Automated Inline Comment Smell Detection and Repair with Large Language Models

Hatice Kübra Çağlar
Computer Science
Bilkent University
Udemy, Inc.
Ankara, Turkey
kubra.caglar@udemy.com

Semih Çağlar
Computer Science
Bilkent University
Ankara, Turkey
semih.caglar@ug.bilkent.edu.tr

Eray Tüzün
Computer Science
Bilkent University
Ankara, Turkey
eraytuzun@cs.bilkent.edu.tr

***Abstract*—Context: Code comments play a critical role in improving code readability, maintainability, and collaborative development. However, comments may deviate from best practices due to software evolution, where code changes are not reflected in comments, as well as practitioner-related issues such as vague descriptions, redundancy, or misaligned intent. These issues lead to various comment smells that degrade software quality. While prior studies have explored comment inconsistencies, most are limited in scope, either addressing a narrow subset of smells or focusing solely on detection without considering repair.**

Objective: This study evaluates the effectiveness of large language models (LLMs) in both detecting and repairing inline code comment smells, using a comprehensive taxonomy of code comment smell types.

Method: We extended a prior data set by incorporating repaired versions of smelly comments, resulting in 2,211 unique instances. Four LLMs—*GPT-4o-mini*, *o3-mini*, *DeepSeek-V3*, and *Codestral-2501*—are evaluated under zero-shot and few-shot prompting strategies. To account for non-deterministic behavior in LLM outputs and ensure robustness, each configuration is executed five times. Detection performance is measured using accuracy, macro F1 score, and Matthews correlation coefficient (MCC); repair is evaluated using SBERT similarity, METEOR, and ROUGE-L. Our multi-stage pipeline feeds detection outputs into the repair phase, where the detection result with the highest macro F1 score is used to simulate the best possible repair scenario. Median scores across runs are reported for comparison.

Results: *o3-mini* with few-shot prompting achieves the highest median detection performance: macro F1 of 0.41, MCC of 0.50, and accuracy of 0.72, exceeding the baseline of GPT-4. For repair, *Codestral-2501* in the zero-shot setting yields the best results with a median SBERT score of 0.61, followed by *DeepSeek-V3* and *GPT-4o-mini* at 0.53, and *o3-mini* at 0.46. Few-shot prompts improve detection, while zero-shot prompts are more effective for repair.

Conclusion: Lightweight LLMs such as *o3-mini* can achieve strong detection performance when guided by effective few-shot prompts. For example, *o3-mini* with few-shot prompting attains the highest median detection results: macro F1 of 0.41, MCC of 0.50, and accuracy of 0.72, surpassing the GPT-4 baseline. In contrast, repair tasks benefit more from zero-shot prompting, though they introduce challenges such as overfitting and the risk of generating new smells. Our findings support the development of practical tools, including a GitHub-integrated comment repair assistant, and motivate future work on dynamic prompt selection and multilingual benchmark construction.

***Index Terms*—Comment Smells, Code Comment Smell Detection, Code Comment Repair, Code Comment Update, Large Language Models**

I. Introduction

Documentation, particularly in the form of code comments, is crucial for understanding, maintaining, and collaborating on software systems [1]–[4]. Their importance is highlighted in multiple studies. Misra et al. [1] found that comments significantly expedited issue resolution in their study of 625 GitHub repositories, where increased relevant comments reduced the time taken to address problems. Similarly, Hartzmann and Austin [4] emphasize that comments are indispensable for the long-term sustainability of large systems. Woodfield et al. [2] conducted experiments with 48 experienced developers, demonstrating that participants working with commented modular code performed significantly better on a 20-question comprehension quiz, answering more questions correctly. Spinellis and Diomidis [3] and Yang et al. [5] warn that inconsistencies between comments and code can mislead developers, potentially introducing errors. Thus, while comments help developers navigate complex algorithms, elaborate data structures, and highlight edge cases, maintaining their relevance and alignment with the evolving code is essential for maximizing their utility and avoiding confusion [3]. Comments are classified as *method comments*, which describe the purpose and functionality of methods, and *inline comments*, which clarify specific lines or sections of code [6]. In this study, we will focus on inline comments.

Several issues can arise during the comment-writing process that deviate from best practices. One common problem is writing comments that restate what the code already expresses, offering little added value. Another issue is that some comments, while initially accurate, might become outdated as the code evolves, leading to inconsistencies between the comment and the actual behavior of the code. These misleading or low-value comments can confuse developers and diminish the role of comments in supporting code comprehension [3], [7]. Also, redundant or irrelevant comments may be more harmful than their absence, as they clutter the code and distract developers from their primary tasks [3]. Yang et al. [5] show that inconsistencies between code and comments can mislead developers and potentially introduce bugs. In our study, we refer to these suboptimal commenting practices as *code comment smells*.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00088

Recent studies [8]–[10] emphasize the negative impact of code comment smells, reinforcing the need for better practices in writing and maintaining inline comments. Addressing these problems is critical for improving code clarity, maintainability, and overall developer productivity.

Rani et al. [11] conducted a comprehensive review of research on code comment quality, identifying key attributes and highlighting gaps in current practices. They stressed the importance of establishing standardized definitions and developing tools that support multiple programming languages to better evaluate and enhance comment quality. Automated tools are particularly valuable in this context, as they improve the detection of undesirable patterns, such as code comment smells, which might otherwise go unnoticed due to human limitations or oversight.

Existing approaches to detecting code comment smells [5], [8], [9], [12]–[16] primarily target a narrow subset of smell types—often lumped together as *comment inconsistencies*, or *incoherent comments* [7]. While these methods excel at flagging mismatches between code and its accompanying commentary, they lack the generalizability needed to cover the full spectrum of comment-quality issues. To broaden the scope of smell detection, Oztas et al. [17] applied both traditional machine learning models and GPT-4 to the richer eleven-category taxonomy proposed by Jabrayilzade et al. [18], [19]. However, their work remained confined to a single-prompt, detection-only setting. In parallel, several comment-repair frameworks [10], [12], [13], [20]–[22] have been developed to automatically update or correct comments. However, these approaches primarily target Java datasets and address only specific inconsistency types, without considering a comprehensive taxonomy of comment smells. Furthermore, the majority of the existing work on comment smell detection and repair focuses exclusively on method-level comments, whereas our study specifically targets inline code comments, a distinction that will be elaborated upon in Section II.

Large Language Models (LLMs) have emerged as powerful tools for a wide range of software engineering tasks. Leveraging training on massive code corpora, LLMs are capable of deeply understanding code semantics and identifying subtle issues. In contrast to traditional machine learning approaches, which require costly task-specific pretraining, struggle to generalize to new codebases, and often overfit on imbalanced datasets [17], LLMs can process code and comments across programming languages without retraining. This flexibility makes them particularly well-suited for detecting and repairing inline comment smells.

Through this approach, our study addresses a key gap in the literature by moving beyond prior work that has been constrained in terms of granularity, coverage of smell taxonomies, or task scope. To the best of our knowledge, this is the first study to investigate the repair of inline code comment smells. To structure the investigation and systematically evaluate model performance, we formulate the following research questions:

RQ1: To what extent can LLMs detect inline code comment smells under varying prompt input combinations?

To address this question, we evaluate four different LLMs under two distinct prompting strategies. Each configuration is executed five times to reduce the impact of output variability due to inherent model randomness. Detection performance is measured using standard quantitative metrics, and comparisons between models are based on median performance scores to ensure robustness.

RQ2: To what extent can LLMs repair inline code comment smells under varying prompt input combinations?

To maximize performance and better understand the capabilities of LLMs in comment repair, we use the best-performing detection outputs as inputs for the repair phase. Repairs are conducted using the same set of LLMs, prompt variants, and multiple iterations. The dataset provides ground-truth repaired versions for each smelly comment, enabling quantitative evaluation of the generated repairs. Specifically, we compute semantic similarity scores between generated and ground-truth comments, reporting the median scores across five runs to ensure robustness against variability. Repeated runs further allow us to assess the consistency and reliability of each model in producing contextually appropriate, non-smelly comment revisions.

II. Background and Related Work

A. Classification of Inline Code Comment Smells

Jabrayilzade et al. [19] investigate inline code comment smells and propose a taxonomy comprising eleven distinct types: *misleading*, *obvious*, *no comment on non-obvious code*, *commented-out code*, *irrelevant*, *task*, *too much information*, *attribution*, *beautification*, *non-local*, and *vague*. The taxonomy is grounded in a multivocal literature review and validated through an online survey of 41 practitioners. Their study spans 2,447 inline comments from eight open-source projects (four Java, four Python), selected to reflect diverse domains and codebase sizes. The results show that 44% of the analyzed comments exhibit smells, with *obvious comments* being the most prevalent. The authors further assessed the community's receptiveness to manually repairing such smells by submitting 53 pull requests or issues, of which 27% were accepted. This work highlights the impact of comment smells on code comprehension and maintainability, offering a practical and empirically grounded taxonomy for further research and tooling.

In contrast to our study, prior research has frequently addressed comment quality using generalized categories, such as *inconsistent* comments [8]–[10], [14], [15], [21], [22], or labeled them as *outdated* [8], [9], [12], [13], *incorrect* [8], [9], or *misleading* [8], [9]. While these efforts have laid valuable groundwork, they often lack the taxonomy needed to capture the nuanced range of comment quality issues.

To fill this gap, we build on the taxonomy introduced by Jabrayilzade et al. [19], whose fine-grained categorization and annotated dataset provide a robust foundation for training and evaluating LLMs in both detection and repair tasks. The full

taxonomy, along with representative examples, is presented in Table I.

B. Code Comment Smell Detection

The detection of code-comment inconsistencies have received significant attention due to their critical impact on software maintainability and developer productivity [21]. Various approaches, ranging from natural language processing (NLP) to deep learning, have been proposed to address these issues.

iComment, employs NLP, machine learning, and static analysis to extract programming rules from comments and verify their consistency with code. Evaluated on large-scale C and C++ projects, including Linux, Mozilla, Wine, and Apache, iComment extracted 1,832 rules with 90.8–100% accuracy, detecting 60 inconsistencies comprising 33 bugs and 27 misleading comments, of which 19 were confirmed by developers [9].

Focusing on Java, tComment introduces a tool that detects Javadoc inconsistencies, specifically null value properties and related exceptions. Using a modified version of the Randoop tool for dynamic analysis, it inferred 2,479 properties with 97–100% accuracy across seven open-source Java projects. It identified 29 true inconsistencies, 39 false alarms, and led to the resolution of 5 confirmed inconsistencies, demonstrating its utility in enhancing code reliability [15].

Exploring semantic consistency, a study using a Siamese Recurrent Network (RNN-LSTM) analyzed code-comment pairs in Java projects. By considering semantic meaning and sequence ordering, this method achieved a recall of 91.7% and a precision of 63.2% when evaluated on 2,881 pairs, identifying inconsistencies even in cases of sequence reorderings. Notably, it flagged 115 out of 180 deliberately reordered consistent pairs as inconsistent, highlighting its robustness in detecting nuanced issues [14].

Addressing documentation quality, RepliComment focuses on identifying cloned comments in Java code, particularly Type I (exact duplicates) and Type III (minor variations). Evaluated on 10 popular Java projects, the tool detected over 11,000 comment clones, with 12% deemed high-severity issues. Compared to its predecessor, RepliComment achieved a precision of 79%, reduced false positives by 8%, and maintained a false-negative rate of just 0.3%, emphasizing its effectiveness in mitigating copy-paste issues [23].

A broader analysis examined code-comment inconsistencies in Java projects across 1,500 GitHub repositories. Analyzing over 3.3 million commits and 1.3 billion AST-level changes, it categorized inconsistencies into six major types, such as outdated descriptions and missing updates after refactoring. The study found that 13–20% of code changes triggered comment updates, with categories like variable declaration and selection changes showing higher likelihoods. Manual inspection revealed 69 types of comment changes, with 25 linked to fixing inconsistencies caused by issues like renaming or technical debt [8].

Igbomezie et al. [7] introduced Co3D, a practical coherence detection tool leveraging word embeddings and LSTM models to assess the alignment between code and its accompanying comments. Their work demonstrates that simpler models, such as Word2Vec-based LSTM architectures, can offer competitive performance compared to larger pre-trained models like CodeBERT, challenging the assumption that LLMs are always superior. Empirical results show that Co3D's LSTM variant achieved an overall accuracy of 92.8% and an F1 score of 95.2%, matching or outperforming the CodeBERT-based configuration on several datasets. Notably, on the CoffeeMaker dataset, both the simple RNN and LSTM achieved perfect precision and recall (1.000), surpassing CodeBERT. These results highlight lightweight models' promise for code-comment coherence tasks in constrained environments.

Oztas et al. [17] expanded comment smell detection by incorporating a broader taxonomy and linking comments to their corresponding code snippets, producing a refined dataset of 2,211 samples. They trained traditional ML models on six major smell types and evaluated GPT-4 on all eleven. While Random Forest achieved the best ML results (69% accuracy, 0.44 MCC), GPT-4's accuracy improved from 34% to 55% with code context, though its MCC remained at 0.28. Their study establishes a contextual baseline for multi-class comment smell detection.

C. Code Comment Smell Repair

This section reviews automated tools for resolving code-comment inconsistencies (CCI), primarily in Java projects [5], [10], [12], [13], [20]–[22], highlighting their role in improving code quality and reducing maintenance effort.

CUP (Comment UPdater) [12] is a tool that automates "Just-In-Time (JIT)" comment updating using a neural sequence-to-sequence model enhanced with features like a unified vocabulary, co-attention mechanisms, and pre-trained embeddings. Evaluated on over 108K samples from real-world Java projects, CUP achieved 16.7% accuracy, a RED of 0.958, and significantly reduced developers' manual editing efforts. Metrics such as accuracy, Recall@5, AED, RED, BLEU-4, and METEOR demonstrated CUP's superiority over rule-based and information retrieval-based baselines [13]. Despite this, limitations in handling non-trivial updates were noted. A heuristic-based alternative, HEBCUP, was introduced, which utilized sub-token-level analysis to achieve a higher accuracy of 25.6% and a 1,700x speedup compared to CUP, advocating for simpler and faster solutions.

CUP2 [20], an enhanced two-stage framework comprising an Obsolete Comment Detector (OCD) and an updated CUP model, further improved performance. By integrating neural network models with unified vocabularies, pre-trained fastText embeddings, and pointer generators, CUP2 addressed diverse code-comment changes effectively. Evaluated on a dataset of 4 million samples, CUP2 achieved a 41.8% identical match rate, a RED of 0.843, and higher Precision, Recall, and F1-Score than previous approaches, reducing developer effort significantly. This framework outperformed both rule-based and machine-learning baselines, further establishing the value of automated tools in ensuring comment quality.

TABLE I
TAXONOMY OF INLINE CODE COMMENT SMELLS [19].

Smell type	Description	Example

Misleading	Comments that do not accurately represent what the code does	public int add(int x, int y){
		// Returns x - y
		return x + y;
		}

Obvious	Comments that restate what the code does in an obvious manner	count = 0; // assigning count a value of 0
No comment on non-obvious code	Non-obvious piece of code is left without a comment	if (count >55 && count <70) { ... }
Commented-out code	A code piece that is commented out	//facade.registerProxy(newSoundAssetProxy());
Irrelevant	Comments that do not intend to explain the code	/* I dedicate all this code, all my work, to my wife, Darlene, who will have to support me and our children and the dog once it gets released into the public.*/
Task	Comments explaining the work that could/should be done in future or was already completed	// TODO: clear and optimize this code later

Too much information	Overly verbose comments	// this makes a new scanner, which can read from
		// STDIN, located at System.in. The scanner lets us look
		// for tokens, aka stuff the user has entered.
		Scanner sc = new Scanner(System.in);

Attribution	Comments that give information about who wrote the code	/* Added by Rick */
Beautification	Comments that aim to distinguish the parts of the code	//********** // VARIABLES //**********

Non-local	Comments that provide systemwide information or mention code that is not near	public void setFitnessePort(int fitnessePort) {
		// Port on which fitnesse would run. Defaults to 8082
		this.fitnessePort = fitnessePort;
		}

Vague	Comments that are not clearly understandable	taskTmpPath = ttp; //_task_tmp

Yang et al. [5] introduced CBS (Classifying Before Synchronizing), a composite approach that performs both detection and repair of code-comment inconsistencies. CBS first predicts whether a given Code-Comment Inconsistent (CCI) sample is better handled by a heuristic-based approach (HebCUP) or a deep learning model (CUP), based on five engineered features. Using this prediction, it selectively applies HebCUP or CUP for repair. Evaluated on a dataset of 9,204 testing samples from 1,496 Java repositories, CBS achieved an accuracy of 27.90%, Recall@5 of 35.67%, an Average Edit Distance (AED) of 3.562, a Relative Edit Distance (RED) of 0.937, a BLEU-4 score of 73.16, and an Effective Synchronized Sample (ESS) ratio of 35.85%. Notably, CBS outperformed CUP by 38.41% in accuracy and HebCUP by 8.53%, highlighting its ability to integrate model specialization for improved synchronization.

A hybrid deep learning system leveraging syntactic encoding and multi-head attention achieved an F1-score of 87.1% and a BLEU-4 score of 77.2. This system, evaluated on a dataset of 40,688 examples from 1,518 Java projects, outperformed models like CodeBERT and rule-based methods, demonstrating promise for real-time comment maintenance [10]. Similarly, DocChecker, a framework built on the UniXcoder model, attained state-of-the-art results for the ICCD task with a 74.3% F1-score and 72.3% accuracy. Additionally, DocChecker excelled in code summarization with a BLEU-4 score of 33.64, surpassing models like CodeT5 and GPT-3.5. These evaluations utilized datasets like Just-In-Time [10] and CodeXGLUE [24], further solidifying its broad applicability across multiple programming languages [21].

C4RLLaMA, a fine-tuned version of CodeLLaMA employing Chain-of-Thought (CoT) processing and a specialized loss function, represents an advancement in CCI detection and rectification. Achieving a superior result with an F1-score of 89%, accuracy of 89.6%, and rectification metrics such as BLEU-4 (82.6%) and METEOR (89.2%), C4RLLaMA outperformed baselines including CodeBERT and Longformer. Using 40,688 Java code-comment pairs from Panthaplackel [10], it demonstrated effective alignment with tags like @param and @return. However, challenges in handling external dependencies and complex semantics persist [22].

D. Granularity and Scope of Related Work

With respect to smell granularity, Tan et al. [9] also explored inline comment smell detection (without repair). However, the unavailability of their source code and dataset prevents experimental comparison, and their taxonomy diverges from ours, making direct alignment infeasible. The only directly comparable study is by Oztas et al. [17], which also targets inline comment smells for detection; using the same dataset,

we demonstrate that our approach achieves superior detection performance.

Other related works [5], [7], [8], [10], [12]–[15], [20]–[22], [24] focus primarily on method-level comments. Given these differences in comment granularity and taxonomy, direct comparisons with their results are not possible. To the best of our knowledge, our study is the first to address the repair of inline code comment smells.

III. Methodology

This study addresses the detection and repair of inline code comment smells using four LLMs under two prompt configurations: zero-shot and few-shot. As illustrated in Figure 1, our methodology consists of five key stages: dataset construction, comment smell classification, detection performance evaluation, repair of smelly comments, and repair quality evaluation.

A. Dataset Construction

Our study builds on the dataset introduced by Jabrayilzade et al. [19], which consists of 2,448 inline comments labeled with one of eleven comment smell types. Oztas et al. [17] extended this dataset by removing duplicates and linking comment-label pairs with relevant code segments for better input completeness. We further enhanced this version by adding manually curated repairs for smelly comments, resulting in a final dataset of 2,211 samples.

Both authors independently reviewed the code segments, comment segments, and corresponding labels before contributing to the repair process. To establish a consistent repair heuristic, the authors held collaborative meetings to determine appropriate repair strategies based on the underlying causes of each smell type, with guidance provided by the third author, who has over 15 years of programming experience. The final repair heuristic is detailed in Section III-B. Repaired comments were initially written by the first author, who has over five years of programming experience, and subsequently reviewed by the second author, who also has five years of experience. The original smell labels served as reference points for evaluating the quality of the repairs. In cases where the second author disagreed with the proposed fixes, disagreements were resolved by the third author. Standard inter-rater reliability (IRR) metrics such as Cohen's Kappa [25] and Krippendorff's Alpha [26] are not applicable, as comment repair is an open-ended generation task without a fixed label space.

Although the original taxonomy defines eleven smell types, our dataset includes only nine, as the *no comment on non-obvious code* and *attribution* categories are absent. It also contains a *not a smell* class to capture high-quality comments that provide context, clarify intent, warn about edge cases, or follow best commenting practices.

The label distribution is notably imbalanced, which may not accurately reflect the true performance of models on this task. Specifically, 56.41% of the samples are labeled as *not a smell*, followed by 30.62% *obvious*, 5.49% *task*, 2.27% *beautification*, 2.22% *vague*, 0.73% *misleading*, 0.32% *too much information*, 0.18% *irrelevant*, and 0.18% *non-local*. The extremely limited number of samples in certain classes can lead to artificially high or low performance metrics, making them unreliable indicators of a model's overall capability. This imbalance, particularly the scarcity of underrepresented classes, may limit the models' ability to generalize, as further discussed in Section IV and Section VI.

B. Repair Strategy

Repair strategies are designed for each smell type, as summarized in Figure 1:

- ***Remove**:* For *attribution*, *beautification*, *commented-out code*, *irrelevant*, *obvious*, and *task* smells, the expected action is removal. These comments do not contribute to explaining the code's logic or clarifying its functionality, and their elimination reduces redundancy in the codebase.
- ***Rewrite**:* For *misleading* and *vague*, the goal is to correct or clarify the original comment, resolving ambiguity or factual inaccuracy.
- ***Summarize**:* For *too much information*, verbose comments should be shortened to keep only essential content.
- ***Move**:* For *non-local* comments, the appropriate repair action is relocating the comment to a more contextually relevant part of the code. However, due to the limited context available in our dataset, this relocation is typically signaled rather than performed. In cases where relocation is not feasible, such comments are removed to avoid potential confusion or misalignment.
- ***Generate**:* Although part of the original taxonomy, the *no comment on non-obvious code* category is excluded from our study, as it does not appear in our dataset and is not applicable to the repair task. Since it concerns missing comments, it aligns more closely with comment generation and should be explored in a different scope.

C. Model Selection and Configuration

We selected four LLMs for our comment smell detection and repair experiments, each offering a distinct balance of reasoning ability, language understanding, and performance efficiency. From the range of available models, broadly classified as having high, medium, or low reasoning capabilities, we intentionally focus on those with medium reasoning capacity, as they strike a practical balance between performance and resource demands. The selected models are *GPT-4o-mini* (2024-12-01-preview) and *o3-mini* (2024-12-01-preview) from OpenAI, *Codestral-2501* (2024-05-01-preview) from Mistral, and *DeepSeek-V3* (2024-05-01-preview) from DeepSeek AI. Below, we provide the justification for each model:

- ***GPT-4o-mini (OpenAI):*** A lightweight yet capable variant of GPT-4, offering strong language understanding and fast inference. Well-suited for both comment classification and repair tasks at scale [27].
- ***o3-mini (OpenAI):*** An instruction-tuned compact model from OpenAI's internal "o" series, optimized for both classification and generative tasks like comment smell detection and repair [28].

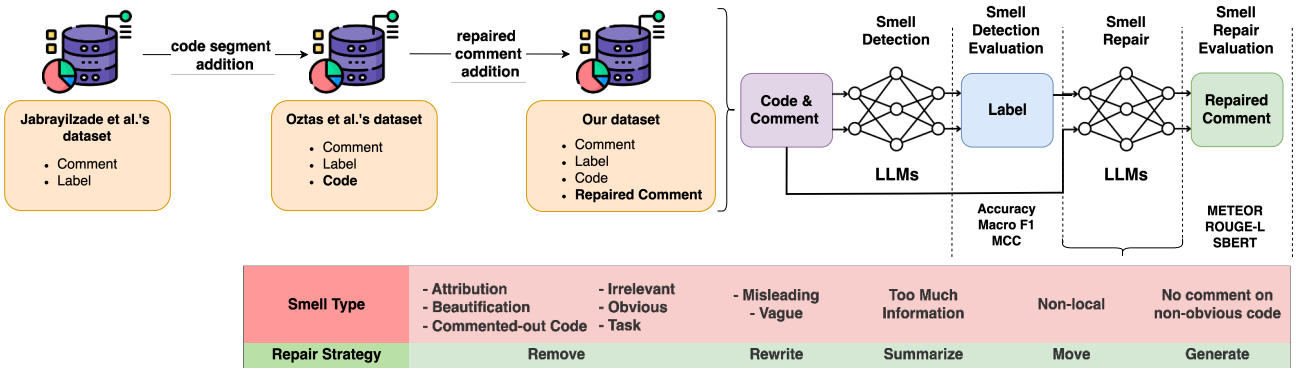

Fig. 1. Summary of our Methodology

- ***Codestral-2501 (Mistral AI):*** Selected for its strong code understanding and dual support for instruction and completion, making it well-suited for both detecting and repairing comment smells [29].
- ***DeepSeek-V3 (DeepSeek AI):*** A general-purpose, multilingual LLM with strong reasoning capabilities in both language and code. Its open-source nature and versatility make it suitable for scalable comment smell analysis [30].

To account for the inherent stochasticity of language model outputs, each model–prompt configuration was executed across five independent iterations during both the detection and repair phases. This multi-run strategy enhances the robustness of our evaluation by mitigating variance introduced through random sampling. Following OpenAI's [31], Mistral AI's [32] and DeepSeek's [33] guidance, and supported by our own empirical observations, we set the temperature to 0.1 for all models during the detection phase, except for *o3-mini*, which does not expose temperature as a configurable parameter. A low temperature setting is particularly effective in classification tasks, as it reduces hallucinations, label drift, and output variability, especially in few-shot settings where models are otherwise prone to overgeneration. Additionally, for models that support alternative sampling strategies, such as *Codestral-2501* and *DeepSeek-V3*, we explicitly set `top_p` to 1.0, enabling the models to consider the full probability distribution during token sampling. This aligns with recommendations by both OpenAI [31] and Mistral AI [32], which note that temperature and top-p control output randomness and should generally not be altered simultaneously.

In contrast, comment repair is inherently generative and non-deterministic, as multiple valid rewrites can convey the same intent. To accommodate this variability, we relaxed the decoding constraints during the repair phase by setting the temperature to 0.3, encouraging greater stylistic diversity while maintaining semantic alignment with the code and the predicted label. This configuration balances creativity and coherence, and is supported by both OpenAI's [31] and DeepSeek's [33] documentation, which recommend moderate temperatures (e.g., 0.2–0.7) for generation tasks where output variation is desirable.

D. Prompt Design for Comment Smell Detection and Repair

To evaluate the impact of prompt design on model performance, we constructed two prompt variants for each of the four LLMs tested: (1) zero-shot and (2) few-shot. This applied to both the smell detection and comment repair phases. Our prompt designs follow prompt engineering principles outlined by OpenAI [34] and the structured prompting heuristics proposed by Marvin et al. [35], which emphasize task clarity, input-output structure, and output constraints.

Smell Detection Prompts. The zero-shot prompt includes a brief task description, the full taxonomy of eleven smell types plus *not a smell*, and a code-comment pair formatted with consistent delimiters. The model is explicitly instructed to select one label from the predefined list, following best practices for clarity and constrained output. This design follows prompt engineering best practices outlined by Marvin et al. [35] and OpenAI [28], including clear instructions, constrained output space, and unambiguous formatting.

The few-shot prompt adds two examples per smell type, covering all nine categories in our dataset, including *not a smell*. The number of examples was capped due to token limitations, ensuring a balance between representativeness and prompt length. Examples were synthetically generated using *Claude Sonnet 3.7* (Anthropic) for its strong code understanding and consistent output quality [36], and were manually reviewed for alignment with the taxonomy presented in Table I. *Claude Sonnet 3.7* was excluded from evaluation to ensure fairness, and no dataset examples were used to prevent leakage. The prompt leverages structured exemplification to guide classification across smell types.

Repair Prompts. For the repair task, we used a two-tiered prompting strategy. The zero-shot prompt includes a brief task description, the smell taxonomy, the code-comment pair, and the predicted smell label from the detection step. The model is instructed to produce a semantically faithful but stylistically improved version of the comment.

The few-shot prompt extends this by adding two labeled examples for each repairable class—*misleading*, *too much information*, and *vague*—generated using *Claude Sonnet 3.7*. Each example consists of a code-comment pair, the corre-

sponding smell label, and the repaired comment. The number of examples was limited due to token constraints, balancing coverage and prompt efficiency. This setup exposes the model to label-aligned rewrites and leverages structured exemplification to enhance generalization.

In both tasks, we used consistent formatting, explicit role instructions, and label constraints to ensure reliable, reproducible behavior across runs. These prompt configurations were designed to measure the incremental value of guided reasoning and demonstration in improving LLM performance for both classification and generation tasks in the comment quality domain. The prompt templates are included in the replication package [1].

E. Evaluation Pipeline for Comment Smell Detection and Repair

For both detection and repair, each of the four LLMs was evaluated under both zero-shot and few-shot prompt configurations.

Detection Evaluation. During this phase, models were provided with the complete smell taxonomy shown in Table I and tasked with assigning one label per code-comment pair. The predictions were compared with the ground truth labels using accuracy, MCC, and macro F1 score. Accuracy offers a simple, interpretable baseline and is commonly used to benchmark classification tasks, but it can be misleading in imbalanced settings. Therefore, macro F1 was chosen as the primary metric, as it gives equal weight to all classes and better reflects performance in underrepresented categories [37]. MCC was added as a complementary measure due to its robustness in multi-class and imbalanced classification scenarios [38].

To ensure stability across stochastic LLM outputs, we report the median macro F1 score—the third value in sorted results—across five runs for each model–prompt configuration. This approach aligns with established practices in language model evaluation, where median reporting is used to mitigate the influence of outliers and better represent typical model performance [39].

Although the median was used for evaluation, the best-performing run (i.e., the highest macro F1 score) was used to generate inputs for the repair phase to ensure the highest possible label quality for downstream comment rewriting.

Repair Evaluation. The repair phase was guided by the predicted labels of the best-performing detection run. Before repair, we removed all comments labeled as *not a smell* from the dataset, as these comments are not considered defective and thus require no repair. However, since LLMs may produce false positives in this category, we computed and visualized a binary confusion matrix (smell vs. not a smell) to quantify detection noise and its propagation into the repair phase.

Among the remaining samples, we applied a filtering and simplification strategy aligned with our repair heuristic. Specifically, comments labeled as *beautification*, *commented-out code*, *irrelevant*, *task*, *obvious*, and *non-local* were excluded from generation-based evaluation and replaced with `null`. This decision reflects our repair policy: these categories are best addressed through removal—either because the comment is redundant (*obvious*, *beautification*), non-informative or unrelated (*task*, *irrelevant*, *commented-out code*), or refers to context not visible in the code snippet due to the scope of our dataset (*non-local*).

As a result, the actual comment generation task was limited to three smell types: *too much information*, *misleading*, and *vague*. The corresponding filtered samples were submitted to each LLM using their respective repair prompts (zero-shot and few-shot), and the generated outputs were compared against the repaired comments in the dataset, which were manually prepared by us as previously described.

To assess the quality of repaired comments, we employed three metrics: METEOR [40], ROUGE-L [41], and SBERT cosine similarity [42]. METEOR and ROUGE-L are widely adopted in natural language generation tasks due to their simplicity, interpretability, and strong historical use in benchmarking systems based on surface-level lexical overlap [40], [41]. However, their effectiveness diminishes in tasks involving paraphrasing or semantic preservation. Novikova et al. [43] highlight that METEOR and ROUGE-L often correlate poorly with human judgments, as they tend to favor lexical similarity over semantic adequacy. Similarly, Blagec et al. [44] note that such metrics may inadequately reflect model performance due to their reliance on surface-level features, overlooking deeper semantic fidelity. To address these limitations, we incorporated SBERT [42], which computes cosine similarity between sentence embeddings, providing a semantic-level assessment of similarity between the model's output and the reference. Because the repair task preserves meaning while allowing stylistic variation, SBERT was chosen as the primary metric.

Consistent with our approach in the detection phase, we reported the median (third-highest) SBERT score across five runs for each configuration to ensure robustness against stochastic variation and outliers.

IV. Results

To evaluate the capabilities of LLMs in detecting and repairing inline code comment smells, we tested four models—*GPT-4o-mini*, *o3-mini*, *DeepSeek-V3*, and *Codestral-2501*—across two prompt configurations: *zero-shot* and *few-shot*. Each model–prompt pair was executed over five independent iterations to account for stochasticity in generation. The detection task focused on classifying comments into one of nine predefined smell categories (or *not a smell*), while the repair task targeted three categories requiring actual rewriting: *misleading*, *too much information*, and *vague*. The following subsections address each research question in turn.

RQ1: To what extent can LLMs detect inline code comment smells under varying prompt input combinations?

Each model–prompt configuration was executed across five independent runs, and the median macro F1 score was used for evaluation to ensure robustness against stochastic variation. As shown in Table IV, *o3-mini* with few-shot prompting

[1] https://figshare.com/s/b71935b0c12dd3330251

achieved the highest median macro F1 score (0.41), outperforming larger models such as *GPT-4o-mini* (0.22, few-shot) and *DeepSeek-V3* (0.27, few-shot). Accuracy and MCC scores followed a similar trend, confirming the relative advantage of *o3-mini* in this task.

Table II presents the detailed class-wise performance of the best *o3-mini* run, which achieved an overall accuracy of 0.72 and macro F1 of 0.47. The model performed well on high-frequency classes such as *not a smell*, with an F1 score of 0.79, as well as *commented out code* and *beautification*, with F1 scores of 0.90 and 0.72, respectively. However, performance dropped significantly on underrepresented classes like *non-local*, *too much information*, and *vague*, which received F1 scores of 0.00, 0.25, and 0.23, respectively. These categories had very few examples in the dataset, hindering generalization. This class imbalance is the primary factor contributing to lower macro F1 scores across iterations. The confusion matrix is available in the replication package.[2]

This class imbalance severely impacts the macro F1 score, which assigns equal weight to each class regardless of its frequency. Consequently, poor performance on rare classes disproportionately lowers the overall macro F1, even when the model performs well on dominant categories. For instance, despite achieving high accuracy and strong weighted averages, the macro F1 remains modest due to consistently low recall and precision on minority classes. This highlights a key challenge in multi-class classification with imbalanced data and underscores the importance of either rebalancing datasets or incorporating targeted techniques such as class-specific prompting or synthetic data augmentation to improve generalization across low-frequency labels.

Importantly, our best-performing configuration still surpasses the GPT-4 detection results reported by Oztas et al. [17], which used a similar dataset and prompt-based methodology. This finding demonstrates that carefully tuned lightweight models like *o3-mini* can match or even outperform larger models in structured classification tasks.

To ensure high-quality inputs for the repair phase, we selected the best-performing *o3-mini* few-shot run as the source of predicted labels for downstream comment rewriting.

RQ2: To what extent can LLMs repair inline code comment smells under varying prompt input combinations?

The repair evaluation was conducted exclusively on comments labeled as *misleading*, *vague*, or *too much information*—the only categories designated for semantic rewriting in our repair taxonomy. Comments from other smell categories were excluded from LLM-based rewriting and replaced with `null`, following our removal policy. However, their treatment is still reflected in the overall class-wise performance of the best detection run, shown in Table II.

While the repair task focused only on these three rewrite-eligible classes, the evaluation was influenced by upstream detection errors in multiple ways. As shown in Table III, 370 smelly comments were misclassified as *not a smell*, and therefore omitted from repair. Simultaneously, 197 false positives led to unnecessary rewriting of comments that did not require any changes. Additionally, some comments were incorrectly assigned to the wrong repair category, for example, labeling a clear or irrelevant comment as misleading or vague. These intra-category misclassifications prompted semantically inappropriate rewrites, further degrading repair quality by confusing the rewriting objective and introducing hallucinations or distortions.

[2]https://figshare.com/s/b71935b0c12dd3330251

TABLE II
PERFORMANCE OF O3-MINI BEST OUTPUT

Class Label	Precision	Recall	F1-score	Support
Beautification	0.76	0.68	0.72	50
Commented out Code	0.91	0.89	0.90	35
Irrelevant	0.38	0.75	0.50	4
Misleading	0.44	0.44	0.44	16
Non-local	0.00	0.00	0.00	4
Not a Smell	0.74	0.84	0.79	1249
Obvious	0.69	0.55	0.61	675
Task	0.87	0.65	0.74	121
Too Much Information	1.0	0.14	0.25	7
Vague	0.21	0.26	0.23	50
Macro Avg	0.55	0.47	0.47	–
Weighted Avg	0.72	0.72	0.71	–
Overall Accuracy		**0.72**		2211

TABLE III
BINARY CONFUSION MATRIX OF THE BEST DETECTION OUTPUT

True \ Predicted	Not a Smell	Smell
Not a Smell	1052	197
Smell	370	592

Each model–prompt configuration was executed across five independent runs, and the median SBERT similarity score was used to evaluate repair quality. As shown in Table V, *Codestral-2501* with zero-shot prompting achieved the highest median score of 0.61. It was followed by *DeepSeek-V3* with 0.53 in the few-shot setting, *GPT-4o-mini* with 0.53 in the zero-shot setting, and *o3-mini* with 0.46 in the few-shot setting. While few-shot prompting led to modest improvements for some models, such as *DeepSeek-V3*, it performed worse for others, likely due to overfitting to example phrasing or introducing unintended stylistic drift.

This result underscores the open-ended nature of comment rewriting. Unlike classification, which benefits from label exemplars, rewriting requires balancing fidelity and conciseness, which can be disrupted by verbose or overly specific few-shot prompts. In contrast, well-scoped zero-shot prompts tended to produce more focused and directive rewrites.

Notably, model architecture and domain specialization played a significant role. *Codestral-2501*, despite lagging in detection performance, achieved the best repair results, suggesting that code-focused training enhances a model's ability to perform semantically faithful rewrites of technical content.

TABLE IV
DETECTION EVALUATION RESULTS ACROSS MODELS, PROMPTS, AND METRICS BY ITERATION (MEDIANS AND BEST OUTPUT HIGHLIGHTED)

Model Name	Metric	Iteration				
		#1	#2	#3	#4	#5
GPT-4o-mini (zero-shot)	Accuracy	0.53	0.53	**0.53**	0.54	0.53
	Macro F1	0.21	0.20	**0.20**	0.22	0.20
	MCC	0.25	0.25	**0.25**	0.26	0.25
GPT-4o-mini (few-shot)	Accuracy	0.48	0.49	**0.49**	0.49	0.49
	Macro F1	0.22	0.23	**0.22**	0.22	0.22
	MCC	0.23	0.24	**0.24**	0.24	0.24
o3-mini (zero-shot)	Accuracy	0.73	**0.72**	0.71	0.71	0.72
	Macro F1	0.34	**0.32**	0.31	0.28	0.35
	MCC	0.52	**0.52**	0.50	0.49	0.51
o3-mini (few-shot)	Accuracy	**0.72**	0.71	0.72	**0.71**	0.71
	Macro F1	**0.47**	0.38	0.39	**0.41**	0.42
	MCC	**0.50**	0.50	0.50	**0.49**	0.49
DeepSeek-V3 (zero-shot)	Accuracy	0.60	0.60	**0.60**	0.60	0.60
	Macro F1	0.21	0.19	**0.18**	0.18	0.18
	MCC	0.30	0.31	**0.32**	0.32	0.32
DeepSeek-V3 (few-shot)	Accuracy	0.60	**0.61**	0.60	0.61	0.61
	Macro F1	0.27	**0.27**	0.25	0.27	0.27
	MCC	0.34	**0.34**	0.34	0.34	0.35
Codestral-2501 (zero-shot)	Accuracy	0.57	**0.60**	0.60	0.60	0.60
	Macro F1	0.16	**0.16**	0.17	0.16	0.17
	MCC	0.21	**0.26**	0.26	0.26	0.26
Codestral-2501 (few-shot)	Accuracy	0.59	0.59	**0.59**	0.59	0.59
	Macro F1	0.15	0.17	**0.18**	0.18	0.18
	MCC	0.29	0.29	**0.29**	0.29	0.29

TABLE V
REPAIR EVALUATION RESULTS ACROSS MODELS, PROMPTS, AND METRICS BY ITERATION (MEDIANS HIGHLIGHTED)

Model Name	Metric	Iteration				
		#1	#2	#3	#4	#5
GPT-4o-mini (zero-shot)	METEOR	0.35	0.38	**0.35**	0.36	0.35
	ROUGE-L	0.35	0.38	**0.35**	0.35	0.36
	SBERT	0.51	0.53	**0.52**	0.51	0.53
GPT-4o-mini (few-shot)	METEOR	0.29	0.28	**0.28**	0.27	0.26
	ROUGE-L	0.30	0.30	**0.30**	0.30	0.29
	SBERT	0.49	0.49	**0.49**	0.49	0.48
o3-mini (zero-shot)	METEOR	0.21	0.24	0.26	**0.24**	0.21
	ROUGE-L	0.23	0.24	0.26	**0.24**	0.23
	SBERT	0.45	0.47	0.48	**0.46**	0.45
o3-mini (few-shot)	METEOR	0.22	0.23	0.21	0.20	**0.22**
	ROUGE-L	0.24	0.25	0.23	0.21	**0.24**
	SBERT	0.45	0.44	0.46	0.43	**0.44**
DeepSeek-V3 (zero-shot)	METEOR	0.30	0.36	0.33	0.33	**0.33**
	ROUGE-L	0.32	0.35	0.32	0.31	**0.33**
	SBERT	0.52	0.54	0.53	0.52	**0.52**
DeepSeek-V3 (few-shot)	METEOR	0.35	0.32	**0.32**	0.31	0.31
	ROUGE-L	0.35	0.34	**0.33**	0.32	0.33
	SBERT	0.53	0.54	**0.53**	0.52	0.52
Codestral-2501 (zero-shot)	METEOR	0.39	**0.39**	0.40	0.37	0.39
	ROUGE-L	0.40	**0.40**	0.41	0.39	0.41
	SBERT	0.60	**0.60**	0.61	0.59	0.61
Codestral-2501 (few-shot)	METEOR	0.27	0.26	0.25	**0.25**	0.25
	ROUGE-L	0.33	0.30	0.31	**0.31**	0.32
	SBERT	0.52	0.50	0.49	**0.51**	0.51

V. DISCUSSION

A. Implications for Practitioners

Toward IDE-Integrated Smell Repair Tools: LLMs like *o3-mini* demonstrate strong potential for powering practical tools that improve code comment quality. Building on this insight, we initiated the development of a tool that triggers upon GitHub pull request merges, with evaluation planned in Azure DevOps pipelines. This approach can be extended by the community to support real-time comment suggestions directly within GitHub's editor or IDEs, assisting developers during code authoring.

Dataset Quality vs. Tool Performance: It is important to note that model performance on our dataset does not directly reflect a deployed tool's real-world effectiveness. Despite careful curation, the dataset is limited to two programming languages—Java and Python—exhibits significant label imbalance, and depends on human-annotated labels. As a result, any industrial deployment should ideally be complemented by a user study to evaluate real-world efficacy, usability, and generalizability across diverse development environments.

Smelly Comments vs. Smelly Code: Towards Holistic Repair Tools: In some cases, poor comments are symptoms of deeper code issues. For instance, a vague or misleading comment may result from poorly named variables or functions. While our framework treats comment repair in isolation, a more holistic approach would also include code refactoring suggestions. Future tools could include an option to identify whether a fix should target the comment, the code, or both.

Single-Pass vs. Double-Pass Repair: Fixing certain smells—such as *misleading* or *non-local*—can sometimes result in new issues, like introducing *obvious* comments. For example, correcting an inaccurate comment or relocating a comment could make its content self-evident in the new context. While a second repair pass could catch and remove these newly introduced smells, we opted for a single-pass design to maintain efficiency. Future tools could include an optional second pass, giving users the flexibility to balance quality improvements against additional computational cost.

B. Implications for Researchers

Prompt Sensitivity and System Constraints: Our findings highlight that LLMs are highly sensitive to prompt structure. Overly constrained system messages can interfere with learning, and token limitations restrict the number of examples that can be included in the few-shot prompts, limiting generalization and reducing performance. Therefore, prompt tuning is critical for effective comment classification and repair.

Generalizability of the Taxonomy: Some of the smells in the taxonomy, such as *attribution* or *task*, may be handled differently depending on team conventions. Moreover, some comments arguably fall into multiple categories (e.g.,

a comment could be both *vague* and *too much information*). Our single-label classification approach simplifies evaluation but may not fully capture this complexity. Future work could explore multi-label or hierarchical taxonomies to reflect real-world usage better.

Temperature and Decoding Trade-offs: For detection, we fixed the temperature at 0.1 to reduce randomness and ensure classification consistency. For repair, a moderate temperature of 0.3 yielded more diverse yet semantically faithful rewrites. This suggests that different sub-tasks benefit from different levels of model creativity. Researchers should carefully calibrate decoding parameters for generative NLP tasks that aim to preserve semantic intent.

Evaluation Challenges: Our decision to report median results (3rd-best of five runs) helps mitigate stochastic noise without cherry-picking outliers. However, selecting the best run for repair input creates a mismatch between the evaluation strategy and the pipeline setup. An alternative would be to perform a double evaluation: one based on the best run and another using the median to assess robustness.

Training Data Limitations and Bias: Our dataset, though extended, remains limited in size and balance. Rare categories like *irrelevant* or *too much information* have so few examples that even strong models cannot generalize well. Moreover, manual labels and repairs reflect human judgment and annotation bias. Some inconsistencies in the dataset (e.g., borderline cases labeled differently) may have also impacted model learning and evaluation.

Suggestions for Future Methodology: We generated few-shot examples using *Claude Sonnet 3.7* to ensure consistency and neutrality; however, future work could explore exemplar selection strategies that adapt to model behavior or dataset-specific patterns. We also suggest enhancing few-shot prompting with BM25 retrieval to dynamically select semantically relevant exemplars [45], which may improve model performance. Additionally, incorporating chain-of-thought reasoning [46] could support more coherent comment rewrites. Finally, developing a balanced, multilingual benchmark dataset—curated by a diverse team of experienced practitioners—would enable fairer, more comprehensive, and generalizable evaluations for comment smell detection and repair tasks.

VI. Threats to Validity

A. Internal Validity

Our study adopts the taxonomy of inline code comment smells introduced by Jabrayilzade et al. [19]. Like any manually defined classification scheme, this taxonomy is based on certain assumptions that influence the detection and repair processes. Notably, it presumes that each comment corresponds to a single smell category, whereas in practice, comments may exhibit multiple overlapping issues. While this simplification aligns with the dataset's labeling protocol, it may not fully reflect the complexity of real-world commenting behavior.

The dataset was refined by three individuals, with disagreements during the repair annotation phase resolved by the third author. Despite this effort to ensure consistency, the potential for human bias and limited perspective remains. Furthermore, Oztas et al. [17] manually linked each comment to a relevant code segment to provide LLMs with sufficient context for detection. Although this step aimed to improve model accuracy, it introduces a risk of inconsistency due to the inherently subjective nature of code-comment pairing.

Another source of internal threat stems from the distribution of smell categories within the dataset. The dataset is highly imbalanced, over half of the samples are labeled as *not a smell*, while some categories, such as *beautification* (2.2%), are severely underrepresented. This imbalance limits the generalizability of model performance, particularly in multi-class classification settings. To mitigate this, we used macro F1 as the primary evaluation metric, as it gives equal weight to all classes regardless of their frequency. Despite this adjustment, models consistently struggled with low-frequency categories, and in some cases, failed to identify certain underrepresented smells altogether, for example, *non-local* was entirely overlooked.

We also acknowledge the inherent limitations of LLMs. Despite careful prompt engineering and parameter tuning, models may exhibit an overreliance on surface-level patterns, limited domain-specific reasoning, and a lack of interpretability [47]. These challenges can lead to subtle misclassifications or suboptimal repairs, especially in edge cases that are underrepresented in the dataset. To mitigate individual model weaknesses and improve overall robustness, we employed a diverse set of LLMs, leveraging their complementary strengths, and two prompt strategies were used for better generalization.

Few-shot examples were generated using *Claude Sonnet 3.7* for consistency and neutrality. However, relying on a single model may introduce stylistic biases and limit generalizability, particularly for models with different reasoning patterns. Additionally, synthetic examples may not fully reflect real-world variability.

B. External Validity

The external validity of our findings is constrained by the characteristics of the dataset and the models used. Limited to eight open-source projects written in Java and Python, the dataset may not reflect the full diversity of programming languages, commenting styles, or development contexts. As such, our findings may not generalize to other ecosystems (e.g., JavaScript, C++, Rust) or industrial-scale codebases.

Moreover, the taxonomy we adopted—while comprehensive within the context of the original study—may not fully capture all relevant types of comment smells across projects, teams, or domains. Commenting practices can vary significantly based on coding conventions, project maturity, or organizational standards, which limits the universal applicability of our classification approach.

VII. Conclusion and Future Work

This paper presents a comprehensive evaluation of LLMs for the dual tasks of detecting and repairing inline code

comment smells. We assess four models—*GPT-4o-mini*, *o3-mini*, *DeepSeek-V3*, and *Codestral-2501*—across zero-shot and few-shot prompts, analyzing their effectiveness over five iterations per configuration. Using GPT-4 as a baseline [17], our results show that lightweight models can match or even outperform it in detection accuracy. Few-shot prompting consistently improves detection performance, while concise zero-shot prompts are more effective for repair, as they help the model produce clearer and more focused revisions without unnecessarily changing the tone or style of the original comment.

We also find that LLM performance is highly sensitive to prompt structure and dataset composition. Repair introduces unique challenges, including the risk of overfitting in example-heavy prompts and the possibility of generating new smells.

To support practical adoption, we are developing a GitHub-integrated comment repair tool, which provides automated suggestions upon pull request merges and will be evaluated in Azure DevOps pipelines.

In summary, our contributions are:

- A dual-focus study on detection and repair of inline code comment smells using LLMs, covering a diverse taxonomy beyond prior work.
- An enhanced dataset from the previous work [19], including 2,211 labeled and repaired inline comments.
- A systematic evaluation of four LLMs with two prompting strategies across multiple runs, analyzing robustness and generalization.
- Empirical evidence that our best configuration surpasses GPT-4's detection performance as reported in a prior work [17].

VIII. Acknowledgements

We acknowledge the generous support of Microsoft Azure cloud credits, which played a vital role in enabling and accelerating our experiments with large language models.

References

[1] V. Misra, J. S. K. Reddy, and S. Chimalakonda, "Is there a correlation between code comments and issues? an exploratory study," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 110–117. [Online]. Available: https://doi.org/10.1145/3341105.3374009

[2] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, p. 215–223.

[3] D. Spinellis, "Code documentation," *IEEE Software*, vol. 27, no. 4, pp. 18–19, 2010.

[4] C. S. Hartzman and C. F. Austin, "Maintenance productivity: observations based on an experience in a large system environment," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, ser. CASCON '93. IBM Press, 1993, p. 138–170.

[5] Z. Yang, J. W. Keung, X. Yu, Y. Xiao, Z. Jin, and J. Zhang, "On the significance of category prediction for code-comment synchronization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3534117

[6] Y. Huang, H. Guo, X. Ding, J. Shu, X. Chen, X. Luo, Z. Zheng, and X. Zhou, "A comparative study on method comment and inline comment," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3582570

[7] M. D. Igbomezie, P. T. Nguyen, and D. Di Ruscio, "When simplicity meets effectiveness: Detecting code comments coherence with word embeddings and lstm," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 411–416. [Online]. Available: https://doi.org/10.1145/3661167.3661187

[8] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 53–64.

[9] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 145–158. [Online]. Available: https://doi.org/10.1145/1294261.1294276

[10] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, pp. 427–435, May 2021. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16119

[11] P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, and O. Nierstrasz, "A decade of code comment quality assessment: A systematic literature review," *Journal of Systems and Software*, vol. 195, p. 111515, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222001911

[12] B. Lin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "Automated comment update: How far are we?" in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 36–46.

[13] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 585–597. [Online]. Available: https://doi.org/10.1145/3324884.3416581

[14] F. Rabbi and M. S. Siddik, "Detecting code comment inconsistency using siamese recurrent network," in *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, 2020, pp. 371–375.

[15] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.

[16] C. Wang, H. He, U. Pal, D. Marinov, and M. Zhou, "Suboptimal comments in java projects: From independent comment changes to commenting practices," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3546949

[17] İpek Öztaş, U. B. Torun, and E. Tüzün, "Towards automated detection of inline code comment smells," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025)*, 2025. [Online]. Available: https://arxiv.org/abs/2504.18956

[18] E. Jabrayilzade, O. Gürkan, and E. Tüzün, "Towards a taxonomy of inline code comment smells," in *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2021. [Online]. Available: https://doi.org/10.1109/SCAM52516.2021.00024

[19] E. Jabrayilzade, A. Yurtoğlu, and E. Tüzün, "Taxonomy of inline code comment smells," *Empirical Software Engineering*, vol. 29, no. 58, Apr 2024.

[20] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li, "Just-in-time obsolete comment detection and update," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 1–23, 2023.

[21] A. Dau, J. L. Guo, and N. Bui, "DocChecker: Bootstrapping code large language model for detecting and resolving code-comment inconsistencies," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, N. Aletras and O. De Clercq, Eds. St. Julians, Malta: Association for Computational Linguistics, Mar. 2024, pp. 187–194. [Online]. Available: https://aclanthology.org/2024.eacl-demo.20

[22] G. Rong, Y. Yu, S. Liu, X. Tan, T. Zhang, H. Shen, and J. Hu, " Code Comment Inconsistency Detection and Rectification Using a Large Language Model ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 432–443. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00035

[23] A. Blasi, N. Stulova, A. Gorla, and O. Nierstrasz, "Replicomment: Identifying clones in code comments," *Journal of Systems and Software*, vol. 182, p. 111069, Dec 2021.

[24] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[25] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, p. 37–46, Apr 1960.

[26] K. Krippendorff, *Computing Krippendorff's Alpha-Reliability*, Jan 2011. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=de8e2c7b7992028cf035f8d907635de871ed627d

[27] OpenAI, "GPT-4o mini: Advancing Cost-Efficient Intelligence," https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, 2024, accessed: 2025-05-30.

[28] OpenAI, "OpenAI Platform Documentation," https://platform.openai.com/docs/guides/gpt, 2024, accessed: 2025-05-19.

[29] Mistral AI, "Introducing Codestral," https://mistral.ai/news/codestral/, 2024, accessed: 2025-05-19.

[30] DeepSeek AI, "DeepSeek-V3 Release Blog," https://deepseek.com/blog/deepseek-v3.html, 2024, accessed: 2025-05-19.

[31] OpenAI, "OpenAI API Reference," https://platform.openai.com/docs/api-reference, 2024, accessed: 2025-05-19.

[32] M. AI, "Api reference - mistral ai," 2025, accessed: May 30, 2025. [Online]. Available: https://docs.mistral.ai/api/#tag/chat

[33] DeepSeek, "Api reference – deepseek api," 2025, accessed: May 30, 2025. [Online]. Available: https://deep-seek.chat/docs/api/

[34] OpenAI, "Text prompt design best practices," https://platform.openai.com/docs/guides/text, 2023, accessed: 2025-05-11.

[35] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *Data Intelligence and Cognitive Informatics*, I. J. Jacob, S. Piramuthu, and P. Falkowski-Gilski, Eds. Singapore: Springer Nature Singapore, 2024, pp. 387–402.

[36] Anthropic. (2025) Claude 3.7 sonnet and claude code. Accessed: 2025-05-30. [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet

[37] Y. Yang and X. Liu, "Re-examining the text categorization methods: A comparison of term-based and phrase-based approaches," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1999, pp. 42–49.

[38] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, pp. 1–13, 2020.

[39] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. Cosgrove, C. D. Manning, C. Ré, D. Acosta-Navas, D. A. Hudson, E. Zelikman, E. Durmus, F. Ladhak, F. Rong, H. Ren, H. Yao, J. Wang, K. Santhanam, L. Orr, L. Zheng, M. Yuksekgonul, M. Suzgun, N. Kim, N. Guha, N. Chatterji, O. Khattab, P. Henderson, Q. Huang, R. Chi, S. M. Xie, S. Santurkar, S. Ganguli, T. Hashimoto, T. Icard, T. Zhang, V. Chaudhary, W. Wang, X. Li, Y. Mai, Y. Zhang, and Y. Koreeda, "Holistic evaluation of language models," 2023. [Online]. Available: https://arxiv.org/abs/2211.09110

[40] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, 2005, pp. 65–72.

[41] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," Computational Linguistics Laboratory, University of Southern California, Technical Report, 2004.

[42] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2019, pp. 3982–3992. [Online]. Available: https://aclanthology.org/D19-1410

[43] J. Novikova, O. Dušek, and V. Rieser, "Why we need new evaluation metrics for nlg," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 2241–2252.

[44] M. Blagec, M. Sedlak, and J. Šnajder, "To overlap or not to overlap? empirical study of lexical overlap measures for evaluation of generated text," *arXiv preprint arXiv:2204.11574*, 2022.

[45] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2450–2462.

[46] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[47] S. Lappin, "Assessing the strengths and weaknesses of large language models," *Journal of Logic, Language and Information*, vol. 33, pp. 9–20, 2024.