [59] (for dependencies) and Graph Neural Networks (GNNs) [60] (for cross-project generalization) further bolster these capabilities. While such methods face challenges, including handling dynamic program states, our *AACC* approach takes a distinct path. *AACC* employs LLM models to classify developer-resolved NPEs by analyzing commit information (messages and patches), focusing on human-intended solutions, which contrasts with automated patch generation and its inherent semantic equivalence complexities.

C. Transformer and LLM approaches

Transformer models and Large Language Models (LLMs) are pivotal for code understanding and NPE analysis. For instance, APT-LLM [61] uses transformers to detect code anomalies, while CodeBERT [62] parses code structure, though its specific NPE application is still evolving. Crucially, LLMs can identify NPE fixes within commit data by leveraging contextual understanding [61], [62], and transformer-generated commit embeddings also promise more accurate fixes classification [61]. Specialized LLM training on large code-focused datasets represents a key future direction for enhancing accuracy. Our *AACC* approach builds on these LLM advancements for robust NPE fix identification, uniquely integrating their contextual capabilities with our specialized BES, AKB, and PA components. This synthesis, incorporating feedback-driven learning, targets high accuracy in classifying developer-applied fixes, thereby advancing NPE analysis.

VI. Conclusion and Future Work

In this paper, we introduced *AACC*, a novel approach designed to categorize commit patches as NPE fixes accurately. *AACC* leverages code structure and commit message context, incorporating Best Example Selection, an Augmented Knowledge Base, a Prioritise Agent, and Iterative Refinement. Evaluated with ChatGPT-4o, *AACC* demonstrates a significant performance leap, outperforming state-of-the-art methods by achieving a substantial F1-score improvement from 72.07% to 98.03%, marking a significant advancement in NPE fix identification. Future work will expand the *AACC's* approach across more diverse programming languages and conduct developer studies to further assess its practical utility.

VII. Data Availability

The replication package is publicly available [24]

Acknowledgement

This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037).

REFERENCES

[1] D. A. Tomassi and C. Rubio-González, "On the real-world effectiveness of static bug detectors at finding null pointer exceptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 292–303.

[2] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009.

[3] S. Banerjee, L. Clapp, and M. Sridharan, "Nullaway: Practical type-based null safety for java," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 740–750.

[4] C. Trubiani, R. Pinciroli, A. Biaggi, and F. A. Fontana, "Automated detection of software performance antipatterns in java-based applications," *IEEE Transactions on Software Engineering*, 2023.

[5] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[6] R. Madhavan and R. Komondoor, "Null dereference verification via over-approximated weakest pre-conditions analysis," *ACM Sigplan Notices*, vol. 46, no. 10, pp. 1033–1052, 2011.

[7] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 133–143.

[8] "Intellij idea," https://www.jetbrains.com/idea/, 2021, accessed: 2021.

[9] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[10] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 459–465.

[11] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 332–343.

[12] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[13] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda, "Verifying dereference safety via expanding-scope analysis," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 213–224.

[14] H. A. Khan, Y. Jiang, Q. Umer, Y. Zhang, W. Akram, and H. Liu, "Has my code been stolen for model training? a naturalness based approach to code contamination detection," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3715765

[15] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *Proceedings of the fourth ACM international workshop on security and privacy analytics*, 2018, pp. 31–39.

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[17] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[18] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[19] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[20] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques for source code analysis," *arXiv preprint arXiv:2110.09610*, 2021.

[21] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, p. 4793, 2021.

[22] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 349–358.

[23] W. Akram, Y. Jiang, Y. Zhang, H. A. Khan, and H. Liu, "Llm-based method name suggestion with automatically generated context-rich prompts," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3715753

[24] NPE-Identification, "Nullpointerexception," 2025, accessed: 2025-03-14. [Online]. Available: https://github.com/NPE-Identification/NullPointerException

[25] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," *arXiv preprint arXiv:2009.03300*, 2020.

[26] Wielded, "GPT-4o Benchmark: Detailed Comparison with Claude and Gemini," https://wielded.com/blog/gpt-4o-benchmark-detailed-comparison-with-claude-and-gemini, 2024, accessed: 2025.

[27] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[28] Y. Hong, Q. Li, D. Ciao, S. Huang, and S.-C. Zhu, "Learning by fixing: Solving math word problems with weak supervision," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 6, 2021, pp. 4959–4967.

[29] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[30] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[31] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[33] W. Zhong, C. Li, K. Liu, J. Ge, B. Luo, T. F. Bissyandé, and V. Ng, "Benchmarking and categorizing the performance of neural program repair systems for java," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 1, pp. 1–35, 2024.

[34] A. Grishina, M. Hort, and L. Moonen, "The earlybird catches the bug: On exploiting early layers of encoder models for more efficient code classification," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 895–907.

[35] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[36] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[37] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[38] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[39] E. Basic and A. Giaretta, "Large language models and code security: A systematic literature review," *arXiv preprint arXiv:2412.15004*, 2024.

[40] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[41] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[42] Y. Yang, E. Ronchieri, and M. Canaparo, "Natural language processing application on commit messages: a case study on hep software," *Applied Sciences*, vol. 12, no. 21, p. 10773, 2022.

[43] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[44] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th*

ieee/acm international conference on software engineering, 2024, pp. 1–13.

[45] MITRE Corporation, "CWE-476: NULL Pointer Dereference." [Online]. Available: https://cwe.mitre.org/data/definitions/476.html

[46] BugSwarm Project, "BugSwarm Dataset." [Online]. Available: http://www.bugswarm.org

[47] P. survey, "patch prioritization," https://docs.google.com/forms/d/e/1FAIpQLSe9-_yAJllLIT2UP2vJcNsR92I33Gw9dTw3OFh3YMI_uP-l3A/viewform?usp=dialog, 2025, accessed: 2025-09-11.

[48] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and T. Cong, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 7, pp. 1–34, 2024.

[49] P. Bhandari and G. Rodríguez-Pérez, "Buggin: Automatic intrinsic bugs classification model using nlp and ml," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2023, pp. 2–11.

[50] Y. Liu, S. Mechtaev, P. Subotić, and A. Roychoudhury, "Program repair guided by datalog-defined static analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1216–1228.

[51] J. Lee, S. Hong, and H. Oh, "Npex: Repairing java null pointer exceptions without tests," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1532–1544.

[52] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM computing surveys (CSUR)*, vol. 44, no. 2, pp. 1–42, 2008.

[53] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri, "A theoretical study of hardware performance counters-based malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 512–525, 2019.

[54] C. Li, A. Raghunathan, and N. K. Jha, "A trusted virtual machine in an untrusted management environment," *IEEE Transactions on services computing*, vol. 5, no. 4, pp. 472–483, 2011.

[55] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17*. Springer, 2014, pp. 109–129.

[56] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li, X. Lian, G. Meng, X. Peng, H. Sun, L. Shi, B. Wang, C. Wang, J. Wang, T. Wang, J. Xuan, X. Xia, Y. Yang, Y. Yang, L. Zhang, Y. Zhou, and L. Zhang, "Deep learning-based software engineering: progress, challenges, and opportunities," *SCIENCE CHINA Information Sciences*, vol. 68, no. 1, p. 111102, 2025. [Online]. Available: https://doi.org/10.1007/s11432-023-4127-5

[57] M. Evangelou and N. M. Adams, "An anomaly detection framework for cyber-security data," *Computers & Security*, vol. 97, p. 101941, 2020.

[58] I. Ullah and Q. H. Mahmoud, "Design and development of a deep learning-based model for anomaly detection in iot networks," *IEEE Access*, vol. 9, pp. 103 906–103 926, 2021.

[59] S. Jacob, Y. Qiao, Y. Ye, and B. Lee, "Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks," *Computers & Security*, vol. 118, p. 102728, 2022.

[60] C. Wang and H. Zhu, "Wrongdoing monitor: A graph-based behavioral anomaly detection in cyber security," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2703–2718, 2022.

[61] S. Benabderrahmane, P. Valtchev, J. Cheney, and T. Rahwan, "Apt-llm: Embedding-based anomaly detection of cyber advanced persistent threats using large language models," in *2025 13th International Symposium on Digital Forensics and Security (ISDFS)*. IEEE, 2025, pp. 1–6.

[62] X. Zhao, X. Leng, L. Wang, N. Wang, and Y. Liu, "Efficient anomaly detection in tabular cybersecurity data using large language models," *Scientific Reports*, vol. 15, no. 1, p. 3344, 2025.