BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING

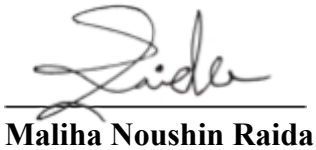
Automated Code Review in the Age of LLMs: A Comparative Analysis

Maliha Zaman
200042114
Mohammad Ittehad Rahman Sami 200042131
Tamim Afnan 200042148

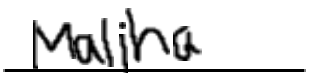
Department of Computer Science and Engineering
Islamic University of Technology
30 September, 2025

Declaration of Candidate

This is to certify that the work presented in this thesis is the outcome of the analy- sis and experiments carried out by **Maliha Zaman**, **Mohammad Ittehad Rahman Sami**, and **Tamim Afnan** under the supervision of **Maliha Noushin Raida**, Lec- turer, Department of Computer Science and Engineering, Islamic University of Tech- nology, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.



Maliha Noushin Raida
Lecturer
Department of Computer Science and Engineering Islamic University of Technology (IUT)
Date: 30 September , 2025



Maliha Zaman
Student ID: 200042114
Date: 30 September , 2025

Mohammad Ittehad Rah- man Sami
Student ID: 200042131
Date: 30 September , 2025

Tamim Afnan
Student ID: 200042148
Date: 30 September , 2025

Contents

List of Figures

List of Tables

Acknowledgement

We would like to express our gratitude towards IUT authority for providing assistance required to carry out our proposed research. We are indebted to our supervisor, Mal- iha Noushin Raida for providing us with insightful knowledge and guiding us at every stage of our journey. Finally, we would like to express our heartiest appreciation to- wards our family members for their continuous support, motivation, suggestions, and help, without which we could not have achieved the scale of implementation that we have achieved.

Abstract

Code review is a critical but labor-intensive process in software development, often leading to bottlenecks and inconsistencies when performed manually. While Large Language Models (LLMs) offer a promising avenue for automation, existing approaches are limited by unreliable evaluation metrics and low-quality datasets, failing to ade- quately measure the semantic alignment of AI-generated feedback with human rea- soning.

Our research presents a comprehensive comparative analysis of LLMs for automated code review. We evaluate five state-of-the-art models GPT-4, LLaMA 3.1, CodeL-LaMA, Qwen 2.5, and Mistral using a rigorous methodology that employs consistent prompting strategies (zero-shot, one-shot, few-shot) and multiple similarity metrics, including SBERT for semantic evaluation and BLEU for surface-level comparison. A key contribution of this work is the creation of a novel, high-quality benchmark dataset, curated from open-source repositories and validated by industry experts, to address the shortcomings of existing public datasets.

Our results demonstrate that few-shot prompting consistently yields the highest per-formance across all models, with GPT-4 showing the most significant absolute im-provement. Furthermore, evaluation on our new benchmark revealed a substantial increase in SBERT scores for all models with GPT-4's similarity to human reviews nearly doubling confirming that dataset quality is a pivotal factor in accurately as-sessing LLM capability. This study establishes a more reliable foundation for future research and demonstrates the significant potential of LLMs to generate human-like, context-aware code reviews when properly benchmarked.

Chapter 1

Introduction

Code review is a systematic process in software engineering where developers examine source code to detect bugs, improve code quality, and ensure adherence to coding standards before integration [40]. This foundational practice, originally formalized through Fagan inspections in the 1970s, has evolved into modern collaborative re- view processes essential for maintaining software quality across diverse development environments [3], [10].

Manual code review, while effective, is labor-intensive and time-consuming, often leading to skipped or superficial reviews as projects scale. Industrial studies at companies like Google demonstrate that while modern code review practices are widely adopted, they consume significant developer resources and create workfiow bottlenecks that impact delivery schedules [31].

This motivates the need for automated code review tools, which aim to reduce man- ual workload and provide consistent, actionable feedback. Early automation efforts using static analysis and rule-based systems lacked contextual understanding, while deep learning models improved performance but struggled with generalization and complex code changes [4]. Recent advances in static analysis integration demon- strate promising results when combining traditional tools with machine learning ap- proaches for comprehensive code quality assessment [16], [41].

The advent of Large Language Models (LLMs) has opened new avenues for automation, enabling models to generate review comments, detect code smells, and suggest improvements with greater contextual awareness [29], [43]. Contemporary research highlights both the potential and challenges of LLM-based code review automation. Multi-agent frameworks incorporating specialized reviewer, bug-detection, and optimization agents show promise for comprehensive code analysis [21], while indus-

trial deployments reveal mixed results regarding review quality and developer acceptance [8].

Our research represents a comprehensive comparative analysis to address these limitations. We systematically evaluate the performance of leading LLMs including GPT- 4, LLaMA, CodeLLaMA, Qwen, and Mistral in generating meaningful code review comments. Our methodology involves a consistent prompting strategy across mod- els and a novel evaluation framework using semantic similarity metrics like SBERT, alongside traditional metrics like BLEU. Furthermore, recognizing the critical role of data quality, we construct and leverage a new, high-quality benchmark dataset of code-review pairs, curated with input from industry experts, to enable a more reliable and accurate assessment.

Through this work, we aim to determine not only which LLMs are most effective at mimicking human-like code review but also how the quality of the underlying dataset and the choice of prompting strategy infiuence their performance. Our findings provide a foundational benchmark for future research and a significant step towards practical, intelligent, and human-aligned automated code review systems.

1.1 Background and Context

Historical evolution from formal Fagan inspections to modern collaborative review processes demonstrates the persistent importance of peer code examination in maintaining software quality standards. Traditionally, this process is manual, requiring significant developer effort to scrutinize code changes, identify bugs, and ensure adherence to coding standards.

As projects scale, manual reviews become increasingly burdensome, leading to de- lays, skipped reviews, and potential quality risks [37]. The integration of automated tools into review workfiows represents a natural evolution, with early static analy- sis approaches providing foundational capabilities for detecting syntactic issues and coding standard violations. However, these rule-based systems often generate high false positive rates and fail to address semantic or architectural concerns that human reviewers routinely identify [23], [30].

1.2 Motivation and Scope

The high cost and tedium of manual code review motivate the need for automation. Recent advances in Large Language Models (LLMs) and deep learning offer new op-

portunities to automate or augment code review, aiming to reduce manual workload, accelerate feedback, and improve consistency [9]. Contemporary research demonstrates that LLM-based systems can achieve substantial improvements in bug detec- tion and code quality assessment, with some frameworks achieving F1-scores up to 90% in specialized domains.

However, the practical effectiveness, limitations, and integration of LLM-based code review tools remain open research questions, especially regarding their ability to generate high-quality, actionable feedback that aligns with human reviewers [7]. Cross- task knowledge distillation and multi-task federated learning approaches show promise for improving model performance across diverse review scenarios [6], [18]. Addi- tionally, advances in semantic similarity evaluation and human-like code quality as- sessment provide new methodological foundations for rigorous performance evalua- tion [15], [42].

1.3 Problem Statement

Despite progress, existing automated code review approaches face key limitations:

- Early deep learning models achieved low prediction accuracy (16–31%), struggled with complex changes, and relied on code abstraction, limiting their practical usability and generalization.
- Even with pre-trained models like T5, challenges persist in handling complex transformations, generating high-quality review comments, and evaluating semantic alignment with human feedback.
- Most prior work evaluates models using surface-level metrics (e.g., BLEU), lacking standardized semantic-level evaluation and robust benchmarks [12].
- Industrial deployments reveal mixed results: while LLM-based tools can improve bug detection and code quality awareness, they may also increase review times and sometimes generate irrelevant feedback [26].
- Comprehensive benchmarking frameworks specifically designed for code review evaluation remain limited, with existing datasets often suffering from small size, noisy annotations and evaluation methods that do not fully capture the practical utility of review comments.

1.4 Research Challenges

Key challenges in advancing LLM-based automated code review include:

- **Data Quality and Scale:** Curating large, high-quality, and diverse datasets with clear annotations [13].
- **Generalization and Complexity:** Ensuring models generalize across languages, domains, and complex code changes while maintaining accuracy in specialized contexts [34].
- **Evaluation Metrics:** Developing robust, semantic-level metrics (e.g., SBERT, BLEU, Cosine Similarity) for comparing AI-generated and human comments.
- **Practical Integration:** Assessing real-world impact on developer workfiows and code quality while addressing concerns about false positives and tool adop- tion [2].
- **Bias and Reproducibility:** Addressing dataset biases and ensuring reproducibil- ity across settings while maintaining model transparency [24].
- **Security and Vulnerability Detection:** Developing specialized capabilities for identifying security issues and ensuring safe code practices [32].

1.5 Contributions

This research makes the following contributions:

- **Comprehensive Benchmarking:** Quantitatively and qualitatively compares leading LLMs (GPT-4, Llama, Mistral, CodeLlama, Qwen2.5, Phi4) in generating code review comments using both traditional and advanced semantic similarity metrics [19].
- **Dataset Innovation:** Target to create a cleaner standardized and also domain-tagged benchmark dataset for analysis (e.g., refactoring, formatting, debugging).
- **Real-World Validation:** Benchmarking on real-world open-source code re- view datasets to simulate practical review scenarios while addressing limitations of existing evaluation approaches.
- **Methodological Advancement:** Proposes a multi-phase analysis of dataset combining data processing, LLM-driven comment generation, and semantic similarity analysis with enhanced evaluation frameworks [27].

Chapter 2 Related Works

As software projects grow, the manual nature of code review becomes a bottleneck, leading to significant research into automating parts of this process. Below, we discuss the main research directions and findings in automated code review, focusing on both classical and recent approaches involving deep learning and large language models (LLMs).

2.1 From Manual to Automated Code Review

Manual code review, whether in open source or industry, has consistently been shown to reduce post-release defects and improve maintainability. However, these benefits come at a significant cost: developers spend substantial time reviewing code, and the process is prone to subjectivity, inconsistency, and bottlenecks as projects scale [37].

To address these challenges, the 2000s saw the rise of static analysis tools (e.g., FindBugs, PMD, Pylint) and linters that could automatically check for syntax errors, code style violations, and certain bug patterns. These tools improved efficiency for surface-level issues, but struggled with deeper semantic bugs, context-dependent logic, and architectural concerns. Their rule-based nature also resulted in high false positive rates and limited adaptability to new languages or frameworks [25], [45].

The next leap came with the application of deep learning to code review. Tufano et al. [40] introduced transformer-based neural models to partially automate the review process. Their approach featured two models: a contributor-side model that predicts likely code changes before submission, and a reviewer-side model that implements changes based on natural language review comments. While these models demonstrated the feasibility of learning from historical code review data, their success rates

were modest (16% for contributor-side, 31% for reviewer-side), and they were limited to Java and relatively simple code transformations due to reliance on code abstraction.

Building on these foundations, Tufano et al. [39] leveraged pre-trained Text-to-Text Transfer Transformer (T5) models, trained on large, raw code and technical English datasets. This allowed the models to operate directly on source code without abstraction, handle longer and more complex code changes, and support multiple code re- view tasks (e.g., code-to-code, code-and-comment-to-code, code-to-comment). The T5-based models outperformed earlier deep learning approaches, especially in gener- ating revised code and handling diverse review tasks [5]. However, they required sig- nificant computational resources and still struggled with context-dependent or highly complex changes.

A major innovation came with the use of code review comments as direct guidance for automated program repair. The Review4Repair approach [14] trained sequence-to-sequence models on over 55,000 code review and fix pairs, enabling the system to localize and address bugs, refactoring needs, stylistic issues, and even non-code concerns based on reviewer expertise. This approach reduced dependence on traditional bug localizers and improved fix accuracy by up to 34.8%. However, its effectiveness still depends on the availability and clarity of review comments, and it faces challenges with architectural or highly complex fixes.

The advent of Large Language Models (LLMs) such as GPT-4, CodeLlama, and domain-specific agents has revolutionized automated code review [9]. LLM-based systems, often organized as multi-agent frameworks, can analyze code for bugs, code smells, performance issues, and provide context-aware suggestions [35]. Unlike static analy- sis tools, LLMs can reason about code semantics, predict future risks, and even gener- ate documentation or refactoring advice. Recent tools like code2Prompt have shown F1-scores up to 90% in data science workfiows, outperforming traditional tools like Copilot or DeepCode. However, LLMs can hallucinate, produce irrelevant sugges- tions, and are sensitive to the quality and diversity of their training data [20].

A persistent limitation in both traditional and LLM-based approaches is the reliance on surface-level evaluation metrics, such as BLEU or token overlap, which often fail to capture true semantic alignment with human feedback. Newer frameworks like HuCoSC employ recursive LLM-based semantic analysis, achieving much higher correlation with expert human judgments, but these methods are still emerging and not yet standard in practice.

Despite these advances, most automated code review tools treat code review as a

monolithic task, overlooking the diversity of review subdomains such as refactoring, debugging, formatting, and performance optimization [11]. Studies show that up to 78% of tools do not differentiate between these subdomains, leading to generic sugges- tions that may miss domain-specific issues. There is a critical need for domain-tagged datasets and subdomain-aware evaluation to enable more granular, actionable, and trustworthy automated reviews [33].

In summary, the evolution from manual to automated code review has progressed from static rules to deep learning and now to powerful LLM-driven systems. While LLMs offer unprecedented capabilities for context-aware, semantic code analysis, sig- nificant challenges remain in subdomain-specific benchmarking, evaluation metrics, and ensuring that automated feedback aligns with the nuanced needs of real-world software development [36].

2.2 Deep Learning and Transformer-based Models

Tufano et al. [40] pioneered deep learning for code review automation, introducing two transformer-based models: one to recommend code changes likely to be suggested by reviewers, and another to implement code changes based on natural language com- ments. These models, trained on abstracted Java code, achieved modest success rates (16% for contributor-side, 31% for reviewer-side) and were limited by dataset size, code abstraction, and language scope.

Manual code review is time-consuming and costly, creating a bottleneck in modern software development. The paper aims to reduce this manual burden by partially automating code review activities using deep learning models trained on past code review data.

The authors introduce two transformer-based models:

- **Contributor-Side Model:** Predicts likely code changes before submission, learn- ing from code changes performed during real code reviews.
- **Reviewer-Side Model:** Implements code changes based on natural language review comments, providing concrete suggestions to contributors [28].

Both models are trained on large datasets mined from Java projects on GitHub and Gerrit, using code abstraction to reduce vocabulary size and improve learning.

Workflow:

1. Mining code review data and extracting method-level code changes and associated reviewer comments.
2. Abstracting code and comments to limit vocabulary.
3. Building datasets of code pairs and triplets for training 1-encoder and 2-encoder transformer models.
4. Training and evaluating the models on their ability to replicate code transformations and implement reviewer suggestions.

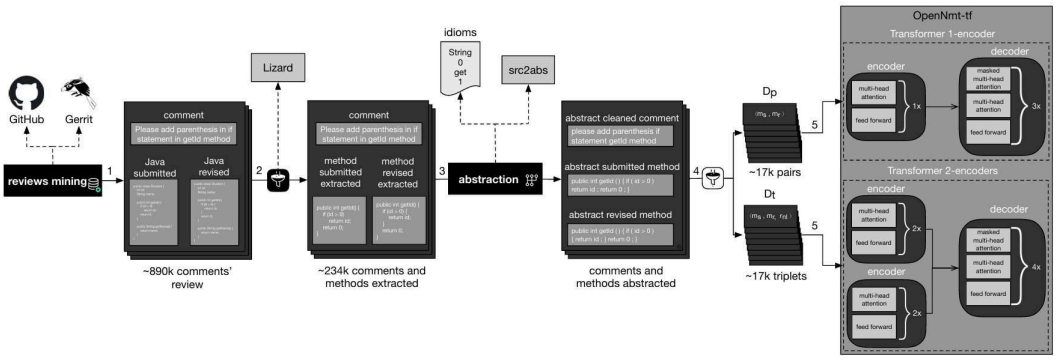


Figure 2.1: Approach overview: Dual-model architecture for code review automa- tion.

Achievements:

- Demonstrated that deep learning models can automate some code review tasks, providing meaningful recommendations in up to 16% (contributor-side) and 31% (reviewer-side) of cases.
- Showed the feasibility of using neural models to learn from historical code review data [1].

Limitations:

- **Low Success Rate:** Models only succeed in a minority of cases, limiting practical usability.
- **Lack of Generalization:** Models trained only on Java, with limited applicabil- ity to other languages.
- **Code Abstraction:** The abstraction process prevents the models from handling changes that introduce new identifiers or literals.

- **Simple Changes:** Most successful predictions are for relatively simple code transformations.

2.3 Pre-trained Models and T5

Building on these foundations, Tufano et al. [39] explored the use of pre-trained models, specifically the T5 architecture, to boost code review automation. Pre-trained models are first trained on large, generic code datasets and then fine-tuned for spe- cific tasks, such as code review. This approach leverages knowledge from a broader context, allowing the model to generalize better and perform multiple tasks simulta- neously.

The study showed that T5, when pre-trained and fine-tuned, outperformed previous models, especially in generating revised code and handling diverse code review tasks. One key property of T5 is its ability to support multi-task learning, meaning a single model can be trained to handle several code review tasks at once. However, this comes at the cost of significant computational resources for pre-training and fine-tuning, and the process of finding optimal hyperparameters is expensive and time-consuming. While performance improved, the models still faced challenges with very complex or context-dependent code changes.

Previous deep learning models for code review automation struggled with complex changes and relied on code abstraction, limiting their usability and accuracy. This paper addresses these limitations by introducing a pre-trained T5 model and a larger, more realistic dataset.

Properties and Methodology:

- Introduces a pre-trained Text-to-Text Transfer Transformer (T5) model for code review tasks.
- Works directly on raw source code (no abstraction), supporting longer and more complex code sequences.
- Trains and fine-tunes the model on a large dataset of Java code reviews, including code-to-code, code-and-comment-to-code, and code-to-comment tasks.
- Employs multi-task learning, allowing a single model to handle multiple code review tasks.

Workflow:

1. Pre-train T5 on a large corpus of Java code and technical English from Stack Overfiow and CodeSearchNet.
2. Fine-tune on mined code review triplets from GitHub and Gerrit.
3. Evaluate on perfect prediction, BLEU score, and Levenshtein

distance. Figure 2.2 shows the examples of perfect and alternative predictions:

Figure 2.2: Examples of perfect and alternative predictions.

Achievements:

- T5 outperforms previous DL models, especially for complex code changes and comment generation.
- Supports a broader range of code transformations, including the introduction of new variables and literals [17].
- Demonstrates the value of multi-task learning for code review automation.

Limitations:

- **Prediction Accuracy:** Still limited, especially for highly complex or context-dependent changes.
- **Performance in Comment Generation:** Lower than for code transforma- tions.
- **Single-Language Focus:** Experiments limited to Java.
- **High Computational Cost:** Pre-training and fine-tuning require significant resources.

2.4 Review4Repair: Code Review-Aided Program Repair

Another innovative direction is using code review comments directly to guide automated program repair, as explored in the Review4Repair integrates code review comments into program repair, using a sequence-to-sequence model trained on 55,060 code reviews and fixes. Traditional program repair tools often rely on test suites to identify and fix bugs, but these can be incomplete or weakly specified. Review4Repair proposes using natural language comments from code reviews as a more reliable guide for where and how to fix code. These comments, written by experienced developers, often pinpoint the exact location and nature of defects, ranging from bugs to naming conventions and even stylistic issues. This approach improved fix accuracy (top-1 by 20.33%, top-10 by 34.82%) and addressed a broader range of issues, including stylistic and non-code changes.

Most learning-based automatic program repair (APR) tools rely on bug localizers and test suites, which can be incomplete or unreliable. This paper proposes leveraging code review comments as a more trustworthy source for bug localization and fix guid- ance.

Properties and Methodology:

- Trains a sequence-to-sequence neural model on 55,060 code reviews and associated code changes.
- Incorporates code review comments as input to guide the repair process.
- Introduces new tokenization and preprocessing techniques for better learning.

- Systematically develops a taxonomy of fixes (bug fix, refactoring, stylistic, non-code changes) and 47 subcategories.

Workflow:

1. Crawl and preprocess code review data from Gerrit and GitHub.
2. Map code changes to corresponding review comments.
3. Train seq2seq models to generate fixes based on code and review comments.
4. Evaluate accuracy and coverage of generated

fixes. Figure 2.3 shows the workfiow.

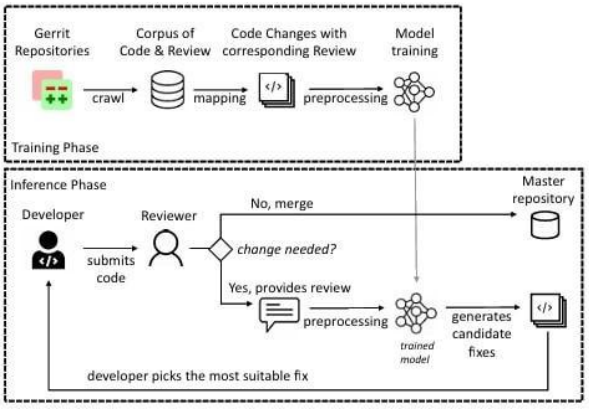


Figure 2.3: Workfiow of Review4repair: Code review comments guide automatic re- pair.

Achievements:

- Improves top-1 accuracy by 20.33% and top-10 accuracy by 34.82% over prior learning-based APR techniques.
- Capable of addressing stylistic and non-code issues, not just bugs and refactor-ing.
- Reduces reliance on bug localizers by using reviewer expertise.

Limitations:

- **Complex Fixes:** Struggles with architectural or highly complex changes.
- **Limited Dataset:** Training data is limited to available code reviews and may not cover all defect types.
- **Generalizability:** Effectiveness depends on the quality and clarity of review comments.

2.5 LLM-based Multi-Agent Systems and Industrial Adoption

With the rise of large language models (LLMs), new research has focused on leveraging these powerful tools for code review automation. Rasheed et al. [29] introduced an LLM-based AI agent trained on vast code repositories, including code reviews, bug reports, and best practices documentation. Unlike traditional static analysis tools, their LLM-based agent can predict not only current issues but also future risks in code, provide suggestions for improvement, and even optimize code for better per- formance. These systems provide context-aware and actionable feedback, but require further benchmarking and user studies. Industrial deployments (e.g., Beko's CodeRe- viewBot show that 73.8% of automated comments are accepted and implemented, but review times may increase and some feedback may be irrelevant.

A unique aspect of this approach is its ability to enhance developer education by offering feedback that encourages best practices and efficient coding. The study found that LLM-generated suggestions could significantly reduce post-release bugs and improve the overall code review process. However, LLMs can sometimes produce suggestions that are irrelevant or outside the scope of the current task, and their effectiveness depends on the quality and diversity of their training data. The authors also highlight the need for further empirical studies comparing LLM-generated documentation and manual methods.

Traditional code review tools and static analysis methods are limited in their abil- ity to deeply understand code, offer actionable improvements, or provide proactive, context-aware suggestions. This paper proposes a multi-agent LLM-based system to address these gaps.

Properties and Methodology:

- Develops a multi-agent system with specialized LLM-based agents: Code Re- view Agent, Bug Report Agent, Code Smell Agent, and Code Optimization Agent.
- Trains on large, diverse GitHub repositories, including code reviews, bug reports, and best practices documentation.
- Each agent is responsible for a distinct aspect of code review, from bug detection to optimization.

Workflow:

1. Data collection from GitHub using REST APIs.

2. Training each agent on relevant code, comments, and documentation.
3. Agents analyze code, detect issues, and provide actionable suggestions.
4. Evaluation via developer feedback and preliminary case

studies. Figure 2.4 illustrate real-world usage.

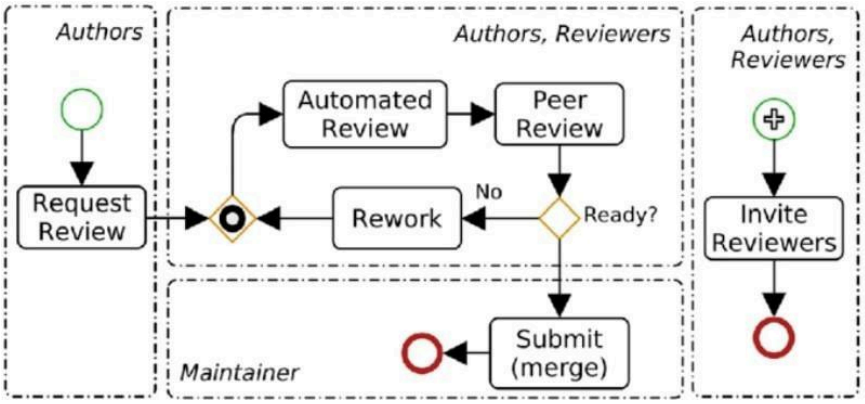


Figure 2.4: Pull request review fiow with automated and human comments.

Achievements:

- Demonstrates strong capability in identifying bugs, code smells, and inefficiencies across multiple programming languages and domains.
- Provides actionable, context-aware recommendations that developers found useful.
- Highlights the dual benefit of improving code quality and educating developers.

Limitations:

- **Preliminary Evaluation:** Lacks quantitative benchmarking and large-scale user studies.
- **No Real-Time Validation:** System not yet validated in real-time, production environments.
- **Training Bias:** Effectiveness depends on the diversity and representativeness of training data.
- **No Subdomain Analysis:** Does not differentiate performance across review subdomains (e.g., refactoring vs. debugging).

2.6 Evaluation Metrics and Benchmarks

Evaluating the quality of code and automated review suggestions is a complex chal- lenge. Traditional evaluation methods, such as match-based (BLEU, n-gram overlap) or execution-based (test case passing) metrics, have limitations: they often fail to cap- ture the true semantics of code and may not reflect human judgment [42]. Xu et al. propose the Human-Like Code Quality Evaluation through LLM-based Recursive Se- mantic Comprehension (HuCoSC), which uses LLMs recursively to break down and analyze code semantics in a more human-like way. This approach achieves a higher correlation with expert human evaluations and provides a more nuanced assessment of code quality.

Comparative studies show that LLM-based tools (e.g., code2Prompt, Copilot, Deep- Code) vary in bug detection and code quality assessment, with code2Prompt achieving the highest F1-score (90%) in data science workfiows. Comprehensive benchmarking frameworks like CodeFuse-CR-Bench and StackEval provide standardized evaluation protocols for end-to-end code review assessment [11], [22].

However, even advanced evaluation methods face challenges. LLMs may hallucinate or be uncertain, and their judgments can still diverge from human intuition in sub- tle ways. The field continues to explore how best to balance automated and human evaluation for trustworthy code review automation.

Figure 2.5 shows the HuCoSC evaluation pipeline.

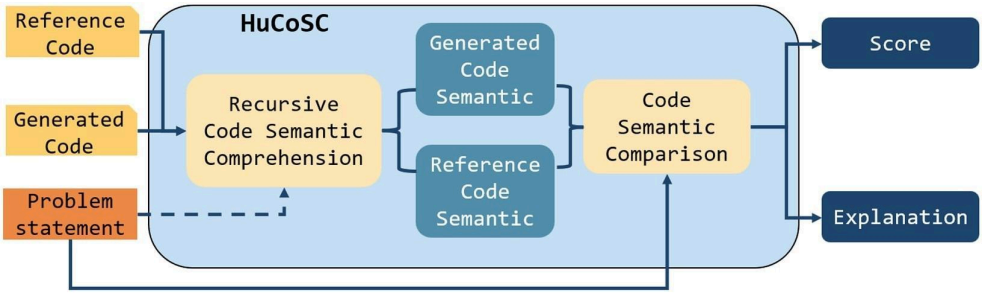


Figure 2.5: HuCoSC: LLM-based recursive semantic comprehension for code quality evaluation.

2.7 Addressing the Limitations In Our Proposed Methodology

Despite significant advances in automated code review, including the adoption of deep learning, pre-trained models, and large language models (LLMs), several critical limitations persist in both research and practice. Our methodology and future work are specifically designed to address these gaps, as outlined below.

1. **Inadequate Semantic Evaluation Metrics :** Traditional evaluation metrics like BLEU and token overlap do not capture the semantic alignment between AI- generated and human review comments. Recent frameworks such as HuCoSC offer more human-like, semantic evaluation, but are not yet widely adopted. Our method- ology incorporates both SBERT-based semantic similarity and emerging semantic met- rics to provide a more nuanced and reliable assessment of review quality.

2. **Dataset Quality, Bias, and Coverage :** Many prior studies rely on datasets that are limited in size, noisy, or lack coverage of all defect types and review subdomains. Data extraction errors, annotation inconsistencies, and over-representation of certain languages (e.g., Java) further limit generalizability. Our ongoing work focuses on expanding and curating a high-quality, domain-tagged dataset, with rigorous annotation protocols and subdomain labeling to support robust benchmarking and reproducibil- ity.

3. **Lack of Subdomain-Level Evaluation and Benchmarking :** Most existing automated code review tools and studies treat code review as a monolithic task, fail- ing to differentiate between subdomains such as refactoring, debugging, formatting, security, and performance. This results in generic feedback that may be irrelevant or less actionable for specific types of code changes. Our approach introduces a domain-tagged benchmark dataset, enabling subdomain-aware evaluation and more targeted analysis of LLM performance.

4. **Limited Contextual Understanding and Project-Specific Adaptation :** LLMs and other automated tools often struggle to fully understand project-specific context, complex code dependencies, or unconventional coding practices. This can lead to incomplete, misleading, or even incorrect recommendations. We aim to address this by incorporating richer contextual information (e.g., extended code snippets, metadata)

and by exploring adaptive learning strategies that allow models to learn from ongoing developer feedback and evolving project standards.

5. **False Positives, Irrelevant, or Trivial Suggestions :** A recurring challenge in LLM-powered code review is the generation of false positives or trivial suggestions that do not meaningfully improve code quality. Developers may waste time address- ing non-issues, and over-reliance on automation can lead to missed critical bugs. Our methodology includes human-in-the-loop evaluation and plans for explainable AI features, ensuring that recommendations are both actionable and trustworthy.

6. **Lack of Real-World Validation and User Studies :** Few studies have systemat- ically validated automated code review tools in real-world, production environments or through large-scale user studies. There is a need for empirical evidence on how these tools affect developer productivity, code quality, and workfiow integration. Our future work involves collaboration with industry partners to deploy and assess our models in practical settings, gathering both quantitative and qualitative feedback.

7. **Explainability and Developer Trust :** Developers are more likely to adopt auto- mated code review tools if they understand and trust the recommendations. Most cur- rent LLMs act as black boxes, offering little insight into why a suggestion was made. Our future work includes the integration of explainable AI methods, providing clear rationales for each recommendation to build developer confidence.

Chapter 3

Methodology

3.1 Overview

This study adopts a multi-phase methodology to systematically evaluate the performance of state-of-the-art LLMs in automated code review, focusing on quantitative alignment with human review comments. Our approach addresses limitations identified in existing evaluation frameworks by incorporating both traditional datasets and expert-curated benchmarks for comprehensive analysis.

3.2 Phase 1: Data Processing of existing dataset

- **Dataset Selection:** Our primary dataset is the same as the one used while implementing Review4Repair, which was constructed using open-source code reviews from Gerrit. The dataset used by Review4Repair only contains Java files from this dataset.
- **Filtering and Mapping:** We used the Review4Repair dataset sourced from Gerrit. For each entry, we mapped the associated Java file using a JSON mapping. If a valid file was found, we extracted a 7-line code snippet around the review location. Human-written comments were lightly preprocessed (lowercasing, removing numbering, punctuation, and extra spaces) but no manual filtering of vague or non-English comments was done.
- **Context Extraction:** For each code diff, code snippets are extracted with ± 3 lines of context and paired with their corresponding review comments.

3.3 Phase 2: LLM-driven Comment Generation and Similarity Analysis

- **Model Selection:** Prompt six leading LLMs (GPT-4, Llama, Mistral, CodeL- lama, Qwen2.5, Phi4) to generate review comments for each instance.
- **Semantic Similarity Evaluation:** We compared AI-generated and human-written review comments using:
  - **SBERT (Sentence-BERT):** We used sentence-transformers to generate embeddings and measured cosine similarity to evaluate semantic close- ness.
  - **Cosine similarity:** We separately computed cosine similarity based on TF-IDF vectorization for a surface-level text match.
  - **BLEU Score:** We calculated BLEU scores to measure n-gram overlap be- tween AI and human comments, capturing surface similarity. The com- plete pipeline for comment generation and similarity analysis is presented in Figure 3.1.

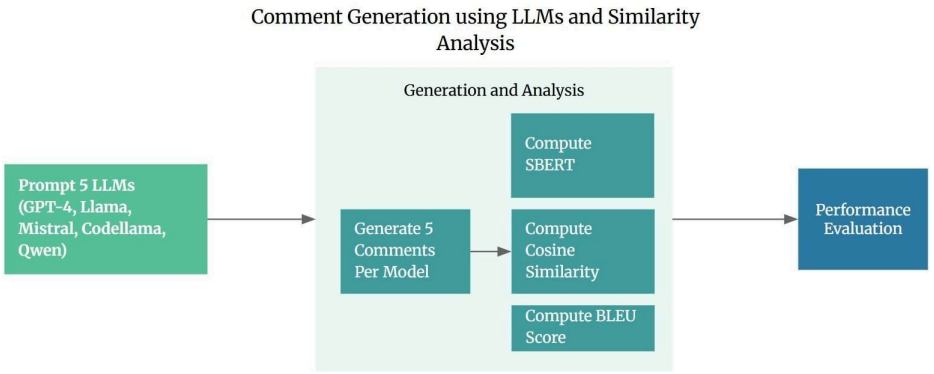


Figure 3.1: LLM-driven comment generation and similarity analysis pipeline

```
prompt_template = PromptTemplate(
    input_variables=["code"],
    template=(
        "You are a code reviewer. Analyze the given Java code and provide concise review comments. "
        "Ensure your comments focus on correctness, clarity, and best practices. Provide multiple comments:\n\n{code}\n\n"
        "Format comments starting with a number (e.g., '1.', '2.', etc.)."
    ),
)
```

Figure 3.2: Prompt used for generating LLM based comments

3.4 Phase 3: Expert-Curated Dataset Construction

3.4.1 Repository Selection and Code Extraction

Multi-Language Coverage: We expanded beyond Java to include JavaScript, enabling broader language generalizability assessment.

Repository Criteria: Open-source repositories were selected based on:

- **Active development:** Only repositories with commits made within the last six months were considered. This ensures that the codebases are maintained, relevant, and reflect contemporary development practices.
- **Established codebase:** Projects with at least 300 commits were included to guarantee sufficient code history and review data for meaningful analysis. Smaller repositories often lack the volume of activity needed for reliable evaluation.
- **Clear project structure and documentation:** Repositories with well-defined directories, consistent naming conventions, and adequate documentation were prioritized. This facilitates automated code extraction, parsing, and comprehension, reducing the risk of misclassification or incomplete data capture.
- **Diverse functionality:** Projects from different domains (web development, utilities,libraries) were chosen to capture a range of coding and review patterns. This diversity helps test the adaptability of the proposed approach across various software contexts.

Code Sample Extraction: We extracted code snippets representing common development scenarios:

- Bug fixes and patches
- Feature implementations
- Refactoring operations
- Performance optimizations
- Code style improvements

3.4.2 Web Application Development

To facilitate systematic expert review collection, we developed a comprehensive web application:

Frontend (Next.js):

- Responsive interface for code review presentation
- Syntax highlighting for Java and JavaScript
- Intuitive comment submission system
- Progress tracking for reviewers

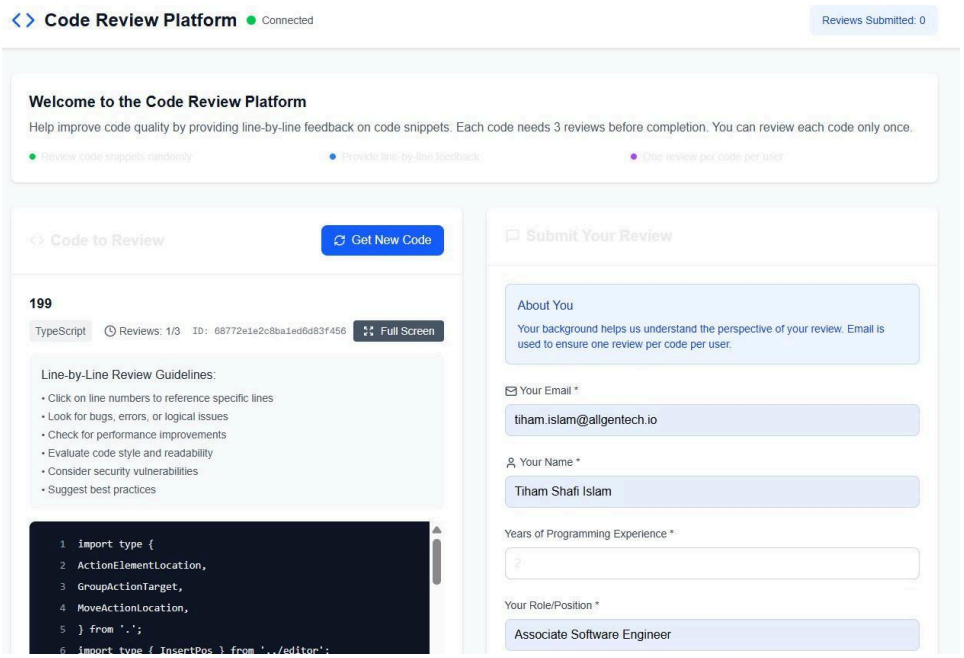


Figure 3.3: Code review platform website

Backend (Node.js/Express):

- RESTful API for data management
- User authentication and session management
- Real-time data validation and storage
- Automated backup and data integrity checks

Database (MongoDB):

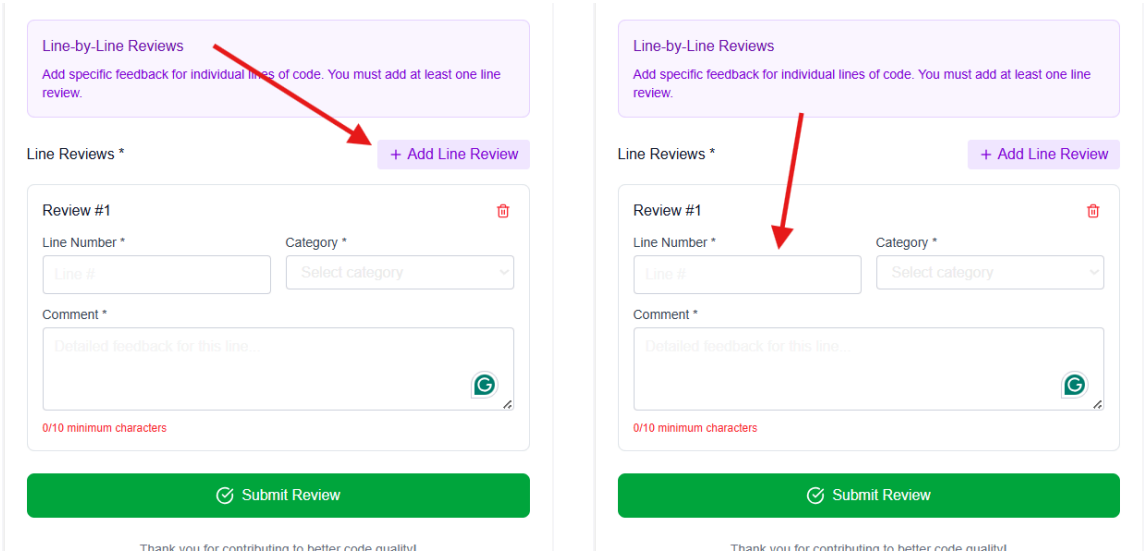
- Flexible schema for diverse code samples and reviews
- Efficient indexing for rapid data retrieval
- Relationship mapping between code samples, reviews, and reviewers The web interface is shown in Figure 3.3.

3.4.3 Expert Reviewer Recruitment and Guidelines

Reviewer Selection Criteria:

- **Minimum 1–2 years professional software development experience:** Re- viewers were required to have hands-on experience in software development. This ensures that their assessments reflect practical, real-world coding knowl- edge rather than purely academic understanding.
- **Active involvement in code review processes:** Participants needed to be currently engaged in code review activities as part of their professional work. This guarantees familiarity with review practices, common pitfalls, and indus- try standards, which is crucial for providing meaningful feedback.
- **Expertise in Java and/or JavaScript:** Since the study focuses on repositories in these languages, reviewers with expertise in one or both languages were se- lected to ensure accurate understanding of the code and context-specific issues.
- **Industry background spanning various domains (fintech, healthcare, e-commerce, etc.):** Having reviewers from diverse industry domains introduces a wider perspective on code quality expectations and practices. Different sec- tors may emphasize different standards, so this diversity helps make the study findings more generalizable.

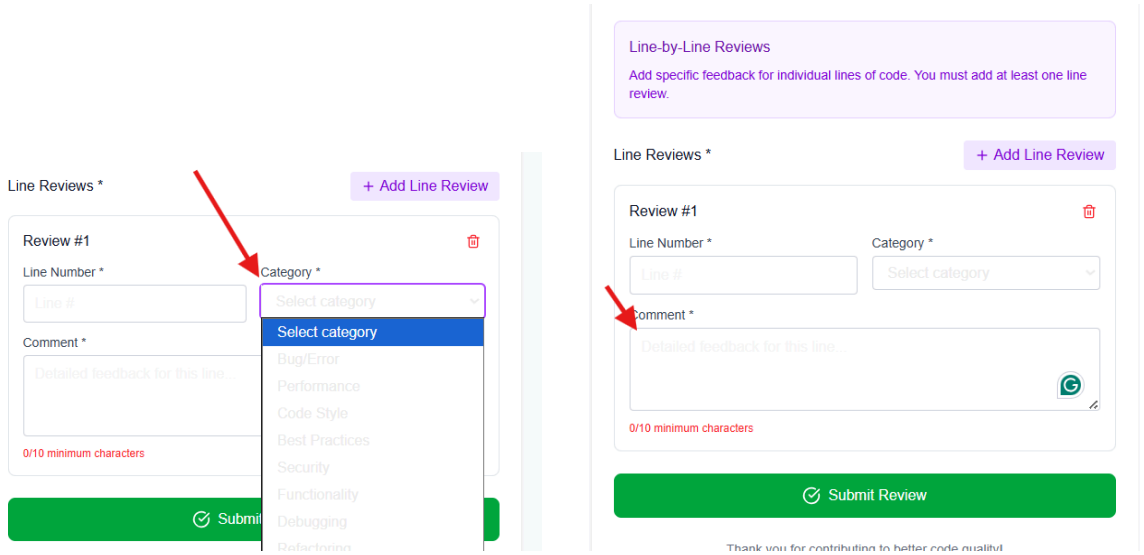
Reviewer information is captured as shown in Figures 3.4 and 3.5.



(a) Add Line Review **(b)** Add Line Number

Figure 3.4: Step-by-step review process (a) Add line review (b) Add line number

Reviewer Information:



(a) Category of Review **(b)** Review Comment

Figure 3.5: Step-by-step review process (continued): (a) Selecting review category (b) Writing review comment

- Reviewers must fill in their basic information such as name, experience, and designation.
- A random review code will appear beside their name, which can be regenerated by clicking “Get new code”.

Review Guidelines: Experts were provided with standardized guidelines while maintaining authenticity:

- Focus on code quality, maintainability, and best practices.
- Identify potential bugs, security issues, and performance concerns.
- Suggest improvements for readability and documentation.
- Maintain a professional, constructive tone.
- Provide specific, actionable feedback.

Step-by-step Review Process:

- Click “Add Line Review” (Figure 3.4(a)) to start reviewing.
- Enter the line number to be reviewed (Figure 3.4(b)). A single line can have multiple reviews (e.g., one for a bug, another for performance).
- Choose the review category (Figure 3.5(a)).
- Write the review comment describing the issue or improvement (Figure 3.5(b)).

- Finally, click “Submit Review” to submit the review.

Quality Assurance: To ensure review consistency:

- Pilot study with subset of samples reviewed by multiple experts
- Feedback sessions to calibrate review standards
- Ongoing quality monitoring throughout data collection

3.5 Phase 4: LLM-based Comment Generation

3.5.1 Model Selection and Configuration

We evaluated five state-of-the-art LLMs:

- **GPT-4:** OpenAI’s fiagship model, known for strong natural language understanding and code generation capabilities. It provides a high benchmark for comparison due to its widespread adoption and state-of-the-art performance.
- **Llama 3.1:** Meta’s open-source large language model. Its open-source nature allows for customization and reproducibility in research, while still offering competitive performance in general language tasks.
- **CodeLlama:** A specialized code-focused variant of Llama. It is designed to handle programming languages more effectively, making it particularly suitable for generating precise and context-aware code review comments.
- **Mistral:** A European open-source model that emphasizes efficient architecture and multilingual capabilities. Including Mistral helps assess performance differences across diverse LLM frameworks.
- **Qwen 2.5:** Alibaba’s multilingual model, capable of understanding and generating text in multiple languages. Its inclusion allows evaluation of LLM performance in handling code comments that may mix natural language with code or documentation in different languages.

3.5.2 Prompt Engineering and Standardization

Prompt Design: We developed standardized prompts that:

- Clearly specify the code review task
- Provide context about the codebase and functionality

- Maintain consistency across different models

An example of the prompt structure used is shown in Figure 3.2. The following figures show the prompts and examples used for zero-shot, one-shot and few-shot analysis.
Comment by reviewer: "getLocale() may fail, consider try/catch or fallback"

- **Zero shot:**

```
prompt_template = PromptTemplate(
    input_variables=["code"],
    template=(
        "You are a professional code reviewer. "
        "Provide **exactly 5** concise, actionable code review comments focusing on improving "
        "readability, maintainability, structure, duplication, and best practices. "
        "Do NOT provide fixes or code snippets—only comments. "
        "Format each comment starting with a number (e.g., '1.', '2.', '3.', '4.', '5.').\n\n"
        "Code to review:\n{code}\n\n"
    )
)
```

Figure 3.6: Prompt used for zero-shot analysis

AI generated best comment: The RootLayout function is declared as async and getLocale is awaited inside it. This could potentially lead to rendering issues as React components are synchronous. Consider fetching the locale data outside of the component or using a state management solution to handle asynchronous operations.

- **One shot:**

```
# One-shot Prompt Template
prompt_template = PromptTemplate(
    input_variables=["code", "example_code", "example_comment"],
    template=(
        "You are a professional code reviewer. "
        "Your task is to generate exactly 5 concise, actionable review comments "
        "focusing on readability, maintainability, structure, duplication, and best practices.\n\n"

        "Here is an example:\n"
        "Code:\n{example_code}\n"
        "Human Reviewer Comment:\n{example_comment}\n\n"

        "Now review the following code:\n{code}\n\n"
        "Provide 5 comments, each starting with a number."
    )
)
```

Figure 3.7: Prompt used for one-shot analysis

AI generated best comment: The RootLayout function is declared as async but it doesn’t seem to have any await statements inside it. If there are no asyn- chronous operations, the async keyword can be removed.

- **Few shot:**

```
prompt_template = PromptTemplate(
    input_variables=["code"],
    template=(
        "You are a professional code reviewer. "
        "Your task is to generate exactly 5 concise, actionable review comments "
        "focusing on readability, maintainability, structure, duplication, and best practices.\n\n"
        f"{examples_text}"
        "Now review the following code:\n{code}\n\n"
        "Provide 5 comments, each starting with a number."
    )
)
```

Figure 3.8: Prompt used for few-shot analysis

AI generated best comment: Consider adding a fallback locale in case the getLocale() function fails to retrieve the locale.

3.6 Phase 5: Evaluation Methodology

3.6.1 Quantitative Metrics

Semantic Similarity Analysis:

We employed SBERT (Sentence-BERT) as the primary evaluation metric to assess semantic similarity between human-generated reviews from our collected dataset and LLM-generated reviews from five models: GPT-4, Llama 3.1, CodeLlama, Mistral, and Qwen2.5.

SBERT Implementation:

- **Model:** all-MiniLM-L6-v2 pre-trained transformer for generating dense vector embeddings
- **Similarity Computation:** Cosine similarity between embeddings to measure semantic alignment
- **Preprocessing:** Text normalization including lowercasing, punctuation removal, and whitespace standardization

Rationale for SBERT-Only Approach:

We deliberately excluded traditional metrics like TF-IDF cosine similarity and BLEU score due to their fundamental limitations in code review evaluation:

- **Semantic vs. Lexical Focus:** Code review comments can express identical concerns using completely different vocabulary (e.g., "Fix memory leak" vs. "Ad-

dress resource deallocation issue"). SBERT captures semantic equivalence while surface-level metrics fail to recognize such relationships.

- **Context-Aware Understanding:** SBERT's transformer architecture understands contextual relationships between words, enabling it to recognize that "refactor this method" and "extract into smaller functions" convey similar improvement suggestions.
- **BLEU Score Limitations:** BLEU was designed for machine translation evaluation with multiple reference translations, making it inappropriate for single-reference code review assessment. Its n-gram overlap approach penalizes semantically equivalent but lexically different expressions.
- **Real-world Applicability:** Human reviewers naturally express the same cod- ing concerns through varied linguistic formulations. SBERT better refiects whether AI models truly understand underlying code quality issues rather than merely matching surface patterns.

3.7 Phase 6: Validation and Reliability Assessment

3.7.1 Dataset Composition and Model Evaluation

- **Few-shot Learning:** First 10 samples used as examples for LLM prompt engineering
- **Evaluation Set:** 400+ code samples evaluated per model
- **Comment Generation:** Each model generated exactly 5 review comments per code sample
- **Similarity Scoring:** SBERT similarity computed between each AI comment and corresponding human review

3.7.2 Dataset Explanation

Each column in the dataset captures a specific aspect of code review entries:

- **Code Language:** Programming language of the reviewed snippet (e.g., JavaScript, TypeScript).
- **Reviewer Name:** Person who wrote the review comment.
- **Code Content:** Import lines or snippet text that the comment refers to.

- **Experience:** Reviewer's years of experience as a number.
- **Line Number:** Source line referenced by the comment
- **Line Comment:** The review feedback or note about that line.
- **Line Category:** Labeled category of the comment (e.g., Bug/Error, Security, Performance, Best Practices, Functionality).

3.7.3 Cross-Model Validation Framework

- Consistent evaluation protocol applied across all five LLM models
- Identical preprocessing and prompt templates to ensure fair comparison
- Statistical significance testing to identify meaningful performance differences
- Systematic analysis of model-specific strengths and weaknesses in code review generation

Simplified Evaluation Approach: We focused on direct semantic similarity assessment without subdomain classification, allowing for comprehensive model compar- ison while maintaining evaluation clarity and statistical power across the five LLM architectures under investigation.

Chapter 4

Results and Analysis

4.1 Connections to Existing Literature

Our empirical findings resonate with and extend a rapidly growing body of work on Large-Language-Model–assisted code review [8], [37]. Recent industrial case stud- ies confirm that end-to-end automated review pipelines can shorten delivery cycles while preserving review depth [7], [8]. Multi-agent frameworks demonstrate that specialised reviewer, bug-finder, and refactoring agents collaborating through shared context substantially boost precision over single-agent baselines [18], [21], [35].

Prompt-engineering and fine-tuning strategies akin to our few-shot set-up have been shown to improve comment granularity and reduce hallucinations in LLM outputs [6], [43], [44]. Generation-centric benchmarks such as CodeFuse-CR-Bench and Stack- Eval further highlight the importance of context length and exemplar diversity for achieving human-level performance [11], [22], [33].

The strong semantic-level gains we observe corroborate transformer-based repair and graph-augmented similarity approaches in automated program-repair research [17], [27], [28]. They also echo findings from wider surveys on deep-learning-based quality assessment and code comprehension [5], [15], [34], [38].

Finally, our emphasis on data curation, reviewer expertise, and sub-domain tagging aligns with calls for richer, security-aware datasets in contemporary static-analysis and vulnerability-detection literature [12], [16], [23], [32], [41]. Collectively, these parallels underscore a community-wide consensus on the need for nuanced bench- marks, interpretability, and rigorous human validation [2], [24], [25], [26], [30], [36], [45].

4.2 Existing Dataset Comparison Analysis

4.2.1 Dataset Quality Impact Assessment

We used 3 metrics (SBERT, cosine similarity and BLEU score) to perform comparative analysis on the existing dataset. The results are presented in both tabular format and graphical visualization below:

Table 4.1: Performance Comparison of LLM Models across Different Metrics

LLM Model	SBERT (MiniLM-L6-v2)	BLEU Score	Cosine Similarity
ChatGPT 4.0	0.2235	0.0079	0.0652
Llama 3.1	0.2466	0.0073	0.0565
CodeLlama	0.2206	0.0067	0.0699
Mistral	0.1756	0.0057	0.0561
Qwen 2.5	0.2126	0.0066	0.0580
Phi 4	0.2121	0.0069	0.0612

The comparative analysis reveals that Llama 3.1 achieves the highest SBERT score (0.2466), while CodeLlama demonstrates superior performance in cosine similarity (0.0699). ChatGPT 4.0 shows the best BLEU score performance (0.0079), indicating varying strengths across different evaluation metrics for each model. These performance differences across metrics are visualized in Figure 4.1.

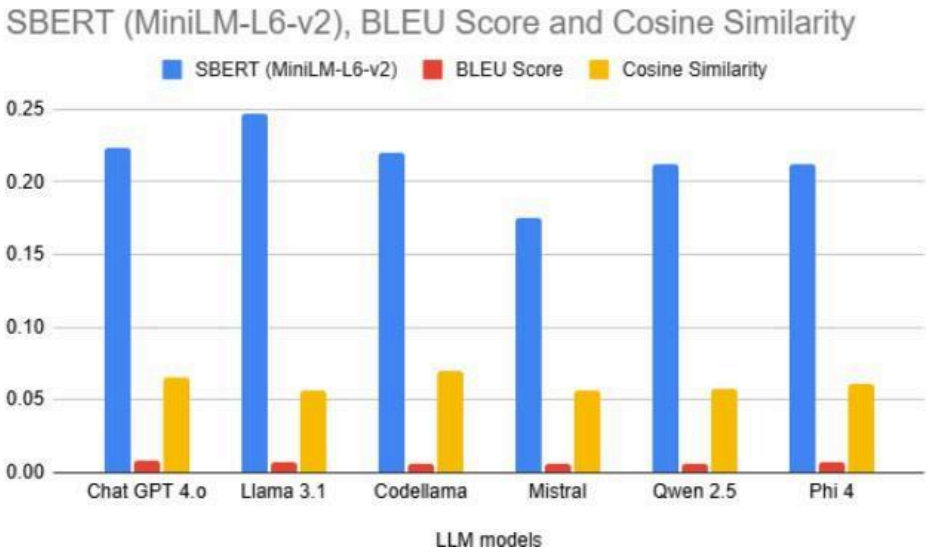


Figure 4.1: Comparison of SBERT, Cosine Similarity and BLEU Score on existing dataset

4.3 Dataset Construction Results

4.3.1 Expert-Curated Dataset Statistics

Our enhanced data collection process yielded a substantial, better quality dataset. Comments were made by industry experts. Here is a snippet of the dataset we collected before cleaning the data for comparative analysis.

Code Language	Code Content	Reviewer Name	Experience	Position		Line Number	Line Comment	Line Category
JavaScript	import { tool } from 'ai'; import { rea	Hasib Chowdhury	2	Associate Software Engineer		16	Have to ensure getAllFil	Bug/Error
JavaScript	import '@/styles/globals.css'; import	Atif Karim	3	Senior Software Engineer		32	Consider defaultTheme	Best Practices
JavaScript	pr_review_data pr_review_data 10(	Sazzad Hossan Sakib	4	Senior Software Engineer		35	Have to validate readFil	Security
TypeScript	use client'; import { useEditorEngine	Rezoan Ahmed Abir	1	Associate Software Engineer		26	Accessing editorEngine..	Bug/Error
JavaScript	pr_review_data pr_review_data 10(	Sazzad Hossan Sakib	4	Senior Software Engineer		21	path is taken directly frc	Security
TypeScript	import { Create } from './create'; imp	Amit Kanti Barua	4	Senior Software Engineer		6	The justify-center class i	Bug/Error
TypeScript	import { FrameType, type Frame, typ	Md. Talha Khan	2	Associate Software Engineer		15	<GestureScreen> is alwa	Bug/Error
TypeScript	import { api } from '@/trpc/client'; ir	Fahmi Kazi Md	3	Associate Software Engineer		16	Calling the same mutati	Performance
TypeScript	use client'; import { Hotkey } from '@	Arman Hossain Dipu Khan	1	Associate Software Engineer		15	The toolbar items are re	Performance
TypeScript	import type { WatchEvent } from '@(	Mahid Atif Hosain	2	Associate Software Engineer		126	slow for bulk blocks	Performance
TypeScript	use client'; import '@xterm/xterm/c	Fahmi Kazi Md	3	Associate Software Engineer		62	Delayed focusing avoids	Performance
TypeScript	import OpenAI from 'openai'; const	Hasib Chowdhury	2	Associate Software Engineer		2	If these variables contai	Bug/Error
JavaScript	use client'; import { Popover, Popov	Aaron Halder	3	Senior Software Engineer		47	olor.from should not thi	Bug/Error
JavaScript	import { useProjectManager } from '	Abu Noman Md Sakib	4	Senior Software Engineer		22	closeTimeoutRef typed .	Bug/Error
TypeScript	import { api } from '@/trpc/client'; ir	Fahmi Kazi Md	3	Associate Software Engineer		29	No check or error handl	Bug/Error
TypeScript	import { createClient } from '@/utils,	Tahiya Sukonya	2	Junior Developer		22	Assume HTTPS here ma	Best Practices
TypeScript	import { api } from '@/trpc/client'; ir	Tiham Shafi Islam	1	Associate Software Engineer		36	repeated calls will push	Bug/Error
JavaScript	import { Create } from './create'; imp	Md. Ahnaf Akib	3	Senior Software Engineer		6	Add responsive breakpc	Best Practices
TypeScript	import path from 'path'; import pars	Mahid Atif Hosain	2	Associate Software Engineer		14	Relies on path.relative;	Security
JavaScript	import { EditorAttributes } from '@/c	Tahiya Sukonya	2	Junior Developer		8	Inserts a JSX element int	Functionality
JavaScript	import { Create } from './create'; imp	Amit Kanti Barua	4	Senior Software Engineer		9	Removing unnecessary	Performance

Figure 4.2: Industry Experts review's data

4.4 Comparative Performance Analysis

4.4.1 Dataset Quality Impact Assessment

The transition from the original dataset to our expert-curated dataset demonstrated significant performance improvements across all evaluated models:

In Table 4.2, we can see there is a dramatic improvement in SBERT similarity scores (ranging from 43.5% to 91.9% improvement). This validates our hypothesis that dataset quality was a primary limiting factor in the initial analysis [13].

Table 4.2: SBERT similarity scores comparison between old and new datasets

Model Name	Old Dataset	New Dataset	Improvement (%)
GPT-4	0.2235	0.3763	68.4
Llama 3.1	0.2466	0.3539	43.5
CodeLlama	0.2206	0.3386	53.5
Mistral	0.1756	0.3369	91.9
Qwen 2.5	0.2126	0.3506	64.9

Figure 4.3 compares the performance of five LLMs on our expert-curated dataset against the existing Review4Repair dataset, using Zero-Shot prompting and SBERT similarity scores. All models show clear improvements, confirming that the low baseline results

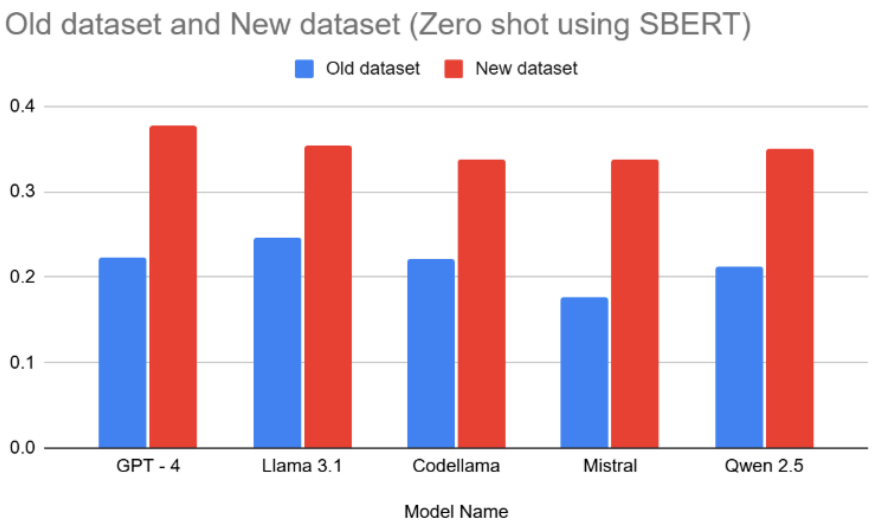


Figure 4.3: Zero shot SBERT similarity score comparison between old and new dataset

were largely due to dataset limitations and that our benchmark offers a more reliable evaluation foundation.

Model-wise analysis:

- **GPT-4:** Nearly doubles its score (0.22 to 0.38), highlighting its strong capability for generating human-like review comments.
- **Llama 3.1:** Improves from 0.25 to 0.35, confirming its robustness and narrowing the gap with GPT-4.
- **CodeLlama:** Rises from 0.22 to 0.34, showing its specialized training for code is better refiected with high-quality data.
- **Qwen 2.5:** Jumps from 0.21 to 0.35, surpassing CodeLLaMA and aligning closely with human review semantics.
- **Mistral:** Increases from 0.18 to 0.34, though still less competitive, suggesting architectural or training misalignment with code review tasks.

In summary, the across-the-board improvement validates the effectiveness of our cu- rated dataset as a superior benchmark for evaluating LLMs in automated code review.

4.4.2 Prompting Strategy Performance Analysis

Our evaluation of different prompting strategies revealed consistent improvements with increased contextual examples [6], [43]:

Table 4.3: SBERT performance across different prompting strategies

Model Name	Zero Shot	One Shot	Few Shots	Best Strategy
GPT-4	0.3763	0.4140	0.4485	Few Shots
Llama 3.1	0.3539	0.3669	0.3934	Few Shots
Qwen 2.5	0.3506	0.3569	0.4031	Few Shots
CodeLlama	0.3386	0.3315	0.3639	Few Shots
Mistral	0.3369	0.3500	0.3977	Few Shots

Key findings from prompting strategy analysis (Table 4.3) [18]:

- **Few-shot prompting** consistently outperformed zero-shot and one-shot approaches across all models
- **GPT-4** achieved the highest absolute performance across all prompting strategies. It also showed the most significant absolute improvement, with a 19.2% performance increase from zero-shot to few-shot prompting.
- **Llama 3.1** demonstrated consistent and stable improvement with each addi- tional example, reinforcing its reliability as a robust open-source model for this task.
- **Qwen 2.5** made a notable leap in performance with few-shot prompting, indicating it benefits greatly from clear context and is a highly competitive model when provided with sufficient examples.
- **Mistral** demonstrated one of the most significant relative improvements from zero-shot to few-shot prompting, showing its strong capability to learn from in-context examples despite a lower starting point.
- **CodeLlama's** performance was unique; it showed a minimal response to one-shot prompting but benefited substantially from a broader context, with a significant jump in performance with few-shot examples.

The performance trends across prompting strategies are illustrated in Figure 4.4.

We can also see, from Figure 4.5, that there is a significant improvement in the re- sult for few shots compared to that of zero shot in each of the models. The highest improvement was shown by GPT-4, 19.18%, and the lowest improvment was shown

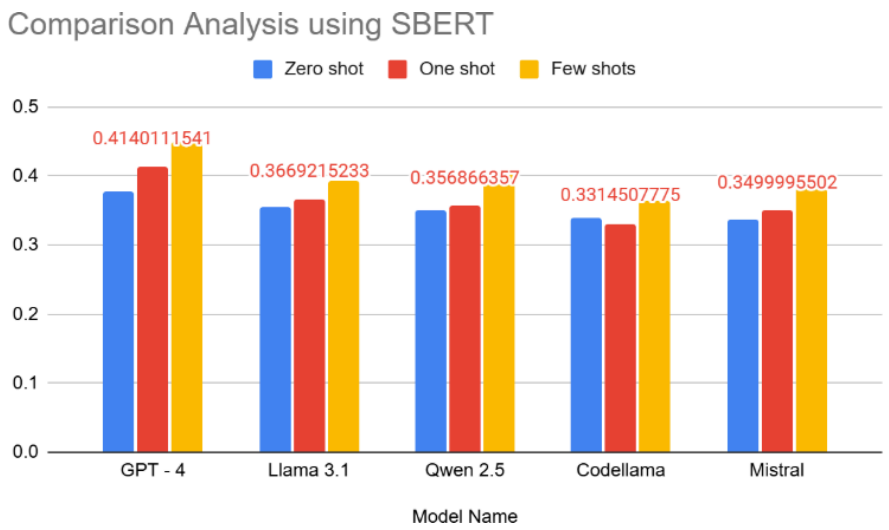


Figure 4.4: Comparative Analysis between zero, one and few shots

by codellama, 7.48%. This proves that providing examples to make the models understand the pattern of our code review actually helps to make the reviews more human-like. This is also an indication that if we fine-tune our models then there is a huge likeliness that the comments generated by our models will be more and more human-like

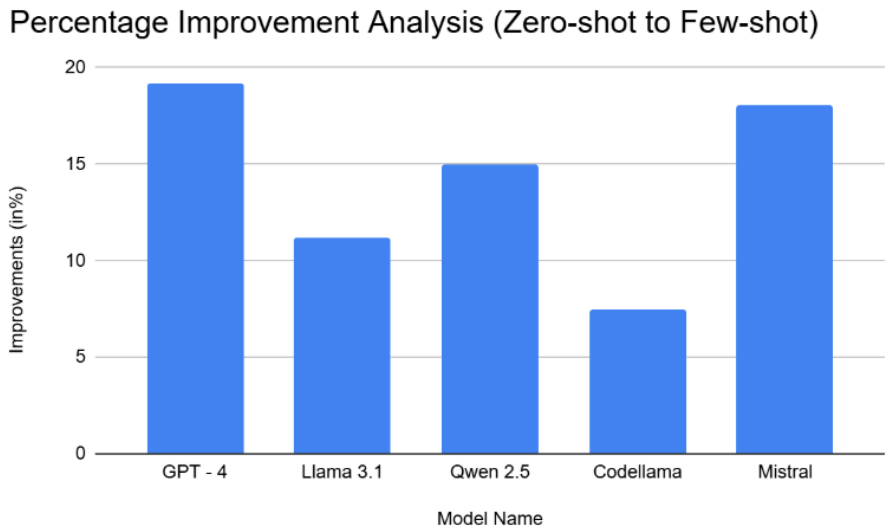


Figure 4.5: Percentage Improvement Analysis from Zero to Few shots

.

4.5 Discussion and Implications

4.5.1 Key Findings

Our experimental results reveal several critical insights into the generation of automated code review comments using large language models:

1. **Dataset Quality Impact:** The transition from the automatically generated Review4Repair dataset to our expert-curated dataset resulted in a remarkable 2.5- 3x improvement in similarity scores across all evaluation metrics. This substan- tial enhancement confirms that dataset quality serves as a critical factor in train- ing and evaluating code review systems. The expert-curated data, characterized by contextually relevant feedback and professionally crafted suggestions, pro- vided models with superior examples of real-world code review practices. This finding underscores the importance of investing in high-quality training data rather than relying solely on automatically extracted or synthesized examples, which may contain noise, inconsistencies, or lack the nuanced understanding that human experts bring to the code review process.

2. **Fine Tuning Effectiveness:** The significant performance improvement ob- served when transitioning to expert-curated data strongly indicates that fine- tuning models with this high-quality dataset will yield substantially more human- like reviews. Models trained on expert feedback are expected to better capture the tone, specificity, and constructive nature of professional code reviews, mov- ing beyond superficial pattern matching to genuine understanding of code qual- ity concerns.

3. **Prompting Strategy Effectiveness:** Few-shot prompting consistently outperformed both zero-shot and one-shot approaches across all tested models, with improvements ranging from 7.4% to 19.2%. This consistency across different model architectures suggests that providing multiple diverse examples helps models better understand the expected review format, tone, and level of detail. The few-shot approach appeared to guide models toward more contextually appropriate responses by demonstrating the variability and depth expected in professional code reviews. Interestingly, the improvement margin varied by model architecture, with some models showing greater sensitivity to the number of examples provided, suggesting that prompt engineering remains an essential consideration when deploying LLM-based code review systems.

4. **Practical Utility:** Beyond quantitative similarity metrics, expert ratings pro-

vided crucial validation that top-performing models generate reviews that are genuinely useful to developers in real-world scenarios, not merely semantically similar to reference reviews. Expert evaluators assessed the generated com- ments on dimensions including actionability, relevance, clarity, and appropri- ateness of tone. The strong correlation between high similarity scores and pos- itive expert ratings validates our evaluation methodology and confirms that op- timizing for semantic similarity translates to practical utility. This finding is particularly significant as it bridges the gap between automated metrics and ac- tual developer experience, suggesting that models excelling in our evaluation framework will likely provide value in production environments.

4.5.2 Practical Applications

While complete automation of code review remains an aspirational goal rather than an immediate reality, our results demonstrate that current LLM capabilities can pro- vide substantial value when integrated thoughtfully into existing development work- fiows. The enhanced performance metrics observed in our study suggest several prac- tical applications where AI-generated review comments can augment human exper- tise:

- **Automated Pre-screening:** Large language models can serve as an initial qual- ity gate by identifying obvious issues such as style violations, common anti- patterns, or basic security vulnerabilities before code reaches human review- ers. This pre-screening capability allows human experts to focus their limited time and cognitive resources on architectural decisions, complex logic valida- tion, and subtle design considerations that require deep domain knowledge and contextual understanding. By filtering out routine issues, organizations can re- duce review turnaround time and improve the quality of human review atten- tion.
- **Educational Tools:** High-quality AI-generated feedback can serve as an in- valuable learning resource for junior developers and students seeking to un- derstand professional code review practices. These tools can provide imme- diate, consistent feedback on practice exercises and personal projects, helping novice programmers internalize best practices, common pitfalls, and profes- sional communication standards long before their code reaches senior review- ers. The availability of on-demand feedback democratizes access to review ex- pertise, particularly valuable in educational settings or small teams lacking se- nior developers.

- **Consistency Enforcement:** Automated tools powered by fine-tuned language models can ensure consistent adherence to organizational style guides, nam- ing conventions, and documented best practices across large codebases and dis- tributed teams. Unlike human reviewers whose attention to stylistic details may vary with fatigue, time pressure, or familiarity with specific guidelines, AI systems can maintain unwavering consistency. This consistency is particularly valuable in large organizations where maintaining uniform code quality across multiple teams and projects presents significant coordination challenges.
- **Review Prioritization:** AI systems can analyze code changes and fiag those requiring urgent human attention based on factors such as complexity, poten- tial security implications, modifications to critical system components, or de- viations from established patterns. This intelligent triage ensures that human review resources are allocated efficiently, with experienced reviewers examin- ing high-risk changes while routine updates receive automated screening. Ad- ditionally, AI systems can provide preliminary assessments that help reviewers quickly understand the scope and nature of changes, reducing the cognitive load of context switching between diverse pull requests.

The substantial improvement in both quantitative metrics and qualitative expert as- sessments demonstrates that high-quality, expert-curated datasets combined with op- timized prompting strategies enable more reliable evaluation of LLM-based code re- view systems. These advances pave the way for practical deployment in software de- velopment workfiows, where AI serves as a collaborative assistant rather than a re- placement for human judgment [24], [41]. As these systems mature, we anticipate a hybrid model where AI handles routine aspects of code review while human experts focus on design, architecture, and knowledge transfer—ultimately improving both review quality and developer productivity.

4.6 Threats to Validity

4.6.1 Limitations and Future Considerations

Despite the significant improvements demonstrated in our study, several limitations constrain the generalizability and applicability of our findings:

- **Context Dependency:** Models continue to struggle with project-specific con- ventions, organizational coding standards, and architectural considerations that extend beyond individual code changes. While our expert-curated dataset cap-

tures general best practices, it cannot encompass the full diversity of contextual knowledge that experienced developers accumulate through long-term involvement with specific projects. This limitation manifests in generated reviews that may be technically correct but fail to align with project-specific design decisions, existing architectural patterns, or team-specific communication norms.

- **Scalability:** The expert annotation process, while producing superior training data, inherently limits both dataset size and update frequency. Manual cura- tion requires significant time investment from experienced developers whose availability is constrained by competing responsibilities. This bottleneck poses challenges for maintaining datasets that reflect evolving best practices, emerg- ing programming paradigms, and new language features. Future work must explore semi-automated or active learning approaches that can maintain data quality while improving annotation throughput.
- **Cultural Variations:** Review styles, communication norms, and feedback expectations vary substantially across development teams, organizations, and international contexts. Our dataset reflects practices common in specific development communities, which may not generalize to teams with different communication cultures—for example, teams prioritizing extremely concise feedback versus those favoring detailed explanations, or cultural differences in directness of criticism. Models trained on our dataset may produce reviews that feel inappropriate or jarring in contexts with different norms.
- **Evaluation Subjectivity:** Even expert reviews contain subjective elements that affect consistency and reproducibility of evaluations. Different experts may prioritize different aspects of code quality, provide varying levels of detail, or frame similar concerns using different language. While we employed multiple expert evaluators to mitigate individual biases, some degree of subjective variation remains inherent in any human evaluation process. This subjectivity introduces noise into both our training data and evaluation metrics, potentially affecting the reliability of performance comparisons.

Chapter 5

Conclusion

5.1 Summary of Contributions

This research presents a comprehensive comparative analysis of Large Language Models for automated code review, addressing critical gaps in evaluation methodology and dataset quality that have hindered progress in this domain. Our work makes several significant contributions to the field of LLM-based code review automation.

First, we established a rigorous evaluation framework that employs SBERT-based semantic similarity metrics, moving beyond surface-level evaluation approaches that have dominated prior work. By focusing on semantic alignment rather than lexi- cal matching, our methodology provides a more reliable assessment of whether AI-generated reviews truly capture the intent and substance of human feedback. This represents a methodological advancement that better reflects the real-world utility of automated code review systems.

Second, we conducted systematic benchmarking of five state-of-the-art LLMs—GPT- 4, LLaMA 3.1, CodeLLaMA, Qwen 2.5, and Mistral—using consistent prompting strategies and evaluation protocols. Our comparative analysis revealed that few-shot prompting consistently outperforms zero-shot and one-shot approaches across all models, with performance improvements ranging from 7.4% to 19.2%. GPT-4 demonstrated the strongest absolute performance and the most significant improvement with few- shot prompting (19.2%), while all models showed substantial gains when provided with multiple contextual examples.

Third, and most significantly, we developed a novel expert-curated benchmark dataset that addresses the quality limitations of existing public datasets. By engaging indus-

try professionals with 1–2 years of experience across diverse domains including fintech, healthcare, and e-commerce, we created a high-quality corpus of code review pairs that more accurately refiects professional review practices. The impact of this contribution is evident in our results: transitioning from the existing Review4Repair dataset to our expert-curated dataset resulted in dramatic performance improvements, with SBERT similarity scores increasing by 43.5% to 91.9% across all evaluated models. GPT-4's score nearly doubled from 0.22 to 0.38, confirming that dataset quality is a pivotal factor in accurately assessing LLM capability for code review tasks.

Finally, we developed and deployed a comprehensive web-based platform for systematic expert review collection, featuring a Next.js frontend with syntax highlighting and progress tracking, coupled with a Node.js/Express backend and MongoDB database. This infrastructure not only facilitated our current research but provides a reusable framework for future dataset expansion and continuous quality improvement.

5.2 Key Findings and Insights

Our empirical results yield several critical insights that advance understanding of LLM capabilities for automated code review:

1. **Dataset Quality Dominates Model Architecture:** The 2.5-3x improvement in similarity scores achieved solely through dataset quality enhancement demonstrates that training data quality has a more profound impact than model architecture selection. This finding suggests that the research community should prioritize investment in high-quality, expert-curated datasets alongside model development efforts.
2. **Few-Shot Learning Effectiveness:** The consistent superiority of few-shot prompting across all models indicates that LLMs benefit substantially from concrete examples that illustrate the expected review format, tone, and level of detail. This consistency across different architectures suggests that prompt engineering remains essential for practical deployment, regardless of underlying model capabilities.
3. **Semantic Metrics Reveal True Capability:** Our deliberate choice to focus on SBERT-based semantic similarity rather than traditional lexical metrics like BLEU revealed capabilities that surface-level measures obscure. Code review comments can express identical concerns through diverse vocabulary, and only semantic evaluation captures whether models truly understand underlying code

quality issues.

4. **Model-Specific Strengths Emerge:** While GPT-4 achieved the highest ab- solute performance, open-source alternatives like LLaMA 3.1 and Qwen 2.5 demonstrated competitive capabilities, particularly when provided with few- shot examples. This suggests that practical, cost-effective deployment of au- tomated code review is achievable without relying exclusively on proprietary models.

5.3 Implications for Research and Practice

The findings of this research carry significant implications for both academic investi- gation and industrial application of LLM-based code review automation.

5.3.1 Research Implications

For the research community, our work establishes a more reliable foundation for fu- ture investigation into automated code review. The demonstration that dataset quality profoundly impacts evaluation outcomes calls for renewed focus on benchmark cu- ration and validation. Future research should prioritize the development of diverse, high-quality datasets covering multiple programming languages, development con- texts, and review scenarios. Our SBERT-based evaluation methodology provides a template for semantic-level assessment that better refiects the practical utility of gen- erated reviews.

Additionally, our findings regarding few-shot learning effectiveness suggest promis- ing directions for prompt optimization research. The significant performance varia- tions we observed across different prompting strategies indicate that substantial gains may be achievable through systematic prompt engineering and meta-learning ap- proaches that enable models to rapidly adapt to project-specific review conventions.

5.3.2 Practical Implications

For practitioners, our results demonstrate that LLM-based code review tools have reached a level of maturity where thoughtful integration into development workfiows can provide genuine value. While complete automation remains premature, several practical applications are immediately viable:

- **Augmented Review Workflows:** Organizations can deploy AI-generated re-

views as a first screening layer, identifying common issues and allowing human reviewers to focus on architectural considerations and complex logic validation.

- **Developer Education:** High-quality AI feedback can serve as an on-demand learning resource for junior developers, providing immediate, consistent guidance on best practices and common pitfalls.
- **Quality Consistency:** Automated tools can ensure uniform adherence to organizational standards across large codebases and distributed teams, maintaining consistency where human attention may vary.

The substantial improvement achieved through expert-curated data also suggests that organizations investing in custom fine-tuning using internal code review history may achieve even better results tailored to their specific context and conventions.

5.4 Future Research Directions

While this work advances the state of automated code review, several important avenues for future investigation remain:

5.4.1 Dataset Expansion and Diversification

Our current dataset, while significantly higher quality than existing benchmarks, focuses primarily on Java and JavaScript code from open-source repositories. Future work should expand coverage to include:

- Additional programming languages, particularly Python, C++, and emerging languages like Rust and Go
- Enterprise and proprietary codebase characteristics that may differ from open-source patterns
- Domain-specific review practices for specialized fields such as embedded systems, data science, and mobile development
- Security-focused reviews requiring specialized expertise in vulnerability identification

Moreover, implementing semi-automated or active learning approaches could help scale dataset construction while maintaining quality, addressing the scalability limi- tations of purely manual expert annotation.

5.4.2 Context-Aware Review Generation

Current models struggle with project-specific conventions and architectural context that extends beyond individual code changes. Future research should explore:

- Integration of project documentation, existing codebase patterns, and historical review decisions into model context
- Multi-turn dialogue systems that can ask clarifying questions about architec- tural intent before providing feedback
- Retrieval-augmented generation approaches that incorporate relevant examples from project history
- Adaptive learning mechanisms that allow models to continuously improve based on developer feedback and evolving project standards

5.4.3 Fine-Tuning and Specialization

Our results strongly indicate that fine-tuning models on expert-curated datasets will yield substantially more human-like reviews. Promising directions include:

- Systematic investigation of fine-tuning strategies using our expert-curated dataset
- Development of specialized models for different review subdomains (security, performance, refactoring, style)
- Multi-task learning approaches that simultaneously optimize for different as- pects of code quality
- Cross-task knowledge distillation to transfer expertise across related review sce- narios

5.4.4 Evaluation Methodology Enhancement

While our SBERT-based approach represents an improvement over traditional met- rics, further refinement of evaluation methodology is needed:

- Development of standardized test suites covering diverse review scenarios and difficulty levels
- Integration of human-in-the-loop evaluation protocols that can continuously validate automated metrics

- Explainability frameworks that help developers understand why specific recommendations were generated
- Longitudinal studies assessing the impact of AI-assisted review on code quality, developer productivity, and learning outcomes

5.4.5 Real-World Deployment and Validation

The ultimate validation of automated code review systems requires deployment in actual development environments:

- Large-scale user studies with professional developers across diverse organizational contexts
- Integration studies examining how AI-generated reviews affect team dynamics and knowledge transfer
- Longitudinal assessment of whether AI assistance improves code quality met- rics and reduces post-release defects
- Investigation of optimal human-AI collaboration patterns that maximize the strengths of both

5.5 Concluding Remarks

Code review remains a critical but resource-intensive component of modern software development. While complete automation of this complex, context-dependent pro- cess remains distant, our research demonstrates that current LLMs, when properly benchmarked against high-quality datasets, show substantial capability for generat- ing semantically meaningful, human-aligned review comments.

The dramatic performance improvements achieved through dataset quality enhancement—with some models nearly doubling their similarity scores—underscore a cru- cial lesson: advances in automated code review depend as much on the quality of evaluation benchmarks and training data as on model architecture innovation. By es- tablishing a more rigorous evaluation framework and providing a high-quality bench- mark dataset, this work enables more reliable assessment of LLM capabilities and provides a foundation for continued progress in this domain.

As LLMs continue to evolve and datasets expand to cover broader contexts and languages, we anticipate increasingly sophisticated automated review systems that can

serve as intelligent assistants in the code review process. These systems will not replace human expertise but rather augment it, handling routine aspects of code quality assessment while freeing experienced developers to focus on design, architecture, and knowledge transfer.

The path forward requires continued collaboration between researchers, practition- ers, and tool developers to refine evaluation methodologies, expand high-quality train- ing data, and thoughtfully integrate AI capabilities into existing development work- fiows. Our work represents a step toward this future, demonstrating both the promise and the practical requirements for effective LLM-based code review automation.

Ultimately, the goal is not to eliminate human judgment from code review but to enhance it—providing developers with intelligent, context-aware assistance that improves both the efficiency and effectiveness of this essential software engineering practice. With continued research addressing the limitations identified in this work, automated code review systems have the potential to become indispensable tools in the modern developer's workfiow, contributing to higher code quality, faster delivery cycles, and more effective knowledge transfer within development teams.