Code Review Automation using Large Language Models with Retrieved Augmented Generation

Krishna Sri Ipsit Mantri
Department of Computer Science
Purdue University
West Lafayette, IN, USA
mantrik@purdue.edu

Pavan Chaitanya Penagamuri
Department of Computer Science
Purdue University
West Lafayette, IN, USA
ppenagam@purdue.edu

Nishchal Jagadeesha
Department of Computer Science
Purdue University
West Lafayette, IN, USA
jagadees@purdue.edu

Suraj Surya Narayana
Department of Computer Science
Purdue University
West Lafayette, IN, USA
ssuryan@purdue.edu

Keerthana Ashokkumar
Department of Computer Science
Purdue University
West Lafayette, IN, USA
kashokku@purdue.edu

ABSTRACT

Manual code review, a cornerstone of software development, is often time-consuming and hinders developer productivity. Large Language Models (LLMs) and Retrieval Augmented Generation (RAG) hold promise for automating aspects of this process. This research investigates the application of LLMs and RAG to generate automatic code reviews, focusing on improvements over existing LLM-based approaches. There are two key contributions of this work. First, we propose a novel approach of leveraging RAG to tailor the LLM to code review generation without expensive fine-tuning, enhancing accuracy and context-specificity compared to using off-the-shelf LLMs. We perform a comprehensive study of using RAG in the two tasks of code review automation - review comment necessity prediction and review comment generation. Second, we developed a GitHub App to integrate this model as a review assistant, providing immediate feedback and suggestions to developers. This project aims to significantly reduce the burden of manual code review, facilitating a more efficient and effective development process. Our code is available at this **GitHub link**. This is a private repository, so we have invited **tianyi-zhang** as collaborator to access the repository.

CCS CONCEPTS

• **Software and its engineering → Software Automation and Tools**;

KEYWORDS

Code Review Automation, Large Language Models, Retrieved Augmented Generation, Deep Learning

ACM Reference Format:
Krishna Sri Ipsit Mantri, Pavan Chaitanya Penagamuri, Nishchal Jagadeesha, Suraj Surya Narayana, and Keerthana Ashokkumar. 2024. Code Review Automation using Large Language Models with Retrieved Augmented Generation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code review is a critical component of the software development lifecycle, where someone other than the author inspects the source code to identify issues and improve the quality of the software. This examination focuses on various aspects like functionality, efficiency, security, readability, maintainability and coding style. A typical code review checklist is as follows [9]:

(1) Verifying feature requirements
(2) Assessing readability
(3) Testing maintainability
(4) Checking for security vulnerabilities
(5) Speed and performance considerations
(6) Confirming adequate documentation
(7) Inspecting naming conventions

Manual code review is time-consuming, especially for large codebases. Automation alleviates this burden, allowing developers to focus on more creative tasks. Automating basic checks can provide immediate feedback to developers, helping them identify and fix issues quickly. Smooth integration with existing tools and workflows is important for practical adoption of the automated code review techniques.

Large Language Models (LLMs) and Retrieval Augmented Generation (RAG) have made significant strides in recent years, impacting various code related tasks. LLMs can generate basic functions, complete code snippets and even generate entire programs based on prompts and specifications. LLMs can automatically generate documentation for existing code, improving code understanding and maintainability. RAG can retrieve relevant code examples and provide more content specific and informative context.

While Large Language Models (LLMs) have demonstrated remarkable effectiveness in various code-related tasks, their application to automatic code review generation remains relatively unexplored. Existing research primarily focuses on utilizing off-the-shelf LLMs directly, yielding suboptimal results due to a lack of context-specific understanding. Additionally, fine-tuning LLMs specifically for code review is deemed an expensive and resource-intensive undertaking.

This project addresses these limitations by proposing a novel approach that leverages LLMs in conjunction with Retrieval Augmented Generation (RAG) for automatic code review generation.

RAG empowers the LLM to tailor its responses to the specific context of code reviews, leading to enhanced accuracy and contextual relevance compared to conventional LLM usage. Furthermore, the project contributes to the field by developing a GitHub App that integrates this model as a review assistant. This app provides developers with immediate feedback and suggestions, streamlining the code review process and contributing to increased developer productivity.

Each aspect from the above checklist requires a senior developer's inherent programming prowess, and their knowledge of the entire code repository. LLMs have achieved human-level performance in various tasks, and LLMs that have been fine-tuned for code generation have demonstrated the abilities of a human software engineer. Taking inspiration from this, we propose to automate code reviewing task using LLMs + RAG.

To rigorously assess the proposed approach, we will address the following research questions. Efficacy: Does our RAG-enhanced model outperform off-the-shelf LLMs in generating automatic code reviews? If so, to what extent? This comparison will quantify the added value of RAG in tailoring the LLM to the specific domain of code review. Efficiency: Does our model achieve comparable performance to fine-tuned LLMs, while avoiding the associated resource demands? This evaluation will demonstrate the cost-effectiveness of using RAG as an alternative to expensive fine-tuning. We tackle the process of code review generation as a two-step process. This is a common standard followed in the previous works. First step is review comment necessity prediction. This is a binary classification task and we use Precision, Recall and F1-score as metrics. Second step is review comment generation. We will employ the BLEU score as a metric to assess the quality and coherence of generated code reviews across different models. This established metric reflects the level of n-gram overlap between generated text and human-written references, providing a quantitative measure of performance.

2 RELATED WORK

In the landscape of automated code review, several existing works have contributed significant advancements in leveraging large language models (LLMs) for enhancing software development processes. Notably, *CodeReviewer*, [6] a domain specific pre-trained language model by taking code diffs as input to generate code review comments. It implements an encoder-decoder Transformer architecture akin to the T5 model, integrating four pre-training tasks—diff tag prediction, denoising code diff, denoising review comment, and review comment generation—to facilitate a comprehensive understanding of code differentials and proficiently generate review comments. The authors demonstrate that this multi-task pre-training approach leads to significant performance improvements on code review tasks compared to models trained on a single task. However, this approach entails substantial data preprocessing and intricate mathematical computations. Additionally, the authors do not extensively tune hyperparameters, and their dataset consists of open-source GitHub repositories rather than industrial projects, which may require specific fine-tuning to capture project-specific contexts.

Conversely, *LLaMA-Reviewer* [8] capitalizes on LLaMA, a pre-trained language model, employing parameter-efficient fine-tuning (PEFT) techniques to adapt LLaMA to code review tasks without necessitating pre-training from scratch. Despite achieving performance comparable to CodeReviewer, LLaMA-Reviewer's fine-tuning methodologies incur resource-intensive processes and introduce additional latency. The authors also state that their model may not generalize beyond the two datasets used for fine-tuning, and each dataset preserves only a single comment per code change, which may lead to potential biases.

A recent work, Improving Automated Code Reviews: Learning from Experience by Lin et al. [7] investigates experience-aware oversampling models for code review, aiming to generate higher-quality reviews based on reviewer expertise. Another recent work, Reimagining code review with RAG to save us from LGTM from watermelon [10] introduces a GitHub application powered by RAG to enhance code reviews, using RAG in a similar manner to how ChatGPT interacts with the internet for latest knowledge.

Our approach is inspired from *Repocoder*, [11] which introduces a repository-level code completion approach, harnessing LLMs and iterative retrieval and generation methods to facilitate code completion within the broader context of GitHub repositories. Our approach draws inspiration from the success of repository-level code completion tasks, enhancing the quality of review comments by providing contextual information from diverse project repositories.

These existing works collectively underscore the evolving landscape of automated code review, emphasizing the significance of leveraging advanced techniques such as pre-trained LLMs and retrieval-augmented generation to streamline software development workflows and enhance code quality.

3 APPROACH

Our approach involves two steps aimed at automating the code review process. First step is *review comment necessity prediction*, in which we predict whether a given chunk of code needs review comment or not. This is a binary classification task. If the result of first step is 'No', then the process for this chunk of code stops here. If the result of first step is 'Yes', then we go to second step in the process. Second step is *review comment generation*, in which we generate a meaningful review for the chunk of the code. The idea of dividing the process into two steps is taken from previous studies and also supported by wide usage of chain of thought prompting when solving complex tasks using LLMs. In both the steps, LLM is prompted with the code snippet. In the first step, LLM is asked to return only Yes/No, in the second step, LLM is asked to return a review comment.

We utilize RAG (Retrieved Augmented Generation) to improve the results in both the steps of the process. We integrate a vector store, Weaviate [4], into our pipeline. In the vector store, each document is a code snippet. Before prompting the LLM in both the steps, we first query the vector store with the chunk of code to get similar code snippets. We pass these code snippets along with the relevant information about the snippet as context, in the prompt to the LLM. Relevant information being 'Yes' or 'No' for the similar code snippets for first step, and review comments for the similar code snippets for the second task. The entire pipeline can be seen in the figure **??**.

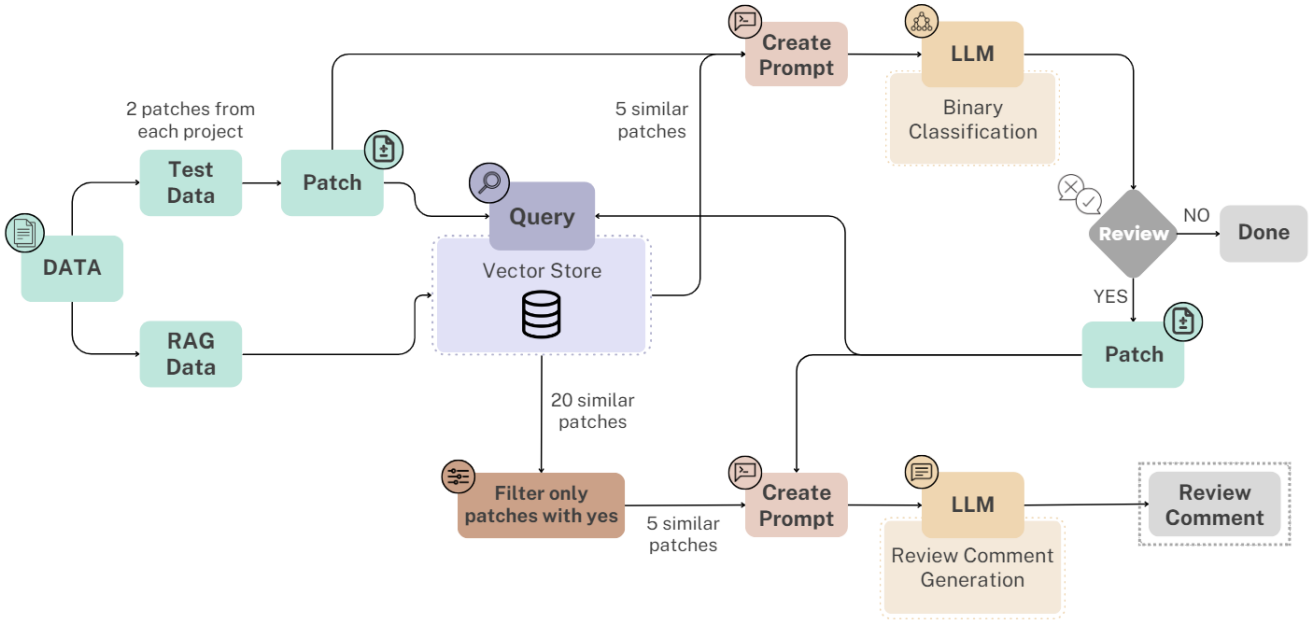

Figure 1: LLM-Model Pipeline

3.1 RAG and Vector Store

Retrieval Augmented Generation (RAG) is an important technique for enhancing the capabilities of Large Language Models (LLMs) beyond their original training data. RAG achieves this by integrating LLMs with an external knowledge base or database, allowing the models to access relevant information that can improve the accuracy, relevance, and timeliness of their outputs. By grounding the LLM's output in relevant external knowledge, RAG helps mitigate the risk of the model generating inaccurate or fabricated information, reducing hallucination. Compared to other approaches like fine-tuning, RAG is a simpler and more cost-effective way to customize LLMs with domain-specific data, especially when the models need to be updated frequently.

RAG typically involves following steps. The external data sources are pre-processed, chunked into appropriate lengths, and indexed using a vector store search engine. When a user query is received, the retrieval mechanism identifies the most relevant sections of the indexed data based on the query. The relevant information from the external sources is combined with the user's query to create a context-rich input for the LLM. The LLM then uses the augmented input to generate the final response, which can be more accurate, relevant, and up-to-date compared to what the LLM could produce on its own.

Vector storage is a database where data, such as text documents, images or other types of content are stored in high dimensional vector representations. They are particularly useful for Retrieval-Augmented Generation (RAG) with Large Language Models (LLMs). Based on semantic similarity between a user's query and the stored vectors, vector stores allow for swift and efficient retrieval of relevant documents or passages. This is crucial for RAG, as the LLM needs to quickly access the most relevant information to generate a high-quality response. There are multiple vector stores available like Pinecone, Weaviate, Elasticsearch, ChromaDB and Redis. Our choice of using Weaviate is based on the open-source nature of the vector store and its ease of setup and use.

We used BAAI/bge-small-en-v1.5 text to vector embedding to convert the code snippets to vectors when storing into the vector store. The choice of using BAAI/bge-small-en-v1.5 embedding is based on the size of embedding and its performance in the Hugging Face dashboard. BAAI/bge-small-en-v1.5 is a pre-trained text embedding model developed by the Beijing Academy of Artificial Intelligence (BAAI). It is a small-scale version of BAAI's general embedding model, called BGE (BAAI General Embedding). The model has a dimensionality of 384. One potential future work could be, to explore other vector stores like ChromaDB and other text to vector embeddings.

3.2 Re-ranking Strategy

For any code patch, a top-k similarity search is performed by the vector store to retrieve similar code snippets. These code snippets are passed in the prompt to LLM to increase the relevance of the answer. It is important to include documents with similar code patch and also the review comments with desired format in the prompt. For this, we have experimented with one novel re-ranking strategy. It is to re-rank the documents retrieved from vector store in such a way that documents which belong to the same project as that of query code patch should have higher score. For this, we performed a weighted re-ranking in which, we introduce a tunable hyper-parameter γ with which we multiply the similarity scores from top-k retrieval. We only multiple with γ, scores of the document which belong to the same project as query code patch. The project information for a code snippet can be stored as metadata in the

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -147,7 +147,12 @@ static void GetScalarFieldOfTable(const StructDef &struct_def,
     getter = "bool(" + getter + ")";
   }
   code += Indent + Indent + Indent + "return " + getter + "\n";
-  auto defaultValue = (is_bool ? "False" : field.value.constant);
+  std::string defaultValue;
+  if (is_bool) {
+    defaultValue = field.value.constant == "0" ? "False" : "True";
+  } else {
+    defaultValue = field.value.constant;
+  }
   code += Indent + Indent + "return " + defaultValue + "\n\n";
 }
```

Answer :

```
{"reviewNeeded" : "true"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, answer whether it needs review or not. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -726,6 +726,8 @@ static void initMemberInfo (statementInfo *const st)
   case DECL_CLASS:
    if (isLanguage (Lang_java))
     accessDefault = ACCESS_DEFAULT;
+   else if(isLanguage(Lang_d))
+    accessDefault = ACCESS_PUBLIC;
    else
     accessDefault = ACCESS_PRIVATE;
    break;
```

Figure 2: Prompt for Review Comment Necessity Prediction

vector store and can be retrieved during the top-k similarity search. But during our preliminary analysis, we found that, vector store is not retrieving documents from same project in the top-k similarity search even for high values of k. This happened for all the code patches in our test data. This is an interesting observation which is opposite to our hypothesis. So we decided not to implement this re-ranking strategy in our workflow.

3.3 Review Comment Necessity Prediction

For each diff chunk, we query the vector store to get top N code snippets. We experimented with various values for N and found that $N = 5$ works best. For the code snippets we also extract the metadata of 'Yes/No' prediction of diff quality task. The prompt for the LLM will contain the code patch and similar patches extracted from vector store along with information about whether these patches required review and we ask LLM to decide whether the code patch requires review. In the prompt, we restrict the LLM to return only yes/no in the response in a pre-determined json format. We perform json extraction to remove any other explanations LLM provides. If LLM answers 'No' then we stop here for this code patch. If LLM answers 'Yes' then we move on to next step. One example prompt is given in figure 2.

3.4 Review Comment Generation

The vector store is queried with the code patch to retrieve top N similar documents. Out of the documents retrieved we take top k documents for which the diff quality task result is 'Yes', that is the documents for which a review comment exists. The reason for doing this is, we only want to involve code snippets with reviews in our prompt in the second step so that LLM doesn't get confused by code snippets with no review comments. After some experimentation, we found that the combination of $N = 20$ and $k = 5$ works the best. There are multiple alternate design choices to the design choice me made. One alternate design choice could be to maintain two different vector stores. One vector store will contain all types of documents while the second vector store will contain only the documents with review comment. In the second step second vector store can be used for querying. This desgin choice wasn't adapoted

PROMPT:

```
Role:

You are a lead software engineer performing code reviews.
```

```
Example Patch:
@@ -538,6 +538,10 @@ const char* PrimaryExpression::decode(const char *pos, const char *end) {
 std::string FunctionCallExpression::toString() const {
     std::string buf;
     buf.reserve(256);
+    if (expr_ != nullptr) {
+        buf += expr_->toString();
+        buf += " ";
+    }
     buf += *name_;
     buf += "(";
     for (auto &arg : args_) {

Answer :
{"reviewComment" : "Why not assign in one line like `buf += expr_->toString() + " ";`"}
```

```
You are a lead software engineer performing code reviews.

For the below patch of code, generate one review comment. Although there could be multiple
issues with the patch of code, only give the most important suggestion. Example patches of code
and expected answers are given above. Use these examples as reference context and generate
response in the same format as the answers. The response should only be in json format.
```

```
Patch:
@@ -308,8 +308,12 @@ std::string PrimaryExpression::toString() const {
         case VAR_INT64:
             snprintf(buf, sizeof(buf), "%ld", boost::get<int64_t>(operand_));
             break;
-        case VAR_DOUBLE:
-            return std::to_string(boost::get<double>(operand_));
+        case VAR_DOUBLE: {
+            int digits10 = std::numeric_limits<double>::digits10;
+            const char *fmt = folly::sformat("%.{}lf", digits10).c_str();
+            snprintf(buf, sizeof(buf), fmt, boost::get<double>(operand_));
+            break;
+        }
         case VAR_BOOL:
             snprintf(buf, sizeof(buf), "%s", boost::get<bool>(operand_) ? "true" : "false");
             break;
```

Figure 3: Prompt for Review Comment Generation

because this will need additional resources because of additional vector store and there is duplication of data. Every time there is a update, update needs to be made at two places which is not a good practive. Another alternate design choice is to retrieve 20 documents in the first step only and use these in the second step. Advantage of this approach is, there would be no need for querying in the second step. But retrieving 20 documents everytime in the first step is unnecessary as every time second step is not needed. So after careful consideration we decided to go with the design choice of having a single vector store and querying it two times and performing filtering during the second step.

The prompt for the LLM will contain the code patch and similar patches extracted from vector store along with information about review comments for the similar patches and we ask LLM to generate a appropriate review comment for the code patch. In the prompt, we restrict the LLM to return review comment in the response in a pre-determined json format. We perform json extraction to remove any other explanations LLM provides. In the prompt, we also restrict LLM to give only one review comment even if there are multiple possible suggestions for code improvement. We ask the LLM to give the most important suggestion. This is done so that we can compare the review comment with the review comment in test data. One example prompt is given in figure 3.

4 DATASET

We use the carefully curated dataset from CodeReviewer [6] for our model evaluation. The dataset contains 328340 code patches for first task i.e diff quality estimation and 138227 code patches for second task i.e review comment generation. There are lower number of code patches for second task because there are code patches

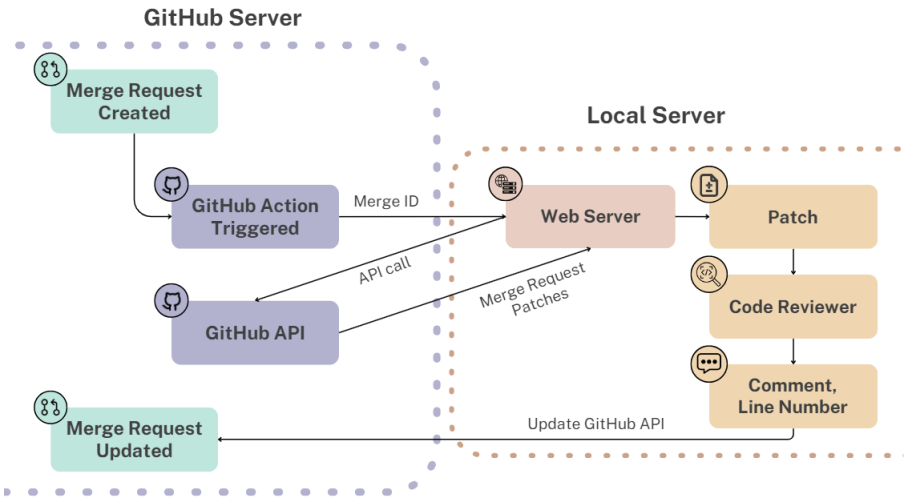

Figure 4: Github Bot Flow

which have no review comments in the first task dataset. We use the diff quality estimation code patches in our model analysis. The CodeReviewer dataset is divided into three parts - train, validation and test. Due to resource constraints all our analysis and evaluation is performed on the test dataset. Another reason for choosing test dataset is because documents in test dataset has the project meta-data information. This information is not present for documents in train dataset. During our initial stages of experiments, we were using the project based re-ranking strategy, which we removed later in our workflow due to the reasons explained before. This was another reason why we used test data of code reviewer dataset as our dataset. The test dataset contains approximately 31 thousand code patches. From the test data, we created a new test data with two documents from each project. Using the remaining data, we created the vector store. This is done so that data corresponding to each project is present in the new test data. The new test data contains 414 code patches. For each document in the vector store, we store code patch and also metadata information about the project to which the code patch belongs to, review comment (if present) and Yes/No for the diff quality estimation task.

5 GITHUB WORKFLOW AND WEB APPLICATION

There are two sides to the GitHub tool we developed for generating automatic code reviews for merge requests in GitHub. The first side is the setup required on the GitHub side. The second side is the setup required on the web server side. On GitHub side, in the repository to which you want this automatic review comment functionality to be added, a GitHub workflow needs to be created. The workflow triggers whenever a merge request is created. The GitHub workflow action makes a API call to the web server. In the API call the merge request ID is sent. The web server URL needs to be configured in the GitHub workflow action.

On the web server side, the GitHub repository name and GitHub API Key needs to be configured. We have deployed a python based web server using flask [5] on our local machines. One future work would be to deploy the web server on cloud. When the web server receives the API call with merge request ID, the web server makes a GitHub API call using the merge request ID to get the code diff file. The code-diff file may consist of multiple chunks of code additions (+++) and code deletions(—). Each code-diff chunk starts with a metadata line indicating the range of line numbers before and after the code change. Utilising these patterns, we use regex to split the whole code-diff file into a vector of code-diff chunks. Each code-diff chunk is passed to our RAG code reviewer model to perform review comment necessity prediction and review comment generation.

The LLM model is prompted to generate output in a structured JSON format, specifying reviews and relative line numbers (offsets) in the code diff chunk where they should be attached. The metadata line associated with each code-diff chunk is leveraged to extract the absolute line numbers, that can be coupled with offsets computed earlier to determine the original line numbers. Finally, the code reviews are automatically posted at the specific line numbers using GitHub's Pull-request API. This entire process is fully automated and triggered whenever a pull request is raised, streamlining the code review process and enhancing collaboration within the development workflow. The entire GitHub bot and web server workflow can be seen in figure ??.

6 EVALUATION SETUP

Our evaluation framework comprises of two core tasks: *Review comment necessity prediction* and *review comment generation*. The review comment necessity prediction task involves determining whether a given code chunk necessitates review comment, which is treated as a binary classification problem. To assess the model's performance in this task, we rely on comprehensive metrics including F1-score, precision and recall. These metrics collectively provide insights into the model's ability to accurately identify code segments requiring review comment, while minimizing false positives and false negatives.

Model	*Run Type*		*Review Comment Necessity Prediction*			*Review Comment Generation*
	RAG	**Binary Class.**	**Precision**	**Recall**	**F1 Score**	**BLEU-4**
GPT3.5	✓	✓	0.529	0.782	0.631	7.98E-04
	✗	✓	0.508	0.847	0.635	7.62E-04
	✓	✗	-	-	-	7.32E-04
LLAMA3 70B	✓	✓	0.624	0.685	0.653	7.97E-04
	✗	✓	0.522	0.81	0.635	5.56E-04
	✓	✗	-	-	-	6.50E-04
Mistral 7B	✓	✓	0.528	0.819	0.642	6.64E-04
	✗	✓	0.498	0.819	0.619	5.90E-04
	✓	✗	-	-	-	6.00E-04

Table 1: Performance Comparison of RAG-LLM Models in Automated Code Review

On the other hand, the review comment generation task focuses on generating coherent and contextually appropriate review comments for identified code segments. To evaluate the quality of the generated comments, we employ the BLEU-4 score metric. BLUE-4 measures the similarity between the generated comments and reference comments by assessing the overlap of n-grams, providing a quantitative measure of the generated comments' similarity to references.

We use three different pre-trained LLMs - LLaMA3 [2], Mistral [3] and GPT-3.5 [1] for our prompting. **LLaMA3** [2], developed by Meta AI, is a state-of-the-art large language model with a staggering 70 billion parameters. It is designed to excel in natural language processing tasks and exhibits high performance in various domains, including code generation and understanding. **Mistral** [3], developed by Mistral AI, is a powerful language model with 7 billion parameters. While relatively smaller in scale compared to LLaMA3, Mistral is still highly effective for code-related tasks, including code review automation. Its design prioritizes efficiency and effectiveness, making it a suitable choice for generating review comments with precision and coherence. **GPT-3.5** [1], developed by OpenAI, is among the largest language models available with 175 billion parameters. Known for its versatility and ability to understand and generate human-like text, GPT-3.5 offers unparalleled capabilities for natural language processing tasks, including code review automation.

In our experimental setup, we conducted three different experiments using above three different pre-trained LLMs on the test dataset:

(1) Review Comment Necessity prediction and Review Comment Generation with RAG
(2) Review Comment Necessity prediction and Review Comment Generation without RAG
(3) Review Comment Generation with RAG

6.1 Review Comment Necessity prediction and Review Comment Generation with RAG

This experiment involves employing RAG in both the review comment necessity prediction and review comment generation tasks. By integrating external knowledge through RAG, we aim to enhance the model's understanding and contextual relevance, thereby improving the accuracy of both prediction and generation processes. The results of this experiment will provide us with the understanding of how good our model is compared to pre-trained models.

6.2 Review Comment Necessity prediction and Review Comment Generation without RAG

In this experiment, we exclude the use of RAG from both the review comment necessity prediction and review comment generation tasks. This serves as a baseline comparison to evaluate the model's performance without the integration of external knowledge. The results of this experiment will provide us with the understanding of the importance of RAG over the usage of off-the-shelf LLMs.

6.3 Review Comment Generation with RAG

This experiment focused solely on the review comment generation task, where RAG was applied to enhance the contextual relevance of the generated comments. This experiment is an ablation study performed to identify the importance of the first step i.e review comment necessity prediction. We evaluate the importance of first step and to see if the LLM produces review comments to the code patches for which review comment is not needed. For this experiment, we modify the prompt slightly and ask the LLM to return a empty review comment if review is not needed for the code patch.

Through above experiments, we sought to comprehensively evaluate the effectiveness and efficiency of our approach in automating the code review process, with a specific focus on the accuracy and contextual relevance of the generated review comments. The results of these evaluations can be seen in Table 1. Example prompts are provided in appendix.

7 RESULTS

In this study, we answer three research questions:

- **RQ1.** *How does RAG-LLM compare to off-the-shelf LLM?*
- **RQ2.** *How does RAG-LLM compare to existing models?*
- **RQ3.** *Is the review comment necessity prediction task important?*

The first question aims to evaluate the effectiveness of Retrieval Augmented Generation (RAG) in comparison to off-the-shelf Large Language Models (LLMs) and existing code review models. The second question compares the performance of RAG-LLM with existing models in terms of precision, recall, and F1-score for both binary classification and review comment generation tasks. The third question focuses on investigating the importance of the binary classification task in the code review automation process by assessing its impact on the on BLEU score, thereby determining its necessity.

7.1 RQ1. How does RAG-LLM compare to off-the-shelf LLM?

In our research, we sought to evaluate the performance of Retrieval-Augmented Generation (RAG) in conjunction with Large Language Models (LLMs) compared to off-the-shelf LLMs in the context of automated code review. Through our experiments, particularly the second experiment where RAG was not utilized in either step of the process, we made several key observations. Firstly, we noted a significant drop in precision accompanied by an increase in recall in the binary classification task for all models. This shift in precision and recall signifies that without RAG, a greater number of both positive and negative samples are being misclassified as positive. Consequently, this results in review comments being generated for code patches that may not necessarily require review. Such misclassification underscores the importance of RAG in ensuring that comments are generated only for patches that truly warrant review. Additionally, our analysis revealed that while the F1-score remained relatively stable for GPT and LLAMA models, there was a noticeable decrease in F1-score for Mistral, dropping from 0.642 to 0.619. This discrepancy suggests that the integration of RAG is particularly beneficial for smaller models like Mistral, underscoring its effectiveness in enhancing the performance of LLMs, especially in scenarios where resource constraints may limit model size. Overall, these findings underscore the efficacy of RAG-LLM in improving the precision of automated code review systems and ensuring that review comments are generated judiciously for code patches in need of attention.

7.2 RQ2. How does RAG-LLM compare to existing models?

In our investigation comparing the performance of Retrieval - Augmented Generation (RAG) with Large Language Models (LLMs) to existing models in automated code review, we found notable differences across various evaluation metrics. Firstly, when assessing metrics such as precision, recall, and F1-score for binary classification, RAG-LLM, particularly the LLAMA3 70B model, demonstrated commendable performance with an F1-score of 0.653. However, these scores were still lower compared to existing models such as CodeReviewer, which achieved an F1-score of 0.715. This outcome was somewhat anticipated, considering that trained or fine-tuned models typically outperform those relying solely on retrieval-based methods like RAG.

Additionally, our analysis revealed stark differences in BLEU-4 scores for review comment generation between RAG-LLM and existing models. Specifically, the BLEU-4 score for LLAMA3 70B with RAG was notably low at 7.97e-4, whereas CodeReviewer achieved significantly higher scores at approximately 5e-2. This substantial contrast can be attributed to several factors. Firstly, while the actual meaning of generated comments may align with ground truth, discrepancies in wording and expression can lead to lower BLEU scores. Furthermore, variations in the focus and interpretation of code patches between LLMs and ground truth annotations may contribute to divergent BLEU scores. Considering these challenges, questions arise regarding the suitability of BLEU as a metric for evaluating review comment prediction.

Alternatively, exploring metrics such as Cosine similarity could offer insights into the semantic similarity between generated and ground truth comments. In our experiments, Cosine similarity yielded promising results, indicating a high degree of semantic alignment between generated and ground truth comments. This suggests that Cosine similarity may serve as a more appropriate metric for assessing the quality and relevance of review comments generated by RAG-LLM models. But still this comparison is not an exact comparison as the datasets for both the models are different. Our model uses a part of original code reviewer test data set for storing in vector store and the rest of it as test data set. A more appropriate comparison would be to use the original Code Reviewer train data set to be stored in vector store and test on the entire original Code Reviewer test data set. This experiment was performed and results are reported in a later section.

7.3 RQ3. Is the review comment necessity prediction task important?

In our investigation into the importance of the review comment necessity prediction task in the automated code review process, we conducted an ablation study to assess the impact of removing this task. Our analysis focused on comparing BLEU scores obtained with and without the review comment necessity prediction task, serving as a proxy for evaluating the necessity of this component in the code review pipeline. Through this comparative analysis, we observed a consistent trend across multiple models, wherein the BLEU scores were notably lower when the review comment necessity prediction task was omitted. For instance, GPT with RAG and review comment necessity prediction achieved a BLEU score of 7.98E-4, whereas the same model without the review comment necessity prediction yielded a slightly lower BLEU score of 7.32E-4. Similarly, LLAMA and Mistral models exhibited similar trends, with lower BLEU scores observed in the absence of the review comment necessity prediction task.

These findings underscore the importance of the review comment necessity prediction task in enhancing the quality and relevance of review comments generated by RAG-LLM models. The decrease in BLEU scores without review comment necessity prediction suggests that this task plays a crucial role in guiding the model to focus on code patches that truly warrant review, thereby improving the overall effectiveness of the automated code review process. Consequently, our results highlight the significance of incorporating review comment necessity prediction into the code review

Rating	*Description*
1	Review Comment is wrong or irrelevant to the code
2	Review Comment has some relevance to the code but lacks insight or clarity
3	Review Comment is correct but is not insightful; a much better review comment is possible
4	Review Comment is relevant and provides some insight, but there is room for improvement
5	Review Comment is correct and very insightful; it addresses the biggest issue in the code

Table 2: Review Comment Assessment Scale

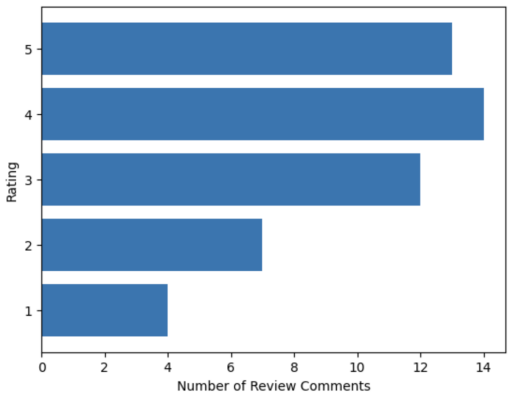

Figure 5: Rating Review Comments

pipeline to ensure that review comments are generated judiciously and accurately.

8 EVALUATION ON ENTIRE TEST DATASET

We performed an evaluation on entire original test dataset of code reviewer test dataset by pushing the entire train dataset of code reviewer into the vector store. This experiment provides a more accurate evaluation of our model performance compared to the pre-trained Code Reviewer model. The results observed for Mistral 7B based RAG model are, Precision is 0.498, Recall is 0.765 and F1-score is 0.604 for the first task of review comment necessity prediction. The BLEU score of 4.4e-4 is observed for the task of review comment generation. F1-score shows that the performance of the model is good but it is not comparable to the pre-trained code reviewer model. Lower F1 score can also be attributed to using a smaller LLM (Mistral) for evaluation due to resource constraints. Other observation is, the model performance is comparable to its performance on the small test dataset. This shows the model performance is good irrespective of the dataset.

9 USER STUDIES

In this section, we present the results of a user study conducted to evaluate the effectiveness of our automated code review system from the perspective of human reviewers. This study involved five human developers, each of whom performed two user studies:

(1) Rating Review Comments
(2) Assigning taxonomy category to review comments

9.1 Rating Review Comments User Study

Each user was provided with code patches along with corresponding review comments. Users were instructed to evaluate each review comment based on its relevance and the insight or help the comment provides, by providing a rating from 1 to 5, as described in table 2. Users assigned scores to each review comment based on their judgement of its quality and usefulness in improving the code. Each user gave rating for ten review comments generated by the RAG code review model. GPT-3.5 LLM model was used for generating review comments for this study. The combined ratings for all the 50 comments are shown in the figure 5.

The results show higher number of review comments with rating 4 and 5. This shows that our model returns insightful and useful comments and also the comments returned address the major issues in the code patches. There are significant number of review comments with the rating 3 which shows the review comments generated by our model are correct but there are better review comments possible. Although our model produces few review comments in the category of 1 and 2, the comments in this category are comments which are wrong or irrelevant to the code, and it is important to address these comments. Further analysis into the comments with this rating showed that the comments try to address issues which are not in the code. One example is a comment which questions a new import as the import is not used in the code patch. It could be possible that the import is used in another code patch in the same merge request or this import could be fixing a import bug in previous merge request. One future work could address this by including other code patches in the same merge request in the prompt to give LLM the context about the other changes in the same merge request to avoid these type of review comments.

9.2 Review Taxonomy Category Assignment User Study

Five users participated in the second study in which each user is provided with 5 prompts. In each prompt there 3 examples of code snippets and their corresponding review comments, the query code patch and review comment returned by our model. Users were tasked with assigning taxonomy category to each review comment (both the example review comments and the review comment returned by our model) based on its nature and content. The taxonomy consisted of 7 categories as listed in table 3. Users carefully analyzed each example comment and the review comment generated and categorize it according to the taxonomy provided. The objective was to understand if the LLM gives review comments in wide range of taxonomy categories and check the influence of the example review comments on the generated review comment. The results can be seen in the figure 6. The plot is only for the review comments generated by our LLM model. The results show that LLM produces review comments across all the taxonomy categories. Only security vulnerabilities category didn't have any review comment, probably because of the less number of review comments considered. One interesting observation is high number of review comments are about

	Category	*Description*
1.	**Readability**	Comment asks for better naming conventions or better structuring
2.	**Efficiency**	Comment suggests improvements for methods or algorithms used
3.	**Security Vulnerability**	Comment points out a security issue
4.	**Maintainability**	Comment suggests writing code in a way that enhances maintainability and scalability
5.	**Correctness**	Comment highlights significant problems with code correctness or syntax
6.	**Not Standard**	Comment suggests an alternate standard way of doing things, such as using a standard library
7.	**Other**	Comments that do not fit into the above categories

Table 3: Review Comment Classification Taxonomy

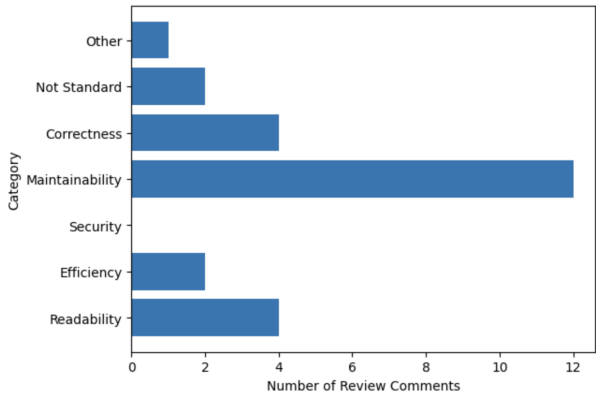

Figure 6: Review Taxonomy

suggestions to improve the code to enhance maintainability and scalability. Our analysis on the affect of context review comment categories on the output review comment showed that there is no relation between them. The taxonomy of context review comments is not affecting the taxonomy of the review comment returned from LLM. Our hypothesis from this results is that LLM uses context review comments to mostly understand the structure, language and format of the review comments but generates the actual review comment based on the patch of code.

10 DISCUSSION AND FUTURE WORK

The major contribution or success of our work showing the effectiveness of RAG for code reviews and improving the existing off-the-shelf LLMs without the overhead of fine-tuning. Our novelty and key feature is using RAG. Other novelty we tried to implement but didn't work out is the project re-ranking strategy. This made us learn that sometimes obvious hypothesis may turn out to be false. Future works could try some other re-ranking strategies. Similar to all LLM based works, our model is also limited by the performance of the underlying LLM. Another limitation of our work is vector store which needs to be constantly updated to stay relevant.

While working on the code reviewer dataset, we found some discrepancies. There is repetition in the data in both in the training and test datasets. There is 19% repetition in the training dataset and 11% repetition in the test dataset. Here the repetition corresponds to the same code patches. There is no overlap observed between the training and test datasets. This repetition raises some doubts about the results reported by Code Reviewer model. All our reported results are obtained after cleaning the data from repeated patches.

Our web server is currently deployed on local and we are able to interact with GitHub by exposing our local IP to internet using a third-party service. This is not a secure way and is not scalable. One potential future work could be to migrate the web server from local to a viable cloud provider. Migrating the web server also requires deploying the vector store in cloud which is currently running on local. The design of web server needs to be changed slightly to be able to serve multiple clients. The setup on GitHub side needs to be simplified to make the app installation easy. This direction of future work will focus on using the existing model and making the GitHub a production ready commercially usable tool.

The prompts for both the review comment necessity prediction task and review comment generation task are designed by trying various handcrafted prompts and picking the ones that work the best. One future work could be to use dspy framework to generate better prompts in a learning fashion. This will involve creating a train and test dataset of examples and creating a optimizer to automatically generate prompt. Using dspy we can exert better control on the LLM output format and eliminate the restricted json parsing currently we have. One more way to improve the prompt would be to include the commit messages of the merge request. This will help the LLM to look for the important places in the code to look for. This direction of future work will focus on improving the model that is runner under in the GitHub tool.

With more resources, a more extensive study can be performed on test data set with the training dataset in vector store. We have only evaluated the performance on entire test dataset on using mistral model because of API constraints. One potential future work could explore the same analysis with GPT-3.5, Llama3 or other LLMs. Similarly due to resource constraints, we have performed only three different experiments with three different LLMs. More experiments with other LLMs could be explored as a future work.

11 CONCLUSION

The integration of RAG with LLMs has proven to be a pivotal advancement in automating code review processes. By leveraging external knowledge sources through RAG, the LLMs were able to generate more accurate and contextually relevant review comments. This approach not only improved the quality of automated reviews but also reduced the likelihood of generating irrelevant or incorrect comments. The development of a GitHub App further exemplifies the practical application of our research, providing developers

with real-time, actionable feedback that seamlessly integrates into their existing workflows. Our work has certain limitations like not being an extensive study of RAG+LLM on code review task and the GitHub tool not being production ready due to the limitation of resources. There are multiple directions of future work possible for our research to handle these limiations in both improving the accuracy of model, making the GitHub bot more accessible for commercial usage by deploying it on cloud and performing more extensive studies with better resources on the application of RAG+LLM for the code review task.

REFERENCES

[1] *'GPT-3.5 is AI model developed by OpenAI'*. URL: https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates.

[2] *'Llama3 is a open-source model developed by Meta'*. URL: https://llama.meta.com/llama3/.

[3] *'Mistral is a open-source model developed by Mistral AI'*. URL: https://mistral.ai/news/announcing-mistral-7b/.

[4] *'Weaviate is an open source, fast, AI-native vector database'*. URL: https://weaviate.io/.

[5] *Flask is a lightweight WSGI web application framework*. URL: https://flask.palletsprojects.com/.

[6] Zhiyu Li 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*. 2022, pp. 1035–1047.

[7] Hong Yi Lin et al. "Improving Automated Code Reviews: Learning from Experience". In: *arXiv preprint arXiv:2402.03777* (2024).

[8] Junyi Lu et al. "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)*. IEEE. 2023, pp. 647–658.

[9] Pluralsight. *Code review checklist: 7 steps to level up your review process*. 2023. URL: https://www.pluralsight.com/blog/software-development/code-review-checklist.

[10] Watermelon. *Reimagining code review with RAG to save us from LGTM*. 2024. URL: https://www.watermelontools.com/post/reimagining-code-review-with-rag-to-save-us-from-lgtm.

[11] Fengji Zhang et al. "Repocoder: Repository-level code completion through iterative retrieval and generation". In: *arXiv preprint arXiv:2303.12570* (2023).

A APPENDIX

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -33,14 +33,14 @@ static PJ_XY fouc_s_s_forward (PJ_LP lp, PJ *P) {     /* Spheroidal, forwa
 static PJ_LP fouc_s_s_inverse (PJ_XY xy, PJ *P) {          /* Spheroidal, inverse */
     PJ_LP lp = {0.0,0.0};
     struct pj_opaque *Q = static_cast<struct pj_opaque*>(P->opaque);
-    double V;
     int i;

     if (Q->n != 0.0) {
         lp.phi = xy.y;
         for (i = MAX_ITER; i ; --i) {
-            lp.phi -= V = (Q->n * lp.phi + Q->n1 * sin(lp.phi) - xy.y ) /
-                (Q->n + Q->n1 * cos(lp.phi));
+            const double V = (Q->n * lp.phi + Q->n1 * sin(lp.phi) - xy.y ) /
+                             (Q->n + Q->n1 * cos(lp.phi));
+            lp.phi -= V;
             if (fabs(V) < LOOP_TOL)
                 break;
         }
```

Answer :

```
{"reviewComment" : "Curious question: in cases like this, would a new address for `V` need to
be created each iteration within this for-loop? Are there advantages of declaring the scope of
the variable as `const double V` within the for-loop rather than with `double V` outside the
for-loop?"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, generate one review comment. Although there could be multiple issues with the patch of code, only give the most important suggestion. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -461,8 +461,14 @@ static EsObject *compile (EsObject *expr, void *engine)
     dsl_throw (UNBOUND_VARIABLE, head);
    if (pb->macro)
    {
-   /* TODO: compile recursively */
-   return pb->macro (expr);
+   EsObject *tail = compile (es_cdr (expr), engine);
+   if (es_error_p (tail))
+    return tail;
+   expr = es_cons (head, tail);
+   EsObject *r = pb->macro (expr);
+   es_object_unref (expr);
+   es_object_unref (tail);
+   return r;
    }
   }
   return es_map (compile, expr, engine);
```

Figure 7: Sample Prompt 1 for Review Comment Generation

PROMPT:

```
Role:

You are a lead software engineer performing code reviews.
```

```
Example Patch:

@@ -15,7 +15,14 @@ namespace Thelia\Coupon;
 use Symfony\Component\DependencyInjection\ContainerInterface;
 use Thelia\Condition\Implementation\ConditionInterface;
 use Thelia\Coupon\Type\CouponInterface;
+use Thelia\Log\Tlog;
+use Thelia\Model\AddressQuery;
+use Thelia\Model\Base\CouponModule;
 use Thelia\Model\Coupon;
+use Thelia\Model\CouponCountry;
+use Thelia\Model\CouponCustomerCount;
+use Thelia\Model\CouponCustomerCountQuery;
+use Thelia\Model\Order;

 /**
  * Manage how Coupons could interact with a Checkout

Answer :
{"reviewComment" : "Be careful, base model is imported here."}
```

```
You are a lead software engineer performing code reviews.

For the below patch of code, generate one review comment. Although there could be multiple
issues with the patch of code, only give the most important suggestion. Example patches of code
and expected answers are given above. Use these examples as reference context and generate
response in the same format as the answers. The response should only be in json format.
```

```
Patch:

@@ -3,8 +3,27 @@
 namespace Thelia\Model;

 use Thelia\Model\Base\OrderCoupon as BaseOrderCoupon;
+use Thelia\Model\Base\OrderCouponCountryQuery;
 class OrderCoupon extends BaseOrderCoupon
 {
+    /**
+     * Return the countries for which free shipping is valid
+     * @return array|mixed|\Propel\Runtime\Collection\ObjectCollection
+     */
+    public function getFreeShippingForCountries()
+    {
+        return OrderCouponCountryQuery::create()->filterByOrderCoupon($this)->find();
+    }
+
+    /**
+     * Return the modules for which free shipping is valid
+     *
+     * @return array|mixed|\Propel\Runtime\Collection\ObjectCollection
+     */
+    public function getFreeShippingForModules()
+    {
+        return OrderCouponModuleQuery::create()->filterByOrderCoupon($this)->find();
+    }
 }
```

Figure 8: Sample Prompt 2 for Review Comment Generation

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -40,11 +40,11 @@ TEST_F(YieldTest, Basic) {
     }
     {
         cpp2::ExecutionResponse resp;
-        std::string query = "YIELD 1+1";
+        std::string query = "YIELD 1+1, '1+1', (int)3.14, (string)(1+1), (string)true";
         auto code = client->execute(query, resp);
         ASSERT_EQ(cpp2::ErrorCode::SUCCEEDED, code);
-        std::vector<std::tuple<int64_t>> expected{
-            {2}
+        std::vector<std::tuple<int64_t, std::string, int64_t, std::string, std::string>>
expected{
+            {2, "1+1", 3, "2", "true"}
         };
         ASSERT_TRUE(verifyResult(resp, expected));
     }
```

Answer :

```
{"reviewComment" : "1+1 < 3"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, generate one review comment. Although there could be multiple issues with the patch of code, only give the most important suggestion. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -125,11 +125,13 @@ TEST_F(IndexTest, TagIndex) {
         std::string query = "SHOW TAG INDEX STATUS";
         auto code = client->execute(query, resp);
         ASSERT_EQ(cpp2::ErrorCode::SUCCEEDED, code);
-        std::vector<uniform_tuple_t<std::string, 2>> expected{
-            {"single_person_index", "SUCCEEDED"},
-            {"multi_person_index",  "SUCCEEDED"},
-        };
-        ASSERT_TRUE(verifyResult(resp, expected));
+        /*
+         * Currently , expected the index status is "RUNNING" or "SUCCEEDED"
+         */
+        for (auto& row : *resp.get_rows()) {
+            const auto &columns = row.get_columns();
+            ASSERT_NE("FAILED", columns[1].get_str());
+        }
     }
     {
         cpp2::ExecutionResponse resp;
```

Figure 9: Sample Prompt 3 for Review Comment Generation

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -36,7 +36,7 @@ module Bolt
       def validate_resolve_reference(opts)
         # TODO: Remove deprecation warning
         if opts.include?('encrypted-value')
-          raise Bolt::ValidationError, "The 'encrypted-value' key is deprecated migrate to to
'encrypted_value'"
+          raise Bolt::ValidationError, "Inventory file parsing error: The 'encrypted-value'
key in the inventory file is deprecated and can no longer be used. Please change the name of
the key to 'encrypted_value' instead, and all will be fine."
         end
         decode(opts['encrypted_value'])
       end
```

Answer :

```
{"reviewComment" : "It is not really an inventory error. Would `Plugin error` help make this
easier to understand?"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, generate one review comment. Although there could be multiple issues with the patch of code, only give the most important suggestion. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -117,6 +117,19 @@ module Faker
         regexify(bothify(fetch(key)))
       end

+      def mercosur_license_plate(legacy_state_abreviation = NOT_GIVEN, state_abreviation: '')
+        key = 'vehicle.mercosur_license_plate'
+        if legacy_state_abreviation != NOT_GIVEN
+          warn_with_uplevel "Passing `state_abreviation` with the 1st argument of
`Vehicle.mercosur_license_plate` is deprecated. Use keyword argument like
`Vehicle.mercosur_license_plate(state_abreviation: ...)` instead.", uplevel: 1
+          state_abreviation = legacy_state_abreviation
+        end
+
+        return regexify(bothify(fetch(key))) if state_abreviation.empty?
+
+        key = key + '.by_state.' + state_abreviation
+        regexify(bothify(fetch(key)))
+      end
+
       def singapore_license_plate
         key = 'vehicle.license_plate'
         plate_number = regexify(bothify(fetch(key)))
```

Figure 10: Sample Prompt 4 for Review Comment Generation

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -171,7 +171,7 @@ class GoGenerator : public BaseGenerator {
     code += ")\n\n";
   }

-  // Begin enum name code.
+  // Begin enum name map.
   void BeginEnumNames(const EnumDef &enum_def, std::string *code_ptr) {
     std::string &code = *code_ptr;
     code += "var EnumNames";
```

Answer :

```
{"reviewNeeded" : "false"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, answer whether it needs review or not. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -461,8 +461,14 @@ static EsObject *compile (EsObject *expr, void *engine)
     dsl_throw (UNBOUND_VARIABLE, head);
   if (pb->macro)
   {
-   /* TODO: compile recursively */
-   return pb->macro (expr);
+   EsObject *tail = compile (es_cdr (expr), engine);
+   if (es_error_p (tail))
+    return tail;
+   expr = es_cons (head, tail);
+   EsObject *r = pb->macro (expr);
+   es_object_unref (expr);
+   es_object_unref (tail);
+   return r;
   }
  }
  return es_map (compile, expr, engine);
```

Figure 11: Sample Prompt 1 for Review Comment Necessity Prediction

PROMPT:

```
Role:

You are a lead software engineer performing code reviews.
```

```
Example Patch:
@@ -25,12 +25,16 @@ use Thelia\Core\Security\SecurityContext;
 use Thelia\Exception\TheliaProcessException;
 use Thelia\Mailer\MailerFactory;
 use Thelia\Model\AddressQuery;
+use Thelia\Model\Base\ProductDocumentQuery;
 use Thelia\Model\Cart as CartModel;
 use Thelia\Model\ConfigQuery;
 use Thelia\Model\Currency as CurrencyModel;
 use Thelia\Model\Customer as CustomerModel;
 use Thelia\Model\Lang as LangModel;
 use Thelia\Model\Map\OrderTableMap;
+use Thelia\Model\MessageQuery;
+use Thelia\Model\MetaData as MetaDataModel;
+use Thelia\Model\MetaDataQuery;
 use Thelia\Model\Order as ModelOrder;
 use Thelia\Model\OrderAddress;
 use Thelia\Model\OrderProduct;

Answer :
{"reviewNeeded" : "true"}
```

```
You are a lead software engineer performing code reviews.

For the below patch of code, answer whether it needs review or not. Example patches of code and
expected answers are given above. Use these examples as reference context and generate response
in the same format as the answers. The response should only be in json format.
```

```
Patch:
@@ -3,8 +3,27 @@
 namespace Thelia\Model;

 use Thelia\Model\Base\OrderCoupon as BaseOrderCoupon;
+use Thelia\Model\Base\OrderCouponCountryQuery;
 class OrderCoupon extends BaseOrderCoupon
 {
+    /**
+     * Return the countries for which free shipping is valid
+     * @return array|mixed|\Propel\Runtime\Collection\ObjectCollection
+     */
+    public function getFreeShippingForCountries()
+    {
+        return OrderCouponCountryQuery::create()->filterByOrderCoupon($this)->find();
+    }
+
+    /**
+     * Return the modules for which free shipping is valid
+     *
+     * @return array|mixed|\Propel\Runtime\Collection\ObjectCollection
+     */
+    public function getFreeShippingForModules()
+    {
+        return OrderCouponModuleQuery::create()->filterByOrderCoupon($this)->find();
+    }
 }
```

Figure 12: Sample Prompt 2 for Review Comment Necessity Prediction

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -23,7 +23,7 @@ class LocalTest extends TestCase
         $local = new Local('/tmp');

         // check if OS is Mac OS X where /tmp is a symlink to /private/tmp
-        $result = 'Darwin' === php_uname('s') ? '/private/tmp' : '/tmp';
+        $result = 'Darwin' === PHP_OS ? '/private/tmp' : '/tmp';

         $this->assertSame($result, $local->getDirectory());
     }
```

Answer :

```
{"reviewNeeded" : "true"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, answer whether it needs review or not. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -356,7 +356,6 @@ class SmartyParser extends Smarty implements ParserInterface
         }

         return $this->internalRenderer('file', $realTemplateName, $parameters);
-
     }

     private function checkTemplate($fileName)
```

Figure 13: Sample Prompt 3 for Review Comment Necessity Prediction

PROMPT:

Role:

You are a lead software engineer performing code reviews.

Example Patch:

```
@@ -118,16 +118,13 @@ namespace RDKit{
       }
     }

-    void iterateCIPRanks(const ROMol &mol, DOUBLE_VECT &invars, UINT_VECT &ranks,bool
seedWithInvars){
+    void iterateCIPRanks(const ROMol &mol, DOUBLE_VECT &invars,
+                         UINT_VECT &ranks,bool seedWithInvars){
       PRECONDITION(invars.size()==mol.getNumAtoms(),"bad invars size");
       PRECONDITION(ranks.size()>=mol.getNumAtoms(),"bad ranks size");

       unsigned int numAtoms = mol.getNumAtoms();
       CIP_ENTRY_VECT cipEntries(numAtoms);
-      INT_LIST allIndices;
-      for(unsigned int i=0;i<numAtoms;++i){
-        allIndices.push_back(i);
-      }
 #ifdef VERBOSE_CANON
       BOOST_LOG(rdDebugLog) << "invariants:" << std::endl;
       for(unsigned int i=0;i<numAtoms;i++){
```

Answer :

```
{"reviewNeeded" : "true"}
```

You are a lead software engineer performing code reviews.

For the below patch of code, answer whether it needs review or not. Example patches of code and expected answers are given above. Use these examples as reference context and generate response in the same format as the answers. The response should only be in json format.

Patch:

```
@@ -308,8 +308,12 @@ std::string PrimaryExpression::toString() const {
         case VAR_INT64:
             snprintf(buf, sizeof(buf), "%ld", boost::get<int64_t>(operand_));
             break;
-        case VAR_DOUBLE:
-            return std::to_string(boost::get<double>(operand_));
+        case VAR_DOUBLE: {
+            int digits10 = std::numeric_limits<double>::digits10;
+            const char *fmt = folly::sformat("%.{}lf", digits10).c_str();
+            snprintf(buf, sizeof(buf), fmt, boost::get<double>(operand_));
+            break;
+        }
         case VAR_BOOL:
             snprintf(buf, sizeof(buf), "%s", boost::get<bool>(operand_) ? "true" : "false");
             break;
```

Figure 14: Sample Prompt 4 for Review Comment Necessity Prediction