Wait, we can use AI for code reviews?

Investigating the usefulness of AI-generated code review comments

Benjamin Ek Marko Pavlov

Examiner: Lena Buffoni
Supervisor: Sebastian Litzinger
External Supervisor: Torvald Mårtensson

Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Computer Engineering
Spring Term 2025 | LIU-IDA/LITH-EX-G–25/024–SE

Upphovsrätt
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida https://ep.liu.se/ .

Copyright
The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: https://ep.liu.se/.

© 2025 Benjamin Ek & Marko Pavlov
CC BY 4.0

Abstract
Recent advancements in artificial intelligence (AI), especially large language models (LLMs), have sparked significant interest in their potential applications. Code review is a widely adopted software development practice that helps participants understand the codebase and maintain software quality. However, this process is highly time-consuming, prompting interest in improving efficiency through advances in code review tools.

This study investigates the applicability and effectiveness of LLMs as assistive tools in the code review process. We generate and systematically evaluate AI-generated code review comments, which are assessed by both the study authors (using scientifically derived criteria) and professional software engineers (reflecting industrial best practices). A review of scientific literature and a series of interviews with engineers reveal substantial theoretical alignment on what constitutes a good code review comment, particularly regarding the identification of functional errors, edge cases, maintainability concerns, and suggestions for more efficient solutions. However, analysis of inter-rater reliability and thematic categorization indicates a substantial divergence of what constitutes a useful code review comment between study authors and engineers in practice.

Our results show that current state-of-the-art LLMs can generate code review comments that are often perceived as useful by both academics and practitioners. Specifically, 69.4% (95% CI [65.5%, 73.1%]) of comments were found useful by the study authors, and 92.8% (95% CI [88.9%, 95.4%]) by professional software engineers. While LLMs are not positioned to replace humans in the code review process, they show promise as effective assistive tools. We highlight challenges related to prompt engineering, subjectivity in usefulness assessments, and the potential of LLMs as a component in future code reviewing tools.

Keywords
Artificial Intelligence (AI), Large Language Models (LLMs), Code Review, Automated Code Review, Software Engineering, Code Quality, Code Review Usefulness

Acknowledgements
We would like to thank our supervisor Sebastian Litzinger for his help in guiding us towards a meaningful study. We would also like to thank our external supervisor Torvald Mårtensson for the opportunity to conduct this study at Saab and his guidance along the way. Finally, we would like to thank our examiner Lena Buffoni. A special thanks to the anonymous software engineers at Saab for participating in interviews throughout this study.

Table of Contents

1. Introduction 1
   1.1. Research Questions 2
2. Background 3
   2.1. Code Reviewing Process 3
   2.2. Expected Benefits 4
   2.3. AI Code Reviewing Software 5
3. Related Works 6
4. Theory 8
   4.1. Large Language Models 8
      4.1.1. Per-Model Performance 8
   4.2. Prompt Engineering 9
      4.2.1. Chain-of-Thought Prompting 9
      4.2.2. Zero-Shot Prompting 9
      4.2.3. Few-Shot Prompting 10
   4.3. Statistical Analysis 10
      4.3.1. Inter-rater Reliability 10
      4.3.2. Chi-Squared Test for Proportions 11
   4.4. Interview Design 12
      4.4.1. Group Interviews 12
      4.4.2. Pilot Testing 12
5. Method 14
   5.1. Establishing Quality Criteria 14
   5.2. Interviews 14
      5.2.1. Phase 1: Eliciting Code Review Quality Criteria 15
      5.2.2. Phase 2: Evaluating LLM-Generated Reviews 16
   5.3. Selection of LLMs for Code Reviews 17
   5.4. Change Data Collection 18
      5.4.1. Milling the CROP 18
   5.5. Prompting 19
      5.5.1. Exploring Prompting Techniques 19
      5.5.2. Automating Code Review Prompting 20
   5.6. Evaluation 20
      5.6.1. Code Review Evaluation Software 21
      5.6.2. Statistical Analysis of Evaluation Data 21
6. Results 23
   6.1. Code Review Quality 23
      6.1.1. Scientific Criteria 23
      6.1.2. Industrial Criteria 24
   6.2. LLM-Generated Code Review Evaluation Result 27
      6.2.1. Author Evaluation Results 27
      6.2.2. Engineer Evaluation 30
      6.2.3. Author and Engineer Evaluation Alignment 32
   6.3. Research Question Answers 34

6.4. Evaluation Dataset 39
7. Discussion 40
7.1. Difficulties Encountered 40
7.2. Study Limitations 41
7.3. Validity Threats 42
7.4. Ethical Implications 43
7.5. Future Work 43
8. Conclusion 45
9. Bibliography 46
10. Appendix 52
10.1. Interview Questions 52
10.1.1. First Interview Phase 52
10.1.2. Generellt om kodgranskningar 52
10.1.3. Förväntningar kring kodgranskningar 52
10.1.4. Förväntningar kring AI-kodgranskningar 53
10.2. Prompts 53
10.2.1. System Instructions 53
10.2.2. User Prompt 55

Table of Tables

Table 1 Strength-of-agreement guidelines proposed by Landis and Koch 11
Table 2 Benchmarking scores for the selected LLMs 18
Table 3 Scientific literature on the usefulness of a code review comment 24
Table 4 Software engineers at Saab on the usefulness of a code review comment 27
Table 5 Comment usefulness ratings for all models from the author evaluation 29
Table 6 Comment usefulness ratings for all models from the engineer evaluation 31
Table 7 Alignment of scientific literature and industrial perspective on the usefulness of a code review comment, N/A refers to the lack of mention by respective party 37

Table of Figures

Figure 1 Zero-shot prompt example for a reasoning model (CoT) 9
Figure 2 Few-shot prompt example for a reasoning model (CoT) 10
Figure 3 Rating screen for an LLM-generated code review comment 21
Figure 4 Engineers' perspectives on defect findings in good code review comments 26
Figure 5 Agreement on comment categorization between authors after collapsing categories 28
Figure 6 Proportion estimate of code review comment usefulness per LLM according to author evaluation 30
Figure 7 Proportion estimate of code review comment usefulness per LLM according to engineer evaluation 32
Figure 8 Proportion estimate of code review comment usefulness per group for subset of comments 33
Figure 9 Percentage of comments in each category after comment categorization relative to total number of rated comments by group 34
Figure 10 Illustrative example of what was considered a unanimously useful code review comment in this study 38

1. Introduction

A code review constitutes a systematic examination of source code by other developers. Code reviews play a crucial role in maintaining code quality, enforcing best practices and catching potential bugs before they reach production. During the review process, developers other than the original author inspect code changes before integration into the main codebase. Code reviews provide valuable feedback for code authors, and constitute a learning opportunity for less experienced coders [1]. Although peer reviewing code is generally considered a good practice, it also results in major time-consumption for developers conducting the review [2].

Sadowski et al. [1] conducted an internal satisfaction survey at Google, which concluded that 97% of developers are satisfied with the code reviewing process. Sadowski et al. also state that code reviews were introduced at Google as a means of maintaining code norms as well as readability. Additionally, they found that there are other perceived benefits of code reviews stemming from education (e.g. learning code norms and design patterns).

Although performed frequently, and usually completed quickly, with most taking less than 24 hours after being requested, some code reviews can take days, and in rare cases, even weeks. A finding by Bosu and Carver [3] suggests that on average, developers spend six hours each week reviewing changes made by their peers. Longer review intervals can make it harder to incorporate the reviewer's feedback without introducing new issues [4].

Code reviews are also prone to human error, especially as codebases grow in complexity and functional faults that block code submission are rarely found by manual code reviews [2]. Approximately half of the functional defects and bugs are missed during manual code reviews [5]. Instead, the majority of identified defects relate primarily to the long-term maintainability of the code [3], [6]. The ratio between evolvability defects and functional defects found in code reviews was found to range between 5:1 and 3:1. Classification of evolvability and functional can be useful when creating code reviewing checklists that cover the majority of defect types [7].

Additionally, the usefulness of a code review varies, and the perceived review quality decreases as the number of files included in the review increases [8]. The decrease in useful feedback is first noticeable when a review contained 20 or more modified files. The reviewers' experience of the codebase is also positively associated with the perceived usefulness of a review [2]. Developers who have previously authored or reviewed a specific software artifact tend to give more useful comments. The amount of useful comments by a reviewer increases with the number of prior reviews on the same software artifact [8].

The code review process, while highly beneficial, has inherent flaws such as its time-intensive nature and challenges related to social dynamics [1], [2]. As a result of code reviews slowing down the development process yet still providing benefit for the developer, the pursuit of more efficient code review methods is increasingly compelling. To accomplish this we propose the usage of large language models (LLMs) as an assistive tool for improving the effectiveness of peer reviewing code. Our objective is to maximize its benefits while mitigating the previously mentioned drawbacks.

Vaswani et al. [9] introduced the transformer architecture in 2017, which has served as a foundational technology for recent advancements in LLMs. LLMs are machine learning models designed to perform natural language processing (NLP) tasks, including question answering and language inference [10]. Additionally, they have shown an ability to support software engineers in software development tasks [11]. These advancements highlight the growing capabilities of LLM assistants, and have prompted this study investigating the applicability of general LLMs (e.g ChatGPT, Gemini, Claude, etc.) in performing code reviews, evaluating their usefulness and potential impact in software engineering workflows.

1.1. Research Questions

This study addresses the following research questions:

RQ1: What specific characteristics define useful code review comments as identified in scientific literature, and how do these align with perspectives from professional software development practice?

RQ2: How do different automated code reviews compare in their perceived usefulness to developers when evaluated against established code review quality criteria?

2. Background

This chapter provides the necessary context to understand the significance of the present study. It begins by examining the code reviewing process in software development and subsequently explores the expected benefits of improvements in the code reviewing process and the potential value of incorporating LLMs in code reviews.

2.1. Code Reviewing Process

The code reviewing process has evolved significantly over time, shifting from formal checklist-based inspections to more agile, tool-supported approaches. Historically, the code reviewing process was primarily performed through code inspections. The code inspection (Fagan inspection), first described by Fagan [12] in 1976, was a formalized process involving planning, overview, preparation, inspection, reworking and follow-up. These inspections were conducted in meetings where both authors and reviewers examined code changes to identify software defects.

Code reviewing practices have changed with advancements in developer tooling and this has led to the use of domain-specific tools that support the logistics of the code reviewing process [1]. Tool-based code reviews typically follow these steps: code authors submit a patch containing changes; reviewers examine the diff and modified files; reviewers initiate discussions linked to specific lines of code with authors and other reviewers; and authors address issues raised in comments by submitting modifications. This cycle repeats until reviewers provide explicit approval.

However, modern code reviews are more lightweight and informal than their predecessors. Rigby and Bird [4] observe that contemporary software engineering processes, such as iterative development, have influenced code reviewing processes. Shorter development cycles have necessitated more frequent code reviews, reducing review intervals from weeks with traditional code inspections to hours or days with modern code review approaches. A prevalent modern code reviewing process is the pull-request based model. This code reviewing process is enabled by version control platforms such as GitHub, GitLab, and Bitbucket. Developers typically fork or create a new branch from the main repository to implement changes. The changes are then pushed

to the developer's own branch. Subsequently, the developer then opens a pull request on the main repository. The pull request contains the differences between the source branch and the changes in the developer's own target branch. Other developers and reviewers can discuss the proposed changes within the pull request, while the code author and other developers are able to submit modifications based on the feedback provided through comments. A repository maintainer can then choose to integrate the changes in the pull request into the source branch once all changes have been reviewed and the proposed changes satisfy project requirements and standards [13].

2.2. Expected Benefits

Several aspects of the code review process could potentially be made more efficient when utilizing LLMs for code reviews. For instance, automated code reviews could potentially reduce the review intervals by providing a code author with near-instant review comments on their authored changes. This can decrease development time and cognitive overhead for a human reviewer. The reduction in cognitive overhead and development time would be particularly significant when dealing with large code changes or numerous modified files, contexts that frequently result in less useful comments [8].

The density of useful comments tends to increase as a project matures. This is partly attributed to increased familiarity with the project and the refinement of code review processes within the project [8]. Implementing LLM-generated code reviews in a project's early stages could potentially accelerate the accumulation of useful comments, allowing the project to reach peak comment usefulness sooner.

Bacchelli and Bird [14] discuss how engineers may tackle the reviewing process differently depending on their familiarity regarding the context of the change. They found that if the change was made in code familiar to the reviewer, a less detailed description of the change was necessary as they could often understand the change by simply looking at the code. However, when the reviewers instead were faced with changes in less familiar files they see more benefit from detailed change descriptions in text. They also explained that when reviewing unfamiliar code they may also spend more time reading parts of the code other than the change itself to build necessary understanding. The need for improved tooling regarding summarizing changes made in code reviews has also been expressed [15]. Detailed LLM-generated summaries could significantly save time for engineers reviewing code changes in unfamiliar parts of a codebase.

2.3. AI Code Reviewing Software

Multiple code reviewing tools leveraging LLMs for generating code review comments have been commercialized (e.g. Diamond, Qodo, CodeRabbit, etc.). This indicates a growing interest in adopting LLMs as a tool for reviewing code. Although, as Greg Foster from Graphite [16] states, *"In the end, though, AI is not a replacement for real human review."* Despite this, there is a lack of scientifically validated metrics on the frequency of useful code review comments generated by LLMs. Thus, this study aims to establish a metric for the usefulness of LLM-generated code review comments.

3. Related Works

There has been a recent increase in interest in leveraging LLMs for automated code reviews. By automating aspects of the code review process, developers can receive assistance when reviewing new changes, allowing them to spend less time on code reviews while simultaneously increasing the quality of the comments generated. Manushree et al. [17] explored the use of LLMs to automate the code review process through a code-review specific LLM. The LLM would review code by leaving comments on code it identified as potentially faulty, along with these comments the LLM would also provide a URL referring to the corresponding best practice document. The authors found that only 54–60% of the comments left by the LLM were perceived as useful by the developers. That study was conducted through 2022–2023 using their own trained LLM made specifically for this purpose. In contrast to this prior study, our work focuses mainly on SotA general commercial LLMs for conducting code reviews.

Others have tried to incorporate LLM-assisted code reviews as plugins in development software. Almeida et al. [18] developed an IDE plugin for ChatGPT supported code reviews. Instead of attempting to automate the code review process, the authors opted to use ChatGPT as an assistant in code reviews conducted by developers. This resulted in a shorter average review time for reviewers using the plugin. The observed changes were an average review time of 22.5 minutes for reviewers not using the plugin, and a 15.2 minutes average review time for reviewers who were using the plugin. The results indicated that utilizing an LLM improved code review efficiency. However, the reviewers used to benchmark the software were all undergraduate students. The present study will be conducted in regards to software engineers at Saab Aeronautics, which may provide insights into the usefulness of LLM-assisted code reviews for experienced engineers in the field.

Some studies have also explored the usage of code review specific LLMs. Yi Lin et al. [19] experimented with the automated code review model *CodeReviewer* against a quantitative measure, as well as five human evaluation tasks. Here the authors successfully increased the review quality of the model through experience-aware oversampling using data mined from GitHub pull requests in a previous study. The results showed

that oversampling training data could result in measurable increases in code review quality. That study focuses primarily on the code review-specific LLM CodeReviewer, and strategies for managing its training data. The present study will instead focus on the effectiveness of LLM-assisted code reviews using general LLMs.

4. Theory

The theory necessary for building an understanding of the method employed in this study is presented below.

4.1. Large Language Models

LLMs are machine learning models designed to perform NLP tasks, including question answering and language inference [10]. LLMs designed for NLP tasks are trained on large amounts of text data, with the aim of learning a generic representation for language processing [20]. An LLM is queried using what is called a *prompt*. After being prompted with either a question or task, LLMs are able to respond in an attempt to answer the question or fulfill the task.

There are several different sub-models for LLMs, specializing at different tasks. This study looks at two of these models, the standard *non-reasoning model*, as well as the *reasoning model*. The reasoning model builds upon the LLM by leveraging what is called chain-of-thought (CoT), which is used to elicit multi-step reasoning abilities for LLMs [21], improving LLM performance for tasks that benefit from concrete reasoning [22].

4.1.1. Per-Model Performance

LLMs differ vastly in terms of performance. This difference can often be attributed to factors such as the amount and quality of training that has been performed. Increasing the scale or quality of training data can lead to better performance and efficiency against various NLP tasks [23]. There are also various commercial LLMs (e.g. ChatGPT, Gemini, Claude) created by several different companies (e.g. OpenAI, Google, Anthropic). These LLMS differ not only in overall performance and training data, but may excel at different specific sets of tasks.

Ultimately, there are several factors that determine the performance of an LLM. Factors such as performance, cost, and model type are all factors to consider when selecting LLMs for a certain task. There do however exist benchmarks describing the performance of each commercially available LLM [24], [25], which aid in the selection process of commercial LLMs for different tasks.

4.2. Prompt Engineering

The quality of a prompt given to an LLM impacts the quality and control of the generated response greatly [26]. There are several possible methods and techniques to employ when creating an effective prompt. Below we describe zero-shot prompting, few-shot prompting and chain-of-thought prompting, which are the prompting techniques considered for use in this study.

4.2.1. Chain-of-Thought Prompting

The CoT prompting method involves chaining together prompt outputs in series of intermediate reasoning steps and telling the models to *think step by step* before arriving at an answer [27]. Each step in the CoT process represents individual *thoughts*, simulating an internal dialogue for the LLM. OpenAI recommends avoiding CoT prompting with reasoning models [28], as these models inherently perform the reasoning processes that CoT prompting typically elicits in non-reasoning models.

4.2.2. Zero-Shot Prompting

Zero-shot prompting involves providing instructions without examples of desired output [29]. This technique works on the assumption that reasoning models are capable of figuring out problems themselves. This inner reasoning is often better at explaining the task than the prompt itself, which would then, in theory, have better results. According to OpenAI, when creating prompts for reasoning models, one should start with zero-shot prompting [28]. Figure 1 shows an example of a zero-shot prompting scenario for a reasoning model, the example is taken from [29].

Input

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A:

(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls.

Figure 1. Zero-shot prompt example for a reasoning model (CoT)

4.2.3. Few-Shot Prompting

In contrast to zero-shot prompting, few-shot prompting incorporates multiple examples of expected output within the prompt [29]. This allows for greater control of the prompt output, as the LLMs get greater visual examples of how to respond to each input. LLMs are generally great at performing few-shot tasks [23], which shows promise in using few-shot prompting to perform the tasks that suits it. This can however negatively impact results for tasks that require out-of-the-box thinking, as few-shot prompting may cause prompt outputs to conform to the examples more than what is wanted for a specific task. Figure 2 shows an example of a few-shot prompting scenario for a reasoning model, the example is taken from [29].

Examples

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Input

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A:

(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are 16 / 2 = 8 golf balls. Half of the golf balls are blue. So there are 8 / 2 = 4 blue golf balls. The answer is 4.

Figure 2. Few-shot prompt example for a reasoning model (CoT)

4.3. Statistical Analysis

Appropriate statistical methods are required to analyze the data collected in this study. Below we present the statistical methods employed for analysis of the evaluation data.

4.3.1. Inter-rater Reliability

Inter-rater reliability is a measure of agreement between two or more subjects on the rating of the same phenomena [30]. The resulting reliability between raters is calculated as a kappa value. Kappa is

necessary when calculating accurate reliability between raters, as it accounts for the expected outcome by chance from a series of ratings, which is often missed when comparing pure percentage agreeability [31]. This study uses two kappa calculations: Cohen's kappa and Fleiss' kappa. Cohen's kappa is used to measure inter-rater reliability between two raters and Fleiss' kappa is used to measure inter-rater reliability between two or more raters. Kappa is computed based on the following equation:

$$\kappa = \frac{P(a) - P(e)}{1 - P(e)}$$

In this formula *P(a)* describes the observed percentage of agreement, and *P(e)* describes the probability of the expected outcome by chance [31]. The resulting kappa value can range anywhere between −1 and 1, where a negative kappa indicates disagreement, a positive kappa indicates agreement, and a kappa of 0 implies that the agreement between raters is no different than chance. Kappa values can have multiple interpretations. This study evaluates kappa values according to strength-of-agreement guidelines proposed by Landis and Koch [32], which can be seen in Table 1.

Kappa	Strength of Agreement
< 0.00	Poor
0.00–0.20	Slight
0.21–0.40	Fair
0.41–0.60	Moderate
0.61–0.80	Substantial
0.81–1.00	Almost perfect

Table 1. Strength-of-agreement guidelines proposed by Landis and Koch

4.3.2. Chi-Squared Test for Proportions

The chi-squared (χ^2) test for proportions is a statistical method used to determine whether there is a statistically significant difference between the observed proportions of one or more categorical outcomes across different groups [33]. The test is particularly suitable for scenarios

where data are summarized in contingency tables, which display the frequencies for each group or category.

The test evaluates the null hypothesis (H_0), which states that the true proportions are equal across the groups. The alternative hypothesis (H_1) posits that at least one of the proportions differs. The test statistic is calculated by comparing the observed frequencies (O_i) with the expected frequencies (E_i) under H_0, using the formula developed by Pearson [34]:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where the sum is over all cells in the contingency table [33]. The chi-squared test assumes that observations are independent and that the expected cell counts are sufficiently large (e.g., all expected counts are at least 1, and at least 80% of expected counts should be 5 or greater) [33], [35].

4.4. Interview Design

Interview protocols can be designed in various ways to suit the specific objectives of a study. This section discusses group interviews and pilot tests, which were used to construct interview protocols in this study.

4.4.1. Group Interviews

Group interviews, sometimes referred to as focus groups, involve a moderated discussion among a small group of participants addressing a set of focused questions [36]. The main objective is not to reach consensus, but rather to elicit a broad range of opinions and experiences related to the topic of discussion. Focus groups can help gain rich, nuanced insights into a topic by exploring shared experiences and diverse perspectives that emerge through social dynamics [36], [37].. The group format encourages participants to interact with one another, enabling them to ask questions, challenge, and elaborate upon ideas presented during the interview. This may reveal insights that may not emerge during individual interviews. This is further encouraged by the use of open-ended questions during the interviews [37]. Open-ended questions can elicit a wide range of answers, some of which are unforeseen by the interviewers [38], [39].

4.4.2. Pilot Testing

The design and wording of interview questions are critical to ensuring clarity, neutrality, and validity of collected qualitative data. Pilot tests serve to identify and eliminate ambiguous, confusing, or leading questions, improve the flow of the interview, and optimize the overall

interview structure [40]. Conducting a pilot test can also help uncover sources of interviewer bias, ensuring that participants' responses are not influenced by the interviewer [41].

Furthermore, pilot tests offer benefits that allow the research team to estimate interview duration, and enhance interviewer confidence and competence. Such testing can be conducted by researchers themselves, with one acting as the interviewer and another as the participant, which provides feedback from both perspectives [41].

5. Method

The methods of extracting quality metrics for code reviews, the selection of dataset, and selection of LLMs used in this study are presented below.

5.1. Establishing Quality Criteria

Establishing objective code review quality criteria is essential for creating evaluation metrics. The perceived quality of a review varies depending on the codebase or the context of where it is conducted [42]. As a result of this broad scope of what constitutes a high-quality code review, the quality criteria of code reviews conducted in this study were derived from two different contexts. This widens the scope of the results, as well as provides a basis for comparison and discussion regarding the differences in each context.

The standards studied encompass both industrial and scientific perspectives regarding code review. The quality criteria of the scientific perspective were extracted through a study of scientific literature regarding code review quality and expectations, and the quality criteria of the industrial perspective were elicited through interviews with professional software engineers working in the industry. The quality criteria of the scientific perspective were established before interviews with the engineers, as the data gathered from scientific literature could be presented in the interviews and used to find potential overlaps between scientific and industrial standards. Once code review quality criteria were extracted from both contexts, guidelines and metrics were formulated based on the results. The performance of the LLMs in the field of code reviews was evaluated based on these developed guidelines and metrics.

5.2. Interviews

A qualitative interview study was conducted with software engineers employed at Saab Aeronautics in Linköping, Sweden. The interviews were conducted with 6 software engineers from two separate departments. The interviews aimed to capture a wide range of opinions to develop a comprehensive understanding of the industrial perspective. Participation was voluntary, and only software engineers with substantial code reviewing experience were included as participants in the study. Prior experience with LLMs was not a requirement. The participating engineers in this study had varying levels of code review expertise.

Interviews

The interviews were conducted to gain insight into participants' thoughts and opinions regarding code reviews and AI. There were two phases of in-person interviews, with the first phase of interviews conducted prior to generating code review comments with LLMs, and the second phase of interviews conducted after the generation of automated code reviews comments with LLMs. During the first interview phase the participants from each department formed a distinct interview group, and in the second interview phase all of the engineers were interviewed at once.

A focus group format was selected for the first interview phase to facilitate deeper discussion and more thoughtful reflection on the research topics. The participants were allowed to freely discuss the questions between each other to obtain multiple perspectives. Notes were taken manually for each interview. During the second interview phase, all participating engineers were interviewed together, where the focus was to introduce the LLM-generated code review comments. Both authors participated in all interviews, alternating the role of interviewer and note-taker. The first phase of interviews lasted around 45 minutes per interview group, and the second phase lasted about 1 hour. Each interview was restricted to 1 hour of planned time, but it was possible to cut them short during the first interview phase once all questions were answered.

The interview protocol was refined by trialing the questions in a pilot test where the authors took turns in acting as interviewer and interviewee. This refinement process eliminated question ambiguity and leading questions, improved the sequence and flow of interview questions, and helped narrow down the total number of questions. The interviews were conducted in Swedish, and the interview questions and protocol is presented in Section 10.1..

5.2.1. Phase 1: Eliciting Code Review Quality Criteria

During the first interview phase the engineers were asked about their perspective on the characteristics of good and useful code reviews, as well as the perceived benefits from having their code changes reviewed. Initially, 25 questions were developed which were later narrowed down to 15 questions for the first phase of interviews. The questions asked were deliberately open-ended, with the goal of obtaining a broader understanding of what is considered useful and what the expected benefits from code reviews are. The questions included in the interviews were based partly on the quality criteria that were developed from scientific literature. This was done to establish concrete similarities and

differences between the scientific and industrial perspectives regarding code review quality.

Prior to asking questions regarding their perspectives on code review quality, data regarding their experience surrounding both getting their code reviewed and conducting code reviews was collected. The participants were asked about how much time they spend reviewing code a typical week, as well as how often they get their own code reviewed. The interview also addressed software engineers' perspective on AI and its potential role in code review processes. Participants were asked to share their views on how AI-assisted code reviews might influence their workflow and development processes, and to identify potential opportunities and challenges they anticipated with the implementation of this technology.

Subsequently, the engineers were presented with questions regarding aspects that constitute a useful code review comment. The primary focus was on aspects mentioned in the code review comments themselves, but topics such as comment phrasing, tone, and length were also brought up to build a comprehensive understanding of what an ideal code review comment is from the ground up. The theoretical framework for the first interview phase is based on Section 4.4., and the finalized interview protocol is presented in Section 10.1..

5.2.2. Phase 2: Evaluating LLM-Generated Reviews

During the second interview phase the engineers evaluated the quality and usefulness of the LLM-generated code reviews. The engineers were presented with code review comments produced by the LLMs, as well as the code that prompted the LLM code review comments. The engineers were asked to evaluate the code review comments in regards to their own expectations established during the first interview phase. To ensure evaluations were based on their current opinions rather than pre-defined standards (even those they helped establish), engineers were not reminded of the previously discussed quality criteria. For each evaluation the engineers were also asked to categorize the code review comment based on the type of issue it addressed. This process allowed each code review comment to be further categorized by the perspective of the engineers. Subsequently, a concrete metric expressing the alignment of scientific literature and industry standards was established.

The code review evaluation interview was meant only to teach the engineers how to use the code review evaluation tool, which is briefly presented in Section 5.6.. During this interview they were shown a

few code review comments just to present the evaluation process. After this introductory session with the tool, engineers were asked to use it individually to evaluate the comments. The results from the second interview phase are presented in Section 6.2.2., and further information regarding the evaluation is presented in Section 5.6..

5.3. Selection of LLMs for Code Reviews

For generating code review comments, four LLMs were selected. The LLMs chosen consisted of four commercial general LLMs that were considered SotA as of April, 2025. Two of the selected models were non-reasoning models and the remaining models were reasoning models. The LLMs used in this study were selected through an examination of general benchmarks, as well as those evaluating LLMs' capabilities in solving software engineering problems that are commonly encountered in real-world industry scenarios. The software engineering related benchmark included in the evaluation was SWE-bench Verified [24] and the general benchmark was LiveBench [25]. It should be noted that the models' benchmarking performance does not necessarily indicate their ability to perform useful code reviews. The benchmarks primarily highlight the possible capabilities of a model, which is valuable when selecting the LLMs. Both reasoning and non-reasoning language models were included to examine whether there would be a significant difference in the quality and usefulness of the generated comments.

At the time of writing, Anthropic's Claude 3.7 Sonnet (claude-3-7-sonnet-20250219) model resolved a high percentage of SWE-bench Verified tasks at 70.3% [43], while also ranking highly with a 64.62 global average on LiveBench. Conversely, Google's Gemini 2.5 Pro Experimental (03-25) model achieved a very high global average on LiveBench [25] with 76.69 and performed competitively on SWE-bench Verified with 63.8% of tasks completed [44]. OpenAI's o4-mini resolved 68.1% of SWE-bench verified tasks [45] and scored 74.40 on LiveBench with a medium reasoning effort. Additionally, OpenAI's GPT-4.1 model scored 54.6% on SWE-bench Verified [46] and with a global score of 62.99 on LiveBench. Consequently, these four models were selected for generating code review comments. Other models such as OpenAI's o3-mini, Anthrophic's Claude 3.5 Sonnet v2, and DeepSeek R1 were also considered during the selection but were excluded due to performing worse at the examined benchmarks. o4-mini and Gemini 2.5 Pro are reasoning models, while GPT-4.1 and Claude 3.7 Sonnet are non-reasoning models.

	LiveBench (2025-04-25)	SWE-bench Verified
Claude 3.7 Sonnet	64.62	70.3%
GPT-4.1	62.99	54.6%
Gemini 2.5 Pro	76.69	63.8%
o4-mini	74.40	68.1%

Table 2. Benchmarking scores for the selected LLMs

5.4. Change Data Collection

This study utilized the code review open platform (CROP) [47] dataset for generating automated code review comments. The CROP dataset was harvested to be used for code review research, and comprises three components: dataset metadata, change discussions and related Git repositories. It contains code review data mined from open source projects that employ Gerrit as a code review platform.

CROP integrates Gerrit patch sets from Gerrit changes with commits from the Git repositories and compiles this into structured metadata. Additionally, the metadata contains references to discussions associated with specific changes or patch sets.

At the time of writing, CROP contains data from 11 different software projects with 50,959 code reviews and 144,906 revisions. The programming languages used in the CROP projects, which include Java, JavaScript, Python C++, and Go, vary across projects. The dataset covers code reviews from different time spans per repository, ranging from September 2009 to November 2017.

5.4.1. Milling the CROP

For this study, the CROP metadata and its included repositories were parsed into a single SQLite database containing changes, change revisions (patch sets) and individual files related to each change revision. The resulting database was then used to randomly select files for review comment generation.

The change revision entries contain the original *Change-Id* from Gerrit and the specific patch set number of the revision together with the initial change commit hash and the commit hash of the patch set. The files table entries are specific files changed related to one revision. The files table includes the file change type (e.g., modified, added, deleted, etc.), the current and old file paths (when applicable), and the blob hashes of the file before and after the change.

5.5. Prompting

Before generating the code review comments using the selected LLMs, a preliminary investigation into prompting techniques was carried out to improve the quality of LLM-generated code reviews. The prompting approach employed in this study focused on code-to-text generation with specific instructions. The three explored prompting techniques are described in Section 4.2..

Several approaches for instructing the LLMs were tested during prompt creation. Through testing different instruction-sets for the purpose of code review, several code review comments created through these different approaches were compared. The resulting code review comments generated throughout this brief trial period provided some insight into how to improve prompt instruction quality. It was found that when prompting the LLMs for code reviews, the resulting code review comments exhibited greater consistency when high-level aspects to consider were explicitly defined. When the LLMs were not provided with very explicit aspects to look for, the resulting code review comments were very inconsistent. Thus, the final prompt used for all the LLMs in this study contained the resulting quality criteria that are presented in Section 6.1..

5.5.1. Exploring Prompting Techniques

Different prompting techniques were evaluated before finalizing the prompt. Initially, zero-shot prompting was compared with few-shot prompting. These two prompting techniques were evaluated on a smaller set of code changes with the *Gemini 2.5 Pro* model. The zero-shot prompt contained no examples, whereas the few-shot prompt included seven examples of code review comments considered useful. The examples were randomly selected from the manually annotated code review comments usefulness dataset by Rahman et al. [48]. For each prompting technique, code review comments were generated for 100 code changes. Few-shot prompting was selected partly through a comparative analysis that looked at comment conciseness, tone neutrality, and comment consistency, and its documented ability to improve response accuracy and consistency [49], thus making comments more effective and reliable [50].

CoT prompting was also considered during the prompt evaluation phase. However, CoT prompting was not used in this study, as half the LLMs evaluated were already dedicated reasoning models. The aim was to explore both reasoning and non-reasoning models for code reviews.

The resulting prompt used in this study can be seen in Section 10.2.

5.5.2. Automating Code Review Prompting

Before generating the review comments, files were randomly selected for review. Comments were then generated for these files with an automated script. The four LLMs employed in this study were prompted with the same system prompt which was previously described, together with the files related to the change for the review. The files included with each review were the before and after files when applicable, and the diff between the before and after state of the changed file. The before file was not included when the file change type was *added*, similarly the after file was not included when the change type was *deleted*. This was done to reduce the amount of input tokens when prompting the LLMs.

Each code review was only conducted for the changes within a single file. Therefore, the response from the LLMs that conducted code reviews could contain multiple review comments for separate changes within a single file. All comments pertaining to a single file for a specific change revision were then fed into a separate comments table in the comment SQLite database which referenced the generated comments to the related file entry which the code review was based on.

5.6. Evaluation

To assess the quality of LLM-generated code reviews, each review was deemed either “useful” or “not useful” and further categorized by the characteristic of useful and not useful code reviews presented in Section 6.1.. This categorization allowed for analysis of what software defects LLMs most frequently find in code reviews, and what category is most frequently associated with a good code review comment in each context. Code review comments were generated by LLMs, and their usefulness was evaluated by two distinct parties. These two parties are: the authors of this study (based on scientific standards) and software engineers (based on industrial standards).

Comments were generated for 25 randomly selected changes by each of the 4 LLMs, for a total of 100 different pairings of LLM and code change, resulting in 278 different code review comments. GPT-4.1 generated 69 comments, o4-mini generated 67 comments, Claude 3.7 Sonnet generated 79 comments, and Gemini 2.5 Pro generated 63 comments. The authors evaluated all 278 of these code review comments. Due to time constraints, the engineers interviewed throughout this study were presented with a subset containing 50 of these code review comments that the authors deemed representative of the dataset as a whole. This

subset was chosen as it contained an even ratio of useful and not useful code review comments during author evaluation, and also a high author usefulness agreeability. Thus, it was expected to give a solid grounds for an author-engineer alignment investigation. These 50 comments were spread across 3 different changes and the 4 different LLMs. Out of these comments GPT-4.1 generated 11, o4-mini generated 15, Claude 3.7 Sonnet generated 16, and Gemini 2.5 Pro generated 8.

5.6.1. Code Review Evaluation Software

To ease the code review evaluation process, a tool for efficiently rating code review comments given a code change was developed. This came in the form of a web application, where each of the generated code review comments for a corresponding change were fetched from the database and displayed. The evaluating parties could then register their name and rate each code review comment to be either useful or not useful alongside what they deemed to be the category of the comment. The results of this would then be saved in the database. Figure 3 shows a code review comment to be evaluated in the evaluation software.

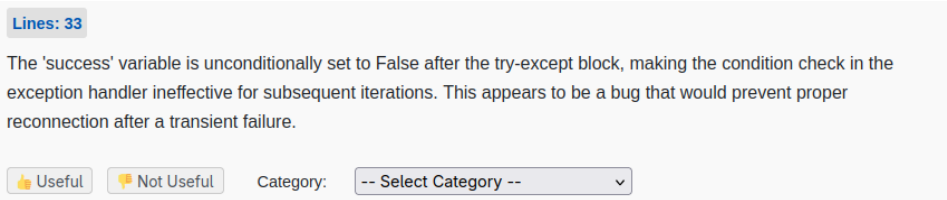

Figure 3. Rating screen for an LLM-generated code review comment

This evaluation tool also allowed for hosting on a web server, which made it possible for the engineers and authors to evaluate code review comments independently of the planned meetings. Individual evaluation allowed each evaluator to spend as much time as they needed for each of the code changes, and minimized influence between the individual engineers and authors.

5.6.2. Statistical Analysis of Evaluation Data

After the evaluation phase concluded, analysis of the data resulting from evaluation was conducted. Since the authors of this study rated each of the LLM-generated code reviews individually, the inter-rater reliability was assessed using Cohen's kappa. Similarly, as the engineers evaluated the code review comments individually, the inter-rater reliability between the engineers was assessed using Fleiss' kappa. The proportion of useful comments was also calculated for each of the LLMs according

to the evaluation results of each group. Finally, the resulting evaluation data between parties was compared.

The inter-rater reliability within the author and engineer evaluation groups was assessed with Cohen's kappa and Fleiss' kappa respectively. Inter-rater reliability was assessed by extracting usefulness and categorization data from the database and calculating Cohen's kappa and Fleiss' kappa coefficients using the *irr* package in R (4.5.0). The analysis between authors included all 278 categorized comments, and the analysis between participating engineers included the 50 comments selected for the engineer evaluation subset. Inter-rater reliability was assessed for the usefulness of comments, categorization of comments and after collapsing comment categories into three supercategories.

Chi-squared tests were employed to compare: (a) the proportions of useful comments among LLMs within each rater group (authors and engineers), and (b) the overall proportion of useful comments between the two rater groups. These analyses were performed using the `prop.test()` function from the *stats* package in R.

6. Results

To evaluate the usefulness of automated code reviews, this study first identifies the characteristics of high-quality human code review comments. The usefulness of LLM-generated code review comments is also evaluated. Code review quality criteria, as well as how different LLMs conform to these quality criteria is presented below.

6.1. Code Review Quality

The quality criteria for code review comments presented below were derived from a combination of scientific literature review and interviews with software engineers.

6.1.1. Scientific Criteria

Several studies also emphasized the importance of code maintainability, highlighting aspects such as code simplicity, readability, high-quality tests and debugging capabilities as critical factors. Other studies consider comments related to maintainability (evolvability defects) to be only *somewhat* useful compared to comments regarding functional and logical defects [8]. The majority of evolvability defect comments fall into one of the following categories: in-code documentation, visual representation of code, code organization and solution approach. Discussions focused on design considerations or new feature proposals and suggestions on alternative output or error messages were also considered *somewhat* useful.

In addition to clarifying what is deemed useful in a code review, some studies also discussed characteristics of comments deemed *not* useful. The criteria often deemed *not* useful for the code author include: incorrect identification of problems in code changes (possibly due to lack of expertise), questions posed by reviewers, and praise for the implementation. These are comments that either serve no purpose regarding the assessment of code quality and error prevention. Rather, they focus on other aspects such as educating the code reviewer rather than contributing to tangible improvements to the code itself.

Table 3 lists the most and least useful practices in code review comments as identified in the scientific literature.

Scientific Criteria

Useful Comments	Not Useful Comments
Pointing out functional and logical defects [7], [8], [14]	Asking for clarification of the functionality of a code snippet [8]
Finding unaccounted for validation issues or edge cases [8], [14]	Comments praising the code author [8]
Finding evolvability (maintainability) defects [7], [8], [42]	Commenting on potential future issues related to the change but not explicitly the change itself [8]
Teaching the code author [8], [14]	Formatting issues and other code style comments [14], [42]
Suggesting a solution that is more efficient or simpler [8], [14], [42]	

Table 3. Scientific literature on the usefulness of a code review comment

6.1.2. Industrial Criteria

Interviewing engineers at Saab gave additional perspectives on the nature of code reviews themselves. When the engineers were asked about the ratio of useful and not useful code reviews the answers varied drastically between the interview groups. The first two engineers that were interviewed claimed that they very rarely saw comments that are explicitly bad or not useful, rather they considered the vast majority of the code review comments they receive as useful or having some value to them. The usefulness of a code review was not exclusively placed in how it helps the codebase stay maintainable and bug free, rather they claimed there is also usefulness in the dialogue created between the reviewer and code author. However, in the second interview group the answers regarding this were the complete opposite. The second interview group claimed instead that they very rarely received code review comments that they considered useful.

Other than the nature of code reviews as a whole, the engineers gave further description of what is deemed a good code review comment. The engineers described various things they deem as useful findings of

a code review comment. Through this, frequent patterns in the nature of a good code review could be seen. The engineers expressed working with many codebases that interact with one another, and thus code that seems good in isolation may break code elsewhere. Finding issues related to functional defects was deemed the most important aspect of a code review. Similarly, the engineers expressed that there may be isolated issues in a code change that may have been missed by the code author. These were most often described as edge cases that the code author did not consider, and thus did not account for. These issues were also apparent as the engineers expressed the need for comprehensive testing for each change, as to make sure the code has no major functional defects.

Other than issues apparent at the time of the change, the engineers also expressed the need for writing code that does not cause issues in the future. The engineers explained that they often encounter code that is difficult to understand. They described this as one-liner solutions that may look nice, but feel impossible to debug. Additionally, one of the engineers explained that "code should not needlessly reinvent the wheel", meaning that a simple problem should be solved with a simple solution. Thus, guiding the code author to a solution that is simpler was deemed very useful. However, the engineers also expressed that this is something that is not always appreciated. If a suggested solution is only marginally more efficient or simple, it would not be worth the effort of implementing in the codebase, they stated.

Concerning the ability to read and understand code, one of the engineers described *"If the code doesn't do what it's supposed to, then it doesn't do what it's supposed to"*, which is to say that functions and other snippets of code should not have undescribed functionality. For instance, if a function is named `quicksort()` it should only perform a quicksort, and have no other functionality that does not align with its naming. If functions have undocumented functionality, these side effects would impede the ability of an engineer in the future in regards to effectively debugging the code.

As the engineers interviewed rarely used the scientifically defined terms referring to defects in code, Figure 4 builds the findings of a good code review from the ground up from what was expressed in the interviews. This categorization is based upon the code review categorization presented by Turzo and Bosu [51].

Industrial Criteria

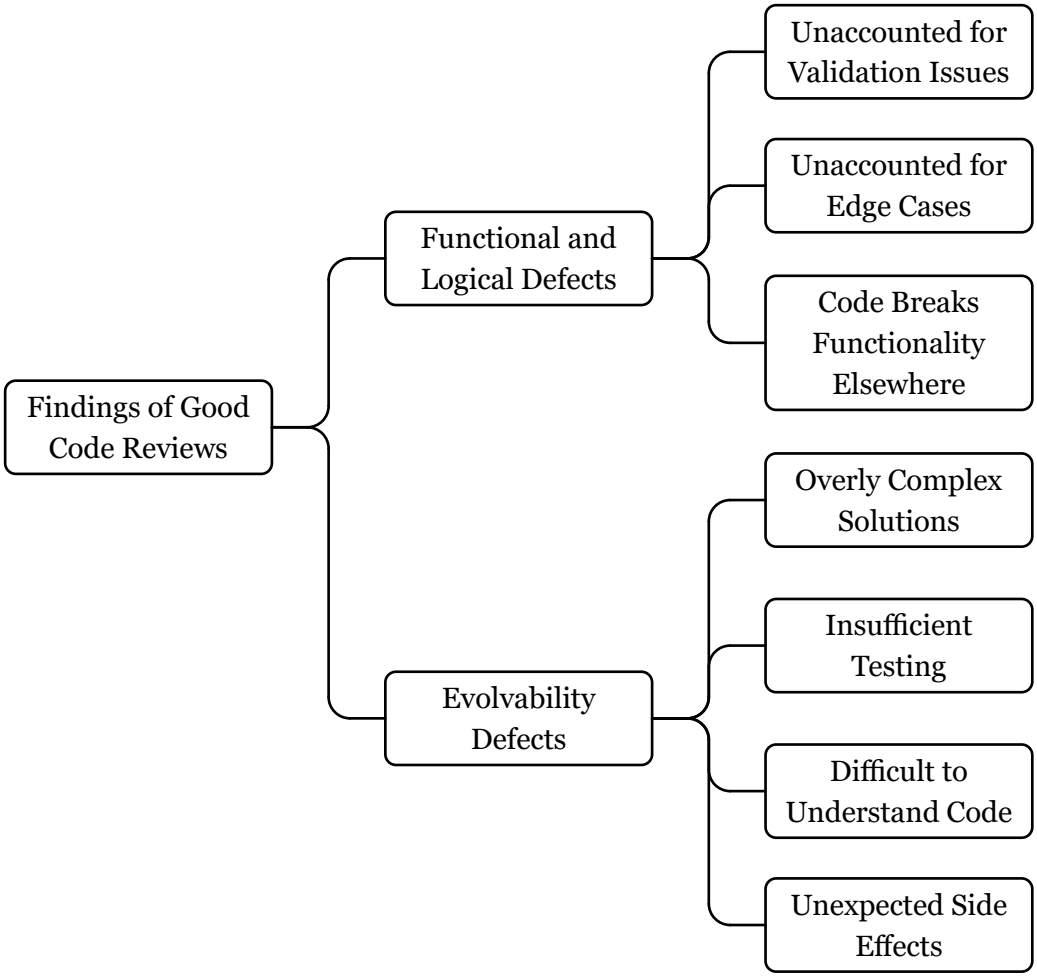

Figure 4. Engineers' perspectives on defect findings in good code review comments

Other than just finding flaws in the code, the engineers also expressed their perspectives on the tone and phrasing of a code review comment. They mentioned it should try to be as humble as possible, and the reviewer should refrain from explicitly stating that something is inherently bad. Rather, they should ask questions in regards to the nature of the change and instead help the author consider its quality themselves. Though if the change has inherent flaws in regards to code standard or other concrete errors, there should be proper background as to why. This aspect also proves useful in teaching the reviewer, especially when the reviewer lacks proper experience with the codebase.

According to the engineers, a good code review comment is not overly wordy, but has enough detail to clearly describe the issue it is referencing. Additionally, they mentioned that one large comment on the change as a whole is not preferred, rather separate small comments

referencing specific parts of the code are easier to digest and are more useful.

The engineers also described that to perform a good code review, the reviewer needs to look at other parts of the code rather than just the changed parts of it. The impact of a change is more apparent when looking at code surrounding it, and can give a deeper understanding of the need of the change. As stated by one of the engineers interviewed, *"As soon as a code review comment mentions parts of the code outside of the change itself, it becomes a good code review comment"*, highlighting the importance of examining the change context during a code review.

Table 4 lists the most and least useful practices in code review comments as identified by engineers at Saab.

Useful Comments	Not Useful Comments
Pointing out functional and logical defects	Pointing out issues without an explanation
Finding evolvability (maintainability) defects	Formatting issues and other code style comments
Finding unaccounted for validation issues or edge cases	Too short or too long comments
Finding a solution that is more efficient or simpler	Using a harsh or unfriendly tone

Table 4. Software engineers at Saab on the usefulness of a code review comment

6.2. LLM-Generated Code Review Evaluation Result

The evaluation of the LLM-generated code review comments is presented below. The evaluations were done by software engineers at Saab, Sweden, as well as the authors of this study. It is important to note that usefulness is inherently subjective; therefore, the reported proportions reflect perceived usefulness by each evaluating party.

6.2.1. Author Evaluation Results

Both authors independently rated all 278 LLM-generated code review comments as either "useful" or "not useful". Inter-rater reliability, measured using Cohen's kappa, was $\kappa = 0.51$ (95% CI [0.40, 0.62]), indicating a moderate level of agreement according to the guidelines of

Author Evaluation Results

Landis and Koch [32]. For the ten-category classification of comments, inter-rater reliability was lower, with $\kappa = 0.12$ (95% CI [0.063, 0.17]), reflecting only slight agreement. When these ten categories were collapsed into three broader super-categories—"Functional and logical defects", "Evolvability defects", and "Other"—inter-rater reliability improved to $\kappa = 0.36$ (95% CI [0.28, 0.45]), corresponding to a fair level of agreement. Figure 5 represents the agreement of comment categorization by the authors after collapsing categories.

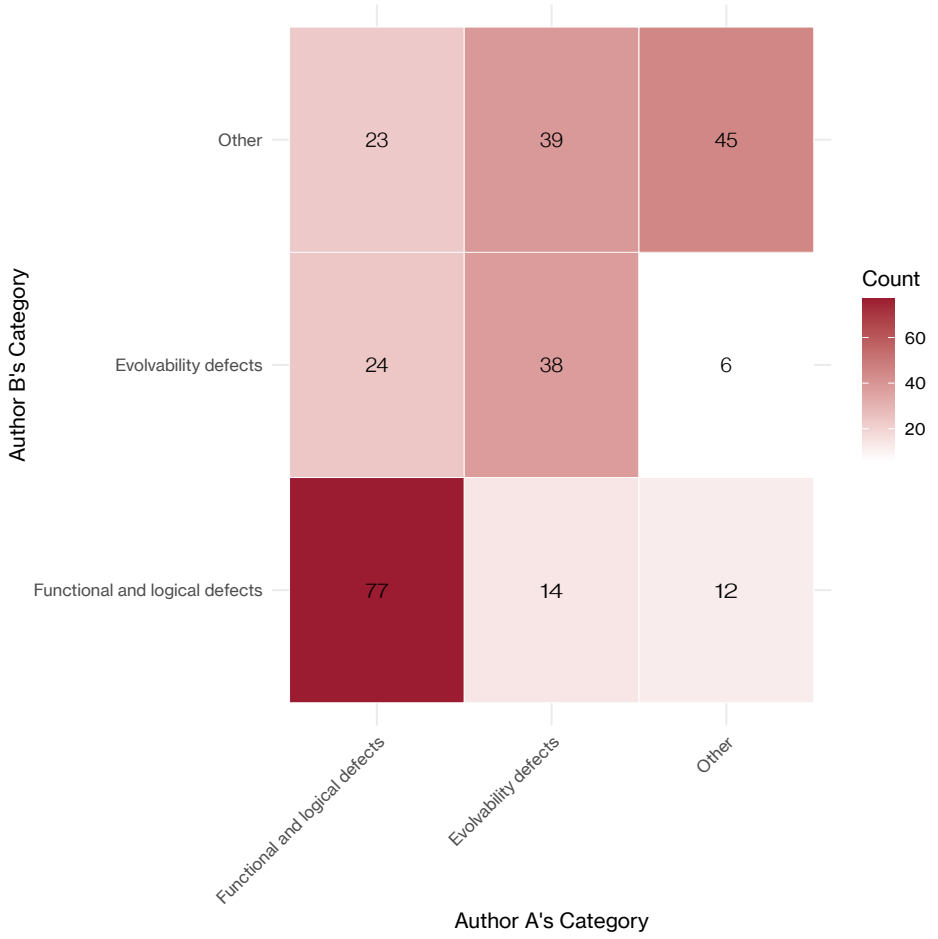

Figure 5. Agreement on comment categorization between authors after collapsing categories

Author evaluation of all LLM-generated code review comments had a perceived usefulness of 69.4% (95% CI [65.5%, 73.1%]). This value describes the total combined usefulness of all LLMs evaluated by both authors. For the subset of comments evaluated by the engineers, the percentage of useful comments was 58% (95% CI [48.2%, 67.2%]). The author evaluation process also revealed that, while both authors

individually rated an identical overall proportion of comments as useful, their agreement on which specific comments were useful was moderate ($\kappa = 0.51$). Table 5 shows the total amount of times each respective LLM was found "useful" and "not useful" according to author evaluation. Figure 6 plots the data of Table 5 based on perceived usefulness frequency. The y-axis shows the proportion of generated code review comments deemed useful for each model, with error bars representing the 95% confidence interval for these proportions.

	Useful	Not Useful	Total
GPT-4.1	98	40	138
o4-mini	95	39	134
Claude 3.7 Sonnet	103	55	158
Gemini 2.5 Pro	90	36	126

Table 5. Comment usefulness ratings for all models from the author evaluation

Author Evaluation Results

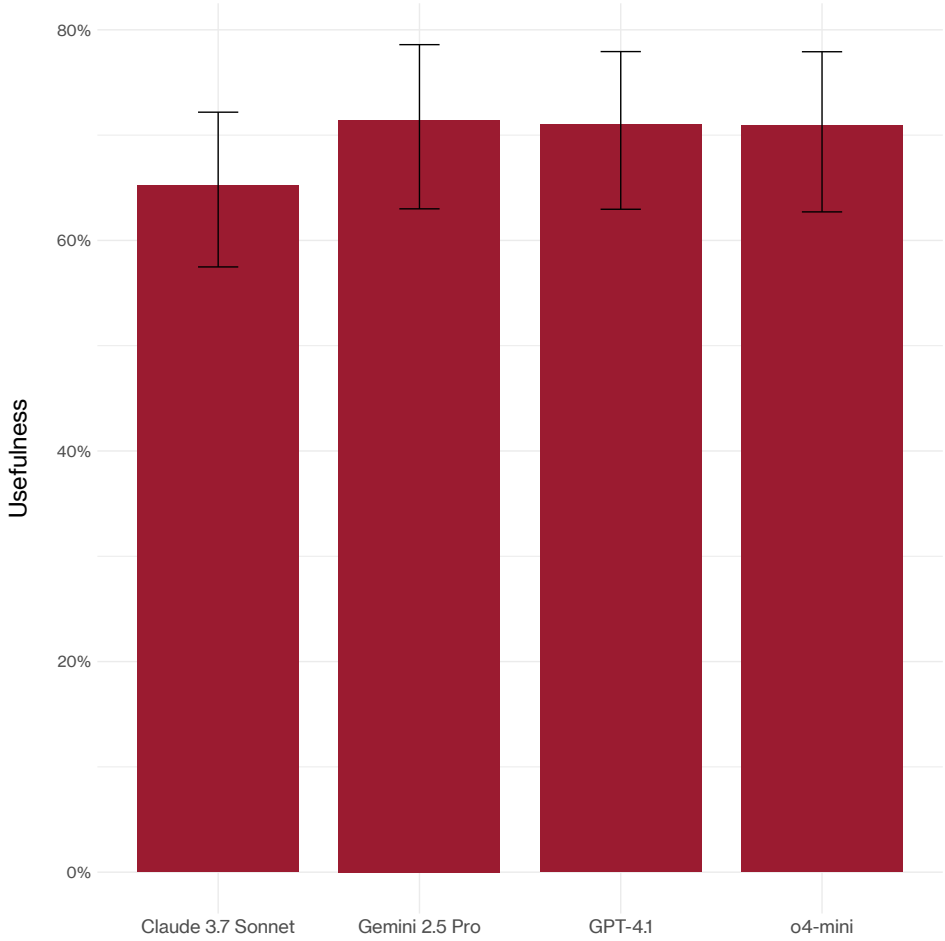

Figure 6. Proportion estimate of code review comment usefulness per LLM according to author evaluation

Per-model usefulness from the author evaluation was estimated as 65.2% (95% CI [57.5%, 72.2%]) for Claude 3.7 Sonnet, 71.4% (95% CI [63.0%, 78.6%]) for Gemini 2.5 Pro, 71.0% (95% CI [63.0%, 77.9%]) for GPT-4.1, and 70.9% (95% CI [62.7%, 77.9%]) for o4-mini. A chi-squared test was conducted to determine whether there were significant differences in usefulness among the four LLMs. The results showed no significant difference ($\chi^2(3, N = 556) = 1.87, p = 0.60$).

6.2.2. Engineer Evaluation

5 out of the 6 participating engineers completed the subset evaluation process. The participating engineers individually rated the subset containing 50 code review comments as either “useful” or “not useful”. Inter-rater reliability, measured using Fleiss’ kappa, was κ = 0.0122, which indicates a slight agreement between the engineers. The inter-rater reliability for the ten-category classification was κ = 0.136 which

indicates a slight agreement and after collapsing the categories into three super-categories, the kappa was $\kappa = 0.319$, which indicates fair agreement.

Table 6 presents the number of "useful" and "not useful" ratings, as well as the total number of comments for each LLM, as evaluated by the engineers participating in this study. The percentage of comments rated as useful was 92.8% (95% CI [88.9%, 95.4%]), representing the point estimate of the proportion of useful comments in this sample.

	Useful	Not Useful	Total
GPT-4.1	52	3	55
o4-mini	69	6	75
Claude 3.7 Sonnet	75	5	80
Gemini 2.5 Pro	36	4	40

Table 6. Comment usefulness ratings for all models from the engineer evaluation

Figure 7 plots the usefulness proportions for each model based on the data from Table 6 for the engineer group, analogous to the presentation in Figure 6.

Engineer Evaluation

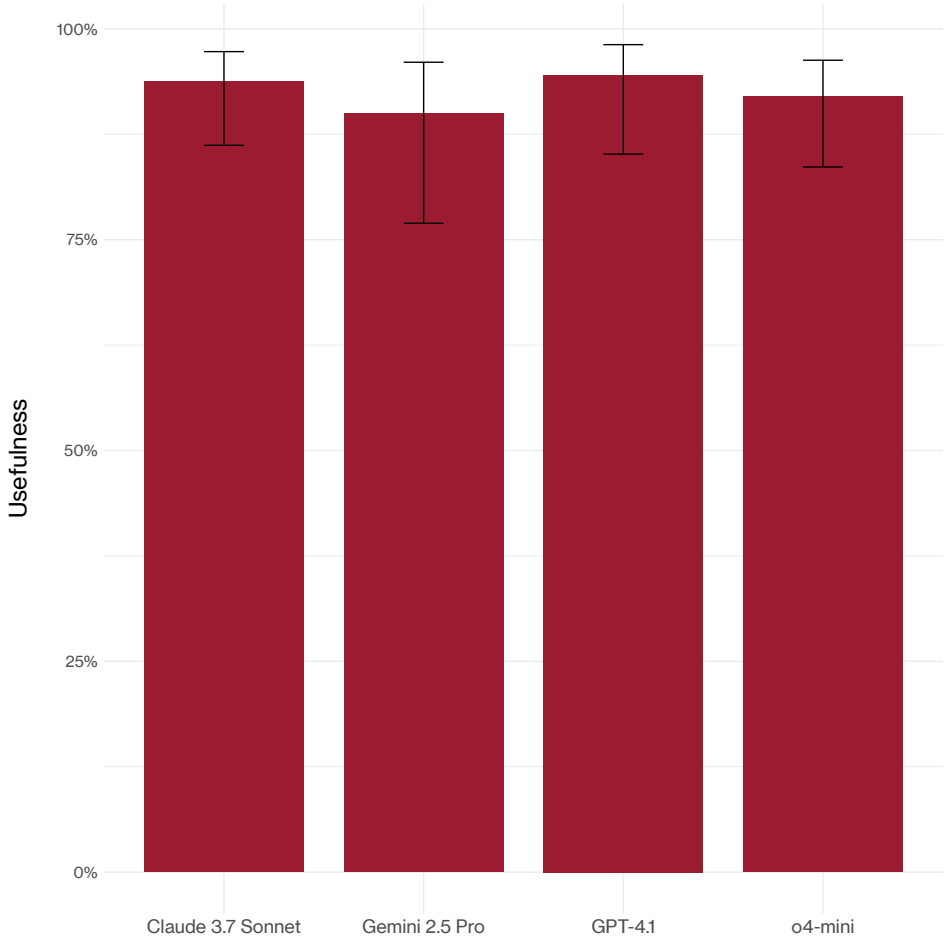

Figure 7. Proportion estimate of code review comment usefulness per LLM according to engineer evaluation

Per-model usefulness based on the engineer evaluation was estimated as 93.8% (95% CI [86.2%, 97.3%]) for Claude 3.7 Sonnet, 90.0% (95% CI [76.9%, 96.0%]) for Gemini 2.5 Pro, 94.5% (95% CI [85.1%, 98.1%]) for GPT-4.1, and 92.0% (95% CI [83.6%, 96.3%]) for o4-mini. No statistically significant differences in usefulness ratings were detected among the models according to the chi-squared test ($\chi^2(3, N = 250) = 0.90, p = 0.83$).

6.2.3. Author and Engineer Evaluation Alignment

Inter-rater reliability among all raters, including participants from both the author and engineer groups, was assessed for the subset of comments using Fleiss' kappa. Agreement was calculated for comment usefulness, the original ten-category classification, and after collapsing categories. The kappa value for comment usefulness was κ = 0.06, indicating a slight agreement, ten-categorization resulted in κ = 0.143, also reflecting slight

agreement, while the collapsed category classification showed moderate agreement with κ = 0.36.

To evaluate whether there is a significant difference in the proportion of useful ratings between the author and engineer groups for the subset of 50 comments, a chi-squared test was conducted. The results revealed a significant difference between the groups ($\chi^2(1, N = 350) = 60.9, p < 0.0001$). Specifically, the engineer group rated the LLM-generated comments as useful 25 to 45 percentage points more often than the author group, with a 95% CI for the difference in proportions of [0.25, 0.45].

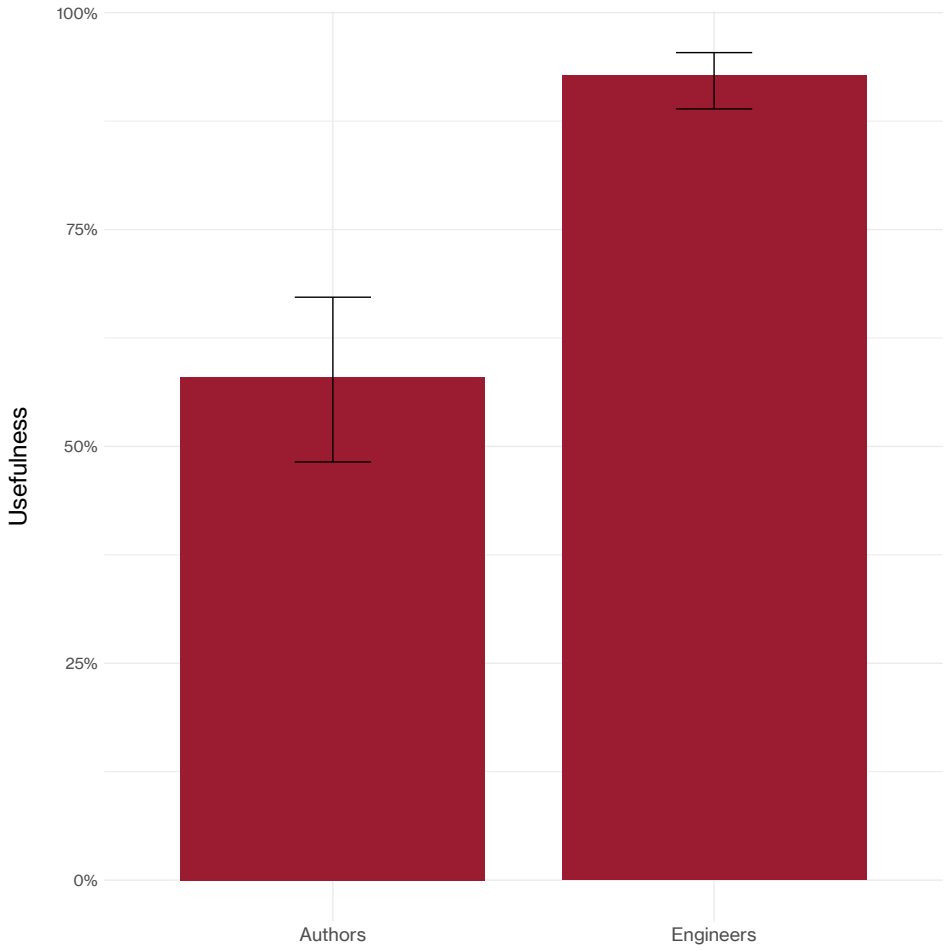

Figure 8. Proportion estimate of code review comment usefulness per group for subset of comments

Although there is noticeable difference between the usefulness proportion between authors and engineers, a brief analysis of the category proportion of each evaluation group showed that both the authors and engineers had a similar distribution of each comment

category. The absolute difference in categorization proportion between both parties never exceeds 5 percentage points. The category with the most significant proportion difference is “Attempting to teach code author”, with a difference of 4.4 percentage points. The differences for some categories are also quite minimal, with the smallest difference being “Unexpected side effect”, at 0.1 percentage points. Figure 9 shows the frequency of each category selected for each party.

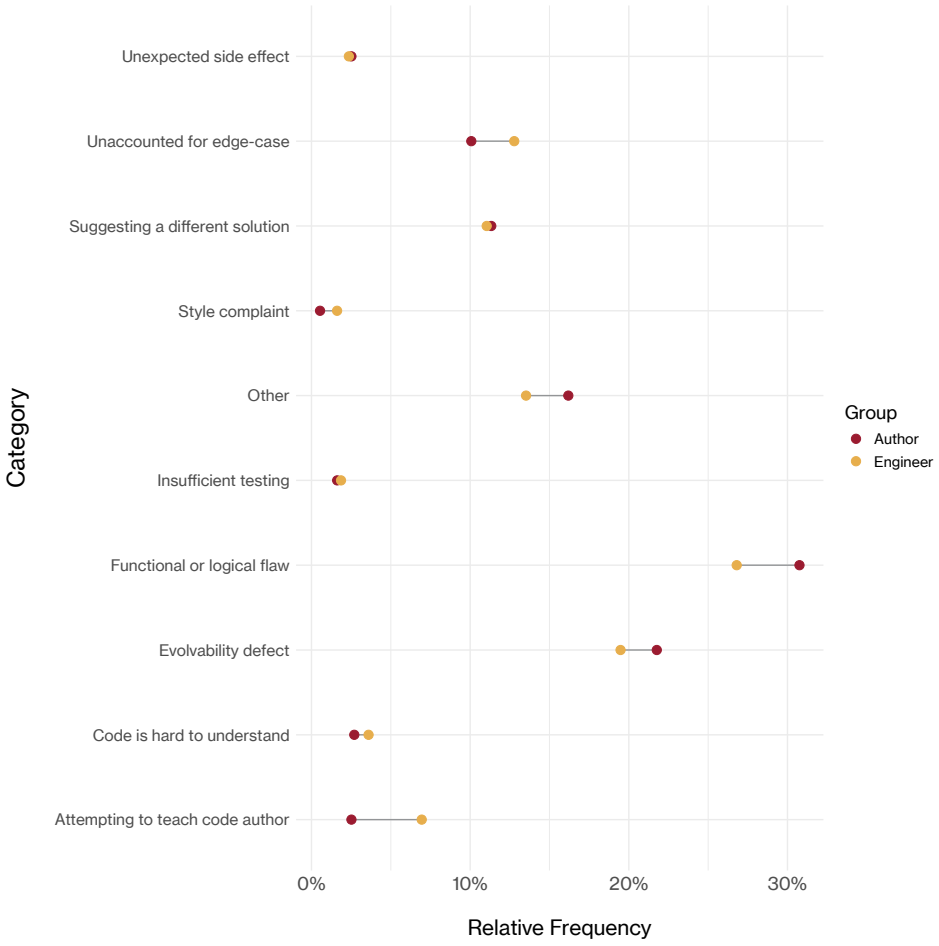

Figure 9. Percentage of comments in each category after comment categorization relative to total number of rated comments by group

6.3. Research Question Answers

This study answers the following questions:

RQ1: What specific characteristics define useful code review comments as identified in scientific literature, and how do these align with perspectives from professional software development practice?

Research Question Answers

This study reveals that the definition of a useful code review comment is nuanced, influenced by personal opinions and varying code standards across codebases. However, through the analysis of both scientific literature and personal perspective from professional software engineers, patterns among the different contexts reveal aspects of a code review comment universally considered useful.

The most concrete characteristic of a good code review comment is the identification of functional and logical defects. This aspect of a code review comment was mentioned in nearly all the scientific literature regarding code review quality examined throughout this study. It is also the aspect each engineer interviewed could come to a universal consensus on in regards to its importance. The findings show that the main purpose of conducting code reviews is that of finding functional defects, and is thus cemented as the most useful characteristic of a code review comment in both a scientific and industrial context.

An additional universally deemed useful characteristic of a code review comment is the mention of evolvability defects in the code change. This refers to the value of the change in the long-term progression of the codebase. A change may work at the time it is completed, but if the change has issues in regards to maintaining it (i.e understanding it or debugging it at a later point in time), issues may arise. The examined scientific literature discusses evolvability defects in great detail, highlighting their importance. The engineers interviewed throughout also concluded that the mention of issues in regards to evolvability are useful and of great importance.

Finally, another universally deemed useful code review characteristic is that of suggesting an alternate solution that is simpler or more efficient. This characteristic is similar to the maintainability flaws previously mentioned, but takes a different approach with a focus on primarily making the software run as efficiently as possible and not needlessly reinventing the wheel. While both scientific literature and the interviewed engineers deem this characteristic useful, its perception within code review is somewhat unique. Although the interviewed engineers stated suggesting more efficient solutions as a useful characteristic, they also explained that in cases where the solution is only marginally more efficient or simple, the perceived usefulness of the comment drops. This is an aspect not captured in the scientific literature examined throughout this study, and shows that the same code review comment may differ in usefulness depending on if it is evaluated in isolation or alongside other comments.

Research Question Answers

The scientific literature aligns strongly with professional perspective in terms of categorization surrounding what a defects are pointed out in a code review comment. However, there is a slight divergence when it comes to code review comment structure, tone, and phrasing. The engineers stressed the importance of a well-written code review comment, which is an aspect the scientific literature rarely mentioned. Although not a direct contradiction, it shows that the human aspect of a code review comment is not as black-and-white as finding defects in code. This was also shown amongst the engineers themselves, as although the engineers interviewed worked on the same codebase and company, the importance of human discussion and interaction varied. Ultimately, the human aspect was frequently deemed important among the engineers, yet is something not frequently expressed in scientific literature.

Overall the scientific literature aligns well with the perspective of professional software engineers, but describes differing nuances that relate to the specific experiences of different engineers and standards of different codebases. Table 7 compiles the useful and not useful aspects discussed in Section 6.1., and visualizes alignment of scientific literature and industrial perspective.

> RQ1: *The specific characteristics that define a useful code review are presented in Section 6.1., Table 7, and in the justification above. The characteristics described in scientific literature align well with perspectives from professional software development practice, but differ in terms of nuance depending on individual developer perspective and experience.*

Research Question Answers

	Scientific Literature	Industrial Perspective
Pointing out functional and logical defects	Useful	Useful
Finding unaccounted for validation issues or edge cases	Useful	Useful
Finding evolvability (maintainability) defects	Useful	Useful
Suggesting a solution that is more efficient or simpler	Useful	Useful
Teaching the code author	Useful	N/A
Formatting issues and other code style comments	Not Useful	Not Useful
Too long or too short comments	N/A	Not Useful
Using a harsh or unfriendly tone	N/A	Not Useful
Pointing out issues without an explanation	N/A	Not Useful
Commenting on potential future issues related to the change but not explicitly the change itself	Not Useful	N/A
Asking for clarification of the functionality of a code snippet	Not Useful	N/A
Comments praising the code author	Not Useful	N/A

Table 7. Alignment of scientific literature and industrial perspective on the usefulness of a code review comment, N/A refers to the lack of mention by respective party

RQ2: How do different automated code reviews compare in their perceived usefulness to developers when evaluated against established code review quality criteria?

This study presents the perceived usefulness of LLM-generated code review comments through both the scientific and industrial perspective. The results show that there are distinct differences in the perceived usefulness among the two different perspectives. The evaluation of LLM-generated code review comments by industry professionals resulted in a noticeably higher ratio of comments perceived as useful than the evaluation by the authors of this study. Figure 10 shows an LLM-generated code review comment that all participating engineers and both study authors perceived as useful.

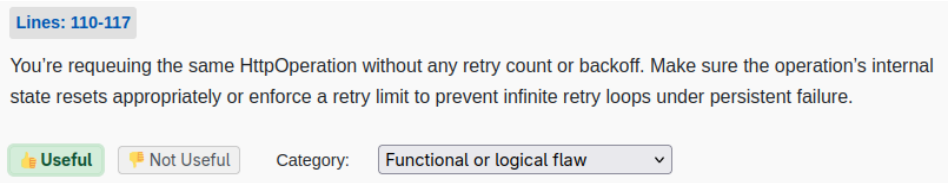

Figure 10. Illustrative example of what was considered a unanimously useful code review comment in this study

Participating engineers and study authors alike found the LLM-generated code review comments to be predominantly useful. The participating engineers perceived the code review comments as useful 92.8% of the time. Meanwhile, the study authors perceived the code review comments as useful 69.4% of the time. However, observing just the engineer evaluated subset of 50 code review comments shows that the increase in usefulness frequency between engineer and author evaluation may be even higher, as the perceived usefulness frequency from author evaluation of the same subset of 50 code review comments was 58%.

The results indicate that although the code review quality criteria show strong alignment between both industrial and scientific perspective, the LLMs generate a higher ratio of code review code review comments when evaluated against industrial code review quality criteria.

> RQ2: *Overall the LLM-generated code review comments were perceived as frequently useful when evaluated against both scientific and industrial code review quality criteria. The perceived usefulness following the industrial quality criteria showed a noticeably higher usefulness frequency than the scientific quality criteria. Exact usefulness frequency metrics are presented in Section 6.2. and the justification above.*

6.4. Evaluation Dataset

The evaluation process yielded a dataset comprising LLM-generated code review comments. This dataset includes corresponding usefulness evaluations from both the authors and engineers. The complete evaluation dataset is provided as supplementary material accompanying this thesis. To link the LLM-generated comments and evaluations within this dataset to specific code changes, the CROP dataset [47] is required.

7. Discussion

Potential shortcomings and limitations impacting the validity and contributions of the study are discussed below.

7.1. Difficulties Encountered

Initial attempts at prompting the LLMs proved challenging, as the LLMs initially struggled with conducting reviews. Rather than finding current flaws in the code, they would primarily focus on non-issues. Several times the LLMs would form comments such as “You replaced x with y, please make sure that they reproduce the same logic”, which are comments that would not be useful in a real-world use case. Additionally, small differences in the prompts showed major differences in token amount used for reasoning, even when there was seemingly no difference in the resulting review comments. This could lead to economic, ecological, and performance drawbacks when implemented at scale.

Establishing code review quality criteria was a complex task throughout this study. The subjective nature of what constitutes a useful code review comment inevitably introduces bias, a factor we acknowledge as a significant limitation in this study’s results. Despite this, we established a reasonably solid foundation for evaluating code review comments. Since the study covers a wide range backgrounds and experience for respective perspectives on the usefulness of a code review comment, we feel that although not flawless, the method employed in this study provides a valuable insight into the actual usefulness of LLM-generated code reviews.

For one of the code changes that this study was going to look at, there was a slight issue with how the code was structured such that when the LLMs quoted the code in their code review comments there was an XML parsing error, causing the evaluation tool to assume all LLMs had found no issues with the change. During the second interview phase, we wanted to select one change with no issues found for the purpose of open discussion. We accidentally picked a file that actually had several code review comments generated, but did not realize until the engineers found several issues with the code during the open discussion. After further inspection, we realized this issue and fixed it. After fixing this issue, all the issues of the code change found by the engineers during this

discussion were then discovered to have also been found by the LLMs, which could be a positive indicator of the LLMs' code review capabilities. However, we decided to remove this example from the final results as it was never intended or accounted for.

7.2. Study Limitations

Several limitations arose throughout the planning and execution of this study. The biggest limiting factor was the time constraint. The limited amount of time impacted the amount of engineers to be interviewed for the study, and also impacted the amount of LLM-generated code review comments to be evaluated not only by the authors of the study, but also by the participating engineers.

The time factor also limited the depth of evaluation for each conducted code review. Each code review comment was limited to binary evaluation (i.e useful or not useful) as well as a categorization based on what the comment is referencing. This becomes an issue because as discussed in this study, the quality of a code review comment may also be impacted by its phrasing, tone, and length. Therefore, a binary 'useful'/'not useful' classification does not capture the full picture. Additionally, the study only looks at the comments created by the LLMs, and not the lack thereof. The LLMs were not evaluated based on missed issues in the code they reviewed, rather they were only reviewed on what they managed to find.

The prompt used for the LLMs in this study was engineered through a set of evaluations comparing several different system prompts. This set of evaluations was only done on *Gemini 2.5 Pro*, and the resulting prompt was used for all models evaluated in this study. The same prompts may have different outputs depending on the model they are used with, which might have given the *Gemini 2.5 Pro* model an unfair advantage when comparing the efficacy of different LLMs in this study. The decision to only use Gemini for the prompt engineering evaluation was due to the cost of conducting this set of evaluations, which was not a factor for Gemini as Google Cloud provides a $300 USD credit to each new user. This decision could also negatively impact the performance of the prompts

Limited codebase context may have also had an impact on the code review usefulness. In the LLM-generated code review comments evaluated in this study, each LLM was limited to single file change context. Providing the LLMs with the entire codebase may have given

them a greater opportunity to comment on changes which impacted files outside of the change.

7.3. Validity Threats

During the LLM-generated code review comment evaluation process, the agreeability between the authors is not very high. This shows the difficulty in properly following evaluation guidelines, as well as potential bias in the author evaluation results. The low inter-rater agreement on comment categorization between authors was likely a result of the vagueness in prior scientific literature regarding definitions of what is to be considered an evolvability defect, what is considered a functional defect, and what is considered neither. Frequently the category disagreement was not due to a functional defect being confused with an evolvability defect, but instead due to the fine line between what is one of the established defect categories, and what is categorized as other.

One major flaw in LLMs is the inconsistency factor, which is not accounted for in this study. There is a small but probable chance that the results of this study are the result of an irreproducible sequence, and thus may not apply in a real world setting. However, throughout our testing we saw consistent results from the LLMs, which makes us believe that the odds are in favor of the results of this study.

Since the study only looks at a rather small number of code changes reviewed by the LLMs, there is a likely chance of sampling error (i.e the observed usefulness frequency might be different when looking at a larger sample). Looking at a larger sample could either reveal that LLMs are better or worse at conducting code reviews than what was observed in this study.

The CROP dataset contains exclusively older code changes (up to November 2017). This means that LLMs may have been trained on a substantial amount of code using similar language versions, potentially affecting their performance on this specific dataset compared to more modern code. It is very likely that the LLMs would perform noticeably worse on very recent revisions of the languages used, as there is likely very limited data surrounding how to properly use them in the dataset used to train the LLMs. For instance, Gemini 2.5 Pro has a knowledge cutoff of January 2025, so code changes using programming languages released after that date could not be properly reviewed by that model.

Although scientific literature and the perspective of the engineers showed significant alignment, the practical evaluation phase of this study

saw a significant divergence to this. One possible explanation for this observation is that the engineers are less receptive of feedback from regular code review comments as re-testing and remaking a pull request is a very tedious process. This study evaluates LLM-generated code review comments, which can provide feedback before any necessary administrative tasks. As a result of this, the immediate feedback is likely more welcome. This suggests that LLM-generated code review comments follow their own separate quality criteria, rather than the already established criteria for regular code review comments.

7.4. Ethical Implications

The integration of LLMs into the code review process may have significant ethical implications, especially for less experienced developers. An overreliance on the LLM tooling can, for instance, cause deskilling among engineers. As previously established in the study, code reviews constitute a learning opportunity. Removing this learning opportunity could make getting used to the codebase significantly slower. An overreliance of LLMs could also create a mental blockade of sort. Engineers may be less concerned about actually writing high quality code due to thinking the code review software will point out all issues in the code, discouraging critical thinking.

Subsequently, if a developer assumes everything the LLM states as fact it could cause severe issues in cases where they are very wrong. If an LLM suggests a different solution to a problem which is then implemented without question, who is to blame in cases where this code is damaging to the codebase? Similar issues may arise as a result of LLM training bias.

There are also concerns of eventual misinterpretation of this study. Employers could read this study and assume that LLMs are an alternative to human code reviewers. This misinterpretation could result in the employer making the decision to reduce their workforce, since humans would be assumed to no longer be necessary for code reviews.

7.5. Future Work

Several factors remain unexplored by this study. For instance, prompt engineering plays a major role in this type of work, and is an area largely untouched by this study. A future study could compare the efficacy of LLM-generated code review comments in regards to different prompting strategies, such as comparing few-shot and zero-shot prompting.

Future studies could also potentially employ the method used in this study, but evaluating it with the code authors themselves. It is quite

difficult for someone to assess the usefulness of a code review comment when they themselves are not on the receiving end. Having the LLM-generated code review comments evaluated by the authors of the reviewed code could potentially yield different results. Aspects such as the initial aim of the code author is something that is not captured throughout this study, and could potentially warrant different study on its own.

This study also does not look into the economic implications of integrating LLMs into the code reviewing process. The LLMs compared in this study vary significantly in cost. It may be worth going for a slightly worse performing LLM in exchange for it being significantly cheaper. Other aspects, such as the feasibility and cost of running LLMs on private servers for security reasons, also warrant exploration.

The code reviews conducted using LLMs in this study are limited to static code analysis. If the LLMs had the ability to use debugging tools or runtime analysis tools a much deeper analysis could be done. Employing both static code analysis and runtime code analysis would likely significantly alter the results, and could be an interesting aspect to explore in a future study.

A comparison between LLM conducted and human conducted was also not done in this study. We exclusively looked at the usefulness of LLM-generated code review comments, and although the results speak on the usefulness of LLM-generated code review comments, it does not explore how they compare to human conducted code reviews. Even if LLM-generated code review comments are somewhat useful in this context, if LLM-generated reviews are significantly less useful than human reviews, their integration into professional workflows may not be warranted.

8. Conclusion

This study investigated the efficacy and most frequent characteristics of code review comments generated by different SotA LLMs. This investigation was conducted through an evaluation based on an interpretation of scientific quality criteria, as well as an evaluation from industry professionals. The study first establishes comprehensive code review quality criteria by synthesizing insight from both scientific literature and experiences of software engineers at Saab. This approach revealed significant alignment on core defect finding targeting primarily evolvability and functional defects alongside nuanced differences in the perceived value of feedback tone and phrasing, providing a solid basis for evaluating code review quality.

Subsequently, we developed and employed a method for generating and evaluating the usefulness of LLM-generated code review comments. The evaluation method was completed by both academics and industry professionals. The findings indicate that the SotA LLMs are capable of frequently generating useful code review comments according to both participating parties.

In conclusion, while LLMs are not a substitution for human expertise, this study demonstrates their growing potential as powerful assistive code review tools. Their ability to provide rapid and consistently useful feedback on certain aspects of code quality suggests that incorporating LLMs into the code review process may become increasingly appealing as the technology advances further. As the LLM technology continues to mature, its integration into software engineering workflows is poised to become increasingly valuable, reshaping how we approach code quality assurance.

9. Bibliography

[1] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 181–190. doi: 10.1145/3183519.3183525.

[2] J. Czerwonka, M. Greiler, and J. Tilford, "Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015, pp. 27–28. doi: 10.1109/ICSE.2015.131.

[3] 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*, Oct. 2013, pp. 133–142. doi: 10.1109/ESEM.2013.23.

[4] 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*, in ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 202–212. doi: 10.1145/2491411.2491444.

[5] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 111–120. doi: 10.1109/ICSM.2015.7332457.

[6] H. Siy and L. Votta, "Does the modern code inspection have value?," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 2001, pp. 281–289. doi: 10.1109/ICSM.2001.972741.

[7] M. Mantyla and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, May 2009, doi: 10.1109/tse.2008.71.

Bibliography

[8] A. Bosu, M. Greiler, and C. Bird, “Characteristics of Useful Code Reviews: An Empirical Study at Microsoft,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 146–156. doi: 10.1109/MSR.2015.21.

[9] A. Vaswani *et al.*, “Attention Is All You Need.” [Online]. Available: https://arxiv.org/abs/1706.03762

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” [Online]. Available: https://arxiv.org/abs/1810.04805

[11] J. T. Liang, C. Yang, and B. A. Myers, “A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, in ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. doi: 10.1145/3597503.3608128.

[12] M. E. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976, doi: 10.1147/sj.153.0182.

[13] “About pull requests.” Accessed: Apr. 16, 2025. [Online]. Available: https://docs-internal.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

[14] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 712–721. doi: 10.1109/ICSE.2013.6606617.

[15] T. Baum and K. Schneider, “On the Need for a New Generation of Code Review Tools,” in *Product-Focused Software Process Improvement*, P. Abrahamsson, A. Jedlitschka, A. Nguyen Duc, M. Felderer, S. Amasaki, and T. Mikkonen, Eds., Cham: Springer International Publishing, 2016, pp. 301–308. doi: 10.1007/978-3-319-49094-6_19.

[16] G. Foster, “Why AI will never replace human code review.” Accessed: May 12, 2025. [Online]. Available: https://graphite.dev/blog/ai-wont-replace-human-code-review

Bibliography

[17] M. Vijayvergiya *et al.*, "AI-Assisted Assessment of Coding Practices in Modern Code Review," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, in AIware 2024. Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 85–93. doi: 10.1145/3664646.3665664.

[18] Y. Almeida *et al.*, "AICodeReview: Advancing code quality with AI-enhanced reviews," *SoftwareX*, vol. 26, p. 101677, 2024, doi: https://doi.org/10.1016/j.softx.2024.101677.

[19] H. Y. Lin, P. Thongtanunam, C. Treude, and W. Charoenwet, "Improving Automated Code Reviews: Learning From Experience," in *Proceedings of the 21st International Conference on Mining Software Repositories*, in MSR '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 278–283. doi: 10.1145/3643991.3644910.

[20] H. Naveed *et al.*, "A Comprehensive Overview of Large Language Models." [Online]. Available: https://arxiv.org/abs/2307.06435

[21] Z. Zhang, A. Zhang, M. Li, H. Zhao, G. Karypis, and A. Smola, "Multimodal Chain-of-Thought Reasoning in Language Models." [Online]. Available: https://arxiv.org/abs/2302.00923

[22] A. Lewkowycz *et al.*, "Solving Quantitative Reasoning Problems with Language Models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., Curran Associates, Inc., 2022, pp. 3843–3857. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/18abbeef8cfe9203fdf9053c9c4fe191-Paper-Conference.pdf

[23] T. Brown *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[24] C. E. Jimenez *et al.*, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?." Accessed: Apr. 07, 2025. [Online]. Available: http://arxiv.org/abs/2310.06770

[25] C. White *et al.*, "LiveBench: A Challenging, Contamination-Free LLM Benchmark," in *The Thirteenth International Conference on Learning Representations*, 2025.

Bibliography

[26] J. Oppenlaender, "A Taxonomy of Prompt Modifiers for Text-To-Image Generation," *Behaviour & Information Technology*, vol. 43, no. 15, pp. 3763–3776, Nov. 2024, doi: 10.1080/0144929X.2023.2286532.

[27] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." [Online]. Available: https://arxiv.org/abs/2201.11903

[28] "Reasoning best practices ." Accessed: Apr. 28, 2025. [Online]. Available: https://platform.openai.com/docs/guides/reasoning-best-practices

[29] T. Kojima, S. (. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large Language Models are Zero-Shot Reasoners," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., Curran Associates, Inc., 2022, pp. 22199–22213. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf

[30] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia Medica*, vol. 22, no. 3, pp. 276–282, Oct. 2012, Accessed: May 20, 2025. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3900052/

[31] K. A. Hallgren, "Computing Inter-Rater Reliability for Observational Data: An Overview and Tutorial," *Tutorials in Quantitative Methods for Psychology*, vol. 8, no. 1, pp. 23–34, 2012, doi: 10.20982/tqmp.08.1.p023.

[32] J. R. Landis and G. G. Koch, "The Measurement of Observer Agreement for Categorical Data," *Biometrics*, vol. 33, no. 1, p. 159, Mar. 1977, doi: 10.2307/2529310.

[33] D. S. Moore, G. P. McCabe, and B. A. Craig, *Introduction to the Practice of Statistics*, 9th ed. New York: W.H. Freeman & Company, 2016.

[34] K. Pearson, "X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, Jul. 1900, doi: 10.1080/14786440009463897.

Bibliography

[35] W. G. Cochran, “Some Methods for Strengthening the Common χ2 Tests,” *Biometrics*, vol. 10, no. 4, pp. 417–451, 1954, doi: 10.2307/3001616.

[36] R. A. Krueger and M. A. Casey, “Focus Group Interviewing,” *Handbook of Practical Program Evaluation*. John Wiley & Sons, Ltd, pp. 506–534, 2015. doi: 10.1002/9781119171386.ch20.

[37] J. Kitzinger, “Qualitative research. Introducing focus groups,” *BMJ (Clinical research ed.)*, vol. 311, no. 7000, pp. 299–302, Jul. 1995, doi: 10.1136/bmj.311.7000.299.

[38] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Apr. 2009, doi: 10.1007/s10664-008-9102-8.

[39] C. Seaman, “Qualitative methods in empirical studies of software engineering,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, Jul. 1999, doi: 10.1109/32.799955.

[40] H. Kallio, A.-M. Pietilä, M. Johnson, and M. Kangasniemi, “Systematic methodological review: developing a framework for a qualitative semi-structured interview guide,” *Journal of Advanced Nursing*, vol. 72, no. 12, pp. 2954–2965, 2016, doi: 10.1111/jan.13031.

[41] R. Chenail, “Interviewing the Investigator: Strategies for Addressing Instrumentation and Researcher Bias Concerns in Qualitative Research,” *The Qualitative Report*, vol. 16, no. 1, pp. 255–262, Jan. 2011, doi: 10.46743/2160-3715/2011.1051.

[42] O. Kononenko, O. Baysal, and M. W. Godfrey, “Code review quality: how developers see it,” in *Proceedings of the 38th International Conference on Software Engineering*, in ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 1028–1038. doi: 10.1145/2884781.2884840.

[43] Anthropic, “Claude 3.7 Sonnet and Claude Code.” [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet

[44] K. Kavukcuoglu, “Gemini 2.5: Our most intelligent AI model.” [Online]. Available: https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/

Bibliography

[45] OpenAI, “Introducing OpenAI o3 and o4-mini.” Accessed: May 08, 2025. [Online]. Available: https://openai.com/index/introducing-o3-and-o4-mini/

[46] OpenAI, “Introducing GPT-4.1 in the API.” Accessed: May 08, 2025. [Online]. Available: https://openai.com/index/gpt-4-1/

[47] M. Paixao, J. Krinke, D. Han, and M. Harman, “CROP: linking code reviews to source code changes,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, in MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 46–49. doi: 10.1145/3196398.3196466.

[48] M. M. Rahman, C. K. Roy, and R. G. Kula, “Predicting Usefulness of Code Review Comments Using Textual Features and Developer Experience,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 215–226. doi: 10.1109/MSR.2017.17.

[49] “Multishot Prompting.” Accessed: Apr. 24, 2025. [Online]. Available: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/multishot-prompting

[50] “Few-Shot Examples with Generative AI.” Accessed: Apr. 24, 2025. [Online]. Available: https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/few-shot-examples

[51] A. K. Turzo and A. Bosu, “What makes a code review useful to OpenDev developers? An empirical investigation,” *Empirical Software Engineering*, vol. 29, no. 1, p. 6, Nov. 2023, doi: 10.1007/s10664-023-10411-x.

10. Appendix

All relevant attachments too large to be in the body of the report can be found below.

10.1. Interview Questions

10.1.1. First Interview Phase

Tack för att ni tar er tid att delta i denna intervju. Syftet med samtalet är att utforska era erfarenheter och åsikter kring kodgranskningsprocesser. Vi kommer att diskutera praktiska aspekter som hur mycket tid ni lägger på kodgranskningar, vad ni anser är värdefulla granskningskommentarer, och er syn på olika kommunikationsstilar i granskningssammanhang. Vi är också intresserade av era tankar kring hur artificiell intelligens potentiellt kan påverka kodgranskningsprocessen. Era insikter kommer att hjälpa oss förstå hur kodgranskningar fungerar i praktiken. Det finns inga rätta eller fel svar – vi värdesätter era ärliga reflektioner baserade på era personliga erfarenheter. Har ni några frågor innan vi börjar?

10.1.2. Generellt om kodgranskningar

1. Hur många timmar spenderar du varje vecka på att granska kod?
2. Hur lång tid brukar en kodgranskning ta dig att genomföra?
3. Hur stor andel av kommentarer uppskattar du vara användbara?

10.1.3. Förväntningar kring kodgranskningar

1. Hur hade ni beskrivit egenskaperna hos en bra kodgranskningskommentar?
2. Hur hade ni beskrivit egenskaperna hos en dålig kodgranskningskommentar?
3. Vad tycker ni om detaljnivån i en kodgranskningskommentar? Hur påverkar detta kommentarens användbarhet?
4. Hur känner ni om kommentarer som påpekar kodstil och konventioner jämfört med de som endast fokuserar på funktionalitet?
5. Hur påverkar en kommentars frasering din mottaglighet till feedbacken som ges?

6. Kan ni dela med er om era tankar kring tonen i granskningskommentarerna och hur det påverkar din respons till dem?
7. Vilka aspekter i en kommentar anser ni vara viktigast för att kodkvaliteten skall hållas hög i framtiden?
8. Påverkar storleken av förändringarna/antalet filer inkluderade i en förändring kvaliteten av kodgranskningen?
9. Hur påverkar kodgranskarens expertis inom de granskade kodändringarna kvaliteten på kommentarerna?
10. Hur känner du kring balansen mellan att peka ut problem jämfört med att föreslå specifika lösningar i granskningskommentarer?
11. Vad tycker du om kommentarer på positiva aspekter i koden jämfört med att endast fokusera på förbättringar?

10.1.4. Förväntningar kring AI-kodgranskningar

1. Hur ser du på artificiell intelligens? Hur ser du på artificiell intelligens som en del av kodgranskningsprocessen?
2. På vilka sätt tror du att AI-assisterad kodgranskning skulle kunna påverka ditt arbetsflöde och utvecklingsprocess? Vilka möjligheter och utmaningar ser du med denna teknik?

10.2. Prompts

10.2.1. System Instructions

```
# Identity
You are an expert code reviewer tasked with analyzing code changes
and providing constructive feedback. Your goal is to identify
potential issues in the changed code and create clear, concise and
useful comments to help improve the code quality.

# Instructions
1. Focus on reviewing and providing feedback only on the code
changes shown in the diff file.
2. Prioritize identifying the following types of issues:
   a. Functional and logical defects
   b. Maintainability and evolvability issues
   c. Unaccounted for validation issues or edge cases
   d. Opportunities for more efficient or simpler solutions
3. Create concise, actionable comments that will be useful for the
code author.
4. Avoid comments that:
   - Ask for clarification of functionality
   - Praise the code author
   - Discuss potential future issues not directly related to the
```

System Instructions

```
change
   - Address formatting or code style issues

Provide your formatted comments based on your findings. Format each
comment as follows:

<comment>
    <lines>[Specify the line number(s) the comment refers to]</
lines>
    <feedback>[Your concise, useful, actionable feedback.]</
feedback>
</comment>

Example output structure (do not use this content, it's just to
illustrate the format):

<comment>
    <lines>15-18</lines>
    <feedback>The nested if statements could be simplified using a
logical AND operator, improving readability and maintainability.</
feedback>
</comment>

<comment>
    <lines>42</lines>
    <feedback>This function lacks input validation, which could
lead to unexpected behavior with edge case inputs.</feedback>
</comment>

<comment>
    <lines>239</lines>
    <feedback>We typically don't work directly with ```this``` in
our knockout models as it's very easy to change the binding context
of 'this 'in knockout.   Using self, is a convention to indicate
that the binding context of this is the viewmodel so you don't
have to think about what "this" is referring too. Most teams are
following this practice.</feedback>
</comment>

<comment>
    <lines>308-320</lines>
    <feedback>Pardon my OCD, but the endpoint
is `sourceCredentials/lookup/` but the handler is
`GetSourceCredentials` - can we change the former to be `/get/` or
the latter to be `Lookup...` ?</feedback>
</comment>

<comment>
    <lines>69</lines>
    <feedback>If all the tests pass without the if check, let's
take it out; I agree that it seems unnecessary.  If it is
```

System Instructions

```
necessary, I don't think it is good practice to is `is` for string
equality checking, string instances aren't guaranteed to share
identity in python. http://stackoverflow.com/questions/1504717/
why-does-comparing-strings-in-python-using-either-or-is-sometimes-
produce  Use == instead. Or even better, just check for a false-y
value: if not netloc_partner_host_name:</feedback>
</comment>

<comment>
    <lines>23-48</lines>
    <feedback>Little nervous to see updates like this coming in. I
assume this was a transitive dependency brought in when you updated
the `vautil` version?</feedback>
</comment>

<comment>
    <lines>64</lines>
    <feedback>ew to your field name, but i see you're following
convention... I realize the reason for the "Flag" is just so vapi
recognizes it as a boolean, but personally I'd rather see the whole
thing camelCased than this mashup of the two</feedback>
</comment>

Remember to prioritize the most important issues, focusing on
functional and logical defects. Your final output should consist
only of the formatted comments and should not include any of your
analysis work from the code review process.
```

10.2.2. User Prompt

```
Here are the files you need to review:

1. The original file before changes:
<before_file>
{before_file}
</before_file>

2. The modified file after changes:
<after_file>
{after_file}
</after_file>

3. The diff file showing the applied changes:
<diff_file>
{diff_file}
</diff_file>
```