# A Note on the Code Quality Score System: LLMs for Maintainable Large Codebases

**Sherman Wong*, Jalaj Bhandari*, Leo Zhou Fan Yang, Xylan Xu, Yi Zhuang, Cem Cayiroglu, Payal Bhuptani, Sheela Yadawad & Hung Duong**
Meta Platforms, Inc.

## Abstract

Maintaining code quality in large-scale software systems presents significant challenges, particularly in settings where a large number of engineers work concurrently on a codebase. This paper introduces the Code Quality Score (CQS) system to automatically detect issues related to code quality and maintainability and provide actionable insights. At its core, the CQS system utilizes two Llama 3 models, each fine-tuned using supervised fine-tuning (SFT) and/or offline reinforcement learning (RL). One model detects common code quality issues related to coding best practices, while the other provides high-quality critiques for LLM-generated code reviews. To maintain good user experience, we also add a set of hand-crafted rules to the system to further filter out incorrect responses/hallucinations. Offline evaluations, based on internal human labeling, show that the CQS system is able to achieve an impressive precision rate for identifying valid code quality issues. This system has already been rolled out to developers at Meta and has consistently achieved 60% week over week user reported helpfulness rate, demonstrating its effectiveness in a real-world environment. In this paper, we present details of the CQS system along with some learnings on curating developer feedback to create training data for LLM fine-tuning.

## 1 Introduction

Evaluating code quality is a crucial part of the software development cycle that requires a significant amount of developer time and effort [19]. Recent studies point to high code quality as one of the key drivers of increased productivity of software engineering organizations and can have a huge impact on the number of defects, issue resolution time as well as time to development [4, 22]. Challenges to maintain a high quality codebase tend to scale with size and complexity in large industrial settings like Meta, with many engineers working concurrently on a continuously evolving codebase. This has led to increased concerns around code maintainability, with issues related to high cyclomatic complexity, legacy code, inadequate documentation etc. requiring resources to manage the technical debt. Despite increasing awareness about its importance, the process of evaluating code quality remains mostly manual and is typically done via peer code review. While traditional static analysis tools like linters are often used to automatically check some aspects of code quality such as formatting, other important (and often nuanced) aspects of code quality such as those related to code modularity, readability, error handling, testability etc. are ambiguous and hard to evaluate by such rule based systems. This, along with high developer satisfaction rates [27], has led to human code review as the dominating paradigm.

Advances in deep learning and natural language processing (NLP) have inspired initial attempts to automate the code review process with large scale pre-trained models, focusing on specific tasks such as code quality estimation, review generation and code refinement [12, 21, 24, 11]. Most of

*Equal contribution. Please email to `shermanwong@meta.com, jalajbhandari@meta.com`

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: The 4th Deep Learning for Code (DL4C) Workshop.

these ideas require domain and task specific pre-training necessitating substantial computational resources. Large language models (LLMs) on the other hand have shown impressive cross domain generalization across a variety of tasks without the need for specific pre-training — in fact, most of the recent advances have leveraged post-training methods to further improve LLM performance on specific tasks.

We take inspiration from some recent work which demonstrates early success of using LLMs for automated code review [14, 25, 6] and develop the Code Quality Score (CQS) system — an LLM-powered system for code quality evaluation and code review at Meta scale. We break down the entire code review process into three distinct tasks, namely, "issue collection", "issue validation" and "action generation" and design CQS to orchestrate multiple components with each component responsible for one of these tasks. For CQS components, we fine-tune open source LLMs (Llama models) with post-training methods like Supervised fine-tuning (SFT) and Direct Preference Optimization (DPO), an offline RL approach. CQS has already been deployed to a large number of developers at Meta with positive feedback and we are leveraging constant developer feedback as a data flywheel to iteratively fine-tune and improve our models.

Our contributions in this paper can be summarized as:

- We develop the Code Quality Score (CQS) system to automatically evaluate code quality and generate code reviews in a scalable manner.
- CQS is powered by open source LLMs (Llama models). We use a multi-stage post-training pipeline to better align with codebase context and developer preferences, and improve performance as compared to the base Llama models.
- To enhance end-to-end developer experience, CQS orchestrates multiple components for code review generation and validation, along with a layer of sanity check before code reviews are send to developers.
- Overall, the CQS system shows an impressive precision rate (based on internal human labeling) for identifying valid code quality issues with a set of code changes. It has been rolled out to more than 5000 engineers at Meta and has achieved an average weekly user reported helpfulness rate of approximately $60\%$.

While research is still in early stages, we envision that systems like CQS can transform the future of software engineering practice in terms of sustainable code management, particularly in large industrial settings. Concretely, we hope that CQS can drastically reduce the manual review burden on senior engineers, while accelerating the on-boarding process for junior developers.

We start by introducing the CQS system below in Section 2 and give details about training and evaluation in the sections that follow. For brevity, we discuss related work in Appendix A.

## 2 A multi-agent design for the Code Quality Score (CQS) system

In this section, we describe the CQS system, including the overall multi-agent design, a description of each agent within the system and the input data format. The CQS system orchestrates multiple LLM based components to generate code reviews. Such a multi-agent architecture allows distinct, specialized roles to be assigned to different agents. We show the system architecture below in Figure 2, which consists of three agents, namely, the 'issue collector' agent, the 'issue validator' agent and the 'post-processing' agent. We describe the role of each agent below.

**The 'issue collector' agent:** The issue collector agent uses an LLM to identify and summarize a set of possible code quality issues (such as those related to modularity, duplication etc.) with a given code change. It scans the proposed code changes in diff format. We prompt the agent to output a summary of the all possible issues identified by the model — such a summary includes an issue tag, function name[2], rationale for the issue, file path and line number. We include a list of potential issue tags in the prompt for the model to choose from but also take into account any novel tags that the model comes up with. See Appendix B.1 for details on how we prompt the issue collector agent. To

[2]For the function name, we prompt the model to output `NULL` in case the identified issue is with code change that is not a part of any function.

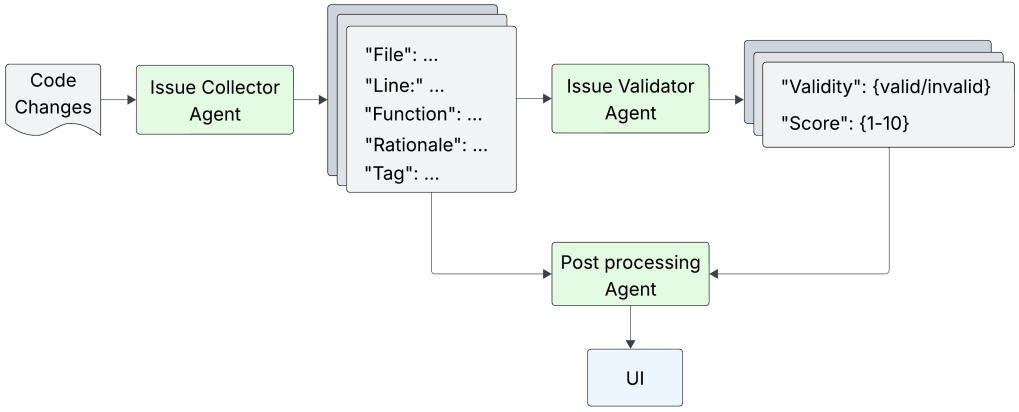


Figure 1: Code Quality Score System Design

power the issue collector agent, we fine-tune a Llama 3.1-70b Instruct model using an initial training round of SFT followed by DPO training. See Section 3.1 for details.

**The 'issue validator' agent:** The set of issue identified by the issue collector agent for a diff are sent to a validator agent . In many cases, we found the set of issues outputted by the collector agent to be not relevant or entirely incorrect. To filter out such instances, the validator agent uses an LLM to critically access correctness and relevance of each issue and output a score on a scale of 0-10. For some issues (categorized based on issue tags), we include specific guidelines in the system prompt to instruct the LLM to focus on particular elements. See Appendix B.5 for details on the prompt we use. To power the validator agent, we fine-tune another Llama 3.1-70b instruct model using SFT to act as LLM-judge. See Section 3.2 for details.

**Post-processing agent:** Validated code quality issues are ingested by the post-processing agent for further filtering. Here we use thresholds for each issue category (as indicated by the tag) to filter out issue which were scored low by the validator agent. We also use a set of hand-crafted rules per issue category for further filtering, before the code review is shown in the UI. Both the thresholds as well as the hand-crafted rules were optimized to improve precision of the overall system.

**A note on Data Format:** We follow past work [12, 25] and use a "diff" format to represent code changes as inputs to the CQS system. An example of this is shown in Figure 5 in Appendix C. The diff format is commonly used to show code changes clearly and is generated by comparing the original version and revised version of code. As compared to using the entire source code file, the diff format is an efficient representation as code changes are marked with "+" and "-" tags at the beginning of each line along with some surrounding line of unchanged code for context and file names. See Figure 5 for a visualization.

## 3 Model Training

In this section, we describe details of the training paradigm we use to fine-tune the issue collector and validator agents.

### 3.1 Issue Collector Agent

The issue collector agent is trained in two phases — the initial model is fine-tuned using SFT — which is followed by a round of offline-RL based fine-tuning using DPO. To the best of our knowledge, this is the current state-of-the-art recipe for fine-tuning foundation models — our motivation for this choice was to teach the model about developer preferences during the SFT phase and perform DPO

training for better generalization[3] to help the model identify issues correctly. We also do ablations with a model only trained using SFT and another model trained with DPO without doing SFT first. Doing a round of SFT training before DPO seems to help improve the recall rate. See Section 5.1 for details.

**Phase 1: Supervised Fine-tuning (SFT)** For SFT training, we collected high quality human reviewed diffs from our internal codebase covering a wide ranging set of tasks, projects and services, including 7 different programming languages; with the majority of code changes evenly split between Python, C++ and PHP. For dataset creation, we applied many filters. For example, we only consider diffs with more than 1 review comment, and discard diffs authored by a bot or those that were solely created for testing purposes. Importantly, we only selected diffs that were aimed specifically to solve code quality issues[4]. For each diff, we classify every human reviewer comment into categories or "tags" and create a "code review" which contains issue tag, comment by the human reviewer, the function name (`NULL` if the comment does not refer to any function), file path and line number for each issue.

To further filter out instances where human review comments were not useful or did not have clear guidance, we prompt a Llama 3.1-70b Instruct model to analyze the quality of the human reviewer comments and to grade them as "good" or "bad". We also instruct the model to rewrite the human comments into "rationales" for the issue. See Appendix B.2 for the prompt we use for this rewriting step. From each code change, we dropped issues where human comments were classified as "bad" by the Llama model and obtained a set of 5000 (diff, code reviews) pairs, each with details about high quality issues derived from human comments. We use this dataset to do SFT. The entire data curation and training pipeline for SFT training is illustrated below in Figure 2.

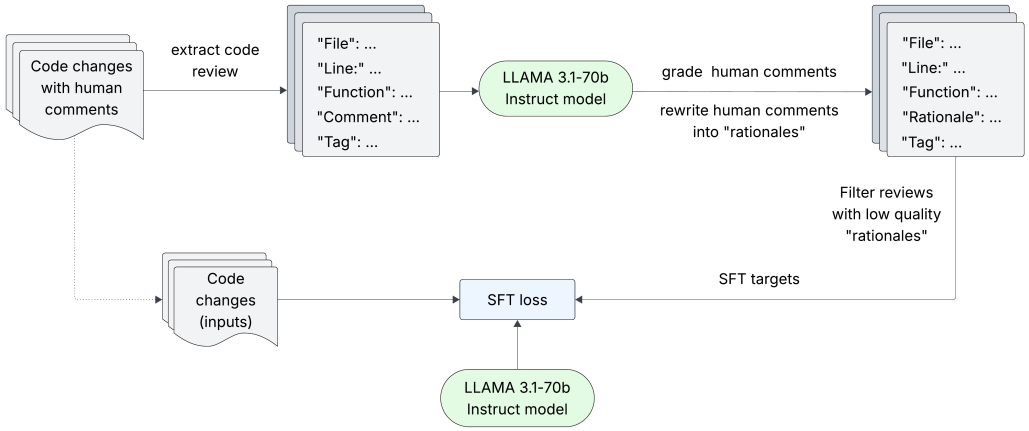


Figure 2: Phase 1 training of the Issue Collector Agent using Supervised Fine-Tuning

**Phase 2: Offline RL using Direct Preference Optimization (DPO)** We further fine-tune the SFT checkpoint using DPO [17], an offline RL algorithm popularly used for fine-tuning LLMs. DPO training requires data in the form of preference pairs, consisting of a 'winner' and a 'loser' response for each input; and attempts to increase the log probabilities of tokens in the 'winner' response while decreasing the log probabilities of tokens in the 'loser' response. Concretely, for a dataset $\mathcal{D}$ of preference pairs, the training objective for DPO is given by:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \left[ \log \sigma \left( \beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)} \right) \right],$$

[3]The preference data for DPO training was sampled from the model itself. By doing a round of SFT before, our goal was to be able to capture developer preferences in the DPO training data as well.

[4]The kind of issues around 'Readability and Maintainability', 'Modularity and Reusability' etc. that we want to identify using the CQS system. See the issue collector agent prompt in Appendix B.1 for details about the major issue categories that we aim for the issue collector agent to identify.

where $\pi_\theta$ is the policy being trained, $\pi_{\text{ref}}$ is the reference model (in our case, the SFT policy), $x$ represents the input (in our case, a diff), $y_w$ and $y_l$ are the 'winner' and 'loser' responses respectively (in our case, a pair of correct and incorrect issues identified), and $\beta$ is the temperature parameter controlling the strength of the KL penalty.

To create preference pairs, we use an LLM-judge to score each issue generated by the SFT policy. An alternate way to score generations is to use human annotations instead — however, obtaining such human feedback was not scalable for us and so we relied on the LLM-judge model. To sample diverse generations, we rely on using high temperature sampling only. Recent work in literature [15, 7] has explored alternate methods to improve diversity of the sampled generations which could be interesting to explore in our setting as well — we leave this exploration for future work.

We instruct the LLM-judge model to score each issue in a generated code review (i.e. each issue in $y_i^{(j)}$ in Equation 1 below) on a scale of 0 to 10 — a score of 0 if the issue identified was found to be entirely incorrect or empty, and a score between 1 to 10 to grade the quality/correctness of the issue.

$$\{y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(N)}\} \sim \pi_{\text{SFT}}(\cdot|x_i)$$
$$s_i^{(j)} = \text{LLM-Judge}(x_i, y_i^{(j)}) \in [0, 10] \tag{1}$$

See Appendix B.4 for the detailed prompt we use to score generations using LLM-judge. We give details about creating preference pairs for DPO training in Appendix D.3.1. The data creation and training pipeline we use for DPO is shown below in Figure 3.

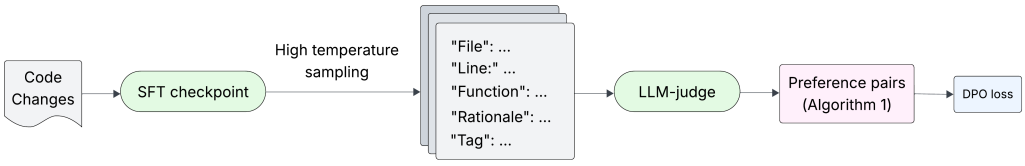


Figure 3: DPO training of the Issue Collector Agent using LLM-judge to create preference pairs

We describe training of the LLM-judge in Section 3.2 below as we also use the same LLM-judge model as a part of the validator agent.

### 3.2 Issue Validator Agent

The issue collector agent is based on the same LLM-judge model that we used for creating preference pairs for DPO training. We give details about training the LLM-judge model below.

**LLM-judge model:** Our motivation for training an LLM-judge model was twofold. First, we wanted to use it both for further fine-tuning the issue collector agent, And second, we found developers more willing to "critique" individual code issues rather than compare two issues and annotate. So we aim to distill this developer feedback into the LLM-judge model. To do this, we follow a recent proposal by Ke et al. [8] and fine-tune a Llama 3.1-70b Instruct model to generate good "critiques" of a code review. Recall that to create preference pairs, we use the LLM-judge model to score code review responses sampled from the SFT model on a scale of 0-10. This requires the LLM-judge model to provide fine-grained distinguish-ability between different sampled responses.

A key finding of Ke et al. [8] is that state-of-the-art LLMs like ChatGPT, GPT-4 etc. lack the ability to generate informative "critiques" to evaluate and grade LLM-generated responses on different real-world NLP tasks such as AlignBench [13] and LLMEval [29]. However, fine-tuning on a small data set of good quality "critiques" improves a models ability to generate granular scores. The resulting model tends to not only perform well on evaluation tasks but can also be used to provide scalable feedback in an fine-tuning loop.

To acquire "critiques" for LLM-judge training, we rolled out the SFT model to a group of developers for beta testing. For each code change, the SFT model generated a code review with different identified issues, and developers were asked to provide both textual feedback and/or click a thumbs

up/down button to indicate if they agreed or disagreed with the issue identified. This way we were able to collect developer feedback and/or "grade" for every issue. To do further data filtering, we use another Llama 3.1-70 Instruct model to analyze, rewrite developer feedback into "critiques and score these (see Appendix B.3 for the exact prompt we use). We then drop low quality critiques to create our training dataset. For SFT training of the judge model, we use code changes and the corresponding model generated code reviews as inputs, and the code review critiques obtained from developer feedback as SFT targets. The entire data curation and training pipeline is illustrated below in Figure 4.

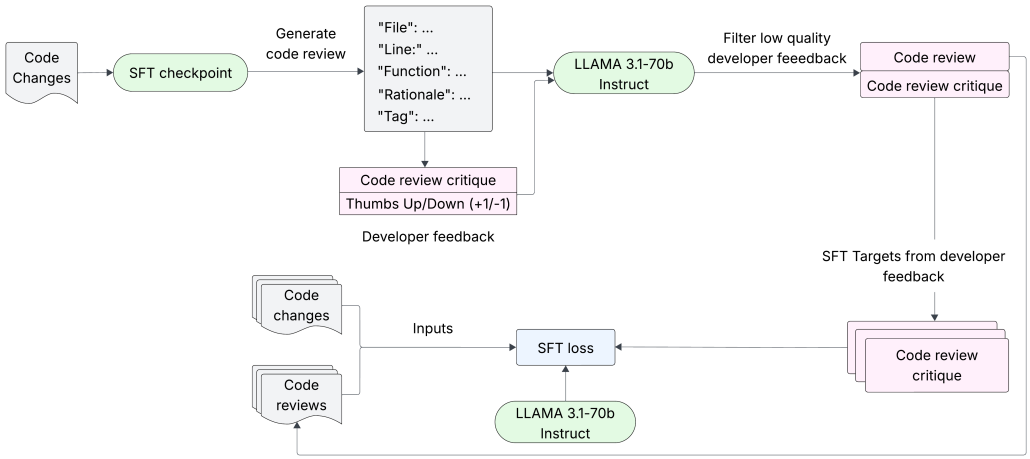


Figure 4: LLM-judge training. We curate high quality developer feedback (code review critique, thumbs up/down) for model generated code reviews sampled from the SFT checkpoint of the issue collector agent. This developer feedback is then used to SFT a Llama model to be good at generating human like critiques and grades for model generated code reviews.

## 4 Evaluation Design

To create an evaluation dataset for CQS, we use multiple frontier LLMs such as GPT-4o, Claude 3.7 Sonnet and Gemini-2.5-pro to generate code reviews for a small set of code changes, different from the ones used for training. We then ask senior engineers to analyze and label each issue (as correct or not), and create a set of (diff, code review) pairs which we use as the ground truth data for evaluating CQS models. During evaluation runs, we use semi-deterministic verification to compute evaluation metrics. We give details below.

**LLM based code reviews:** To generate code reviews from state-of-the-art LLMs, we collect a set of 200 diffs that cover a diverse set of projects, across 7 programming languages with a majority of these being either C++, PHP or Python. To maximize issue coverage we use GPT-4o, Claude 3.7 Sonnet and Gemini-2.5-pro to generate candidate code reviews.

**Human annotation and evaluation dataset:** After generating candidate code reviews, we engage with multiple engineers to review the issues identified. Each diff is assigned to at least two engineers. For each code review per diff, we only keep issue where both engineers agree on issue tag, line number and issue rationale — issues on which engineers disagree are discarded. This way we created a set of approximately 80 (diff, code review) pairs with each diff containing approximately 2-4 consensus issues identified by engineers.

**Evaluation protocol and metrics:** During evaluation runs, we apply both rule based and semantic matching to compare code reviews generated by the CQS system with those in the evaluation dataset, issue by issue. We do this by constructing an automatic verifier that first performs an exact string match check for issue tags. If the tag match is successful, we compare the rationale for each issue —

this is done using a lightweight Llama 3.1-8b model given the comparison task here is simple[5]. In addition, we also eyeballed some issue comparisons as a sanity check for our metrics.

Note: We note here that a major challenge in evaluating the quality of code reviews generated from a system like CQS is that ground truth can be incomplete and biased, since human annotations can be insufficient and often noisy ([25]; [12]). We also struggle with similar issues. Since we use GPT-4o, Claude 3.7 Sonnet and Gemini-2.5-pro to generate candidate issues in the first place, the evaluation data is heavily biased towards these models. While we asked multiple senior engineers to validate each issue, this did not get rid of the bias since we did not ask them to identify possible issues which might have been missed by these models. In addition, during validation, engineers were also able to see the model name for each candidate issue — we think this might have further amplified bias towards the Claude 3.7 Sonnet model.

# 5 Results

In this section, we present evaluation results for the CQS system. We primarily evaluate on two metrics – *precision*, i.e the fraction of issues identified by CQS that were also identified in the evaluation dataset; and *recall*, i.e. the fraction of issues identified in the evaluation dataset that CQS is also able to successfully identify. Our system and prompt design is aimed at improving precision while fine-tuning agents is aimed at optimizing for both precision and recall. We do note that these results should be read in context of the bias in the evaluation dataset as mentioned above in Section 4. Despite this, the CQS system shows an impressive precision rate as shown in Table 2.

## 5.1 Fine-tuning the Issue Collector Agent

In Table 1 below, we compare the precision and recall rates for different fine-tuning recipes we tried while training the issue collector agent. We report the numbers averaged over 10 evaluation runs. As can be seen, the SFT trained model seems to be at par (even slight regression in terms of recall rates) as compared to the base Llama 3.1-70b Instruct model. However, DPO trained models show improvements. While directly fine-tuning the base model with DPO shows a 3% lift in precision, it does not improve recall. On the other hand, a combination of SFT and DPO improves both precision and recall rates by 2% and 1% respectively indicating improved generalization. A takeaway from our results is that using an LLM-judge model (even with very limited data) helps in fine-tuning — it will be interesting to see the performance gains that can be achieved by improving the LLM-judge model using more developer feedback. We leave this investigation for future work.

| Model | # issues found | precision (%) | recall (%) |
|---|---|---|---|
| Baseline Llama-3.1-70b Instruct | 138 | 11.28 | 8.14 |
| Llama 3.1-70b Instruct + SFT | 130 | 11.14 | 7.62 |
| Llama-3.1-70B Instruct + DPO | 103 | 14.56 | 7.82 |
| Llama 3.1-70B Instruct + SFT + DPO | 131 | 13.48 | 9.25 |
| Claude 3.7 Sonnet | 193 | 30.43 | 30.58 |
| GPT-4o | 233 | 24.33 | 29.53 |
| Gemini-2.5-pro | 134 | 32.77 | 22.53 |

Table 1: Evaluation metrics: precision & recall rates across different stage of training. Here "# issue found" refers to the total number of issues identified by the model over the entire evaluation dataset.

A noteworthy observation from our results is that fine-tuning using DPO directly from the base model yields a lower recall rate than following the SFT + DPO recipe. A possible explanation could be that doing SFT on human code reviews as a first step likely helps the model mimic developer preferences more closely.

The huge gap between Llama 3.1 70b models and other state-of-the-art models like Claude 3.7 Sonnet, GPT-4o and Gemini 2.5-pro is also noteworthy. Although the Gemini 2.5-pro model has a better

[5]We didn't use BLEU score as done in [12], given the cost of using a lightweight Llama 3.1-8b model during evaluation runs is small.

precision rate, Claude 3.7 Sonnet seems to be the best performing model overall. We would like to remind the readers again of the caveat that since these models were used as part of the evaluation dataset creation, it is very likely that the precision and recall rates are biased. Nevertheless, the numbers do show a significant advantage of the Claude, GPT and Gemini models over Llama models.

### 5.2 Multi-stage Prompt Optimization

While designing the CQS system, we prioritize precision over recall to ensure developer trust in the service[6]. In Table 2 below, we show how the precision rates change for different system configurations. As compared to only using the issue collector agent, using the issue validator and post -processing agent is really helpful in improving the precision rate of the CQS system.

| Method | Avg precision (%) | Avg recall (%) |
|---|---|---|
| Issue collector agent | 13.48 | 9.25 |
| Issue collector agent + validator agent + post processing agent | 78.20 | 1.20 |
| Claude 3.7 Sonnet (for both issue collection and issue validation agents) + post-processing agent | 51.63 | 11.45 |

Table 2: Precision/recall rates through the different stages of the CQS system.

We also benchmark against a setting where we use two Claude 3.7 Sonnet models, one for issue collection agent and the other as LLM-judge for the issue validator agent, along with the post-processing agent. As can be seen in Table 2 below, while this does lift precision (from 30.43% to 51.63%), it lags in comparison to the CQS system. The LLM-judge model used for scoring as part of issue validator agent seems to be responsible for this gap – off-the- shelf Claude 3.7 Sonnet model is unable to distinguish between valid and invalid issues as compared to the Llama 3.1-70b LLM-judge model we trained. This is expected as prior research shows fine-tuning large language models (even with a small dataset of high quality) can be effective in significantly improving its accuracy as a judge model [8, 26]. We report results of LLM-judge evaluation using a small dataset in Appendix E.

### 5.3 Online Results

CQS has been rolled out to a large number of developers with consistently good user feedback. For instance, it has an average 60% user reported helpfulness rate — which we measured as the percentage of positive feedback that developers shared by clicking a thumbs-up/down button in the code review UI. An important point to note here is that while we see a correlation between improvements in precision and user helpfulness rate, it is not directly proportional. A potential reason could be the small size of our evaluation dataset — model selection based on improvements in our evaluation metrics may not transfer when the system is rolled out to a diverse set of code changes. We currently observe a $\pm 10\%$ confidence interval on the weekly user reported helpfulness rate and are working on increasing the size, quality and coverage of our evaluation dataset.

### 5.4 Discussion: Interesting observations and Limitations

In this section, we share a couple of noteworthy points from our experience designing the CQS system along with some limitations. First, we found LLM-judge model performance to be quite important for DPO training. This is expected as reward/feedback quality is known to be the key component in RL based fine-tuning of LLMs and more generally in RL as well. Overall, we found the use of a fine-tuned LLM-judge model to be quite useful in the absence of human annotations — using LLM-judge, we were able to transfer limited developer feedback into preference signals for fine-tuning, at least partially. In this context, we think that training high quality reward models with limited feedback data, such as our setting, is an important research question that deserves more attention. While current work has leveraged textual reasoning using critiques [1] or supervised fine-tuning techniques [8, 26] to enhance performance of LLM-judge models, other ideas such as

[6]Developers want the system identified issues to be valid and are fine with the system not being able to identify all issues.

using reasoning models with multiple evaluation rubrics might be useful in training robust reward models.

Second, the idea of using the issue validator and post-processing agents was motivated by developer expectations of high precision rates from a system like CQS. As can be seen in Table 1, even frontier models have low stand-alone precision rates ($< 33\%$) for identifying code quality issues. It will be interesting to see how these precision rates improve with the next generation of frontier models. Despite improvements we made with the CQS system, extracting high-quality, actionable code review comments for diff remains a significant challenge. We show examples of a few trivial cases where the CQS system fails in Appendix F.

## 6 Conclusion and Future Work

The Code Quality Score (CQS) system we present in this work has shown good performance with early adoption by a large number of developers and over 60% average weekly user reported helpfulness rate (as indicated by internal feedback). Although Llama models lag state-of-the-art closed source models like Claude model series on issue collection tasks, we were able to fine-tune them using developer feedback to improve their performance on issue collection and validation tasks, and leverage them effectively in a multi-agent system to achieve a better precision rate overall. As Llama models improve and close the gap with other state-of-the art models, we hope to further improve the performance of our system.

We also plan to further enhance the CQS system by leveraging online developer feedback to build a data flywheel that can be used to continuously improve the agents on issue collection and validations tasks. Another promising direction that we plan to pursue is to go beyond code quality issues and expand the CQS system to identify complex issues related to security vulnerabilities and performance. Although AI-based code review systems are still emerging, early results are encouraging in terms of adoption and satisfaction rates. As these systems mature, they are not only expected to reduce human review burden but also bring long-term improvements to the codebase health.

## 7 Broader Impact Statement

We believe systems like CQS can be useful to partially automate code review tasks which are hugely important to software development but time and resource consuming. There are many potential societal consequences of our work which are linked to the adoption of AI tools for automation more broadly, none which we feel must be specifically highlighted here.

## Acknowledgment

We thank the team members and leadership in Meta's monetization organization for their support in developing and deploying the Code Quality Score system, including Daniel Jiang, Wenlin Chen, Hung Duong, Shah Rahman, Ritwik Tewari, Santanu Kolay, Neeraj Bhatia, Matt Steiner and Peng Fan.

## References


[1] Zachary Ankner, Mansheej Paul, Brandon Cui, Jonathan D Chang, and Prithviraj Ammanabrolu. Critique-out-loud reward models. *arXiv preprint arXiv:2408.11791*, 2024.

[2] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Lan Cheng, Emerson Murphy-Hill, Mark Canning, Ciera Jaspan, Collin Green, Andrea Knight, Nan Zhang, and Elizabeth Kammer. What improves developer productivity at google? code quality. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1302–1313, 2022.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240): 1–113, 2023.

[6] CodeRabbit. Awesome coderabbit. `https://github.com/coderabbitai/awesome-coderabbit`, 2024. Accessed: 2025-06-26.

[7] Chan-Jan Hsu, Davide Buffelli, Jamie McGowan, Feng-Ting Liao, Yi-Chang Chen, Sattar Vakili, and Da-shan Shiu. Group think: Multiple concurrent reasoning agents collaborating at token level granularity. *arXiv preprint arXiv:2505.11107*, 2025.

[8] Pei Ke, Bosi Wen, Zhuoer Feng, Xiao Liu, Xuanyu Lei, Jiale Cheng, Shengyuan Wang, Aohan Zeng, Yuxiao Dong, Hongning Wang, et al. Critiquellm: Towards an informative critique generation model for evaluation of large language model generation. *arXiv preprint arXiv:2311.18702*, 2023.

[9] Dongyoung Kim, Kimin Lee, Jinwoo Shin, and Jaehyung Kim. Aligning large language models with self-generated preference data. *arXiv preprint arXiv:2406.04412*, 2024.

[10] Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, et al. Rlaif vs. rlhf: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.

[11] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. Auger: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1009–1021, 2022.

[12] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1035–1047, 2022.

[13] Xiao Liu, Xuanyu Lei, Shengyuan Wang, Yue Huang, Zhuoer Feng, Bosi Wen, Jiale Cheng, Pei Ke, Yifan Xu, Weng Lam Tam, et al. Alignbench: Benchmarking chinese alignment of large language models. *arXiv preprint arXiv:2311.18743*, 2023.

[14] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 647–658. IEEE, 2023.

[15] Minh Nhat Nguyen, Andrew Baker, Clement Neo, Allen Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. Turning up the heat: Min-p sampling for creative and coherent llm outputs. *arXiv preprint arXiv:2407.01082*, 2024.

[16] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[17] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

[18] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[19] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pp. 181–190, 2018.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[21] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th international conference on software engineering*, pp. 237–248, 2022.

[22] Adam Tornhill and Markus Borg. Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*, pp. 11–20, 2022.

[23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[24] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*, pp. 2291–2302, 2022.

[25] Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, et al. Ai-assisted assessment of coding practices in modern code review. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 85–93, 2024.

[26] Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*, 2024.

[27] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software engineering at google: Lessons learned from programming over time*. " O'Reilly Media, Inc.", 2020.

[28] Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*, 3, 2024.

[29] Yue Zhang, Ming Zhang, Haipeng Yuan, Shichun Liu, Yongyao Shi, Tao Gui, Qi Zhang, and Xuanjing Huang. Llmeval: A preliminary study on how to evaluate large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19615–19622, 2024.

[30] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

# A Appendix: Related Work

We discuss some closely related work below.

**AI Assisted Automated Code Review** Initial attempts at automating aspects of the code reviewing process mostly leveraged pre-training recipes with transformer based model architectures. For example, [24] used source code and code reviews from GitHub to train an Text-to-Text Transformer (T5) model [18] while [12] train an encoder-decoder model using code diff hunks (instead of using the source code directly) and corresponding reviews collected from high quality GitHub projects. However, more recently, the focus has shifted away from training models from scratch and towards leveraging the world knowledge from foundation models such as Llama [23] or Claude models, to create systems for code review. Unsurprisingly, using LLMs for code review (and broadly for coding tasks) has outperformed domain specific models trained on code generation/code review tasks. For example, [14] design Llama-Reviewer, a framework aimed to fully automate the code review process by integrating code review generation and code refinement steps, using Llama models. Similarly, [25] develop AutoCommenter, an LLM-based system deployed at Google for detecting violations of coding best practices frequently referenced by human reviewers.

Compared to the Llama-Reviewer framework of [14] which is aimed at automating the code review process end-to-end, we only restrict ourselves to generating code reviews — the Llama-Reviewer framework incorporates performing review necessity prediction, code review generation and code refinement steps in order iterating over these. Another difference between our works is that while [14] use training data collected from GitHub and restrict themselves to parameter efficient fine-tuning methods such as low rank adaptation (LoRA), we curate training data from human reviews and human labeling and design a multi-stage post-training pipeline comprised of SFT and RL fine-tuning approaches (akin to state-of-the-art post-training pipelines).

Compared to AutoCommenter [25], we go beyond only detecting URL based best practice violations and instead focus on a variety of issues referenced by human developers in their code reviews. In addition, CQS orchestrates multiple components[7] to improve precision and developer experience, and is designed to leverage constant developer feedback to iteratively improve based on new issues that may arise in new code context (for e.g. when new frameworks are introduced in the codebase). This way, we are able to better track developer preferences as opposed to mapping to a static set of best practices for reference.

**Fine-tuning LLMs** Large language models trained with self-supervised learning have shown impressive zero-shot/few-shot performance in a variety of tasks [3, 5]. With the explosion of LLMs and their applications in recent years, alignment/fine-tuning has emerged a major research area with the aim to improve performance of LLMs on specific tasks. There is a large body of work exploring novel methods for fine-tuning LLMs. For brevity, we only mention work that we directly leveraged. The Reinforcement Fine-tuning (RFT) pipeline typically consists of an SFT phase, where a pre-trained LLM is fine-tuned with supervised learning on high quality data from the downstream task of interest, followed by RL optimization [30, 16]. Both online RL (using reward models with algorithms such as Proximal Policy Optimization [20]) and offline RL approaches (by sampling multiple generations from the LLM to create preference pairs) exist for the RL fine-tuning phase. As human labeled data is hard to scale, a lot of recent research has focused on RL from AI feedback (RLAIF) [2, 10], where LLM-judge models are used as sources of feedback (both as reward models or to create preference data). In our work, we only explore offline RL approaches based on DPO [17] and its variants due to their simplicity[8].

**LLM-Judge for Preference Optimization** As mentioned above, we use an LLM-judge model to create preference pairs for DPO training. Using LLM-judge models in RLAIF has emerged as an important alternative to using human annotations. Recent work shown impressive results using LLM-judge models as reward models/scoring functions for creating preference pairs [26, 28, 9]. Another closely related line of work has focused on judge models that first verbalize a critique and then map it to a preference score, yielding more transparent and often more data-efficient alignment signals. For

[7] Since [25] do not reveal much details about system design and model training, it is hard to compare the two works in this aspect.

[8] We had limited data and wanted to avoid training reward models with it.

example, the Critique-out-Loud approach of [1] explicitly trains a judge model to verbalize free-form critiques, showing that such rationales can significantly improve pairwise preference classification accuracy. In our work, we also leverage an LLM-judge model which was fine-tuned with SFT, to create preference pairs for DPO training.

# B Appendix: Prompts

In this section, we list out all the prompts that were used for different LLMs.

## B.1 System instruction for the Issue Collector Agent

```
=== Raw Source Control Code Changes BEGIN ===
code changes go here
=== Raw Source Control Code Changes END ===

=== Context for the Code Change BEGIN ===
The following additional information from the author may
provide context about the code changes and their purpose:
Title: title
Summary: summary of code changes
=== Context for the Code Change END ===

Your task is to provide constructive and concise feedback for
the code changes based on the following criteria (each goes
with a tag name in the beginning):
```

- DedupeLogic: Deduplicate logic into shared functions except for logging.
- DictionaryKeyExistenceCheck: Check for dictionary key existence before accessing it.
- UseConstant: Use constants instead of literals, unless it's a constant definition.
- RenamingVariable: Variable names should be pronounceable, easily readable, and reveal intent.
- DomainSpecificName: Use solution domain names, computer science (CS) terms, algorithm names, pattern names, math terms. When there is no name from the solution domain then prefer problem domain names.
- BreakdownLongFunction: Break down long functions into smaller, more focused functions. Only include this issue if the length of the current function is longer than 50 lines.
- ExtractMethod: When we have a block of code that can be extracted into a separate method, we should do so.
- ResourceLeak: A resource handle (e.g., file, socket, connection) is not properly wrapped in a context manager or try-with-resources construct or use a with statement in Python.
- Documentation: Docstring should match the code, and be added for each major unit. Pay extra attention to different types of docstrings.
- DivisionByZero: When performing division operations, ensure that the divisor is checked to be non-zero before proceeding with the calculation.

- Typo: A typo is detected in the code.
- RenamingFunction: Function names should be pronounceable, easily readable, and reveal intent.

In case none of the provided tags are appropriate, you may come up with your own tag.

Code lines are prefixed with the line number, followed by symbols ('+', '-', ' '). The '+' symbol indicates new code added, the '-' symbol indicates code removed, and the ' ' symbol indicates unchanged code. The review should address new code added (lines starting with '+').
Pay attention to what part of the code has changed, you

should only grade the code that has changed, and ignore the rest.

Example output:

```
{
  "function": "[exact function name from code]",
  "rationale": "[clear explanation of the issue and suggested
  improvement]",
  "file": "[file path where the function is located]",
  "line": [numeric line number],
  "tag": "[one of the tags as described above]"
}
```

Note you should not be overly critic and generate many bullet points. Please pin-point the 'problematic_function' with issues, and don't forget to include line numbers. The output field 'problematic_function' is mandatory in the response if the problem is inside a function. The output field 'relevant_file' should include full path of the file (including directory and filename). In your response, do not describe what the code changes is about because the code author already knows about it. Pay attention to the code change.

The output field 'rationale' would be shown to the author of code changes. Please use a polite and suggestive tone for the 'rationale' field, provide a reasoning and give suggestions using a question instead of a command.

### B.2 System Instruction to Rewrite Human Comments Into “Issue Rationales"

You are given a code change and a corresponding code review.

Here are the code changes:
=== Raw Source Control Code Change BEGIN ===
code changes go here
=== Raw Source Control Code Change END ===

Here is the code review in yaml format:
=== Code Review (YAML FORMAT) BEGIN ===

```yaml
{
  "Function": ...,
  "Rationale": ...,
  "File": ...,
  "Line": ...,
  "Tag": ...
}
```

=== Code Review (YAML FORMAT) END ===

Please reason and understand the code review and carefully follow the instructions below.

First, please do an initial assessment of the code review to try to roughly understand what the reviewer’s objective. Then, try to explain if the rationale in reviewer’s comments is logical. You may or may not have a clear understanding of what the rationale might be, so you can propose multiple hypothesis. For example:

“Hypothesis: In file xyz.py, at line 100, the reviewer is referring to an issue in the definition of ‘ProblematicFunction’, and if the reviewer’s comment is not addressed, it could cause X/Y/Z”

After you propose hypothesis, reflect on them and answer the following questions:

1. Is the provided rationale addressing the issue in the give code change or it’s referring to some context outside of the given code change?
2. Is the rationale linking to external references/links?
3. Is the rationale explicitly pointing out issues? Or is it just an inquiry because the review is not sure either?
4. Is the reviewer’s suggestion actionable? Or is it just some general suggestions that does not have clear action items?
5. Is the reviewer nitpicking? For example, saying something like "nit...."

After you have answered these questions, you should have a clear judgment of the "quality" of the review. Typically, bad quality reviews have the following characteristics:

- Out of context: reviews which link to external references or refer to something outside of the given code change.

- Nitpicking.
- Not actionable.
- Reviews which do not detect SPECIFIC issues but are more general suggestions or free discussions.

If the review quality is good, please rewrite the rationale from an AI Assistant's perspective. Here are the hints to derive a good rationale:

- Try to VERIFY each hypothesis you proposed, using facts in the code change content.
- Summarize the verified hypothesis
- Try to refine the summary to a concise form, the final derived rationale should be specific, helpful and actionable.

Finally, please output a <conclusion> in the following format. Note, if it's a bad quality review, you can set 'review_rewrite' field as null.

```
{
<conclusion>
   review_quality: {good/bad}
   review_rewrite: {human comment rewritten by an AI assistant}
</conclusion>
}
```

Please stop after writing the <conclusion> and don't output anything else.

### B.3 System Instruction to Rewrite Human Feedback Into Critiques for LLM-judge Training

```
You are given a code review and a corresponding feedback to
the review from developers.  Your task is to analyze the
developer feedback and grade it, in context of the code
changes and code review.

Here are the code changes:
=== Raw Source Control Diff BEGIN ===
code changes go here
=== Raw Source Control Diff END ===

Here are the code review comments for code changes above
{
  "Function": ...,
  "Rationale": ...,
  "File": ...,
  "Line": ...,
  "Tag": ...
}

And here is the human feedback to the code review.
{
  "human sentiment": {positive/negative (thumbs up/down)}
  "human feedback comments": {}
}
Note that "human feedback comments" could be empty if the
user did not provide any.  The "human sentiment" is positive
if they agree with the review and negative otherwise.

Please make sure you understand the code change, code review
and the human feedback.  Then, come up the following:
 1. human_feedback_quality:  a score from 0-5 about the
    quality of human feedback.  A good feedback should
    specifically point out the problem in the code review.
    Output a score of 0 if you did not understand the
    feedback at all.
 2. critique:  rewrite the human feedback to one sentence
    critique by using your knowledge of software engineering.
    If the feedback is of low quality or not available,
    please write your own based on the positive or negative
    sentiment.

Write your thoughts, and then respond with the following yaml
format:
‘‘‘yaml
"human_feedback_quality": {0-5, 0 means you did not understand
    the human feedback at all}
"critique": {one sentence critique to the code review}
‘‘‘
```

### B.4 System Instruction for LLM-judge Scoring

```
You are a language model that specializes in evaluating
reviews for a code change.  You are given details of a code
change as input and a list of corresponding issues/code
suggestions identified.  Your goal is to inspect and review
the issues/code suggestions, and score.

Be aware - the issues/suggestions may not always be correct
or accurate, and you should evaluate them in relation to the
actual code changes presented.

Carefully review both the suggestion content, and the related
code change.  Mistakes in the suggestions can occur.  Make
sure the suggestions are correct, and properly derived from
the code changes.  Mark each issue/suggestion as valid or
invalid.

Score the suggestions:  high scores (8 to 10) should be given
to correct suggestions that address major bugs and issues,
or security concerns.  Lower scores (3 to 7) should be for
correct suggestions addressing minor issues, code style, code
readability, maintainability, etc.  Don't give high scores
to suggestions that are not crucial, and bring only small
improvement or optimization.  Order the feedback the same way
the suggestions are ordered in the input.  Response should be
a valid YAML and nothing else.

Here are the code changes and the corresponding code review
suggestions.
=== Raw Source Control Code Change BEGIN ===
code changes go here
=== Raw Source Control Code Change END ===

=== Suggestions (YAML Format) BEGIN ===
'''yaml
{
  "Function": ...,
  "Rationale": ...,
  "File": ...,
  "Line": ...,
  "Tag": ...
}
{
  "Function": ...,
  "Rationale": ...,
  "File": ...,
  "Line": ...,
  "Tag": ...
}
=== Suggestions (YAML Format) END ===


Here is an example of the expected output format for
reference.  Please review and score each suggestion above.
=== Example Output Format ===
'''yaml
```

```
{
"Suggestion_content": {rationale}
"Status": {valid/invalid}
"Sentiment": {positive if the suggestion is complimenting the
             code changes, negative if the suggestion is
             criticizing the code changes or neutral if uncear)}
"Line_matching": {yes/no, to check if the line number in the
               issue description is matching the pre-pending
               line number shown in the code change}
"Suggested score": {between 0-10}
"Score reason" : the code review suggestion is correct and
               actionable because ...
}
‘‘‘

Note:  you should pay attention to the line number in the
issue/code suggestion description, and judge if it is
matching the pre-pending line number shown in the diff
format.  If the code suggestions are empty, just return an
empty YAML string.


Now begin.  Please format your output according to the
"Example Output Format" as shown above.  Remember to do the
review and score each code suggestion.
```

### B.5 System Instruction for the Issue Validator Agent

Your input is a difference view of code changes, and a list of code issues for it.
Your goal is to inspect, review the issues and score each of them.
Be aware - the issues may not always be correct or accurate, and you should evaluate them in relation to the actual code changes presented.

Review issues. Carefully review both the issue content, and the related code changes. Mistakes in the issue can occur. Make sure the issues are correct and properly derived from the code changes. Mark each issue as valid or invalid.

Score issues. High scores (8 to 10) should be given to correct issues that address major bugs, or security concerns. Lower scores (3 to 7) should be for issues that address minor concerns for example code style, code readability, maintainability, etc. Don't give high scores to suggestions that are not crucial, and bring only small improvement or optimization to the code. Incorrect issues should be scored as 0. Order the feedback the same way the issues are ordered in the input.

When reviewing issues, pay special attention to the following common mistakes:

- When reviewing documentation issues, verify the presence of existing docstrings. If a docstring is already present in any form (e.g., as a comment, a description, or a formal docstring), without nitpicking about its quality or completeness, mark the issue as invalid.
- When reviewing division by zero issues, carefully examine the code to verify that the variable in question is indeed used as a divisor in a division operation. If the variable is not involved in a division or if zero validation has already been performed, mark the issue as invalid.

Record verbosely your thought process in the output field "motivation".

## C Appendix: Data format

In Fugure 5 below, we give an example visualization of the code changes in diff format which is the input format for the CQS system. We also show a sample code quality issue generated as part of the code review by the CQS system.

src/handler/Handler.cpp

```
...  @@ -412,7 +413,9 @@
413  413    initCache();
414  414
415  415    const auto& rankCfg =
415  -      *cfg->get_rankSettings();
416  +      FLAG_use_old_rank_settings
417  +      ? *cfg->get_rankSettings()
418  +      : *idx::RankSettingsProvider::getSnapshot();
419  419    const auto& structCfg =
420  420        *cfg->get_structConfig();
421  421
```

src/retriever/RetrieverHandler.cpp

```
...  @@ -262,8 +263,10 @@
263  263
264  264    const auto structCfg =
265  265        *rCfg->get_structConfig();
265  -  const auto rankCfg =
266  -      *rCfg->get_rankSettings();
266  +  const auto& rankCfg =
267  +      FLAG_use_old_rank_settings
```

```
File: src/retriever/RetrieverHandler.cpp
Line: 267
Function: handleRetrieval
Rationale: The new code has duplicated logic in both Handler.cpp and RetrieverHandler.cpp for getting the ranking settings.
Tag: DedupeLogic
```

```
268  +      ? *rCfg->get_rankSettings()
269  +      : *idx::RankSettingsProvider::getSnapshot();
270  270    const auto dataLoaders = std::make_shared(
271  271        *rCfg->get_structConfig()->whitelistModels());
272  272
```

Figure 5: An example of code changes in diff format and code quality issues identified by the CQS system

# D Appendix: Training Details

For all training runs, we used 4x8 H100 GPUs with tensor parallelism set to 8. For SFT experiments used greedy decoding for generation.

## D.1 SFT Training of the Issue Collector Agent

We use a dataset of 5000 (diff, code review) pairs for SFT training. Please refer to Figure 2 in Section 3.1 for details about the data collection and training pipeline. We use Llama 3.1-70b Instruct model as the base checkpoint. For training, we use a sequence length of 8192, a batch size of 64 and set the learning rate to be 2e-7, with a warm-up period of 10 steps and a minimum rate of 0.2 with cosine annealing schedule. The training is only done for 1 epoch as we observed a degradation in model quality post that.

## D.2 SFT Training of the LLM-judge model

For training the LLM-judge model, we used a dataset of 1600 (diff + code review, code review critiques). Please refer to Figure 4 in Section 3.2 for details about the data collection and training pipeline. We use Llama 3.1-70b Instruct model as the base checkpoint. For training, we use a sequence length of 8192 and a batch size of 32. We set the learning rate schedule to be exactly the same as the one for training the issue collector agent above.

## D.3 DPO Training of the Issue Collector Agent

### D.3.1 Data Creation

We describe the logic to create preference pairs for DPO training using LLM-judge scores. The pseudo-code is shown in Algorithm 1 below. For each code change, we compare all pairs of model generated code reviews and for each pair, we compare issues with the same issue tag but different scores (as graded by the LLM-judge) – if the score difference is greater than a threshold (chosen empirically), we add this pair of issues to our training dataset for DPO training. This way, we were able to obtain a set of 8400 preference pairs using the same set of 5000 diffs we had curated for phase 1 training.

**Algorithm 1** Make Issue-wise Preference Pairs for DPO

**Inputs**: code changes $x_i$, model generated code reviews $\{y_i^{(j)}\}_{j=1}^{10}$, threshold $\delta$,
preference_pairs $\leftarrow$ [ ]
**for** each $x_i$ **do**
    issue_list($y_i^{(j)}$) $\leftarrow$ extract list of issues from each code review $y_i^{(j)}$.
    **for all** $\{(a, b) \mid a \in$ issue_list($y_i^j$), $b \in$ issue_list($y_i^k$)$\}$ **do**
        **if** $a$.tag $= b$.tag **and** $|a$.score $- b$.score$| \geq \delta$ **then**
            preference_pairs.append($\{x_i$, chosen=$a$, rejected=$b\}$)
        **end if**
    **end for**
**end for**
**return** preference_pairs

### D.3.2 DPO Training of the Issue Collector Agent

For DPO training, we use the SFT checkpoint from phase 1 training as the initial model. We used a sequence length of 8192, a batch size of 32 and set the learning rate to be 2e-7, with a warm-up period of 25 steps and a minimum rate of 0.5 with cosine annealing schedule. DPO training starts from the SFT-ed 70b checkpoints, we perform DPO training on the 8k preference pairs generated from judge results. Experiment setups follow a learning rate of 2e-7, with a warm-up period of 25 steps and minimum rate of 0.5 with cosine schedule. We set the parameter to control KL regularization $\beta = 0.1$.

## E Appendix: LLM-judge Evaluation

Here, we present a small study to study the quality of the LLM-judge model after fine-tuning. Recall that we trained the LLM-judge model using developer feedback on code reviews generated by the SFT checkpoint. From this dataset, we set aside approximately 200 code reviews with thumbs up/down signal to evaluate the LLM-judge model — with 100 positive examples (code reviews with a thumbs up) and 100 negative examples (code reviews with a thumbs down). On this evaluation data set, the fine-tuned LLM-judge model scored a 62% accuracy as compared to a 56% accuracy scored by a Llama 3.1-70b Instruct model.

## F Appendix: Failure Cases

In this section, we show some example cases where the CQS system fails. In our experience, many of the failure cases are those where the model lacks sufficient code context/underlying business logic or cases where the model lacks extensive knowledge of the programming language itself. In such cases, the issue identified by the CQS system might be trivial or misleading.

**Example 1: Missing language context**

```
Code line added:
buffers = new(buffer.data()) EndToEndBuffers

Issue Rationale from CQS:
The constructor now allocates memory for 'EndToEndBuffers'
using the new operator.  Is it a good idea to use a smart
pointer instead to avoid potential memory leaks?
```

Consider the example above where the code line added in the diff is using the "placement new" operator in C++ to construct an "EndToEndBuffers" object at the memory location pointed to by "buffers.data()". The CQS system incorrectly flagged this as a potential memory leak issue. The placement new operator does not allocate new memory but rather constructs an object in place. So, the suggestion to use a "smart pointer" here is incorrect. Smart pointers are expected to own the memory they manage but since the placement new pointer does not allocate memory, the smart pointer would not own the memory. In fact, wrapping a placement new pointer in a smart pointer is likely to lead to undefined behavior.

**Example 2: Nitpicking on extracting shared code**

```
Code line added:
const auto& container = getContainer(ad, adToAugment);

Issue Rationale from CQS:
The call `getContainer(ad, adToAugment)` to obtain
`container` is repeated in this function (lines 461-463),
and similarly in `FuncA` (lines 473-475) and `FuncB` (lines
486-488).  Would it be beneficial to extract this common
logic, perhaps into a private helper function within
`ClassABC`, to improve maintainability and reduce redundancy?
```

In this example, C++ code added in the diff is essentially a straightforward, three-line fetch operation. However, the CQS system suggests to extract this simple fetching logic into a separate function for deduplication. This seems like nitpicking/over-engineering since the duplication is at a couple of more places only.