An Evaluation of Automated Code Review Approaches

Shaquille Pearson

Abstract—Automated code review approaches have gained traction in software engineering, promising to streamline the review process by suggesting comments on code changes. However, verifying the semantic correctness of these automatically generated comments remains a significant challenge, often requiring manual intervention. Current evaluation methods frequently rely on exact text matches, which fail to account for the nuanced semantics of code review comments, leading to misclassification of valid suggestions. In this study, we explore an evaluation framework that leverages pre-trained language models to assess semantic equivalence between generated and ground-truth comments. By addressing this gap, our work aims to enhance the reliability of automated code review tools, reduce the need for manual verification, and foster more practical adoption in real-world development workflows.

Index Terms—Code Review, Automated Code Review, Semantic Equivalence, Pre-trained Language Models, GPT-4, all-miniLM-L6-v2, Comment Analysis, Software Engineering, Natural Language Processing, Embedding Models

1 Introduction

CODE reviews have become a cornerstone of modern software development, ensuring code quality, maintainability, and adherence to best practices. By fostering collaborative knowledge sharing, code reviews enable teams to identify defects early and reduce long-term maintenance costs [3]. However, the increasing complexity of software systems, coupled with the expanding size and diversity of development teams, has magnified the challenges of performing efficient and effective manual reviews. As projects grow, the need for automated solutions to augment the code review process has become increasingly apparent.

Automating aspects of code reviews not only alleviates the workload for developers but also enhances consistency and reduces human error. A particularly challenging aspect of automation is identifying semantically equivalent code review comments—feedback that conveys the same intent or addresses the same issue, even if expressed differently. This task is critical for ensuring that automated systems provide meaningful insights without overwhelming developers with redundant or irrelevant suggestions.

Early efforts in automating code reviews included tools like Gerrit [24] and Phabricator [7], which focused on facilitating collaboration but lacked semantic analysis capabilities. With the advent of embedding models like GPT-3 [5] and PLBART [1], there is an opportunity to address these gaps using state-of-the-art natural language understanding.

Advancements in machine learning and natural language processing (NLP) have spurred significant progress in automating code review tasks. Pre-trained language models, such as Sentence-BERT [23] and GPT-4 [18], have demonstrated the potential to understand and generate text in ways that closely mimic human reasoning. Prior studies by Tufano et al. [26] and Hong et al. [13] have shown that these models can assist in generating and evaluating code review comments. However, many existing approaches rely on exact text matching, which often misclassifies semantically correct comments as incorrect. This limitation underscores the need for methods capable of capturing semantic intent, particularly in programming-specific contexts.

Furthermore, while general-purpose models like all-miniLM-L6-v2 offer cost-effective solutions, their training on natural language data makes them less adept at understanding programming-related nuances. On the other hand, high-performance models like GPT-4 deliver exceptional semantic understanding but come with scalability challenges, including token-based pricing and computational overhead. These trade-offs highlight the importance of evaluating models not only for their effectiveness but also for their efficiency in real-world scenarios.

This study evaluates the effectiveness and efficiency of pre-trained language models in identifying semantically equivalent code review comments. By leveraging multiple prompts across *Comment-Only* and *Code & Comment* contexts, we aim to provide actionable insights into the models' capabilities and limitations. Specifically, we address the following research questions:

(RQ1) How effectively can we identify semantically correct code review comment recommendations? - Identifying semantically correct code review comments is vital for improving developer productivity and ensuring high-quality code reviews. Accurate automation reduces manual effort and minimizes the risk of overlooking meaningful insights. The evaluation results indicate that GPT-4 achieves a superior balance between Precision and Recall, particularly in the *Comment-Only* context, where it effectively captures semantic equivalence. In contrast, all-miniLM-L6-v2 exhibits strong Precision but struggles to generalize to programming-specific tasks due to its reliance on general natural language training.

- *Shaquille Pearson is with the David R. Cheriton School of Computer Science, University of Waterloo, Canada.*
E-mail: {s23pearson@uwaterloo.ca

Manuscript received date; revised date.

(RQ2) How efficiently can we identify semantically correct code review comment recommendations? - Efficient evaluations are essential for scalability, particularly for large-scale or continuous code review tasks. This question addresses whether automated methods can process inputs rapidly and cost-effectively. Both models exhibited rapid processing times, but GPT-4's token-based pricing limits scalability, while all-miniLM-L6-v2 offers a cost-effective alternative with trade-offs in performance.

By highlighting the strengths and limitations of GPT-4 and all-miniLM-L6-v2 in these contexts, this study provides actionable insights for developers, researchers, and tool builders aiming to enhance automated code review systems.

2 Background

Code reviews are a cornerstone of modern software engineering, serving as a critical process for identifying defects, improving code quality, and fostering collaboration within teams [3]. During a code review, developers examine changes made to the codebase, provide feedback through review comments, and refine the code based on this feedback. These comments often include suggestions, corrections, or clarifications aimed at improving the code.

A major challenge in automating code reviews lies in understanding the semantic equivalence of review comments. For example, two comments like "Add input validation" and "Check for invalid inputs" express the same intent but differ in phrasing. Identifying such equivalence is essential for ensuring that automated tools provide accurate and meaningful feedback, reducing the need for manual review.

Semantic equivalence detection requires advanced natural language understanding, as traditional evaluation methods relying on exact text matching often misclassify semantically correct comments as incorrect [13, 26]. This limitation has driven interest in leveraging pre-trained language models, which have demonstrated strong capabilities in natural language tasks.

Embedding models such as Sentence-BERT [23] and CodeT5 [27] have transformed the way text and code are analyzed, providing dense representations for similarity tasks. Similarly, frameworks like Powers et al. [20] have laid the groundwork for evaluating models using robust metrics such as Precision, Recall, and MCC.

However, programming contexts introduce unique challenges. Unlike general-purpose text, code review comments often depend on the surrounding code for context. Addressing this requires embedding models that incorporate programming language semantics, enabling them to capture the interdependencies between code and comments. These foundational concepts set the stage for evaluating pre-trained models, such as GPT-4 and all-miniLM-L6-v2, in identifying semantically equivalent code review comments.

3 Study Design

This section outlines the study design, focusing on (a) the motivation behind the selection of subject systems and communities, and (b) the data extraction, filtering, and analysis procedures that were employed.

3.1 Subject Systems/Communities

The subject systems in this study were chosen based on their relevance to automated code review evaluation tasks, specifically in the context of identifying semantically equivalent code review comments. The dataset utilized in this research consists of **100** manually labeled samples originally derived from prior work by Tufano et al. and Hong et al. [13, 26]. This dataset is uniquely positioned for evaluating semantic equivalence due to its careful curation and focus on Java code and associated review comments.

To ensure the validity and practicality of the study, the dataset was selected based on the following criteria:

- The data must be curated by experts to ensure high-quality labeling of semantically equivalent and non-equivalent comments.
- The dataset should represent realistic code review scenarios, with a balance between positive (semantically equivalent) and negative cases.
- The data must align with the scope of prior work, enabling continuity and comparability with existing research on automated code review tools.

3.2 Data Extraction

The data extraction process for this study was simplified by leveraging the replication package provided by Tufano et al. [26]. This package included a carefully curated dataset of 100 labeled samples, specifically designed for evaluating automated code review systems. The following steps outline how the dataset was utilized and prepared for analysis in this study.

(DE1) Access and Utilize the Replication Package.

The first step involved accessing the replication package provided by Tufano et al., which contains the raw data necessary for this study. This step was essential to ensure the validity and reproducibility of our analysis by relying on a pre-validated dataset curated by experts in the field. The dataset includes labeled examples of code review comments, categorized as semantically equivalent or non-equivalent, alongside the associated Java code snippets.

To implement this step, the replication package was downloaded from the publicly available repository specified in the original study. The dataset included **36** positive samples (semantically equivalent comments) and **64** negative samples. Each sample was formatted to include the target and predicted comments, ensuring compatibility with the evaluation framework of this study. The straightforward structure of the replication package allowed for seamless integration with our analysis pipeline without requiring extensive preprocessing.

(DE2) Align Dataset with Evaluation Framework.

After acquiring the replication package, the next step was to align the dataset with the specific evaluation framework employed in this study.

The alignment process involved organizing the data into configurations compatible with the study's prompts. Specifically:

- **Comment evaluations**, only target and prediction comments were extracted.

- **Code & comment evaluations**, the associated method-level Java code snippets were also included.

Each sample was formatted to fit the input requirements of the models and the ten prompts used for evaluation. The dataset's pre-curated nature minimized the need for extensive filtering or linking between elements, allowing us to focus on testing semantic equivalence under the defined experimental conditions.

By relying on this replication package, the study ensured consistency with prior work while maintaining a high standard of data quality. This streamlined approach also allowed us to allocate more resources to developing and refining the evaluation framework rather than data preprocessing.

3.3 Data Analysis

Once the data was prepared, it underwent analysis to evaluate the performance of two models, GPT-4 and all-miniLM-L6-v2, in identifying semantically equivalent code review comments. This analysis aimed to assess the effectiveness and efficiency of these models under varying levels of context. The following steps outline the methodology used in the data analysis pipeline.

(DA1) Model Selection and Justification.

The analysis relied on two models: GPT-4 and all-miniLM-L6-v2. These models were selected for their complementary strengths and alignment with the study's goals:

- **GPT-4**: Known for its state-of-the-art natural language understanding and generation capabilities [6], GPT-4 was used to evaluate semantic equivalence across a series of tailored prompts. Its ability to process complex inputs, including both code snippets and comments, made it a robust choice for this study.
- **all-miniLM-L6-v2**: A lightweight embedding model optimized for sentence similarity tasks [23], this model was used via the Hugging Face API to generate embeddings for comments. Its efficiency and scalability were crucial for conducting analyses on large datasets without requiring extensive computational resources.

These models were chosen to balance accuracy, interpretability, and resource efficiency. While GPT-4 excels in reasoning and contextual understanding, all-miniLM-L6-v2 provides a computationally efficient alternative for embedding-based similarity evaluations. This combination allowed us to investigate both high-resource and lightweight solutions to the problem of evaluating semantic equivalence.

(DA2) Evolution of Prompt Design. To evaluate the semantic equivalence of code review comments, i iteratively designed and refined a series of ten prompts. These prompts were divided into two main categories:

- **Comment-Only**: Comments were evaluated in isolation to test the models' ability to assess semantic equivalence using natural language alone.
- **Code & Comment**: Comments were accompanied by the relevant code snippet, providing additional context for more realistic evaluation scenarios.

Through this iterative refinement, i aimed to balance clarity, task focus, and alignment with real-world code review practices. The following examples illustrate the two primary categories of prompts:

Example 1: Comment-Only Prompt

```
"In the context of a code review,
two reviewers have made the following ↪
    ↪comments:
Reviewer 1: {target}
Reviewer 2: {prediction}

Do these comments point to the same ↪
    ↪issue?
Focus on the meaning and intent of the
comments.
Respond with 'yes' or 'no'."
```

Example 2: Code & Comment Prompt

```
"Below is a method-level Java code
snippet under review:
Code Snippet:
{input_value}

Two reviewers have provided feedback
about this code:
Reviewer 1: {target}
Reviewer 2: {prediction}

Based on the code and the comments, are↪
    ↪ both
reviewers pointing to the same issue?
Focus on whether the intent or meaning ↪
    ↪of the
comments is equivalent.
Respond with 'yes' or 'no'."
```

To evaluate semantic equivalence under varying conditions, i designed prompts for two key scenarios: comment-only and code & comment. The comment-only prompt (**Example 3.3**) isolates natural language understanding, focusing on whether the comments alone convey the same intent. In contrast, the code & comment prompt (**Example 3.3**) incorporates relevant code snippets to simulate real-world code review practices, where comments are analyzed alongside the underlying code for additional context.

(DA3) Metrics and Statistical Tests.

The performance of each model was evaluated using a range of metrics, including:

- **Precision and Recall**: To measure the models' accuracy in identifying semantically equivalent comments and their ability to capture all relevant cases.
- **F1-Score, Accuracy, and MCC (Matthews Correlation Coefficient)**: These metrics provided a holistic view of model performance, balancing precision and recall while accounting for true positives and negatives.
- **Cosine Similarity**: For all-miniLM-L6-v2, cosine similarity was used to quantify the semantic closeness between embeddings of target and prediction comments [23].

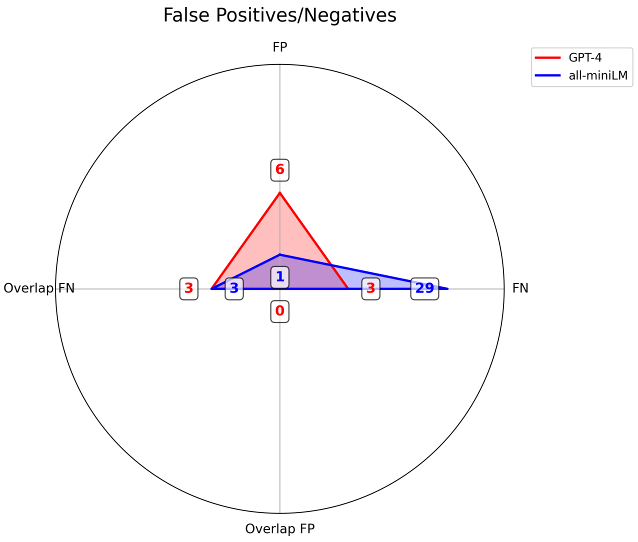

Fig. 1. Flase Positives/Negatives Comparison

By conducting a consistent evaluation across prompts and metrics, the study provided insights into how each model performs under varying contexts. This approach enabled a comprehensive comparison of GPT-4's contextual reasoning capabilities and all-miniLM-L6-v2's efficiency in embedding-based similarity evaluations.

4 Study Results

In this section, we present the results of our study with respect to our research questions. For each research question, we describe our approach to addressing it, followed by the results that we observe.

(RQ1) How effectively can we identify semantically correct code review comments recommendations?

RQ1: Approach.

To answer this research question, we evaluated two pretrained models GPT-4 and all-miniLM-L6-v2 on identifying semantically equivalent code review comments. Table 1 provides a detailed breakdown of model performance across all prompts. The evaluation was conducted under two contexts:

- **Comment-Only**: Models assessed comments without any associated code.
- **Code & Comment**: Models analyzed comments alongside method-level Java code snippets.

RQ1: Results.

The evaluation results, summarized in Table 1, highlight GPT-4's superior performance in identifying semantically correct code review comments. In the *Comment-Only* context, GPT-4 achieved its highest metrics, including an F1-Score of **0.8800**, MCC of **0.8098**, and Accuracy of **0.91**. These results underscore GPT-4's strong natural language understanding capabilities, effectively capturing semantic intent without the need for additional code context.

In the *Code & Comment* context, GPT-4 exhibited a marginal improvement in Recall (e.g., **0.8889** for Prompts 7 and 9), suggesting that code snippets provide some utility in resolving ambiguous cases. However, the overall gains were minimal, indicating that GPT-4 performs effectively with textual content alone in most scenarios.

In contrast, the all-miniLM-L6-v2 model demonstrated strong Precision (**0.8750**) in the *Comment-Only* context but struggled with Recall (**0.1944**). This disparity likely stems from its general natural language training, making it less equipped to interpret programming-specific comments without explicit code-review context.

Figure 1 presents a radar chart comparing false positives (FP), false negatives (FN), and their overlaps for the two models. GPT-4 exhibited a higher number of false positives (6) compared to the all-miniLM-L6-v2 model (3), with minimal overlap between the two (1 shared instance). Conversely, the all-miniLM-L6-v2 model had significantly more false negatives (29) than GPT-4 (3), with 3 instances of overlap.

These results reveal distinct trade-offs between the models. GPT-4 excels at minimizing false negatives, making it more reliable in capturing semantically correct comments, whereas the all-miniLM-L6-v2 model demonstrates caution with false positives, albeit at the cost of Recall. The limited overlap in misclassification types highlights differences in their underlying classification strategies, which could be attributed to their training architectures and contextual understanding.

Overall, GPT-4's balance of Precision, Recall, and F1-Score makes it the more robust option for identifying semantically correct code review comments. Its ability to handle most cases with textual content alone emphasizes its utility for automated code review tasks. The findings also suggest potential opportunities for complementary ensemble techniques, leveraging GPT-4's strength in reducing false negatives and all-miniLM-L6-v2's focus on avoiding false positives.

> *GPT-4 delivers the best balance of Precision, Recall, and F1-Score, particularly excelling in the Comment-Only context (F1-Score: 0.8800, MCC: 0.8098). By contrast, all-miniLM-L6-v2, while achieving strong Precision, struggles with Recall due to its lack of programming-specific training, underscoring GPT-4's suitability for automated code review tasks.*

(RQ2) How efficiently can we identify semantically correct code review comments recommendations?

RQ2: Approach.

To evaluate efficiency, we compared two pre-trained models GPT-4 and all-miniLM-L6-v2 against manual analysis in terms of *processing time* and *cost implications*. The evaluation spanned ten prompts across both *Comment-Only* and *Code & Comment* contexts.

Efficiency factors considered include:

- **Processing Time:** Manual evaluations typically require several minutes to hours for a single pair

TABLE 1
Evaluation Results for GPT-4 and all-miniLM-L6-v2 Across Different Prompts

Model	Prompt	Precision	Recall	F1-Score	Accuracy	MCC
all-miniLM-L6-v2	Prompt (Comment Only)	0.8750	0.1944	0.3165	0.8824	0.8119
GPT-4	Prompt 1 (Code & Comment)	0.8333	0.4167	0.5556	0.76	0.4620
	Prompt 2 (Comment)	0.9231	0.3333	0.4898	0.75	0.4535
	Prompt 3 (Code & Comment)	0.7750	0.8611	0.8158	0.86	0.7059
	Prompt 4 (Comment)	0.8056	0.8056	0.8056	0.86	0.6962
	Prompt 5 (Code & Comment)	0.7561	0.8611	0.8052	0.85	0.6879
	Prompt 6 (Comment)	0.8235	0.7778	0.8000	0.86	0.6931
	Prompt 7 (Code & Comment)	0.8000	0.8889	0.8421	0.88	0.7485
	Prompt 8 (Comment)	0.8049	0.9167	0.8571	0.89	0.7726
	Prompt 9 (Code & Comment)	0.7805	0.8889	0.8312	0.87	0.7303
	Prompt 10 (Comment)	0.8462	0.9167	0.8800	0.91	0.8098

of comments, depending on complexity [2, 17]. In contrast, GPT-4 and all-miniLM-L6-v2 processed responses in near real-time.

- **Cost Considerations:** GPT-4's API incurs token-based costs, which can accumulate for large-scale tasks [19]. On the other hand, all-miniLM-L6-v2, accessed via the Hugging Face API, is free and computationally lightweight [8, 22].

RQ2: Results.

The results highlight the following:

- **Processing Time:**
  - **Manual Analysis:** On average, manual evaluations took **5 to 10 minutes** per pair of comments [17].
  - **GPT-4:** The API delivered responses in approximately *1 to 2 seconds* per prompt [19].
  - **all-miniLM-L6-v2:** Responses were returned in *under 1 second* on average, due to its lightweight architecture [22].
- **Cost Considerations:**
  - **GPT-4:** GPT-4 incurs a cost of approximately *0.03 per 1,000 tokens* [19], which limits its scalability for large-scale or repeated tasks.
  - **all-miniLM-L6-v2:** This model remains free to use and lightweight, making it a cost-effective alternative [8].

> *Compared to manual evaluation, both models exhibit significant efficiency improvements. GPT-4 reduces evaluation time from several minutes to just **1-2 seconds** per prompt but incurs token-based costs. In contrast, all-miniLM-L6-v2 provides responses in under **1 second** and remains cost-free, albeit at the expense of reduced accuracy in programming-specific evaluations.*

5 Practical Implications

This study provides valuable insights for developers, researchers, and tool builders aiming to enhance automated code review processes:

1. **Prioritize tools with strong semantic understanding.** Models like GPT-4 demonstrated exceptional performance in identifying semantically equivalent code review comments, particularly in the *Comment-Only* context. Developers can integrate such models into their workflows to streamline code reviews, reduce manual effort, and maintain high-quality feedback.
2. **Balance precision and recall in evaluations.** The limitations of models like all-miniLM-L6-v2, particularly in Recall, highlight the importance of balanced evaluation metrics. Researchers should design experiments that assess both Precision and Recall, ensuring a comprehensive understanding of model behavior in nuanced scenarios such as automated code reviews.
3. **Optimize scalability for high-performance models.** While GPT-4 delivers exceptional semantic understanding, its scalability is limited by cost and computational demands [18]. In contrast, lightweight models like all-miniLM-L6-v2 offer cost-effective solutions but often underperform in nuanced programming tasks. This trade-off aligns with findings in scalability-focused studies, such as Li et al. [16].

6 Threats to Validity

This section identifies potential threats to the validity of our study and outlines the measures taken to mitigate their impact.

6.1 Construct Validity

A potential threat to construct validity arises from the extent to which the chosen evaluation metrics (e.g., Precision, Recall, F1-Score) capture the models' ability to identify semantic equivalence in code review comments. This threat may emerge if these metrics fail to account for nuances in human interpretation, such as contextual subtleties or implicit meanings within comments. To address this, the models were evaluated across multiple prompts and contexts, ensuring a comprehensive reflection of their semantic understanding capabilities.

6.2 Internal Validity

A threat to internal validity concerns the possibility that the observed performance differences between GPT-4 and all-miniLM-L6-v2 are influenced by factors unrelated to their semantic understanding abilities, such as prompt design or API configurations. This risk could arise if the prompts inadvertently favor one model over the other. To mitigate this, the prompts were iteratively refined to ensure consistency and neutrality across both models, minimizing any unintended bias.

6.3 External Validity

The primary threat to external validity stems from the limited dataset size (100 manually labeled samples), which may not fully represent the diversity of real-world code review scenarios. This threat could manifest if the models' performance varies significantly when applied to larger or more diverse datasets. While this limitation exists, the dataset was curated from prior research to ensure high-quality and realistic code review contexts. Consequently, the findings remain meaningful within the scope of this study, though additional evaluations on larger datasets are recommended for broader generalizability.

7 Related Work

The challenge of improving code review processes has long been recognized as a critical task in software engineering. Early efforts primarily relied on manual reviews, which, while thorough, were resource-intensive and prone to subjective biases. As software systems grew in complexity, the need for automated solutions became apparent. This need led to the development of tools such as ReviewerBot [**?**], which relied on rule-based heuristics to generate code review comments. While effective in addressing simple patterns, ReviewerBot lacked adaptability to nuanced and context-specific scenarios, limiting its practical applicability.

Tools like Gerrit [24] and Phabricator [7] have been widely adopted for collaborative code reviews. These tools facilitate team collaboration and codebase management but rely heavily on manual processes. This reliance has prompted researchers to explore machine learning-based approaches, such as Tufano et al. [26] and Hong et al. [13], who demonstrated the potential of pre-trained models for automating review comments. Despite these advancements, achieving semantic equivalence in code review comments remains an open problem, often requiring manual intervention for validation.

Recent advancements in machine learning and natural language processing (NLP) have paved the way for more sophisticated approaches. CommentFinder [**?**] leveraged machine learning models trained on curated datasets to recommend comments with higher accuracy and relevance. However, its reliance on exact string matches in evaluation often led to the misclassification of semantically equivalent comments. Similarly, Li et al. [15] introduced AUGER, which applies pre-trained models to automate review comment generation but faces challenges in nuanced cases where comments are context-dependent.

Parallel to these developments, embedding models revolutionized NLP and programming tasks by representing text as dense vector representations. Sentence-BERT [23], widely used for semantic similarity tasks, demonstrated the utility of embedding models in capturing natural language semantics. However, models like Sentence-BERT struggled in programming-specific contexts due to their training on general language corpora. To address these limitations, programming-aware models such as CodeBERT [10], GraphCodeBERT [11], and CodeT5 [27] were introduced. These models incorporate programming language semantics, enabling better performance in tasks such as code summarization, defect prediction, and code-to-code retrieval.

Recent research has also explored hybrid models and methods. Ahmad et al. [1] proposed PLBART, which integrates programming and natural language semantics, achieving state-of-the-art performance in program repair and translation. Similarly, UniXcoder [12] offers a unified framework for understanding and generating programming languages, addressing challenges in multi-purpose downstream tasks. Hybrid approaches leveraging knowledge graphs [14] and pre-training on domain-specific datasets [21] have shown promise in bridging the gap between general-purpose and programming-specific embeddings.

Evaluation metrics play a critical role in assessing the effectiveness of these approaches. Powers [20] introduced robust metrics such as Precision, Recall, and MCC, which have become standard for evaluating semantic similarity tasks. Fang et al. [9] surveyed metrics for evaluating program analysis tools, identifying challenges in balancing computational cost and interpretability. Li et al. [16] highlighted the scalability challenges associated with training and deploying deep learning models in software engineering contexts.

While the field has made significant strides, key challenges remain. Studies such as Tufano et al. [25] highlight the potential for deep learning models to predict bug-fixing patches, suggesting synergies between automated code reviews and defect prediction. However, as highlighted by Beller et al. [4], integrating these approaches into CI/CD workflows introduces scalability and performance trade-offs.

8 Conclusions

This study investigated the effectiveness and efficiency of automated methods for identifying semantically equivalent code review comments—a crucial task for enhancing the code review process in software engineering. We evaluated the performance of two pre-trained models, GPT-4 and all-miniLM-L6-v2, across multiple contexts to assess their suitability for this task.

The evaluation was conducted using ten prompts spanning *Comment-Only* and *Code & Comment* contexts, with metrics such as Precision, Recall, F1-Score, Accuracy, and MCC. GPT-4 demonstrated a robust capability for identifying semantic equivalence with balanced performance across metrics, while all-miniLM-L6-v2 provided a cost-effective yet less nuanced alternative.

- **GPT-4 achieves the best balance between Precision and Recall**, particularly in the *Comment-Only* context, making it highly effective for identifying semantic equivalence in code review comments.
- **Code snippets add marginal benefits**, resolving edge cases but providing only slight improvements in overall performance.

- **All-miniLM-L6-v2 offers a cost-effective alternative**, though it struggles with programming-specific nuances due to its reliance on general natural language training.
- Cost considerations underscore the importance of scalable solutions for employing high-performance models like GPT-4 in large datasets or continuous evaluation settings.

These findings offer actionable insights for developers, researchers, and tool builders, emphasizing the need to balance quality, cost, and scalability in designing automated code review systems.

Future Work.

This study has demonstrated the potential of pre-trained language models such as GPT-4 [18] and all-miniLM-L6-v2 [23] in identifying semantically equivalent code review comments. However, it also highlighted the limitations of general-purpose models like all-miniLM-L6-v2 when applied to programming-specific contexts.

Future research will explore embedding models tailored for programming tasks, such as CodeT5 [27], GraphCodeBERT [11], and UniXcoder [12], which integrate programming language semantics into their training. Evaluating these models alongside general-purpose embeddings will provide deeper insights into their effectiveness in capturing the nuanced intent behind code review comments.

Additionally, extending the dataset to encompass a broader variety of programming languages and review scenarios will enhance the generalizability of findings. This expansion will help identify strengths and weaknesses of emerging models across diverse programming contexts, enabling the development of more effective automated code review systems.

References

[1] W. U. Ahmad, S. Chakraborty, and B. Ray, "Unified pre-training for program understanding and generation," *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, pp. 8305–8325, 2021.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[3] ——, "Expectations, outcomes, and challenges of modern code review," *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 712–721, 2013.

[4] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An exploratory study of ci/cd challenges in large-scale projects," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 2, pp. 182–198, 2017.

[5] T. B. Brown *et al.*, "Language models are few-shot learners," 2020.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models Are Few-Shot Learners," 2020.

[7] R. Ellis, A. Matthews, and R. Calloway, "Phabricator: A review tool for collaborative software development," *Proceedings of the 2014 ACM Conference on Computer Supported Cooperative Work*, pp. 478–483, 2014.

[8] H. Face, "all-minilm-l6-v2," 2023, available at https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2.

[9] Q. Fang, D. Lee, and X. Yu, "A survey on the metrics for evaluating program analysis tools," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 44, no. 3, pp. 1–32, 2022.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1536–1547, 2020.

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, N. Duan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020, pp. 8581–8594.

[12] ——, "UniXcoder: Unified Pre-trained Model for Multi-Purpose Programming Language Understanding and Generation," 2022, available at https://arxiv.org/abs/2203.04821.

[13] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "CommentFinder: A Simpler, Faster, More Accurate Code Review Comments Recommendation," in *the Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 507–519.

[14] Z. Hu, Y. Dong, K. Wang, K. Chang, and Y. Sun, "Heterogeneous graph embedding for code representation," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2019, pp. 2423–2432.

[15] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "AUGER: Automatically Generating Review Comments with Pre-Training Models," in *the Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 1009–1021.

[16] X. Li, J. Chen, X. Li, and H. Zhang, "Deep learning for scalability in software engineering: A systematic literature review," *Journal of Systems and Software*, vol. 133, pp. 201–215, 2017.

[17] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[18] OpenAI, "GPT-4 Technical Report," 2023, available at https://openai.com/research/gpt-4.

[19] ——, "Pricing for gpt-4 api," 2023, available at https://platform.openai.com/pricing.

[20] D. M. W. Powers, "Evaluation: From precision, recall,

and f-measure to roc, informedness, markedness, and correlation," in *Journal of Machine Learning Technologies*, vol. 2, 2011, pp. 37–63.

[21] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2020.

[22] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2019.

[23] ——, "Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks," 2019.

[24] P. C. Rigby, C. Bird, A. Hindle, D. M. German, and M.-A. Storey, "Using gerrit for code reviews in open source and industry," *IEEE Software*, vol. 30, no. 6, pp. 46–52, 2013.

[25] M. Tufano, C. Watson, and D. Poshyvanyk, "Deep learning for automatic code review: Predicting bug fixes from code changes," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 284–294.

[26] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using Pre-Trained Models to Boost Code Review Automation," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2022, pp. 2291–2302.

[27] Y. Wang, W. Liu, S. Joty, and A. Cohan, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2021, pp. 8696–8708.

APPENDIX A
PROMPTS USED IN THE STUDY

This appendix provides the full set of ten prompts used to evaluate pre-trained language models for identifying semantically equivalent code review comments. The prompts are presented in the order they were used.

Prompt 1 (Code & Comment):

```
"The input field is a code snippet ↪
    ↪related to
the comment in the Target.
Input Field: {input_value}

Given this context, are the following ↪
    ↪comments
semantically equivalent?
Respond with 'yes' or 'no'.

Target: {target}
Prediction: {prediction}"
```

Prompt 2 (Comment):

```
"Are the following code review comments
semantically equivalent?
Respond with 'yes' or 'no'.

Target: {target}
Prediction: {prediction}"
```

Prompt 3 (Code & Comment):

```
"Assuming the code snippet below is ↪
    ↪being
reviewed.
Code Snippet: {input_code}

If Reviewer 1 makes the comment:
{target_comment}

And Reviewer 2 makes the comment:
{predicted_comment}

Are these two comments pointing to the
same issue?
Respond with 'yes' or 'no'."
```

Prompt 4 (Comment):

```
"In the context of code review in ↪
    ↪software
projects,
If Reviewer 1 makes the comment:
{target_comment}

And Reviewer 2 makes the comment:
{predicted_comment}

Are these two comments pointing to the ↪
    ↪same
issue?
Respond with 'yes' or 'no'."
```

Prompt 5 (Code & Comment):

```
"Assuming the method-level Java code ↪
    ↪snippet
below is being reviewed.
Code Snippet: {input_value}

If Reviewer 1 makes the comment:
{target}

And Reviewer 2 makes the comment:
{prediction}

Are these two comments pointing to the ↪
    ↪same
issue?
Respond with 'yes' or 'no'."
```

Prompt 6 (Comment):

```
"In the context of code review in Java ↪
    ↪software
projects,
If Reviewer 1 makes the comment:
{target}

And Reviewer 2 makes the comment:
{prediction}

Are these two comments pointing to the ↪
    ↪same
issue?
Respond with 'yes' or 'no'."
```

Prompt 7 (Code & Comment):

```
"Below is a method-level Java code ↪
    ↪snippet
under review:
Code Snippet:
{input_value}

Two reviewers have provided feedback ↪
    ↪about this
code:
Reviewer 1: {target}
Reviewer 2: {prediction}

Based on the code and the comments, are↪
    ↪ both
reviewers
pointing to the same issue? Focus on ↪
    ↪whether
the intent
or meaning of the comments is ↪
    ↪equivalent.
Respond with 'yes' or 'no'."
```

Prompt 8 (Comment):

```
"In the context of a code review, two ↪
    ↪reviewers
have made the following comments:
Reviewer 1: {target}
Reviewer 2: {prediction}

Do these comments point to the same ↪
    ↪issue?
Focus on the
meaning and intent of the comments.
Respond with 'yes' or 'no'."
```

Prompt 9 (Code & Comment):

```
"You are a reviewer analyzing a method-↪
    ↪level
Java code snippet under review.
Your task is to determine whether the ↪
    ↪following
two reviewers are pointing to the same ↪
    ↪issue
in their feedback:

Code Snippet:
{input_value}

Reviewer 1: {target}
Reviewer 2: {prediction}

Focus on whether the intent or meaning ↪
    ↪of the
comments is equivalent.
Write 'yes' if both comments are ↪
    ↪pointing to
the same issue or 'no' if they are not↪
    ↪."
```

Prompt 10 (Comment):

```
"You are a reviewer analyzing two code ↪
    ↪review
comments.
Your task is to determine whether both ↪
    ↪comments
are pointing to the same issue.

Reviewer 1: {target}
Reviewer 2: {prediction}

Focus on the meaning and intent of the
comments.
Write 'yes' if both comments are ↪
    ↪pointing to
the same issue or 'no' if they are not↪
    ↪."
```