# AUTOMATED CODE REVIEW: EMPIRICAL EVIDENCE FROM EXPERIMENTS AND INDUSTRY

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Umut Cihan
April 2025

Automated Code Review: Empirical Evidence from Experiments and Industry
By Umut Cihan
April 2025

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray Tüzün (Advisor)

Uğur Doğrusöz

İsmail Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

# ABSTRACT

## AUTOMATED CODE REVIEW: EMPIRICAL EVIDENCE FROM EXPERIMENTS AND INDUSTRY

Umut Cihan
M.S. in Computer Engineering
Advisor: Eray Tüzün
April 2025

Code reviews are essential for software quality. Advances in large language models (LLMs) have enabled AI-powered code reviews, but their reliability and impact on the industry remain unclear. This thesis evaluates LLMs for detecting code correctness and suggests improvements while assessing the adoption of AI-assisted code review tools in practice. The thesis consists of two studies: an experimental evaluation of LLMs in code review and a case study on real-world AI-assisted code reviews. In the experiment, GPT-4o and Gemini 2.0 Flash were tested on 492 AI-generated code blocks and 164 HumanEval benchmark blocks. The models assessed correctness and suggested fixes, with GPT-4o achieving 68.50% accuracy in classification and 67.83% in corrections, outperforming Gemini 2.0 Flash (63.89% and 54.26%). Performance dropped without problem descriptions and varied across code types. The case study examined an AI-assisted code review tool based on Qodo PR Agent, deployed to 238 practitioners across ten projects, in Beko Corporation. The analysis focused on 4,335 pull requests, of which 1,568 received automated reviews. Developers engaged with 73.8% of AI-generated comments, though pull request closure time increased. Surveys indicated minor improvements in code quality but highlighted issues such as faulty suggestions and increased review time. LLM-based code reviews aid in detecting issues and improving code but risk errors. A "Human-in-the-loop" approach is proposed to balance automation with oversight. Despite challenges, AI-assisted reviews enhance bug detection and code awareness, offering valuable, albeit imperfect, integration into software development workflows.

*Keywords:* Code Review, Large Language Models, Pull Requests, AI-Assisted Code Review, Industry Case Study, Code Review Automation.

# ÖZET

# OTOMATIK KOD GÖZDEN GEÇIRME: DENEYLERDEN VE ENDÜSTRIDEN ELDE EDILEN BULGULAR

Umut Cihan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
Nisan 2025

Kod gözden geçirme, yazılım kalitesi için kritik bir süreçtir. Büyük dil modellerindeki (BDM) gelişmeler, otomatik kod gözden geçirmelerini mümkün kılmıştır, ancak güvenilirlikleri ve endüstrideki etkileri belirsizdir. Bu tez, BDM'lerin kod doğruluğunu tespit etme ve iyileştirme yeteneklerini değerlendirirken, BDM destekli kod gözden geçirme araçlarının yazılım endüstrisindeki uygulamalarını inceler. Bu tez, BDM'lerin kod gözden geçirme görevlerindeki performansını değerlendiren deneysel bir çalışma ile BDM destekli kod gözden geçirmelerinin endüstri kullanımını analiz eden bir vaka çalışmasından oluşmaktadır. Deneysel çalışmada, GPT-4o ve Gemini 2.0 Flash, 492 yapay zeka tarafından üretilmiş kod bloğu ve 164 HumanEval veri seti kod bloğunda test edilmiştir. GPT-4o, doğruluk tespitinde %68,50, düzeltmede %67,83 başarı sağlarken, Gemini 2.0 Flash sırasıyla %63,89 ve %54,26 oranlarına ulaşmıştır. Koda dair açıklamalar olmadan performans düşmüş, sonuçlar kod türüne göre değişmiştir. Vaka çalışması, Qodo PR Agent tabanlı bir BDM destekli kod gözden geçirme aracının, Beko şirketinde, 10 projede 238 geliştirici tarafından kullanımını incelemiştir. Analiz edilen 4.335 pull request'in 1.568'i BDM yorumu almıştır. Geliştiriciler, BDM yorumlarının %73,8'ini dikkate almış ancak kod gözden geçirme süresi uzamıştır. Anketler, kod kalitesinde küçük iyileşmeler olduğunu ancak hatalı veya gereksiz öneriler gibi sorunlar yaşandığını göstermiştir. Sonuç olarak BDM tabanlı kod gözden geçirmeleri hata tespiti ve iyileştirmelerde faydalıdır, ancak hatalı sonuçlar üretebilir. Bu riskleri azaltmak için "İnsan-Döngüsünde BDM Kod Gözden Geçirme" önerilmektedir. BDM destekli gözden geçirmeler, bazı eksiklerine rağmen hata tespitini ve kod farkındalığını artırarak yazılım geliştirmeye değerli katkılar sunmaktadır.

*Anahtar sözcükler*: Kod Gözden Geçirme, Büyük Dil Modelleri, Çekme İstekleri, Yapay Zeka Destekli Kod Gözden Geçirme, Endüstri Vaka Çalışması, Kod Gözden Geçirme Otomasyonu.

# Acknowledgement

I wish to convey my deepest gratitude to my supervisor, Assistant Professor Eray Tüzün. His invaluable guidance, support, and the confidence he consistently showed in my abilities were essential throughout my academic endeavors. His profound understanding and skill have been crucial in the development of both my scholarly and individual self.

Furthermore, I extend my sincere appreciation to the distinguished members of my examination committee, Professor Dr. Uğur Doğrusöz and Professor Dr. İsmail Sengör Altıngövde. I am thankful for the significant time they dedicated to examining my work and for their thoughtful consideration.

My heartfelt thanks also go to my beloved family: my mother, Nesrin; my father, Mustafa; and my sister, Melis. Their affection and backing have been a constant source of inspiration and strength during this entire process.

I am truly grateful to my friends, Murat, Dursun, and Ege, for their continuous support and friendship. My special thanks go to İrem for always being there for me, motivating me to be my best, and inspiring me every day.

Lastly, I want to sincerely thank Arda, Vahid, and the other colleagues at BILSEN, as well as Mert, Ömercan, Emircan, and Baykal from Beko. Their teamwork, insightful contributions, and helpful critiques were invaluable to my research.

*Umut*
*April 2025, Ankara*

# Contents

**1 Introduction** **1**

1.1 Research Problem . . . . . 2

1.2 Contribution of the Thesis . . . . . 3

**2 Background and Related Work** **5**

2.1 Code Review . . . . . 5

2.2 Large Language Models (LLMs) . . . . . 6

2.3 Related Work . . . . . 7

**3 Methodology of the Experimental Study** **9**

3.1 Dataset . . . . . 10

3.2 Test Setup . . . . . 10

3.3 Evaluation Metrics . . . . . 11

3.4 Variables . . . . . 13

**4 Methodology of the Case Study 15**

4.1 Study Object . . . . . 16

4.2 Case Study Goal and Research Questions . . . . . 18

4.3 Data Collection From Azure DevOps . . . . . 20

4.3.1 Initial Pull Request . . . . . 21

4.3.2 Pull Request Comments . . . . . 21

4.3.3 Closed Pull Request . . . . . 22

4.3.4 Data Extraction and Analysis . . . . . 22

4.4 Data Collection From Surveys . . . . . 23

4.4.1 Code Review Surveys . . . . . 23

4.4.2 General Opinion Surveys . . . . . 24

4.5 Approach Towards Research Questions . . . . . 24

**5 Results of the Experimental Study 26**

5.1 Mixed Dataset Experiment Results . . . . . 27

5.2 Ground Truth Dataset Experiment Results . . . . . 28

**6 Results of the Case Study 32**

6.1 Code Review Surveys . . . . . 32

6.2 General Opinion Surveys . . . . . 33

6.3 Analysis of the Azure DevOps Data . . . . . 35
6.3.1 Comment Labels . . . . . 35
6.3.2 Commits on Pull Requests . . . . . 36
6.3.3 Pull Request Closure Durations . . . . . 38
6.3.4 Human Reviewer Comments . . . . . 39
6.4 General Opinion Survey Open-Ended Questions . . . . . 41

**7 Discussion 43**

7.1 Revisiting Research Question 1.1 . . . . . 43
7.2 Revisiting Research Question 1.2 . . . . . 44
7.3 Human-in-the-loop LLM Code Review . . . . . 44
7.4 Does LLM-based automated code review considerably improve software development? . . . . . 46
7.5 Implications of the Case Study to Practitioners . . . . . 47
7.6 Implications of the Case Study to Researchers . . . . . 48

**8 Threats to Validity 50**

8.1 Threats to Validity of the Experimental Study . . . . . 50
8.1.1 Internal Validity . . . . . 50
8.1.2 External Validity . . . . . 51
8.1.3 Construct Validity . . . . . 51

8.1.4 Conclusion Validity . . . . . 51

8.2 Threats to Validity of the Case Study . . . . . 52

8.2.1 Internal Validity . . . . . 52

8.2.2 External Validity . . . . . 53

8.2.3 Construct Validity . . . . . 54

8.2.4 Conclusion Validity . . . . . 54

**9 Conclusion** **55**

# List of Figures

3.1 Test Setup . . . . . 11

3.2 Prompt Template . . . . . 12

4.1 Data Collection Timeline . . . . . 15

4.2 Example Review of CodeReviewBot . . . . . 16

4.3 Flow of Pull Request Process . . . . . 17

4.4 Data Collection Overview . . . . . 20

5.1 Correctness Accuracy (%) . . . . . 27

5.2 False Positive Rates (%) . . . . . 28

5.3 False Negative Rates (%) . . . . . 29

5.4 Correction Ratios (%) . . . . . 30

5.5 Regression Ratios (%) . . . . . 30

5.6 Correctness Accuracy (%) with Ground Truth . . . . . 31

5.7 Regression Ratios (%) with Ground Truth . . . . . 31

6.1 Automated Code Review Ratings from Code Review Surveys . . . 33

6.2 Code Review Survey . . . . . 34

6.3 General Opinion Survey Responses (Questions 1-3) . . . . . 35

6.4 General Opinion Survey Responses (Questions 4-6) . . . . . 36

6.5 Comment Labels on Merged PRs . . . . . 37

6.6 Commits After Code Reviews . . . . . 38

6.7 Pull Request Closure Durations Before and After CodeReviewBot (H:MM) . . . . . 39

6.8 Average Number of Human Reviewer Comments . . . . . 40

7.1 Human-in-the-loop LLM Code Review Process . . . . . 45

# List of Tables

3.1 Experiment Configurations . . . . . 14

4.1 Projects Included in Our Study . . . . . 17

4.2 Comment Resolution Policy in Azure DevOps . . . . . 20

4.3 Survey Participants . . . . . 24

# Chapter 1

# Introduction

Code review, a quality assurance process with diverse industry adaptations, involves developers inspecting each other's code modifications [1]. Originating as formal code inspections [2], code review has transitioned into a more streamlined practice known as Modern Code Review (MCR) [3]. MCR is characterized by its informality, tool-based implementation, and routine application [4]. The process commonly yields benefits such as knowledge dissemination, enhanced learning, defect detection, and code refinement [5, 6].

Code review necessitates developers investing time to comprehend their colleagues' work and its contextual relevance within the broader project. Studies indicate varying time commitments to this process. Open Source Software (OSS) developers self-reported an average of 6.4 hours per week dedicated to code review [7], while a Google case study reported a lower average of 3.2 hours [6]. These figures underscore the substantial time investment developers allocate to code reviews.

Developer workloads often lead to delays in code review tasks, consequently postponing the merging of code changes. While Google reports median merge approval times of under four hours, and five hours for large change sets [6], other

projects and companies exhibit significantly longer durations. Studies report median approval times of 15.7 hours for Google Chrome, 20.8 hours for Android at Google, 17.5 hours for AMD, 14.7 hours for Bing, 19.8 hours for SQL, and 18.9 hours for Office at Microsoft [8]. This variation highlights inter-company and inter-project differences. However, median and average statistics fail to capture the true costs associated with delays in critical code changes, which can be substantial. Indeed, a study of Microsoft developers identified timely feedback as the foremost challenge in code review practices [5].

Automation presents the potential to expedite code approval and alleviate the manual effort required from developers. Consequently, numerous initiatives have explored the automation of the code review process [9–19], focusing mainly on the assignment of reviewers [20]. These efforts come from tools that generate code reviews [10–12], detect process smells [21, 22], predict the approval of code changes [23, 24], and fix code according to code reviews [25, 26]. In our study, we focus on the automation of code review generation.

Developers dedicate significant time to code reviews, scrutinizing changes for errors, performance issues, and coding standard violations. While pre-trained models have been investigated for automated code review [23, 24, 27], the advent of ChatGPT [28] has accelerated this trend. This has resulted in the creation of numerous tools, like CodeRabbit and Codium [29, 30], that utilize powerful language models like OpenAI's GPT-4 for automated code review generation [31].

## 1.1 Research Problem

The adoption of automated code review tools is on the rise within the industry [31]. However, a critical gap exists in empirical studies that validate their claimed advantages. The introduction of novel issues may diminish the time efficiencies gained, and the economic benefits associated with decreased developer effort might be eclipsed by the ongoing costs of running these systems. To address this research gap, we conducted a study aimed at answering the following

research questions:

RQ1: How capable are large language models at performing code reviews?

- RQ1.1: How accurately can large language models (LLMs) evaluate code changes for approval or rejection?
- RQ1.2: How effective are the code improvement suggestions generated by large language models (LLMs) in improving code correctness?

RQ2: What are the practical impacts and effectiveness of integrating LLM-based automated code review tools in professional software development environments?

- RQ2.1: How useful are LLM-based automated code reviews in the context of the software industry?
- RQ2.2: How do LLM-based automated code reviews impact the pace of the pull request closure process?
- RQ2.3: How does the introduction of LLM-based automated code reviews influence the volume of human code review activities?
- RQ2.4: How do developers perceive the LLM-based automated code review tools?

## 1.2 Contribution of the Thesis

This thesis combines two distinct studies, each of which addresses one of the main research questions. The first research question is addressed with an experimental study, while the second research question is addressed with an industry case study.

To address the first research question, we developed a setup that uses code blocks along with their unit tests. The setup prompts the LLM to assess code correctness and suggest improvements for each code block. We evaluate the LLM's assessment against unit test results and test its suggestions with the same

unit tests. A suggestion is a correction if the new code passes all unit tests. In code reviews, authors often include comments. To reflect this, we added the problem description to some prompts and omitted it from others, then tested both and reported the results.

To address the second research question, we collaborated with Beko[1], a multinational home appliances company, whose software division adopted an automatic code review tool based on the open-source Qodo (Formerly CodiumAI) PR Agent [30] using GPT-4 Turbo model [32]. This tool provides automatic code review comments for each pull request across 10 projects and 22 project repositories. The study targeting the second research question is accepted to be presented at ICSE 2025 $47^{th}$ International Conference on Software Engineering, Software Engineering in Practice track.

To comprehensively analyze our case study, we utilized three distinct data sources. Initially, we retrieved detailed information from Azure DevOps, the platform hosting our project's repositories. This included pull request metadata, review comments with developer-provided labels indicating implementation, and code diffs comparing initial and final pull request versions. Subsequently, developers provided feedback through short surveys tailored to each pull request. Finally, we conducted a broader survey involving 22 industry practitioners to capture their overarching insights and perceptions.

**Thesis Organization.** Chapter 2 discusses the background and the related work. We describe the methodology for the experiment study in Chapter 3. We describe the research settings with information on the study object, goal, research questions, and the means of data collection for the case study in Chapter 4. Chapter 5 and Chapter 6 list our findings and results for the experiment setup and the case study, respectively. We discuss and elaborate on the potential implications of the results in Chapter 7. Threats to the validity of our study are presented in Chapter 8. Chapter 9 includes our concluding remarks.

[1]https://www.bekocorporate.com/

# Chapter 2

# Background and Related Work

## 2.1 Code Review

The practice of code reviews, where developers inspect changes prior to integration, is widely established [3]. In 1976, Fagan introduced formal inspections, demonstrating their ability to enhance productivity and program quality [2]. However, these formal inspections, while effective in ensuring quality, proved to be overly time-consuming for widespread implementation [33]. Their successful adoption depended heavily on organizational structure and a strong focus on process improvement [33]. More recently, in 2013, Bacchelli and Bird [4] introduced Modern Code Review (MCR), which is characterized by an informal, tool-supported, and regularly conducted process, representing a more agile alternative to its formal predecessor.

Modern Code Review (MCR) is prevalent in companies like Google [6], AMD [8], Microsoft [8], and within open-source software (OSS) projects [34].

A 2018 Google case study [6] highlighted its critical role in development, facilitating codebase learning and ensuring integrity, readability, and consistency. Extensive research has explored code review efficiency and its determinants. For

instance, McIntosh et al. [35], through studies of Qt, VTK, and ITK, demonstrated that review coverage significantly influences quality, while also emphasizing that inadequate reviews negatively impact software quality. A 2013 survey of 287 open-source software (OSS) contributors revealed an average of 6.4 hours per week dedicated to code review [7]. Similarly, a 2018 study of Microsoft developers [5] highlighted timely feedback as the primary challenge, with time management also being a significant concern.

## 2.2 Large Language Models (LLMs)

Natural language processing (NLP) has seen rapid advancements. In 2014, LSTM networks [36] and RNNs [37] improved sequential data processing. The Transformer architecture [38] in 2017 further revolutionized NLP with self-attention, enabling context-aware word importance assessment. Google's BERT [39] in 2018 marked a shift towards pre-trained models. BERT and its successors, like GPT [40], RoBERTa [41], and T5 [42], demonstrated the efficacy of fine-tuning large pre-trained models for specific tasks with limited data.

OpenAI's GPT-2 [43] and GPT-3 [44] demonstrated the versatility of large language models (LLMs). In 2021, Codex [45], fine-tuned for programming, powered GitHub Copilot [46]. ChatGPT [28] and GPT-4 [32] further accelerated LLM adoption in programming. Recent advancements include Anthropic AI's Claude 3 family [47] (March 2024), with Claude 3 Opus surpassing state-of-the-art LLMs, OpenAI's GPT-4o [48] (May 2024), a faster, cross-modal GPT-4 variant, and Google's Gemini 2.0 family [49] (December 2024), which outperforms previous Google models.

## 2.3 Related Work

Recognizing the time-intensive nature of code reviews [5, 7], researchers have primarily focused on automating reviewer selection. This includes various approaches to recommending suitable reviewers [50–57].

To enhance efficiency, automating the review process itself, framed as a code-to-comment task [58], has been explored. Early efforts, such as Gupta and Sundaresan's 2018 deep learning model [27], matched code blocks with historical reviews. Later, Li et al. [24] and Shi et al. [23] used CNN and CNN-LSTM models for change approval prediction. In 2022, Hong et al. [9] introduced CommentFinder for code comment suggestions, while Li et al. [12] and Tufano et al. [11] leveraged pre-trained models, including T5, for review comment generation. Thongtanunam et al. [26] and Google [59] developed tools for automatic code modification during reviews. Recent work has focused on large-scale pre-training [10], progress metrics [60], and LLM agents [19, 61] for code review automation.

In 2024, Tufano et al. [20] performed a qualitative evaluation of previous code review automation approaches [9–11] alongside ChatGPT. Their study, while using an unspecified ChatGPT version, suggested it could serve as a competitive baseline for code revision (comment-to-code), but not for comment generation (code-to-comment) [20]. Distinct from prior work, our experimental study investigates LLMs as a code approver, assessing their ability to make merge decisions and provide suggestions. We introduce an experimental setup to benchmark various models, allowing practitioners to identify optimal models and prompt configurations for their codebases.

The application of LLMs and generative AI in code review has garnered increasing research interest. Davila et al. [31] reviewed grey literature, highlighting the exploration of LLM-based tools like ChatGPT for code review. Fan et al. [14] assessed LLM capabilities in code review generation, commit message generation, and just-in-time comment updates, concluding their promise for these tasks.

Watanabe et al. [62] analyzed GitHub projects with ChatGPT conversation links, finding 30.7% negative reactions, often citing limited added value. Vijayvergiya et al. [15] from Google and the University of Washington demonstrated AutoCommenter, an LLM-backed automated code review system for multiple languages, showing feasibility and high user acceptance.

Our case study addresses the gap in longitudinal research on automated code review tools' impact on software development. Unlike prior work, we examine the effects of an LLM-based tool [30] in real-world industry settings, focusing on development artifacts and developer perceptions. This research aims to provide practitioners with actionable insights into the adoption of LLM-based automated code review.

# Chapter 3

# Methodology of the Experimental Study

Developers have different expectations about what makes a “good” code review [4]. To ensure a robust evaluation, we focus on the merge approval decision, which governs the procedure for integrating change requests into the mainline code. This decision is based on code correctness, defined as the ability of the code to perform its intended functionality in all cases. When new code is submitted, reviewers determine whether it should be merged into the mainline code; if rejected, they offer suggestions for improvement. Similarly, we expect LLMs to deliver a verdict on code correctness and offer suggestions when necessary.

Our dataset comprises code blocks with unit tests, offering an objective standard for code correctness and a means to assess the effectiveness of improvement suggestions. Code blocks that pass all unit tests are deemed ”Correct,” while those that do not are considered ”Incorrect.”

## 3.1 Dataset

We use the HumanEval dataset [63], a popular dataset for evaluating LLMs with interview-level code generation questions, alongside LLM-generated code blocks from Yetistiren et al. [64] for solving HumanEval questions. Our two datasets are 164 canonical solutions from the HumanEval dataset that pass all unit tests and 492 AI-generated code blocks from three tools (ChatGPT 9 Jan '23, Amazon CodeWhisperer Jan '23, and GitHub Copilot v1.70.8099), all written in Python. The datasets are referred to as "Ground Truth" and "Mixed" respectively. The 492 AI-generated blocks are categorized as 234 "Correct" and 258 "Incorrect." We chose this dataset for its unit tests and its diversity of correctness.

## 3.2 Test Setup

In our test setup, we simulate a code review scenario where reviewers must approve or reject a proposed change. When rejecting a change, reviewers typically suggest improvements. Therefore, our prompt instructs the LLM to classify the code as correct or "Incorrect" and to provide a code suggestion. The detailed steps to our methodology can be seen in Figure 3.1.

Prompts can enhance and refine the LLM's capabilities [65]. We optimized our prompt using a chain-of-thought style, a widely used prompting method. Our prompt template is given in Figure 3.2.

Code blocks often include descriptions that explain the code in the form of code comments or pull request descriptions. To mirror this, we use the HumanEval problem descriptions in our prompts. Since such descriptions are not always provided, we had two prompt types. In Figure 3.2, the text in red appears only in the prompts that include problem descriptions while the text in black is the same for each prompt type.

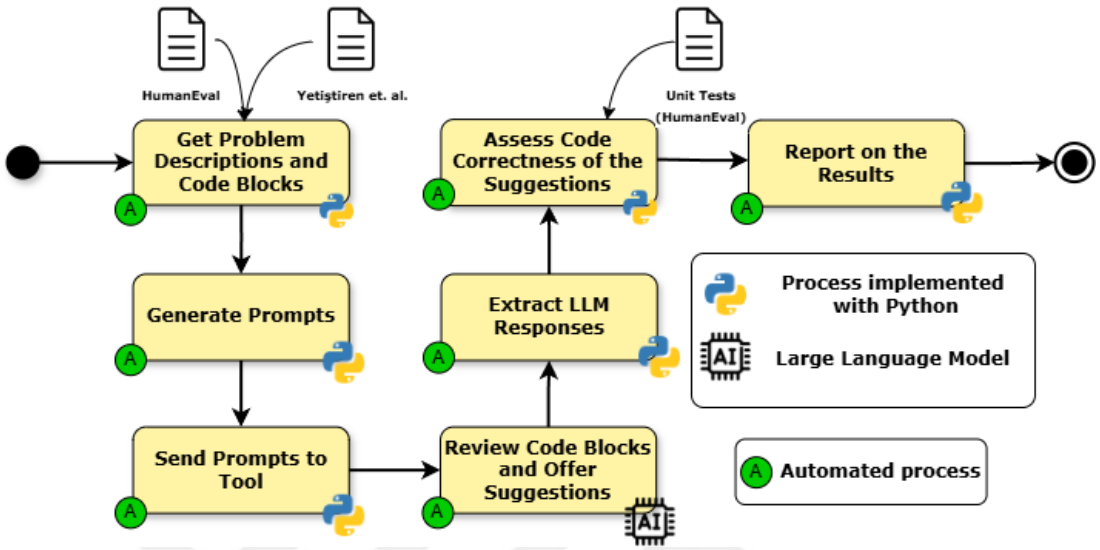


Figure 3.1: Test Setup

The LLM's output provides two distinct pieces of information for our evaluation. The first is the code correctness classification, indicating whether the code is "Correct" or "Incorrect". The second is the code improvement suggestion: a revised code block that should perform better (or remain unchanged if the code is "Correct") than the original.

## 3.3 Evaluation Metrics

We expect reviewers to detect code correctness accurately. To quantify this, we define "Correctness Accuracy" as the proportion of correctness assessments that match unit test results, as seen in equation 3.1. When we classify "Correct" as positive and "Incorrect" as negative, correctness accuracy is equivalent to the model accuracy. Using this classification, we also calculate false positive and false negative rates.

- **Correctness (Model) Accuracy:**

$$\frac{\text{The Number of Accurately Assessed Code Blocks}}{\text{The Number of All Code Blocks}} \tag{3.1}$$

For effective code improvements, suggestions must pass all unit tests. We

You are an experienced Python developer. You will be provided with a code block (and a corresponding problem description). Your task is to understand the intended functionality, review the code, and generate feedback. Please follow these steps: Analyze the Code: Understand what the code is meant to do. Consider Edge Cases: Reflect on all scenarios in which the code should operate. Evaluate Functionality: Determine whether the code successfully implements the intended functionality across these cases. Classify the Code: If the code requires changes, set `classified_type` to Incorrect. If the code does not require any changes, set `classified_type` to Correct. Suggest a Corrected Code Block: In the `complete_code` field, suggest a corrected code block if the `classified_type` is Incorrect; otherwise, you should return the code without changing it.
Important: Use only the classifications Correct or Incorrect (case sensitive).
Now, please review the code based on the following code block:
**#code block**
(And the following problem description: **#problem description**)
You need to respond in the following format. This is a strict requirement.
Example output:
feedback: classified_type: No
code: complete_code: No
Apply the following rules strictly:
- Answer should be a valid YAML, and nothing else. Do not add your thought process or any other text.
- Replace "No" values with your suggestions.
- Be careful about the indentation and syntax of your suggested code block, make sure it can run without problems.

Figure 3.2: Prompt Template

define the "Correction Ratio" as the proportion of suggestions that meet this criterion, as seen in equation 3.2.

- **Correction Ratio:**

$$\frac{\text{The Number of Correct Code Suggestions}}{\text{The Number of Incorrect Code Blocks}} \tag{3.2}$$

We should also consider negative scenarios to assess the impact of code suggestions. A suggestion might worsen a code block—turning "Correct" code into

"Incorrect" code. We refer to such instances as "Regressions" and define a "Regression Ratio", as seen in equation 3.3.

- **Regression Ratio**

$$\frac{\text{The Number of Incorrect Code Suggestions}}{\text{The Number of Correct Code Blocks}} \tag{3.3}$$

## 3.4 Variables

Our test setup comprises three variables. The first variable under consideration is the LLM utilized. GPT-4o by OpenAI is used by code review tools such as Qodo [66], and CodeRabbit [29], making it a relevant candidate for evaluation. Shortly before our study, Google introduced the Gemini 2.0 model family as a competitor to GPT-4o. During our experiments, Gemini 2.0 Flash was the most capable model in the family with API access. We ran our experiments with Gemini 2.0 Flash and GPT-4o (model version: 2024-11-20). The second variable is the existence of the problem description. This will allow us to see how the performance is affected by descriptions such as comments or pull request descriptions. Since the involvement of the problem description can provide the LLM with further information, we consider it an important variable.

The third variable is the dataset. We have a dataset with 492 code blocks (mixed dataset) and the HumanEval dataset's canonical solutions (ground truth dataset). The canonical solutions play the role of a control group and provide us with further insights into the reliability of the LLMs. If the code is correct, the suggestions should not worsen it; therefore, for the ground truth, regression ratios are important.

The three variables resulted in eight distinct test configurations. The results of various configurations allow us to infer the generalizability of the findings. For a variable to be an effective factor, we would expect to see consistent effects on

Table 3.1: Experiment Configurations

| No. | LLM | Problem Description | Dataset |
|---|---|---|---|
| 1 | Gemini | Without | Mixed |
| 2 | GPT-4o | Without | Mixed |
| 3 | Gemini | With | Mixed |
| 4 | GPT-4o | With | Mixed |
| 5 | Gemini | With | Ground Truth |
| 6 | GPT-4o | With | Ground Truth |
| 7 | Gemini | Without | Ground Truth |
| 8 | GPT-4o | Without | Ground Truth |

results across different configurations. In Table 3.1, you can see the test configurations. We aim to provide comprehensive coverage with these test configurations. The data obtained from this study, as well as the code used, are in the thesis replication package[1].

[1]https://doi.org/10.5281/zenodo.15008407

# Chapter 4

# Methodology of the Case Study

The second part of this thesis is an evaluative case study that examines the impact of LLM-based automated code reviews on software development, focusing on effectiveness, pull request closure speed, and human code review volume. This section outlines our research settings. Section 4.1 details the study object, and Section 4.2 presents the case study's goal and related research questions. We employed both quantitative and qualitative data sources. Section 4.3 describes our quantitative data collection from project repositories, and Section 4.4 explains our qualitative data collection through surveys. Finally, Section 4.5 details our approach to addressing the research questions.

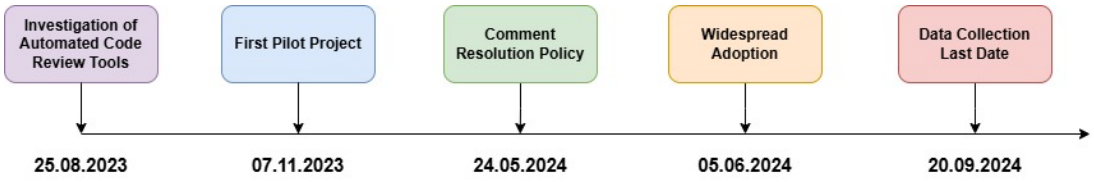


Figure 4.1: Data Collection Timeline

## 4.1 Study Object

We conducted our case study within the software development division of Beko, a multinational company. It operates in the consumer durables and electronics sectors. The software development division of Beko is responsible for developing customer-facing and employee-facing software with 238 practitioners.

Figure 4.1 illustrates Beko's implementation of CodeReviewBot, beginning with an investigation phase on August 25, 2023. Beko assessed various tools, including CodeRabbit [29], Qodo (CodiumAI) [30], and pipeline extensions like Reviewbot, ChatGPT-CodeReview, and Codereview.gpt. Ultimately, they chose Qodo PR-Agent [30], renaming it CodeReviewBot, due to its performance and integration capabilities. The first pilot project adopted CodeReviewBot on November 7, 2023. By June 5, 2024, it was deployed across 10 projects and 22 repositories. On May 24, 2024, a comment resolution policy was communicated to practitioners via email. Data collection for our study concluded on September 20, 2024. Figure 4.2 shows an example CodeReviewBot comment.

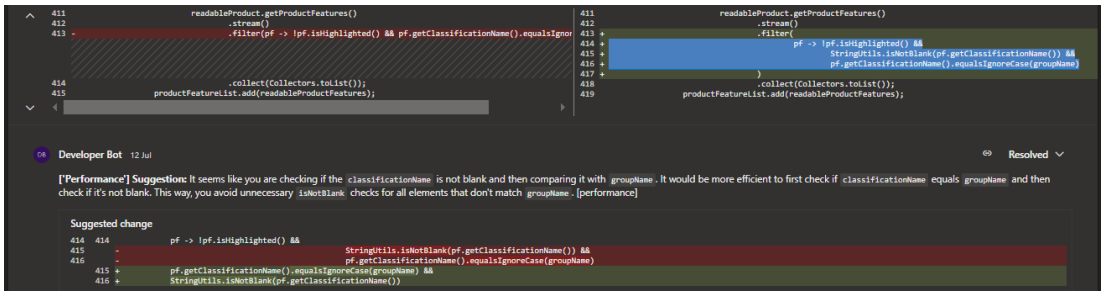


Figure 4.2: Example Review of CodeReviewBot

Using the GPT-4-32K model, CodeReviewBot automates code review comments for pull requests, enhancing but not replacing the traditional modern code review process. Developers continue to contribute their own reviews, as shown in the pull request workflow depicted in Figure 4.3. CodeReviewBot is applied to 22 repositories across 10 projects, covering a diverse range of languages, including Java, JavaScript, C, HTML, SQL, C#, and TypeScript.

The selection of Beko as the study object was motivated by their substantial

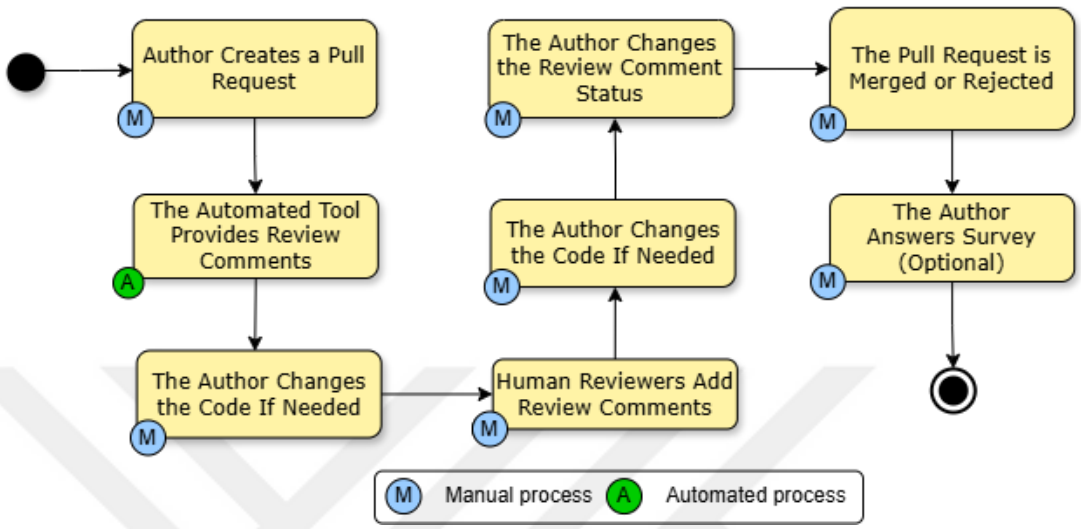


Figure 4.3: Flow of Pull Request Process

Table 4.1: Projects Included in Our Study

| Project | Explanation | # of Developers | Language |
|---|---|---|---|
| **Project #1** | B2B E-commerce Portal | 21 | TypeScript, Java, JavaScript |
| **Project #2** | Enterprise Management Platform | 51 | HTML, JavaScript, C# |
| **Project #3** | Customizable AI Solutions Hub | 22 | C#, HTML, JavaScript SQL |

experience with CodeReviewBot, the organization's scale (238 practitioners), and the variety of projects (10) utilizing the tool. To ensure practitioner privacy, we anonymized individual data and maintained survey confidentiality from Beko authors. We limited our analysis to three projects (Project 1, 2, 3) due to their earlier adoption of CodeReviewBot (November 2023 and April 2024). The other seven projects, implementing the tool around June 2024, lacked sufficient data for analysis. Table 4.1 provides information regarding the three projects.

## 4.2 Case Study Goal and Research Questions

The primary objective of our case study is to investigate the comprehensive impact of automated code reviews within the context of software development, encompassing a variety of perspectives. From an academic standpoint, this research contributes empirical data that aids in understanding the evolving landscape of modern code review and the potential of automated tools. For practitioners, it provides critical insights into how these tools influence development processes. From a business perspective, it evaluates the financial implications of implementing such tools, ensuring that their benefits outweigh the associated costs. And from a quality assurance perspective, it examines whether automated code reviews enhance or diminish the existing code review process. Therefore, this study is a well-motivated and valuable investigation for both industry professionals and academic researchers.

To comprehensively evaluate the impact of LLM-based automated code reviews in our case study, we formulated the following research questions, which were addressed using data gathered from project repositories and surveys:

RQ 2.1: What is the practical usefulness of LLM-based automated code reviews within the software industry?

This question aims to assess the real-world utility of these tools by examining whether the comments they generate are effectively incorporated into the code, thereby reflecting both the tool's usefulness and developer reception.

RQ 2.2: How does the implementation of LLM-based automated code reviews affect the speed of the pull request closure process?

Given that one of the primary motivations for automation is to achieve time efficiency, this question investigates whether integrating automated code reviews accelerates the pull request closure process, indicating enhanced development workflows.

RQ 2.3: How does the introduction of LLM-based automated code reviews influence the volume of human code review activities?

Recognizing the substantial effort required for human code reviews, this question explores whether automation leads to an increase or decrease in human review volume, providing insights into the impact on workload.

RQ 2.4: What are developers' perceptions of LLM-based automated code review tools?

As developers are key stakeholders in the code review process, this question seeks to analyze their perceptions of automated comments and the overall process, providing a comprehensive understanding of their attitudes toward automation and its role in code review.

To comprehensively address our research objectives, we adopted a mixed-methods approach, integrating both qualitative and quantitative data sources. Given the critical focus on the interaction between practitioners and automated code reviews, we collected qualitative data through the administration of pull request surveys and general opinion surveys. Furthermore, we implemented a mandatory comment resolution policy, which required developers to explicitly label how they addressed the comments generated by the automated system. For quantitative data, we employed repository mining, a technique that systematically records digital activities, thereby providing valuable insights into real-world usage patterns. To ensure the robustness and validity of our findings, we triangulated the data gathered from the surveys and comment resolution labels by analyzing historical commit data, pull requests, and related comments within the repository. Figure 4.4 depicts our data collection overview.

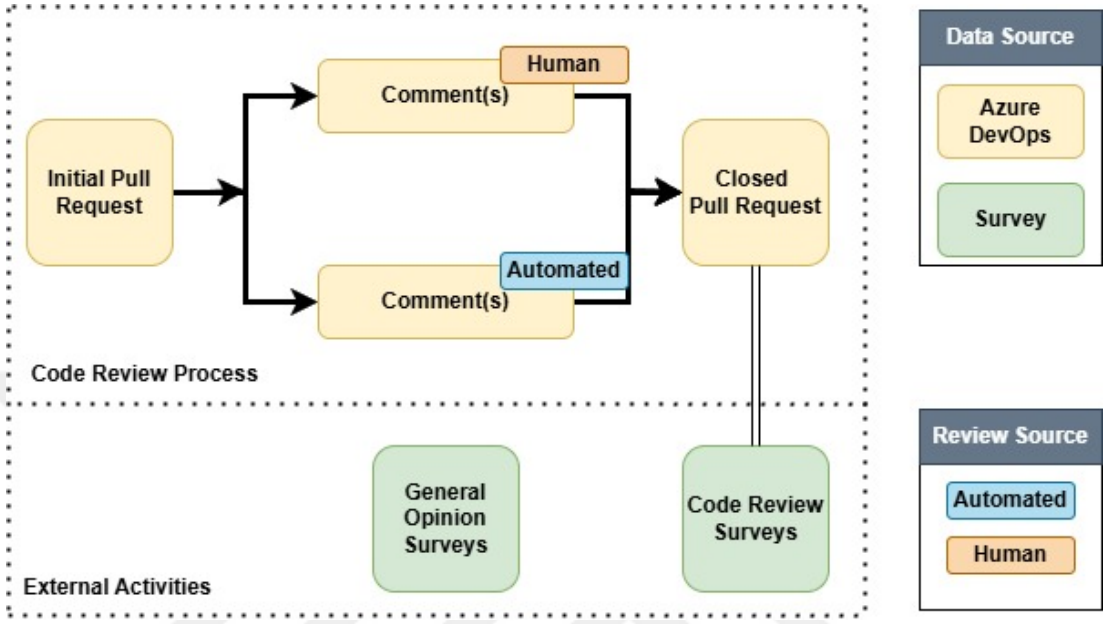


Figure 4.4: Data Collection Overview

Table 4.2: Comment Resolution Policy in Azure DevOps

| Status | Explanation |
|---|---|
| **Active** | This is the default status for new comments. |
| **Pending** | The corresponding comment is being analyzed or waiting for some other thing. |
| **Resolved** | The corresponding comment is successful and its suggestion was implemented. |
| **Won't fix** | The corresponding comment is faulty or cannot be implemented. |
| **Closed** | The corresponding comment was not implemented for some other reasons than "Won't Fix". |

## 4.3 Data Collection From Azure DevOps

To gather comprehensive quantitative data, we extracted information directly from the projects' version control system, specifically Azure DevOps. This data set included detailed pull request information, all associated review comments, and the commit history for each pull request. For a clearer understanding of the data extraction process, we have provided a visual representation of its components in Figure 4.4.

### 4.3.1 Initial Pull Request

The pull request process serves as a method for introducing and reviewing changes to the codebase. Upon creation, a pull request notifies other developers of the proposed modifications. These developers then provide review comments to guide the author, who is expected to implement necessary changes based on the feedback. Once the code meets the required quality standards, the pull request is approved, and the changes are merged into the codebase.

In our specific case study, pull request authors received both automatically generated review comments and comments from other human reviewers. We conducted an analysis of a total of 4,335 pull requests, of which 1,568 were created after the adoption of automated code review tools.

### 4.3.2 Pull Request Comments

Pull request comments serve as a comprehensive record of the code review process, capturing feedback from both human reviewers and CodeReviewBot. We analyzed these comments with respect to volume, timing, authorship, and the extent to which pull request authors utilized them.

It is important to note that the mere presence of a comment does not necessarily indicate its implementation. To accurately assess whether pull request authors benefited from the review comments, we configured a required comment resolution policy within Azure DevOps [67]. This policy mandates that authors explicitly indicate how they addressed each pull request comment. However, it is crucial to acknowledge that this comment resolution policy was implemented after the adoption of CodeReviewBot, which means we lack comparable data from pull requests submitted prior to the tool's integration. Table 4.2 provides a detailed overview of the comment status options available to authors and their corresponding explanations as defined by the comment resolution policy.

### 4.3.3 Closed Pull Request

The initial code changes proposed in a pull request are subject to modification based on the feedback received through review comments. These modifications are recorded as additional commits within the pull request. Therefore, we conducted an analysis of these commits to track the evolution of the code changes. Pull requests ultimately result in either acceptance or rejection. However, it is important to note that practitioners at Beko do not utilize rejection as a primary mechanism for quality assurance. Instead, they reserve rejection for instances where the proposed change is deemed unnecessary or untimely. Consequently, we did not consider pull request acceptance as a reliable indicator of code quality in our analysis.

### 4.3.4 Data Extraction and Analysis

We began our data analysis by extracting raw data from the Azure DevOps server using its API. To facilitate this process, we designed a data schema capable of storing comprehensive information regarding projects, repositories, commits, pull requests, and pull request comments. This schema was then used to organize and store the API responses within a relational database, ensuring structured and accessible data for subsequent analysis.

Following data acquisition, a rigorous preprocessing phase was implemented to guarantee data integrity and precision. Initially, database tables were transformed into CSV file format, subsequently imported utilizing the pandas library. Bespoke scripts were then developed to extract pertinent metrics. A notable anomaly emerged during initial analysis: an unexpectedly elevated volume of active comments, which deviated significantly from the anticipated effects of the comment resolution policy. Upon manual inspection of pull requests, it was discerned that a subset had been closed prior to the generation of comments by CodeReviewBot. Consequently, these post-closure comments were deemed irrelevant for resolution purposes. Subsequently, the merge duration of pull requests

was analyzed. To establish a representative dataset, the elbow method was employed to identify a threshold that encapsulated 93% of the pull requests, thereby ensuring a balanced evaluation. Finally, the remaining 7% of pull requests, identified as outliers, were manually excised to enhance dataset homogeneity.

The investigation also revealed a skew in comment distribution, with two developers exhibiting remarkably elevated comment numbers. Upon scrutiny, it was determined that these developers' Azure DevOps accounts were being utilized by software bots, including SonarQube[1]. As a result, these bot-associated accounts were subsequently excluded from the analysis.

A final data refinement step involved the removal of comments with undefined resolution labels, stemming from deletions or CodeReviewBot's pull request descriptions. Additionally, pull requests targeting branches other than the main branch were excluded, as the comment resolution policy was exclusively applicable to the main branch. Through collaborative data cleaning sessions with both non-practitioner and Beko personnel, dataset corruption was eliminated. The resulting dataset included 4,335 pull requests, with 1,568 subjected to automated reviews.

## 4.4 Data Collection From Surveys

### 4.4.1 Code Review Surveys

A supplementary data source was obtained through a three-question survey administered to pull request authors. Participants were asked to rate the automated review comments on a scale of zero to five and to respond to three survey questions. The first two questions utilized a five-point Likert scale, and the final question was designed to elicit open-ended qualitative feedback. The replication package provides the full text of the code review survey.

[1]https://www.sonarsource.com/products/sonarqube/

Table 4.3: Survey Participants

| Survey Participants | |
|---|---|
| **Experience in Software Development** | **Number of Participants** |
| 0-2 | 4 |
| 2-5 | 9 |
| 5-10 | 6 |
| 10+ | 3 |
| **Position at Beko** | **Number of Participants** |
| Individual Contributor | 16 |
| Lead/Manager | 6 |
| **Total Practitioners** | **22** |

### 4.4.2 General Opinion Surveys

A general opinion survey was developed and administered, yielding responses from 22 developers involved in the three projects within the scope of this research. These practitioners represented a diverse range of experience levels and organizational positions, as detailed in Table 4.3. The survey questions were designed to explore the impact and perceived value of automated code reviews from the developers' perspective. The complete general opinion survey questions are included in the replication package.

## 4.5 Approach Towards Research Questions

To address the diverse aspects of RQ2, a triangulation approach was employed, drawing upon multiple data sources. To address RQ2.1, we integrated review comment labels, post-review commit data, and responses from the general opinion survey. RQ2.2 was answered by extracting pull request closure duration data from Azure DevOps, supplemented by practitioners' perspectives on the development pace from the general opinion survey. RQ2.3, human code review counts were extracted from Azure DevOps, and developers were directly queried about their ability to generate equivalent comments manually. RQ2.4 relied on data from both the developer and pull request surveys. Our data analysis scripts, survey

questions and results are shared in the thesis replication package [2].



[2]https://doi.org/10.5281/zenodo.15008407

# Chapter 5

# Results of the Experimental Study

We evaluated two state-of-the-art LLMs, Gemini and GPT-4o, using the Gemini-2.0-Flash and gpt-4o-2024-11-20 versions with the default model parameters. To ensure reliability, we ran each experiment configuration three times and reported the average results. The standard deviations ranged from 0.35% to 1.61% for correctness accuracy, from 1.02% to 1.93% for false positive rates, from 0.65% to 1.07% for false negative rates, from 0% to 2.88% for regression ratios, and from 0.38% to 1.34% for correction ratios. A chi-square test for variance (using a 5% threshold, df=2, and a critical value of 5.991 at $\alpha = 0.05$) showed that all chi-square statistics were below the threshold. This confirms the consistency of our results, as the observed standard deviations are not statistically significant. We share our data and code in our replication package.

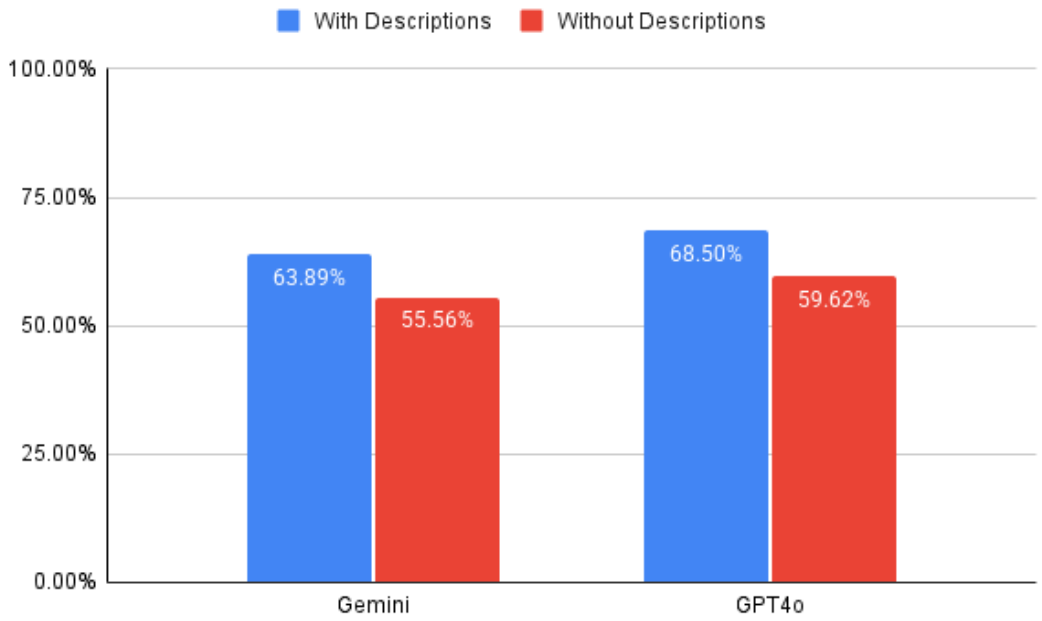


Figure 5.1: Correctness Accuracy (%)

## 5.1 Mixed Dataset Experiment Results

With the mixed dataset, we ran the experiment with and without problem descriptions for both models. GPT-4o outperformed Gemini in code correctness assessments with and without problem descriptions, as shown in Figure 5.1. When provided with descriptions, GPT-4o was accurate 68.50% of the time, compared to Gemini's 63.89%.

Looking at the false positive rates in Figure 5.2, we see that GPT-4o is considerably better than Gemini. Both models perform poorer without the descriptions.

False negative rates in Figure 5.3 differ from the rest of the metrics because Gemini stands better than GPT-4o. Although it may seem contradictory, it complements the false positive rates seen in Figure 5.2.

In terms of correction, GPT-4o performed better than Gemini with and without problem descriptions, as shown in Figure 5.4. GPT-4o had a higher correction ratio at 67.83%, surpassing Gemini's 54.26%.

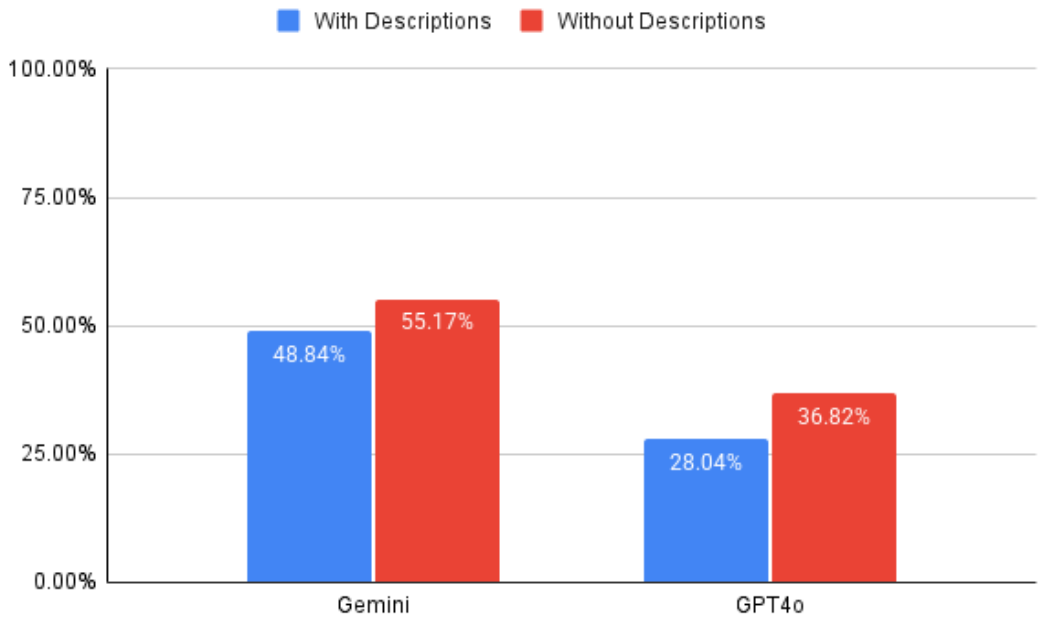


Figure 5.2: False Positive Rates (%)

Looking at the regression ratios, GPT-4o was better in all configurations, as shown in Figure 5.5. GPT-4o had a regression ratio of 10.43%, while Gemini's was 13.53%.

Our experiments showed that both models performed poorer when the prompt did not provide the problem descriptions. This was especially apparent in regression and correction ratios, where differences of up to 22.87% were observed.

## 5.2 Ground Truth Dataset Experiment Results

Using the ground truth dataset, we ran the experiment for both models with and without problem descriptions. We did not calculate correction ratios since all code blocks were already "Correct."

The correctness accuracy results contradicted the mixed dataset findings as seen in Figure 5.6. Gemini outperformed GPT-4o with 66.67%, compared to GPT-4o's 42.07% with problem descriptions. The regression ratio results were

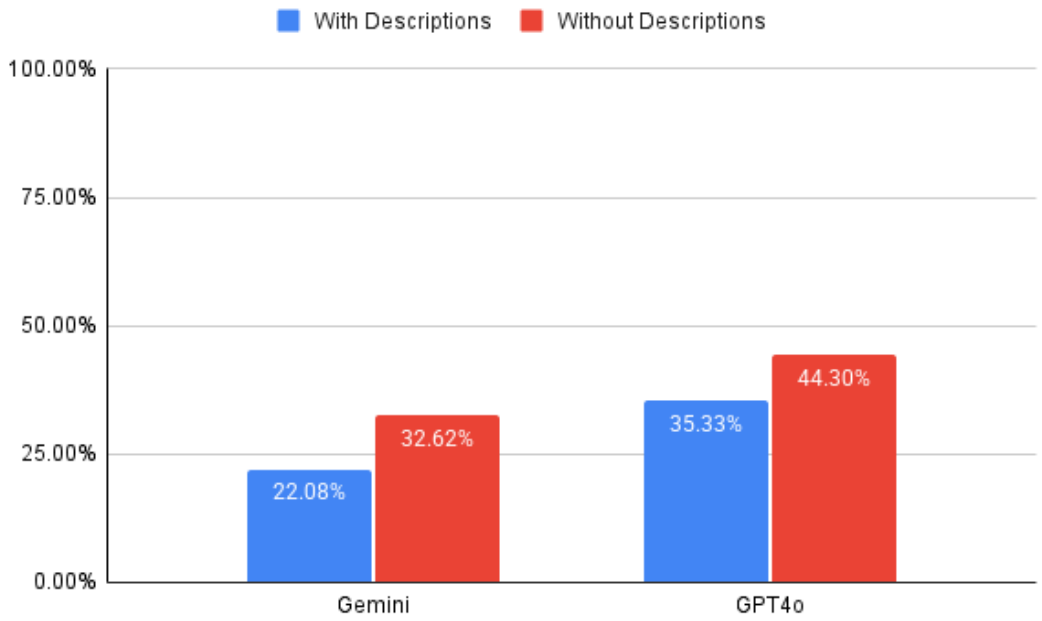


Figure 5.3: False Negative Rates (%)

similar to the mixed dataset results with GPT-4o outperforming Gemini with 9.96% to Gemini's 12.40% with problem descriptions. Both models performed poorer without problem descriptions, though with smaller differences of up to 12.40%.

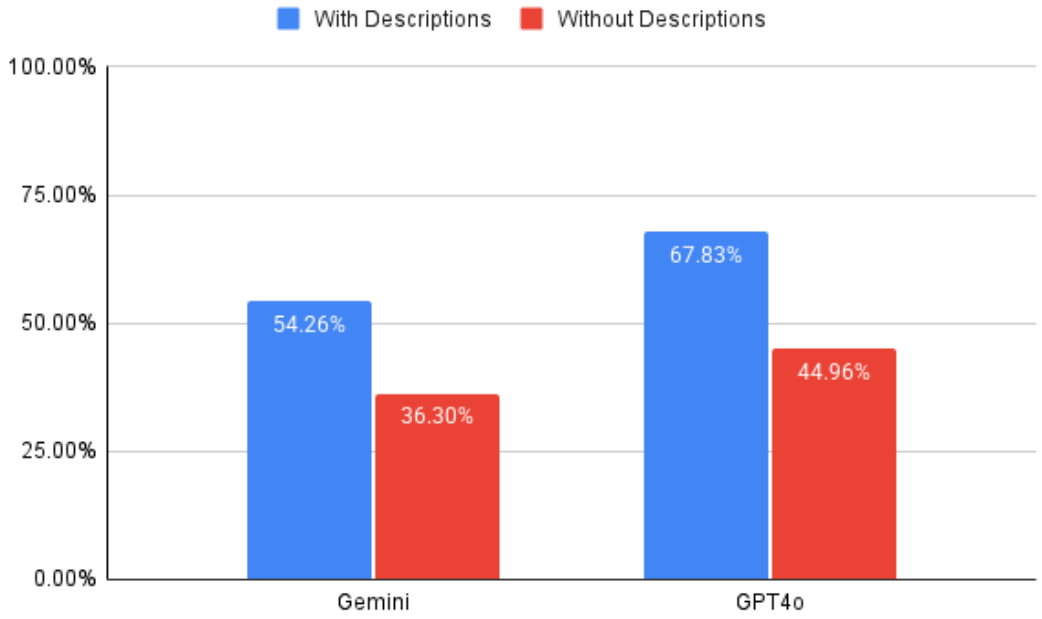


Figure 5.4: Correction Ratios (%)

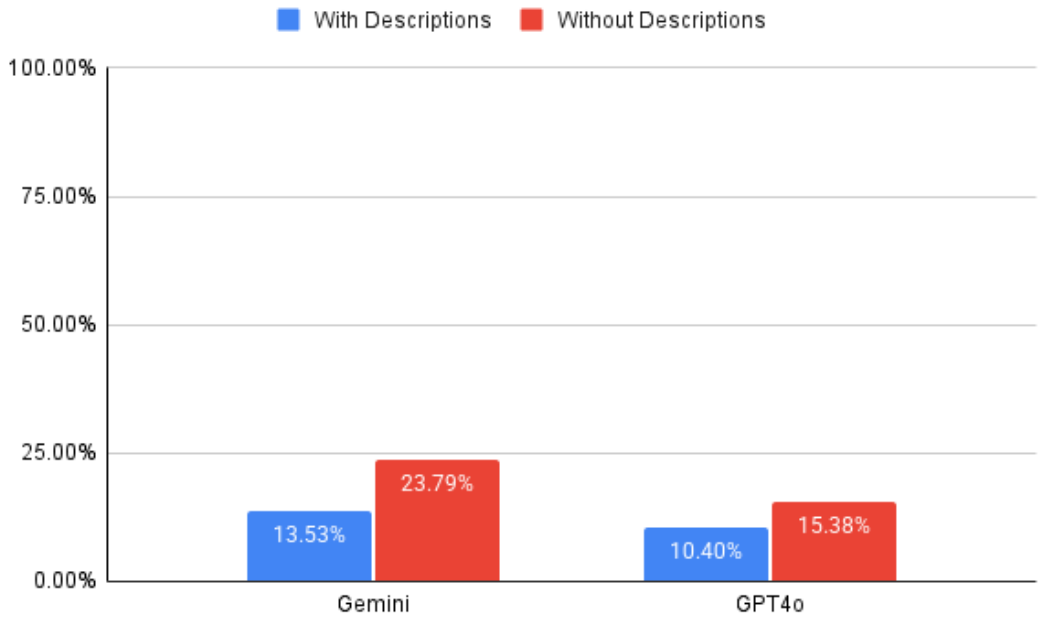


Figure 5.5: Regression Ratios (%)

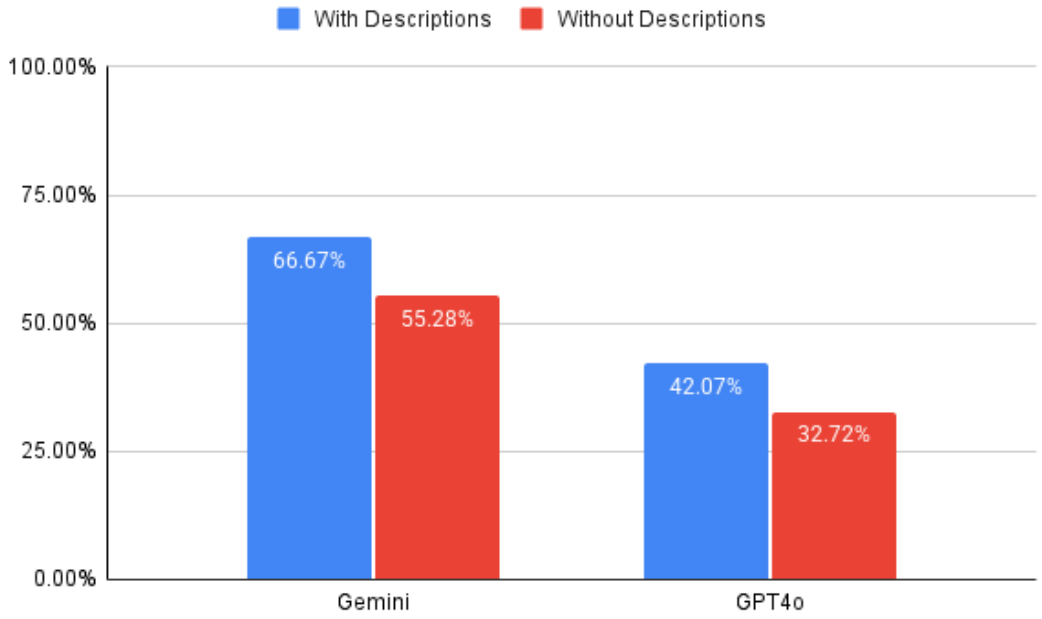


Figure 5.6: Correctness Accuracy (%) with Ground Truth

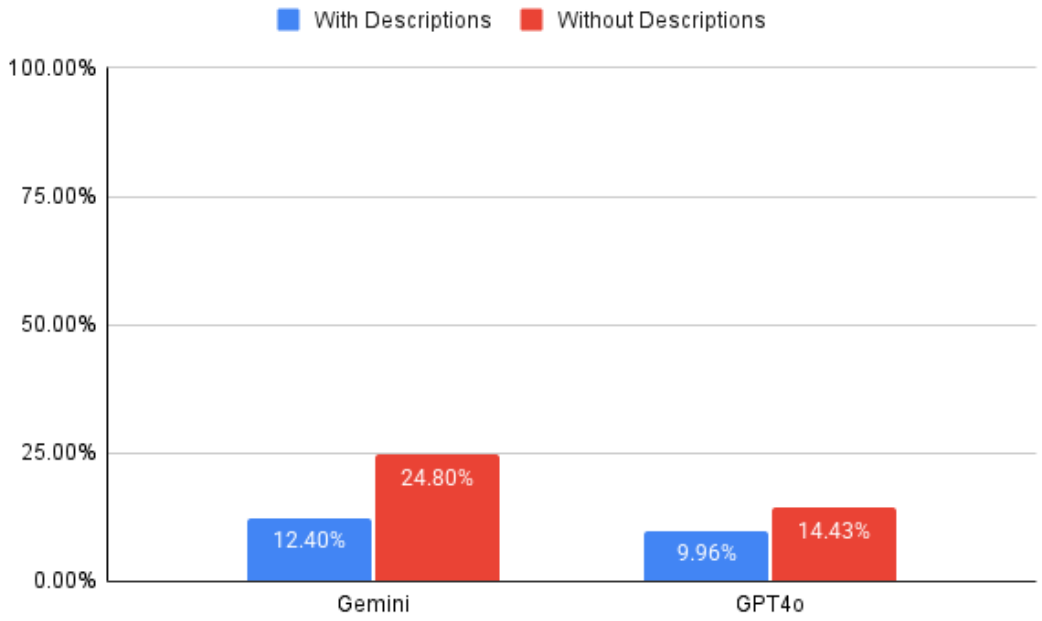


Figure 5.7: Regression Ratios (%) with Ground Truth

# Chapter 6

# Results of the Case Study

## 6.1 Code Review Surveys

In this case study, 38 pull requests subjected to automated code reviews were evaluated by their authors. The authors also completed a survey comprising two multiple-choice questions and one open-ended question, with ten respondents participating. The average rating for automated review comments was 3.46, with a standard deviation of 1.79, as visualized in Figure 6.1. A notable disparity in ratings was observed across projects: Project #1 (4.04), Project #2 (2.72), and Project #3 (3.00). This variation may reflect differences in review quality influenced by project-specific conditions or, alternatively, variations in developer perceptions across the projects.

The authors of the pull requests were presented with a three-question survey. The first two questions, utilizing a rating scale, aimed to gauge the author's agreement with the automated reviews and their evaluation of the review comments' presentation. The third question allowed for free-form, open-ended responses to capture supplemental feedback.

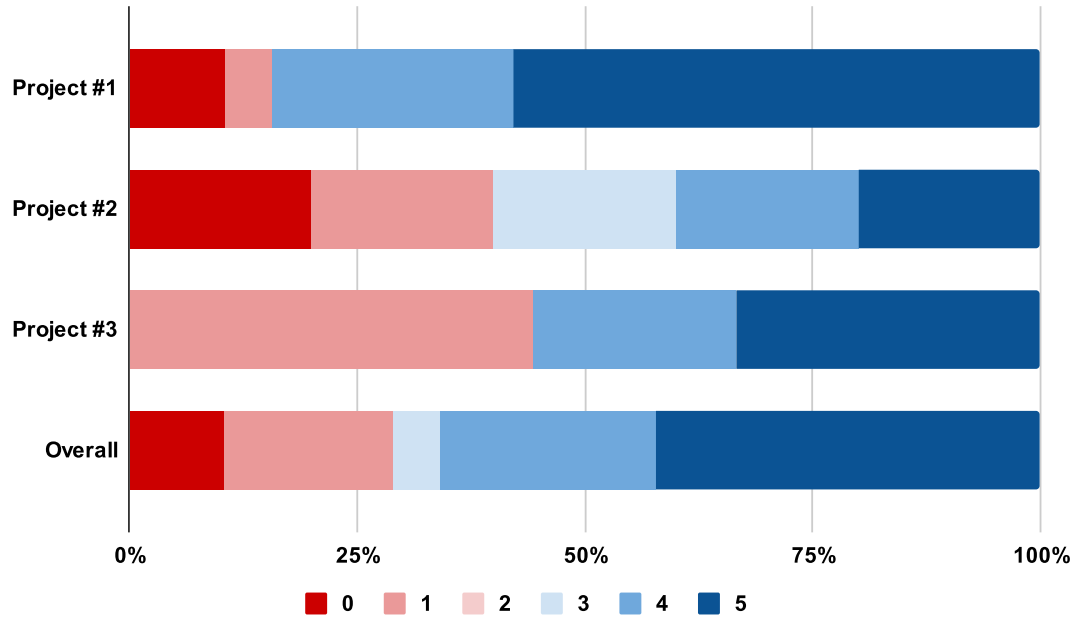


Figure 6.1: Automated Code Review Ratings from Code Review Surveys

Regrettably, the open-ended third question yielded limited responses, primarily consisting of brief affirmations such as "Great." The first two Likert-scale questions, however, received ten responses, as depicted in Figure 6.2. Notably, eight respondents rated their agreement with the comments as "5," while nine rated the presentation as "5." One respondent rated both aspects as "1." The discrepancy in response rates between the numerical ratings and the survey may be attributed to the increased effort required for survey completion, potentially discouraging dissatisfied developers from participating. Nevertheless, the numerical ratings and survey results collectively indicate that developers generally perceived the automated comments as agreeable and well-presented.

## 6.2 General Opinion Surveys

A comprehensive eight-question survey was distributed to project developers, yielding 23 responses, with 22 participants consenting to the publication of their data. This survey aimed to capture developers' overall perceptions regarding automated code reviews. The survey instrument included six multiple-choice

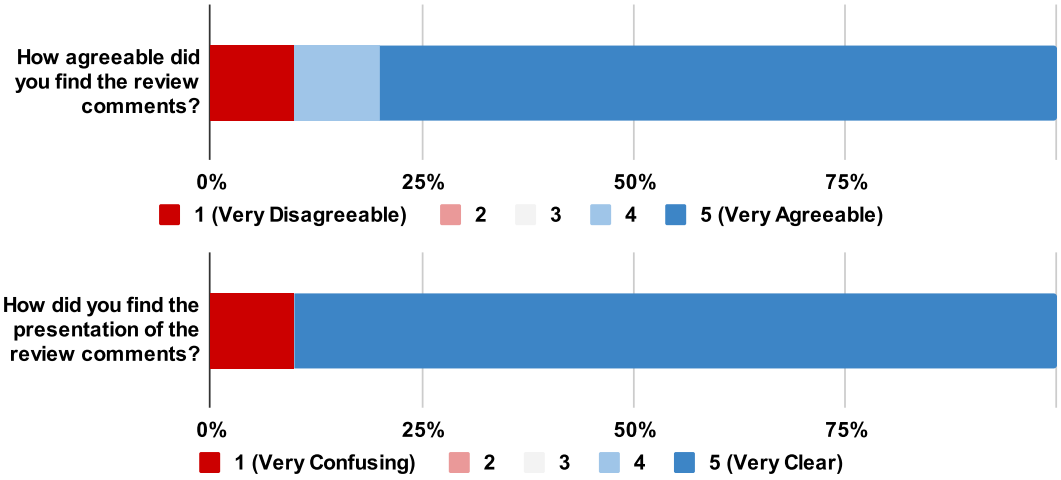


Figure 6.2: Code Review Survey

questions and two open-ended questions.

Results from the multiple-choice questions are presented in Figures 6.3 and 6.4. Specifically, the first question explored the perceived impact of automated code reviews on development speed. Eight developers reported a minor acceleration, four perceived no effect, three reported a minor deceleration, and two reported a significant acceleration.

he second survey question examined the impact of automated code reviews on knowledge sharing among developers. Nine respondents indicated no perceived effect, while eight reported a positive influence. The third question addressed code quality, with the majority (14 respondents) suggesting that automated code reviews resulted in a minor improvement in the overall code quality.

The fourth survey question, addressing the relevance of automated review comments, received ratings of ”4” from nine respondents and ”3” from seven respondents, demonstrating a general consensus on the relevance of the automated reviews to pull requests.

In the fifth question, developers were asked to rate their capacity to detect the issues highlighted by automated code reviews manually. On a scale where ”1” signified inability to identify any issues and ”5” signified the ability to identify all issues, ten respondents selected ”3,” while six selected ”2”.

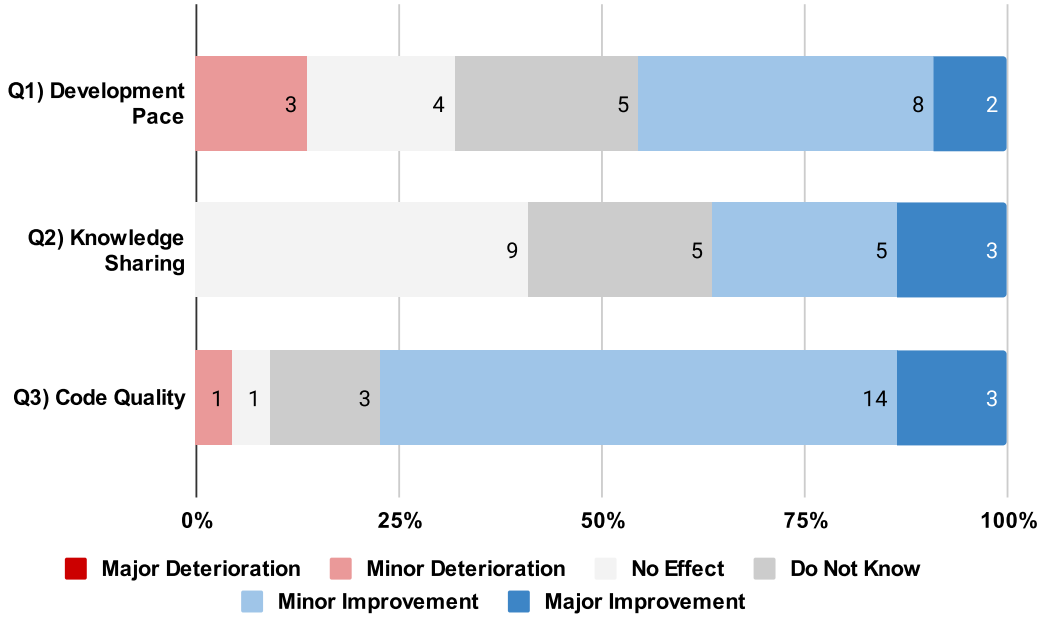


Figure 6.3: General Opinion Survey Responses (Questions 1-3)

The sixth question explored the perceived importance of the issues identified by automated code reviews. Eight respondents rated this importance as ”3,” and six rated it as ”2.” The observed variation in ratings indicates a diversity of perspectives among practitioners regarding the significance of these issues.

## 6.3 Analysis of the Azure DevOps Data

### 6.3.1 Comment Labels

Our analysis involved the extraction of 4408 comments generated by CodeReviewBot. Given the policy's implementation timeline, comments predating the comment resolution policy, as well as those currently active or pending, were excluded. Following this filtering process, 1408 CodeReviewBot comments on merged pull requests were analyzed. The distribution of comment statuses across projects is presented in Figure 6.5. Overall, 73.8% of comments were labeled ”Resolved,” while 21.3% were labeled ”Won't Fix.” Notably, significant inter-project

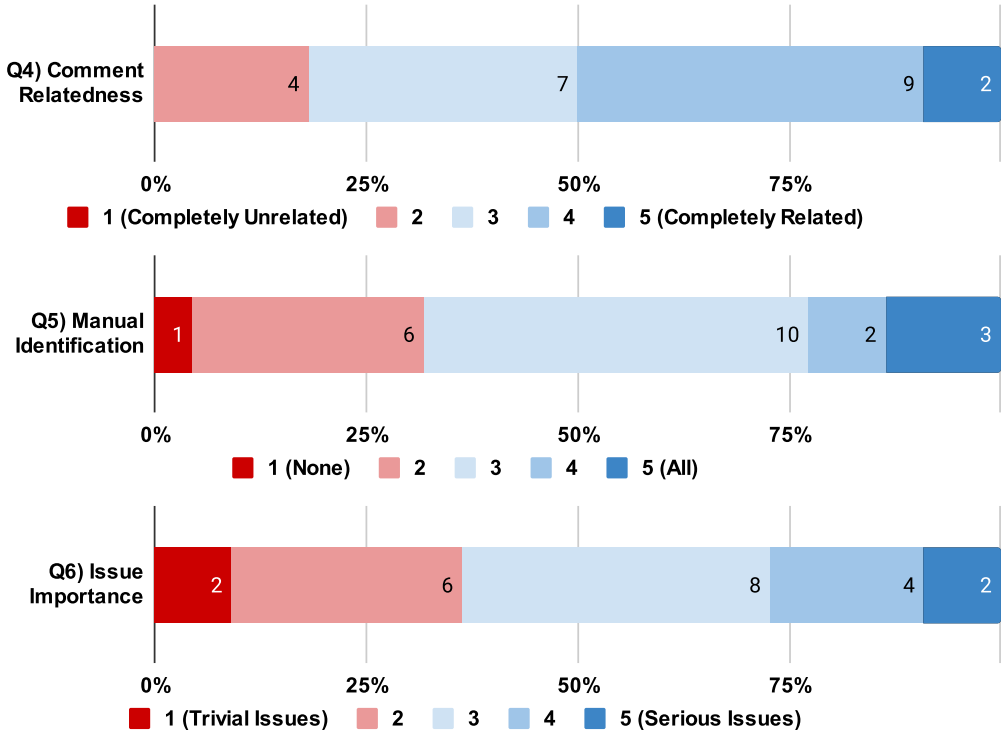


Figure 6.4: General Opinion Survey Responses (Questions 4-6)

variations were observed, with Project #1 and Project #3 exhibiting "Resolved" comment percentages of 55% and 90%, respectively.

### 6.3.2 Commits on Pull Requests

To quantify the impact of the code review process, we analyzed commits made to pull requests following the receipt of review comments, specifically after the implementation of CodeReviewBot. The results, presented in Figure 6.6, indicate that pull requests received 88 commits after CodeReviewBot's comments but prior to human reviewer feedback, and 69 commits after human reviewer comments. It is important to note that authors may address CodeReviewBot comments even after receiving human reviewer feedback. Furthermore, draft pull requests may inherently include subsequent commits. A discrepancy between the number of commits and "Resolved" comment labels is anticipated, given that pull requests can receive multiple comments, and commits pertain to the entire

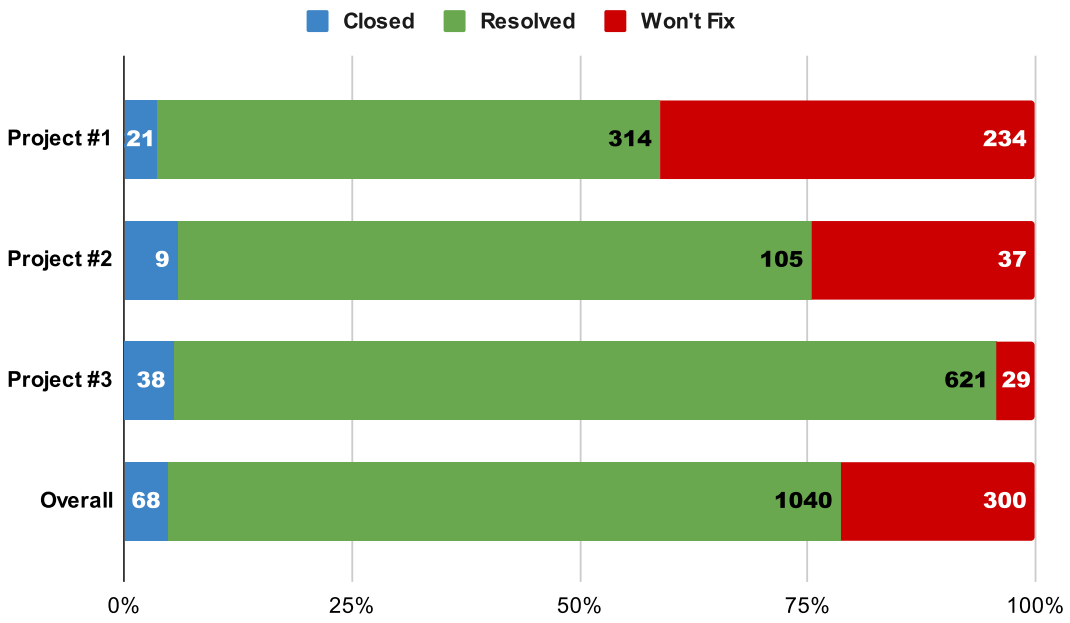


Figure 6.5: Comment Labels on Merged PRs

request. However, deviations from the expected labeling policy, such as using "Resolved" instead of "Closed" or "Won't Fix," may occur. "Closed" labels indicate comments that were not technically erroneous but were not implemented for other reasons.

**RQ2.1: How useful are LLM-based automated code reviews in the context of the software industry?**

Our analysis revealed that 73.8% of CodeReviewBot's comments were accepted and implemented, demonstrating the bot's substantial influence on the code review process. Furthermore, the 88 commits made following CodeReviewBot's comments, prior to human review, suggest proactive code modifications driven by the bot's suggestions. Consistent with these findings, survey respondents largely perceived an improvement in code quality attributable to CodeReviewBot. Based on this convergent evidence, LLM-based automated code reviews are beneficial within the context of this company.

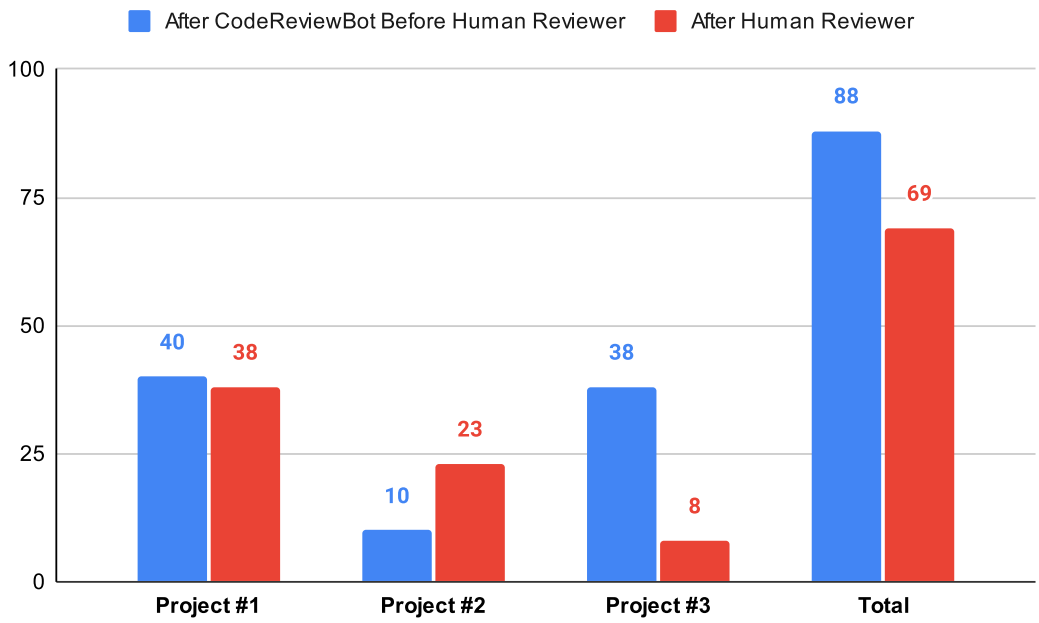


Figure 6.6: Commits After Code Reviews

### 6.3.3 Pull Request Closure Durations

Overall, the mean pull request closure duration increased significantly from five hours and 52 minutes pre-CodeReviewBot to eight hours and 20 minutes post-implementation (t-test, p-value $< 0.001$). However, project-specific trends varied, as illustrated in Figure 6.7. Project 1 exhibited a statistically significant increase, from two hours and 48 minutes to four hours and 38 minutes (t-test, p-value $< 0.001$). Conversely, Project 2 demonstrated a statistically significant decrease, from six hours and six minutes to three hours and seven minutes (t-test, p-value $< 0.001$). Project 3 showed a significant increase, from 20 hours and 22 minutes to 30 hours and 51 minutes (t-test, p-value $< 0.001$).

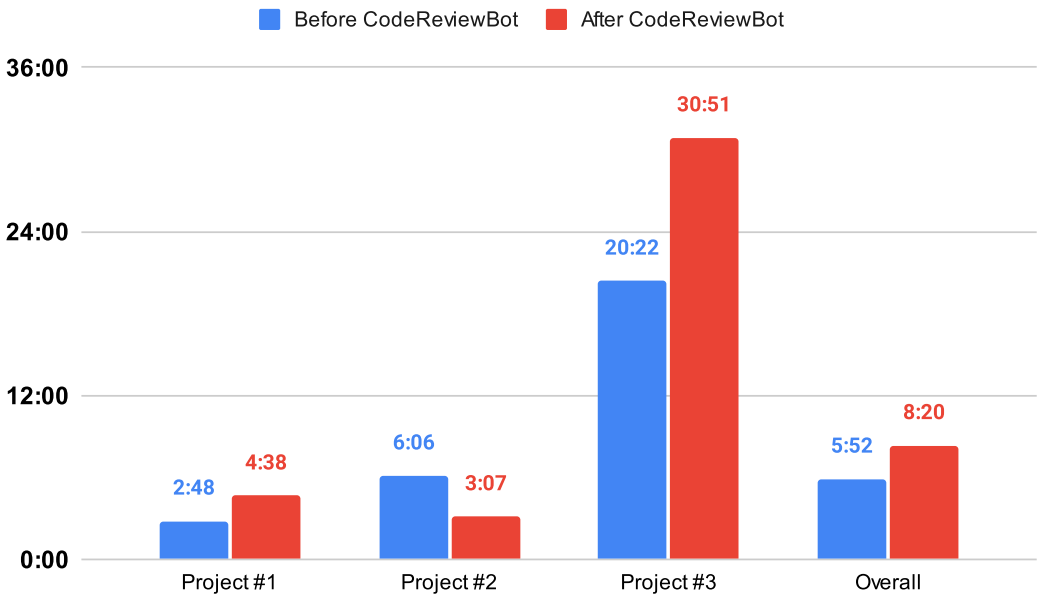


Figure 6.7: Pull Request Closure Durations Before and After CodeReviewBot (H:MM)

**RQ2.2: How do LLM-based automated code reviews impact the pace of the pull request closure process?**

We found that there was a statistically significant increase in the time taken to close pull requests. This increase in the pull request closure process duration is likely due to the developers being required to address additional comments, which were generated from both the bot and human reviewers. The project-specific conditions were shown to have a critical role in determining the impact on pull request closure durations, which varied across different projects.

### 6.3.4 Human Reviewer Comments

Prior to the deployment of CodeReviewBot, human reviewers contributed an average of 0.31 comments per pull request. Subsequently, this average decreased to 0.28 comments per pull request, as illustrated in Figure 6.8. However, a Poisson

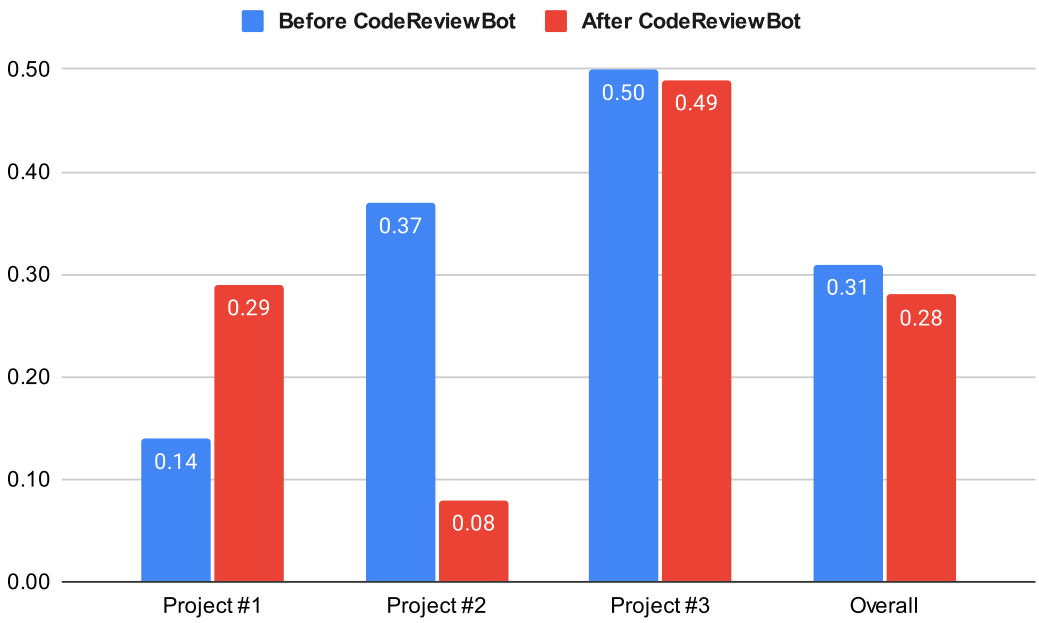


Figure 6.8: Average Number of Human Reviewer Comments

regression analysis revealed that this overall decrease was not statistically significant (p-value $\geq 0.05$). CodeReviewBot's contribution, averaging 3.65 comments per pull request, far exceeded that of human reviewers. Nevertheless, the statistical analysis indicated that trends across projects were not uniform, necessitating project-specific examination:

Post-CodeReviewBot, human comments per pull request significantly increased in Project #1 (from 0.14 to 0.29, p-value $< 0.05$), significantly decreased in Project #2 (from 0.37 to 0.08, p-value $< 0.05$), and remained statistically unchanged in Project #3 (from 0.50 to 0.49, p-value $\geq 0.05$).

RQ2.3: How does the introduction of LLM-based automated code reviews influence the volume of human code review activities?

While the average number of human comments per pull request decreased from 0.31 to 0.28 following the introduction of CodeReviewBot, this reduction was not statistically significant, as determined by Poisson regression analysis. Furthermore, the impact of LLM-based automated code reviews on human reviewer activity exhibited variability across projects, potentially attributable to differences in project dynamics and team practices.

## 6.4 General Opinion Survey Open-Ended Questions

In the general opinion survey, two questions were dedicated to exploring the perceived benefits and drawbacks of automated code review. From 22 responses, 20 were usable, with two being blank. A significant portion of respondents (13/20) highlighted the positive impact on code quality and the enforcement of coding standards. Many respondents also pointed to improvements in the review process, such as faster review cycles and the identification of previously unnoticed code smells and potential bugs. The auto-generated code descriptions were frequently cited as facilitating quicker reviews. Other advantages mentioned included the provision of improvement suggestions and the promotion of best practices.

RQ2.4: How do developers perceive the LLM-based automated code review tools?

While the majority of developers recognized the advantages of automated code reviews, some concerns were raised. Practitioners noted that irrelevant suggestions could slow down reviews and divert developer attention, and that automated systems might miss critical issues detectable by human reviewers. A significant concern, highlighted by one respondent, was the potential for automated reviews to inadvertently change the context of pull requests, potentially leading to severe bugs if changes are applied without careful oversight. Nevertheless, the prevailing view was that automated code review tools are beneficial for improving code quality and maintaining coding standards, especially in identifying quality problems and providing actionable improvement suggestions.

# Chapter 7

# Discussion

## 7.1 Revisiting Research Question 1.1

Our findings suggest that LLMs can evaluate code changes for approval or rejection with moderate accuracy. The highest value we observed was 68.50% (GPT-4o with problem descriptions, mixed dataset). In contrast, Gemini performed worse on the mixed dataset yet better on the ground truth dataset. This raises questions about whether the code type affects LLM performance. To mitigate such concerns, our experimental setup can help practitioners run tests using their own code, guiding them toward the best-performing LLM for their specific needs.

As shown in Figure 5.2 and Figure 5.3, Gemini is more likely to misclassify "Incorrect" code blocks as "Correct," whereas GPT-4o tends to make the opposite error more often. Given these scenarios, higher false negative rates are preferable to higher false positive rates because merging faulty code into the mainline can lead to quality issues, such as bugs. In contrast, the opposite error primarily inconveniences the author, which we consider to be a more minor issue by comparison. For this reason, we believe that the false positive and false negative rates are significant factors when choosing an LLM.

Finally, we consistently observed that LLMs perform better with problem descriptions. This suggests that practitioners adopting automated code review tools should ensure their code changes include clear, helpful comments. It should be noted that we do not refer to code comments specifically. Depending on the tool setup, other descriptions, such as pull request descriptions, can also serve the same purpose.

## 7.2 Revisiting Research Question 1.2

To effectively replace human code reviewers, LLMs need to provide effective code suggestions. In our experiments, we observed that LLMs can correct up to 67.83% of incorrect code (GPT-4o with problem descriptions, mixed dataset). Our results suggest that code improvement suggestions of LLMs are moderately effective in improving code correctness. Without the problem descriptions, the results were consistently poorer.

In terms of regressions, we observed that up to 24.80% of correct code blocks received incorrect code suggestions. The regression rates were higher, and even doubling, without the problem descriptions. The regression and correction ratio results also underlined the importance of comments. Overall, the key takeaway for us is that LLM code suggestions are not reliable enough for full automation. However, this does not mean they cannot be useful when used in moderation.

## 7.3 Human-in-the-loop LLM Code Review

While our results were positive, they also show that LLMs exhibit significant error rates, with regression rates reaching up to 23.79% and inaccurate approval decisions of 44.44% (both from Gemini w/o problem descriptions, mixed dataset). Regressions and inaccurate approval decisions could potentially cause more harm than benefit. These findings raise doubts about the reliability and accountability

of fully automated LLM code reviews. Additionally, there are human-driven motivations for code review, such as knowledge transfer, team awareness, and shared code ownership [4], [5]. To preserve these benefits, full automation is not adequate for code reviews. Based on our results and these motivations, a hybrid process with human involvement is needed.

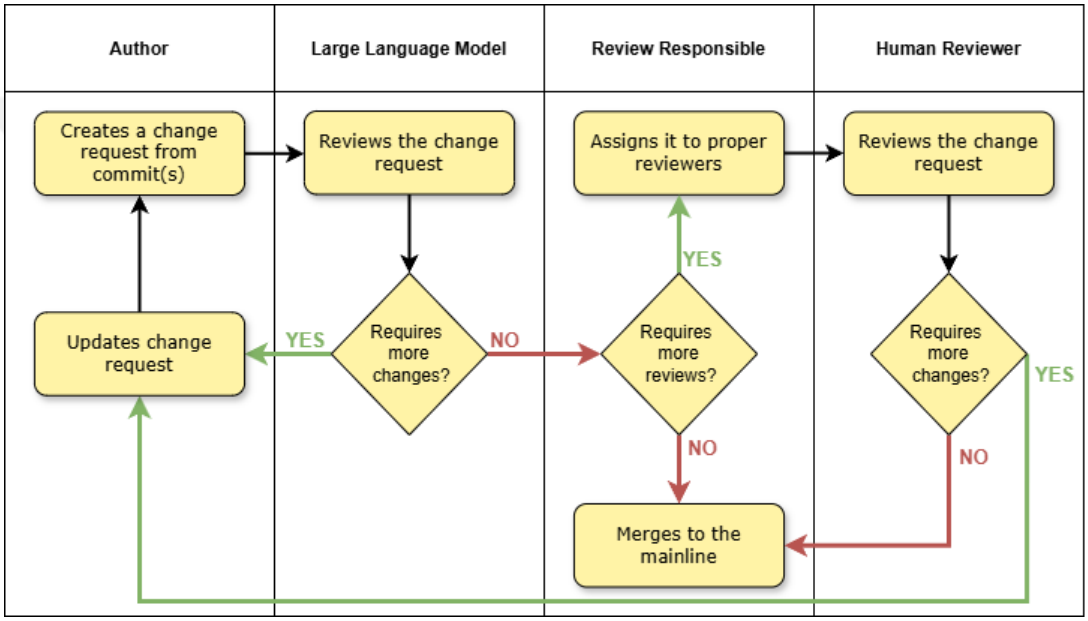


Figure 7.1: Human-in-the-loop LLM Code Review Process

To address these shortcomings, we propose a process that incorporates both human reviewers and LLMs. We call this "Human-in-the-loop LLM Code Review," as depicted in Figure 7.1. The process begins with the author creating the change request, followed by an initial review by the LLM. LLM's suggestions are not automatically implemented. To ensure accountability, the author makes the changes. This iteration continues until the LLM no longer suggests further modifications. Organizations can determine how many of these LLM–author iterations are permitted according to their needs.

Next, a practitioner called "Review Responsible" decides whether the LLM's review is sufficient or if additional human review is necessary. This determination may depend on the criticality or complexity of the proposed change. The change request is merged into the mainline if no further review is required. Otherwise,

human reviewers step in, functioning as a second layer of oversight for reliability. If more changes are needed, the process restarts from the first step. If not, the change request is merged into the mainline.

Overall, this approach reduces manual effort while leveraging human expertise and facilitating knowledge transfer. It mitigates the regressions and faulty assessments encountered in our experiments. The exact implementation of the process can vary according to organizational needs. We see this process as complementary to our experimental setup, helping practitioners establish their own LLM-based automated code review systems.

## 7.4 Does LLM-based automated code review considerably improve software development?

Implementing an LLM-based automated code review tool represents a significant organizational decision, balancing potential benefits against incurred costs. While these tools introduce financial expenses—for instance, in the case study, CodeReviewBot consumed an average of 3,937 tokens per pull request, amounting to $0.48—they also necessitate developers' time investment in addressing review comments.

Analysis of the pull request authors' comment labels revealed that 73.8% of the comments were resolved. Furthermore, we observed 88 commits made after CodeReviewBot's reviews but prior to human reviews, indicating a tangible impact of the automated reviews on the pull requests. Our developer survey indicated that 68.8% perceived a modest improvement in code quality following CodeReviewBot's implementation. Notably, practitioners deemed the issues identified by CodeReviewBot relevant and important, reinforcing the tool's utility in automated code reviews.

Beyond code quality, code reviews are recognized for their role in knowledge dissemination [5]. Our general opinion survey assessed the impact of CodeReviewBot on knowledge sharing. While the majority of respondents reported no effect, none indicated a negative impact.

A primary purported advantage of automated code review is its potential to optimize time and developer effort [58]. To investigate this, we examined pull request closure durations. Although we observed an overall increase in closure times, this may be attributed to developers dedicating additional time to address issues raised by the automated reviewer. The variability in trends across projects suggests active developer engagement with automated feedback, potentially prolonging closure times while enhancing code quality. Moreover, the number of human review comments per pull request did not significantly decrease post-implementation. This implies that while developers invested effort in responding to automated comments, human reviews remained essential. Consequently, the case study did not yield conclusive evidence supporting consistent time or effort savings from the automated code review tool.

Our findings suggest that automated code review can contribute to moderate enhancements in software development practices. However, the decision to implement such a tool warrants careful consideration. The observed benefits within the case study may be influenced by existing code review practices or other quality assurance measures already in place.

## 7.5 Implications of the Case Study to Practitioners

**Over-reliance on automated code reviews**: The general opinion survey highlighted a potential disadvantage: diminished attentiveness among human reviewers. One respondent articulated this concern, stating, 'It may introduce bias, leading reviewers to overlook issues under the assumption that the bot would have

identified them.' This over-reliance on automation poses a risk of critical bugs escaping detection, potentially harming the organization. Consequently, practitioners are advised to conduct thorough evaluations of LLM-based automated code review tools prior to widespread implementation. Organizations should cultivate a comprehensive understanding of these tools' limitations and implement necessary safeguards to mitigate potential risks.

**Unnecessary Review Comments**: Analysis of the comment resolution labels revealed that 26.2% of the tool's comments were designated as 'Won't Fix' or 'Closed,' indicating they were not addressed. These comments may have pertained to issues deemed trivial, irrelevant, or not problematic within the specific context of the pull request. Nevertheless, developers spent time reviewing them. This concern was echoed by survey respondents. One participant in the general opinion survey noted, 'It also makes suggestions that fix code blocks that are not in the scope of the task,' while another stated, 'Sometimes the mistakes it thinks it finds are not mistakes at all.

**Early Identification of Bugs:** Survey results indicated that early bug detection is a notable advantage of automated code review. As one respondent in the general opinion survey articulated, 'It makes finding code defects easier. Developers can see their mistakes quickly.' Participants also emphasized the tool's efficacy in identifying typographical errors and overlooked test code. Another respondent highlighted its effectiveness in 'detecting overlooked code smells.' These observations collectively underscore the tool's potential to enhance code quality.

## 7.6 Implications of the Case Study to Researchers

**Effectiveness of LLM-Based Tools**: The case study offers valuable empirical evidence regarding the effectiveness of LLM-based automated code review tools within real-world industrial contexts. The positive developer reception, coupled

with the significant percentage of comments addressed in pull requests (73.8%), indicates that LLMs possess the potential to enhance the code review process. As one participant in the general opinion survey noted, 'It improved the awareness of the team about code quality.' This observation suggests that the tool's impact extends beyond the immediate code review practice, influencing broader team awareness and potentially fostering a culture of quality.

**Human-AI Interaction Dynamics**: The integration of automated reviews into the code review process introduces novel dynamics in human-AI interaction. A notable finding was the frustration stemming from recursive reviews, as articulated by one respondent: 'With each fix, a new review is generated. However, after the initial review, subsequent comments on revised pull requests often become redundant and unhelpful.' This highlights the need for further research to investigate factors influencing developer trust, satisfaction, and reliance on automated reviews. Such investigations could inform the development of more human-centered design improvements for these tools.

# Chapter 8

# Threats to Validity

## 8.1 Threats to Validity of the Experimental Study

### 8.1.1 Internal Validity

Since we are experimenting with LLMs, the prompt plays a crucial role. We acknowledge that different prompts can yield different results, and ours followed a chain-of-thought approach. Our testing method for code suggestions is also subject to criticism, as it expects a code block rather than a textual suggestion. We chose to do it this way to ensure objective results. We extracted code from the YAML response and ran the corresponding unit tests. Since Python is indentation-sensitive, our prompts warned about indentation and correct YAML formatting. Overall, 94.70% of the code was executed without errors, while 4.08% had indentation errors and 1.08% had YAML format errors. Because our prompt explicitly warned about these issues, we classified erroneous code blocks as "Incorrect," regardless of whether they would pass unit tests if the indentation or YAML errors were fixed.

### 8.1.2 External Validity

In the case study, our scope is limited to Python. Therefore our findings are only directly generalizable to Python. Additionally, due to the stochastic nature of LLMs [68], their outputs are not always identical. To account for this variability, we expanded our sample size by running our experiment three times and reporting the average results.

### 8.1.3 Construct Validity

Our evaluation was conducted on GPT-4o and Gemini, and other LLMs may exhibit different behaviors under the same conditions. The HumanEval dataset [63] consists of simple questions, while the mixed dataset was AI-generated, raising concerns about generalizability. We failed to find a dataset of human-generated code with unit tests with a similar diversity of correctness. To gain deeper and more reliable insights, future experiments should be conducted on code from real software projects. We provide practitioners with our experiment setup and source code, enabling them to generate their own results.

### 8.1.4 Conclusion Validity

In a practitioner setting, unit testing is not always conducted. This may affect the applicability of our results, as practitioners may apply different criteria for code approval or unit tests might not have enough coverage. We used unit tests since it was a reliable way to get objective results. Practitioners may create their own labels to run their own experiments.

## 8.2 Threats to Validity of the Case Study

### 8.2.1 Internal Validity

Given that certain contributors to the case study are affiliated with Beko's software development organization, including one author in a managerial capacity, the potential for bias exists. To mitigate this concern and ensure objectivity, the results and discussions presented herein were primarily formulated by the authors without direct practical involvement in Beko's software development processes.

We acknowledge that the mandatory comment resolution policy may impose a time burden on practitioners. This may lead to frustration, potentially resulting in inconsistent or unreliable comment resolution labels. To address this threat to validity, we employed data triangulation to corroborate our findings. For instance, we cross-referenced comment labels with pull request change data to assess the degree of concordance between these sources.

Our data collection period coincided with the summer months. Beko practitioners indicated that this period was marked by increased developer vacations, resulting in reduced productivity and a slower development pace. Consequently, we acknowledge that seasonal variations may influence our findings, and this limitation should be considered when interpreting our conclusions.

The projects analyzed in the case study were pre-existing and initiated prior to the commencement of the research. To mitigate potential unintended effects and address ethical considerations, we ensured the anonymization of all data, preventing the association of results with individual practitioners. Consequently, the practitioners involved in these projects had no incentive to alter their behavior, as the analysis was conducted independently and retrospectively.

Given the potential for respondent fatigue due to repeated exposure to the pull request survey, we recognized the risk of diminished engagement and careless responses. To mitigate this threat to data quality, we deliberately limited the pull

request survey to a concise set of three questions.

The reliability of our quantitative data analysis is contingent upon the integrity of the data residing within Azure DevOps. Potential database corruption could compromise the validity of our results. To address this risk, we implemented joint data cleaning sessions, wherein we collaboratively evaluated data quality. During this process, we identified two accounts that had been repurposed as bot accounts for a specific period. Consequently, we removed all comments attributed to these accounts. While acknowledging the possibility that some human-generated comments may have been inadvertently excluded, we determined that the majority of these comments were indeed automated bot responses.

While 'Active' and 'Pending' comments typically prevent pull request closure, we observed instances where automated comments were posted after a pull request had been closed. These post-closure comments were predominantly left in an 'Active' state. Consequently, we excluded them from our analysis to maintain data integrity and focus on comments relevant to the active review process.

### 8.2.2 External Validity

The case study's findings are specific to the automated code review tool employed, Qodo PR Agent [30], which utilizes the GPT-4 32k language model [32]. Given this specificity, we acknowledge that alternative LLMs and automated code review tools may exhibit divergent behaviors. Consequently, further research is warranted to ascertain the generalizability of our observations across a broader range of LLMs and tools.

### 8.2.3 Construct Validity

Our data collection methodology involved gathering information from respondents via survey responses and comment resolution labels. Recognizing the inherent potential for misinterpretation in these data sources, we implemented rigorous measures to enhance validity. Specifically, we dedicated significant effort to clearly articulating the survey expectations to respondents. Furthermore, we subjected the survey questions to a thorough review by multiple researchers, enabling us to refine the potentially ambiguous or complex language.

### 8.2.4 Conclusion Validity

The findings of the case study may vary in alternative organizational contexts. Given the inherent complexity and variability of industrial settings, precluding the establishment of statistically generalizable conclusions, we deemed a case study approach most appropriate. To achieve definitive statistical conclusions, a multiple case study design employing our methodology would be necessary. However, recognizing the logistical challenges associated with such an extensive investigation, we opted to limit our scope and refrain from pursuing statistical generalizations.

# Chapter 9

# Conclusion

This thesis presents two distinct studies examining the LLM-based automated code review. The first study is an experimental study evaluating LLM code review capabilities. The second study is a case study examining the effects of LLM-based automated code review tools in industrial settings.

With our experimental setup, we experimentally evaluated the code review capabilities of state-of-the-art LLMs, Gemini 2.0 Flash, and GPT-4o. The LLMs were prompted to assess the correctness of code blocks and provide suggestions if needed. We observed that LLM performance consistently improved when a description of the code's intended functionality was provided. This finding highlights the importance of code comments and pull request descriptions for obtaining better reviews. The results across different datasets showed significant variations, highlighting the need for customized testing for different codebases. Practitioners can use our setup to identify the best-performing LLM and prompt tailored to their needs.

The results of the experimental study indicated that the LLM's assessments had moderate accuracy, and its suggestions were moderately effective. This suggests that a fully automated code review process may be unreliable. We suggested

a hybrid process combining human involvement and LLMs to reduce manual effort and increase reliability while preserving the human-driven benefits of code review. We called it "Human-in-the-loop LLM Code Review." Together with our experimental setup, this process can help organizations establish their own LLM-based automated code review systems.

The second part of this thesis employed an evaluative case study design within the software development division of Beko, a multinational corporation, to investigate automated code reviews. Specifically, we examined the implementation of an LLM-based automated code review tool, derived from the open-source Qodo PR-Agent [30], across ten projects. From this cohort, we strategically selected three projects for in-depth analysis, prioritizing those with extended periods of tool utilization.

The results of the case study indicate that the automated code review tool demonstrated effectiveness, as evidenced by the 73.8% comment resolution rate. Furthermore, a majority of surveyed developers reported a modest improvement in code quality and no adverse impact on knowledge sharing. Notably, the introduction of the tool correlated with a statistically significant increase in average pull request closure times, from five hours and 52 minutes to eight hours and 20 minutes. However, project-specific trends varied, with one project exhibiting a decrease in pull request closure duration. Additionally, we found no statistically significant change in the number of human code reviews following the tool's implementation.

Given that developers constitute the primary user base for the automated code review tool, we prioritized an examination of their perceptions. Survey results indicated that practitioners generally regarded the issues identified by the automated reviews as pertinent and significant to the pull requests. Participants highlighted several advantages that contributed to enhanced code quality, including expedited bug detection, reduction of code smells, heightened awareness of code quality standards, and the promotion of standardized best practices. Conversely, respondents identified disadvantages such as suggestions for code modifications outside the scope of the task and occasional inaccurate recommendations that

diverted developer attention and impeded development efficiency. Furthermore, concerns were raised regarding the potential for over-reliance on automated systems and its associated drawbacks.

The case study demonstrated the potential for automated code reviews to influence software development practices positively. However, it also revealed unintended consequences and inherent limitations. These findings offer valuable guidance for practitioners seeking to implement and leverage LLM-based automated code review tools effectively. To address inter-organizational variability and enhance the generalizability of our results, future research should aim to replicate the case study across diverse software development companies.

Overall, our study shows that LLM-based automated code review tools can be useful for providing code reviews. However, practitioners should be concerned about the reliability issues. Over-reliance on such tools might be more costly than its benefits. Our hybrid code review process proposal, "Human-in-the-loop LLM Code Review," can guide organizations to fine-tune between using LLMs and human reviews.

In future work, we plan to expand our case study to additional companies to examine how organizational differences influence the effectiveness of automated code review tools. Our findings, which reveal project-specific variations, suggest that organizational structures and best practices may play a crucial role in shaping these tools' impact.

Additionally, we aim to extend our experimental study by incorporating diverse prompt types, models, and datasets. As the debate on prompt design continues, we seek to provide empirical evidence for code review. Given the rapid advancement of LLMs, our future work will include a broader benchmark of state-of-the-art models. We also plan to curate an open-source code dataset to enhance the generalizability of our results.

# Bibliography


[1] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "A faceted classification scheme for change-based industrial code review processes," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 74–85.

[2] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[3] N. Davila and I. Nunes, "A systematic literature review and taxonomy of modern code review," *Journal of Systems and Software*, vol. 177, p. 110951, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000480

[4] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[5] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2018.

[6] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[7] A. Bosu and J. C. Carver, "Impact of peer code review on peer impression formation: A survey," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 133–142.

[8] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 202–212. [Online]. Available: https://doi.org/10.1145/2491411.2491444

[9] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder: a simpler, faster, more accurate code review comments recommendation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 507–519. [Online]. Available: https://doi.org/10.1145/3540250.3549119

[10] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," 2022.

[11] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," *CoRR*, vol. abs/2201.06850, 2022. [Online]. Available: https://arxiv.org/abs/2201.06850

[12] 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," 2022.

[13] J. Yu, P. Liang, Y. Fu, A. Tahir, M. Shahin, C. Wang, and Y. Cai, "Security code review by large language models," 2024. [Online]. Available: https://arxiv.org/abs/2401.16310

[14] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code change related tasks," 2024. [Online]. Available: https://arxiv.org/abs/2407.02824

[15] M. Vijayvergiya, M. Salawa, I. Budiselić, D. Zheng, P. Lamblin, M. Ivanković, J. Carin, M. Lewko, J. Andonov, G. Petrović *et al.*, "Ai-assisted assessment of coding practices in modern code review," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 85–93.

[16] C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation," *Information and Software Technology*, vol. 175, p. 107523, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001289

[17] M. Nashaat and J. Miller, "Towards efficient fine-tuning of language models with organizational data for automated software review," *IEEE Transactions on Software Engineering*, pp. 1–14, 2024.

[18] D. Tang, K. Kim, Y. Song, C. Lothritz, B. Li, S. Ezzini, H. Tian, J. Klein, and T. F. Bissyandé, "Codeagent: Collaborative agents for software engineering," 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267412469

[19] Z. Rasheed, M. A. Sami, M. Waseem, K.-K. Kemell, X. Wang, A. Nguyen, K. Systä, and P. Abrahamsson, "Ai-powered code review with llms: Early results," 2024.

[20] R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli, and G. Bavota, "Code review automation: Strengths and weaknesses of the state of the art," 2024.

[21] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, p. 106737, 2022. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106737

[22] M. F. Gön, B. Yetiştiren, and E. Tüzün, "Towards unmasking lgtm smells in code reviews: A comparative study of comment-free and commented reviews," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 163–174.

[23] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 4910–4917, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4420

[24] H.-Y. Li, S.-T. Shi, F. Thung, X. Huo, B. Xu, M. Li, and D. Lo, "Deepreview: Automatic code review using deep multi-instance learning," in *Advances in Knowledge Discovery and Data Mining: 23rd Pacific-Asia Conference, PAKDD 2019, Macau, China, April 14-17, 2019, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 318–330. [Online]. Available: https://doi.org/10.1007/978-3-030-16145-3_25

[25] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," *CoRR*, vol. abs/1901.09102, 2019. [Online]. Available: http://arxiv.org/abs/1901.09102

[26] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: Automated code transformation to support modern code review process," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 237–248.

[27] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[28] OpenAI, "Chatgpt," https://www.openai.com/chatgpt, 2023, accessed: 2024-04-28.

[29] "What is coderabbit?" May 2023. [Online]. Available: https://docs.coderabbit.ai/

[30] "qodo-ai/pr-agent," September 2024. [Online]. Available: https://github.com/qodo-ai/pr-agent

[31] N. Davila, J. Melegati, and I. Wiese, "Tales from the trenches: Expectations and challenges from practice for code review in the generative ai era," *IEEE Software*, vol. PP, pp. 1–8, 01 2024.

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

[33] F. Shull and C. Seaman, "Inspecting the history of inspections: An example of evidence-based technology diffusion," *IEEE Software*, vol. 25, no. 1, pp. 88–90, 2008.

[34] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: which problems do they fix?" ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 202–211. [Online]. Available: https://doi.org/10.1145/2597073.2597082

[35] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, 04 2015.

[36] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[37] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[39] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[40] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[41] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[42] 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," *CoRR*, vol. abs/1910.10683, 2019. [Online]. Available: http://arxiv.org/abs/1910.10683

[43] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[44] 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," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[45] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes,

A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[46] "Github copilot," May 2022. [Online]. Available: https://github.com/features/copilot

[47] "Introducing the next generation of claude," Mar 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-family

[48] "Hello gpt-4o," May 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[49] "Introducing gemini 2.0: our new ai model for the agentic era," Dec 2024. [Online]. Available: https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/

[50] W. H. A. Al-Zubaidi, P. Thongtanunam, H. K. Dam, C. Tantithamthavorn, and A. Ghose, "Workload-aware reviewer recommendation using a multi-objective search-based approach," in *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 21–30. [Online]. Available: https://doi.org/10.1145/3416508.3417115

[51] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal, C. Maddila, S. Mehta, and B. Ashok, "Whodo: automating reviewer suggestions at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 937–945. [Online]. Available: https://doi.org/10.1145/3338906.3340449

[52] J. Jiang, D. Lo, J. Zheng, X. Xia, Y. Yang, and L. Zhang, "Who should make decision on this pull request? analyzing time-decaying relationships and file similarities for integrator prediction," *Journal of Systems and Software*, vol. 154, pp. 196–210, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300962

[53] H. Ying, L. Chen, T. Liang, and J. Wu, "Earec: Leveraging expertise and authority for pull-request reviewer recommendation in github," in *2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*, 2016, pp. 29–35.

[54] E. Mirsaeedi and P. C. Rigby, "Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1183–1195. [Online]. Available: https://doi.org/10.1145/3377811.3380335

[55] A. Ouni, R. G. Kula, and K. Inoue, "Search-based peer reviewers recommendation in modern code review," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 367–377.

[56] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: Code reviewer recommendation in github based on cross-project and technology experience," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 222–231.

[57] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 141–150.

[58] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," 2021.

[59] A. Froemmgen, J. Austin, P. Choy, N. Ghelani, L. Kharatyan, G. Surita, E. Khrapko, P. Lamblin, P.-A. Manzagol, M. Revaj, M. Tabachnyk, D. Tarlow, K. Villela, D. Zheng, S. Chandra, and P. Maniatis, "Resolving code review comments with machine learning," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 204–215. [Online]. Available: https://doi.org/10.1145/3639477.3639746

[60] X. Zhou, K. Kim, B. Xu, D. Han, J. He, and D. Lo, "Generation-based code review automation: How far are we?" in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, 2023, pp. 215–226.

[61] X. Tang, K. Kim, Y. Song, C. Lothritz, B. Li, S. Ezzini, H. Tian, J. Klein, and T. Bissyandé, "Codeagent: Autonomous communicative agents for code review," 01 2024, pp. 11 279–11 313.

[62] M. Watanabe, Y. Kashiwa, B. Lin, T. Hirao, K. Yamaguchi, and H. Iida, "On the use of chatgpt for code review: Do developers like reviews by chatgpt?" in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 375–380. [Online]. Available: https://doi.org/10.1145/3661167.3661183

[63] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[64] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," 2023.

[65] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[66] "Qodo-ai pr agent documentation," February 2025. [Online]. Available: https://qodo-merge-docs.qodo.ai/

[67] "Branch policies and settings," may 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/repos/git/branch-policies?view=azure-devops&tabs=browser#check-for-comment-resolution

[68] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, "Large language models: A survey," 2024.