Code Reviews on a Budget: Memory-Efficient Fine-Tuning with QLoRA and RAG for Big Code Applications

Sumukh Naveen Aradhya$^{1[0009-0004-4003-0420]}$, Melody Moh$^{1[0000-0002-8313-6645]}$, and Teng-Sheng Moh$^{1[0000-0002-2726-102X]}$

Department of Computer Science, San Jose State University, San Jose, CA, USA
`{sumukhnaveen.aradhya, melody.moh, teng.moh}@sjsu.edu`

Abstract. In this technological era, where Artificial Intelligence and Machine Learning are revolutionizing various domains, Large Language Models (LLMs) are surfacing as a powerful tool for managing and analyzing large-scale data, including software codebases. In the process of software development, having reliable code reviews is highly essential to ensure code security, maintain quality, and manage huge code repositories. This paper aims to survey numerous existing methodologies that aid in creating efficient code review automation agents and investigate the suitability of various existing methods for fine-tuning open-source models in the context of code review automation. Parameter-efficient fine-tuning (PEFT) methodologies, such as Low-Rank Adaptation (LoRA) and Quantized Low-Rank Adaptation (QLoRA), have been explored, with an additional focus on a hybrid model that combines QLoRA with Retrieval Augmented Generation (RAG) to determine efficient ways to reduce the amount of memory required for fine-tuning while ensuring inference quality is not affected. The context has been used from general-purpose LLMs, specifically Meta's Llama 3.2 3B model. Experiments show that the hybrid approach reduces memory utilization by nearly 17% while achieving low entropy values. The results also show improved performance over baseline systems in both efficiency and inference stability, highlighting the potential of this hybrid technique for real-world code review automation.

Keywords: Code Review Automation · Large Language Models · LoRA · QLoRA · QDyLoRA · Retrieval Augmented Generation

1 Introduction

The software development life cycle in any company has always been dynamic, where maintaining code quality is as important as developing innovative solutions. Among the most labor-intensive processes is code review. This is the process of analyzing and assessing a piece of code. It is a practice that aims to improve software quality, prevent bugs, and foster knowledge sharing among developers. However, with increasingly complex software being developed in a fast-paced, data-intensive environment with rapidly shrinking development cycle timelines, traditional code review practices do not scale well because they rely heavily on manual scrutiny.

The main tools for such developments are LLMs, which have shown great performance in understanding large codebases (data) and generating code. These models, when fine-tuned with domain-specific knowledge, provide reliable opportunities for automating complex tasks such as code review. This research builds on studies such as LLaMA-Reviewer, the work by Tufano et al. [1] and the latest advances in Retrieval Augmented Generation (RAG) [7] models by exploring an integration with Quantized Low-Rank Adaptation (QLoRA) fine-tuning [13]. This research tries to bridge the gap between manual expertise and machine efficiency in leveraging LLM to replicate reviewers' insights by contextually assessing code in large and complex codebases.

Automation is not just the goal of this work. The purpose of this paper is to explore the relevance of state-of-the-art fine-tuning methodologies for the code review automation domain by trying to find ways to reduce memory utilization during fine-tuning process while maintaining inference quality. This is to address the challenges posed by current big data-scale software projects.

2 Related Work

This section presents an overview of existing work in automating code review tasks.

Tufano et al. [1] researched transformer-based models trained on Java datasets to automate large code snippets. These models reported having moderate accuracy and challenges like noisy comments and domain restriction. Later research [9] showed strong performance in smaller code snippets but difficulty in handling complex / large multiple components. This necessitated the need for better evaluation metrics and cleaner data. Another study [8], proposed AI agents that learn from bug reports to find inefficiencies, bugs, and style issues. These agents, unlike static analysis tools, proactively optimize code and offer actionable developer feedback, making them a promising enhancement in LLM-based development. Building on this, [10] explored a pre-trained T5 models fine-tuned on 168k Java code-review triplets. This T5 model demonstrated an improvement 93% over older methods, but still failed to meet the readiness for deployment, raising the question whether pre-trained LLMs can generalize effectively in practical large-scale code review tasks. Trans-Review, CodeBERT and CodeT5 [14] were evaluated and CodeT5 outperformed others. A new metric, Edit Progress, was proposed to address the strictness of exact match scores. The findings suggest CodeT5's potential as a strong baseline for review comment generation. Samsung showed real-world deployment [12], providing a reliable Code Review Bot that utilized static analysis and typo-checking functionality. It handled 80k+ reviews and achieved high defect fix rates, showcasing the scalability of automation in heterogeneous, enterprise-sized environments.

After the 2024 CrowdStrike breach, the review of code security with LLMs has gained a lot of traction. The work in [3] has compared the variants GPT-4 and LLaMA between prompt types using a 534-file dataset and reported that GPT-4 performed best with CWE-enriched prompts, though the results varied with large-scale data. Big-data related limitations included verbosity, hallucination, and contextual gaps. In IRIS, a neuro-symbolic approach was introduced [5] by combining static analysis with GPT-4 to infer taint specs and examine false positives by 80%. It outperformed CWE-Bench-Java, showing that hybrid models are capable of overcoming Big Data limitations of symbolic tools [5]. Another paper [6] compared zero-shot and chain-of-thought prompt-based models where GPT-4 based models achieved 95.6% accuracy in vulnerability detection, highlighting the value of prompting as a way to attain high-performance security review in large-scale data domain.

Some of the main papers for this research include [2] where LLaMA was fine-tuned using LoRA and prefix-tuning with <1% trainable parameters. This hybrid parameter efficient fine tuning (PEFT) approach showed competitive performance using limited resources. The research builds on this by integrating QLoRA [13], a memory efficient variant that uses 4-bit quantization and LoRA, allowing large-scale fine-tuning on a single 48GB GPU. To also address hallucination and improve context, Retrieval Augmented Generation (RAG) [7] has been explored, which combines parametric transformers with dense retrievers. The RAG Token and RAG Sequence outperform traditional models in fact-rich generation [7]. With nonparametric memory and dynamic knowledge injection, RAG offers scalability and adaptability for Big Data-scale review tasks. This paper aims to combine RAG with QLoRA for a context-aware, efficient, and scalable large-scale corporate technology code review framework in the upcoming sections.

3 Methodology

In this section, methodology followed to analyze different approaches for fine-tuning an LLM in the code review automation domain is explored. First, existing fine-tuning methodologies and common patterns are identified. Next, an explanation on the focus on LoRA and QLoRA, despite availability of other fine-tuning methods is elaborated. The section then describes data used for experiments - understand their strengths and their weakness to be able to better judge the quality of the results. The training infrastructure has also been explained. Finally, the section discusses individual approaches and introduces a hybrid methodology.

3.1 LoRA, QLoRA, QLoRA+RAG - Surely there's something common here!

All three approaches LoRA, QLoRA, and QLoRA+RAG share the same purpose of enabling efficient, scalable fine-tuning of Large Language Models, particularly for Big Data and low-resource environments. All approaches utilize

Parameter-Efficient Fine-Tuning (PEFT), which avoids updating the full model by introducing trainable low-rank matrices into transformer attention layers. During training, only these small matrices are updated, significantly lowering memory, computation, and training time. LoRA uses 16/32-bit base weights, while QLoRA uses more efficient 4-bit quantization. The joint QLoRA+RAG approach maintains quantized fine-tuning and enhances it with context-aware retrieval, which makes it most suitable for high-volume and dynamic code review tasks observed in Big Data-based software development.

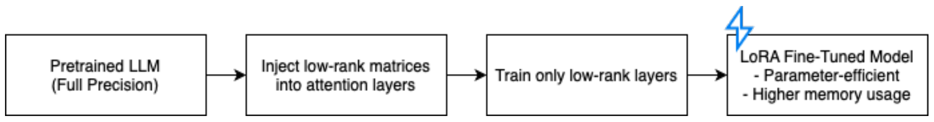
Pretrained LLM (Full Precision)
Inject low-rank matrices into attention layers
Train only low-rank layers
LoRA Fine-Tuned Model - Parameter-efficient - Higher memory usage

Fig. 1. LoRA Fine-Tuning Flow

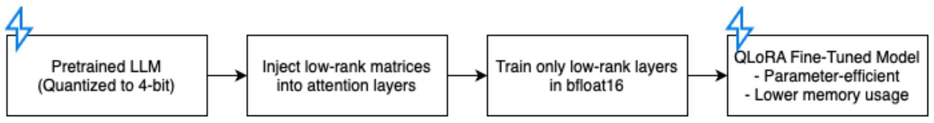
Pretrained LLM (Quantized to 4-bit)
Inject low-rank matrices into attention layers
Train only low-rank layers in bfloat16
QLoRA Fine-Tuned Model - Parameter-efficient - Lower memory usage

Fig. 2. QLoRA Fine-Tuning Flow

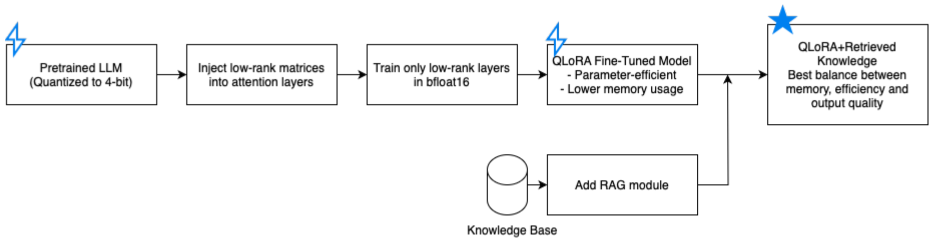
Pretrained LLM (Quantized to 4-bit)
Inject low-rank matrices into attention layers
Train only low-rank layers in bfloat16
QLoRA Fine-Tuned Model - Parameter-efficient - Lower memory usage
QLoRA+Retrieved Knowledge Best balance between memory, efficiency and output quality
Add RAG module
Knowledge Base

Fig. 3. QLoRA+RAG Fine-Tuning Flow

The diagrams above show the evolution of LoRA to QLoRA and finally QLoRA+RAG in terms of model efficiency, training precision and output quality. Figure 1 shows LoRA fine-tuning based traditional full-precision pretrained LLM where only the injected low-rank matrices are trained. This is more efficient than traditional fine-tuning but can be improved further. Figure 2 shows the improved quantized version of this where the pretrained model is quantized to 4-bit and low-rank matrices are trained in bfloat16, which makes it more memory-efficient.

Finally, Figure 3 enhances QLoRA further by integrating RAG module to allow for context to be fed during inference for domain-specific consistent output.

3.2 Justification for focusing on LoRA and QLoRA

Full fine-tuning of LLMs like LLaMA 3.2 is very memory and compute-intensive and therefore not very feasible in most real-world data-intensive usecases. This work focuses on Parameter-Efficient Fine-Tuning (PEFT) approaches that brings a balance between model output quality and memory usage. Specifically, Low-Rank Adaptation (LoRA) and Quantized Low-Rank Adaptation (QLoRA) are compared. LoRA brings in a small number of trainable parameters through low-rank adapters, and QLoRA further enhances this by using 4-bit quantization for the base model—minimizing memory usage without hurting performance. These approaches are aligned with the overall goal of making fine-tuning memory-efficient. The choice of LoRA and QLoRA is due to:

- **Strong cost-quality tradeoff:** In comparison to alternatives such as Adapter-Fusion or Prefix Tuning, LoRA and QLoRA are more memory efficient with higher task-specific accuracy on difficult domains
- **Open-source compatibility:** They are both supported natively by Hugging Face's PEFT and Transformers libraries making it easily useable

3.3 Dataset and setup

First, open-source benchmark dataset was obtained for code review automation tasks from Tufano et al [1] [9] [10]. A combination of datasets have been used in the research to make sure the training is more robust and includes context from many sources. One part of the dataset contains src2-train which contains abstracted code, src1-train containing the natural language review on the code and tgt containing code that has the natural language review implemented [1]. Considering the use case only requires to take code as input and produce natural language review as an output, only the combination of src1-train and src2-train was considered. Along with this, a dataset from [10] was also used. This has both code and natural language review as two columns in the same dataset - train.tsv.

3.4 Data pre-processing

After having created a combined dataset, the data was preprocessed. This process started with train.tsv, which directly maps code snippets to natural language reviews and appended it by integrating data from src2-train.txt (containing original code snippets), and src1-train.txt(containing corresponding reviews) into one unified dataset. This merged dataset, train_master.csv [16], provides a complete and robust training corpus with two columns - one for code samples and the other for corresponding natural language reviews. This combined dataset now guarantees better diversity and coverage, hence laying the foundation for training the open-source large language model in the generation of high-quality, structured code reviews. Pre-processing steps includes: (a) Tokenizing both the code and the reviews with truncation and padding to ensure uniform sequence lengths across all inputs (b) Replacing padding tokens in the review (target) sequences with `-100` to make sure they are ignored during loss computation, as required by most transformer-based training objectives (c) Transforming the final dataset into the Hugging Face `Dataset` format, for easy integration with the fine-tuning pipeline and compatibility with the `Trainer` API

Dataset strengths

– Dataset contains real-world code samples extracted directly from Github repositories
– The `Review` column provides contextual and specific feedback for each snippet, making annotations actionable and useful
– Reviews are diverse as it provides good naming convention, detailed feedback on logic and functionality, details on the code structure, code maintainability and also provides potential improvements or information on unresolved decisions in the code snippet

Dataset weaknesses

– The reviews primarily offer high-level suggestions and often lack in-depth analysis
– There is potential bias in the reviews due to human subjectivity, and some classes such as security vulnerabilities may be underrepresented
– The dataset does not include any internal scoring system to evaluate the quality of relevance of a review
– Some outliers exist in the dataset, such as vague or non-informative reviews (e.g., *"Shall we revisit?"*), which may introduce noise during training

3.5 Choosing a Large Language Model

The choice of a suitable base Large Language Model (LLM) is essential in determining the efficiency, scalability, and final performance of the downstream task. The research also evaluated a number of different closed-source and open-source LLMs based on their architecture, availability, and performance in code review-based tasks.

Other LLMs Considered

– **CodeGen (Code-specific model):** This model is pre-trained on very large code corpora and is heavily specialized towards software tasks. While this offers a good zero-shot performance in code understanding, it is also larger in terms of size (ranging from 6B to 15B parameters). This means that it would be harder to specialize it for specific styles or domains without extensive retraining

- **LLaMA 2 (7B, 13B) and LLaMA 3 (7B+):** Meta's LLaMA family is a popular choice of model because of its proven performance on a diverse set of tasks. As larger variants are explored, performance was seen to improve, particularly on reasoning-intensive tasks, but at the cost of much higher memory and compute requirements. This rendered them less suitable for fine-tuning on typical cloud GPU setup like this
- **Closed-source State-of-the-Art Models (OpenAI GPT-4o, OpenAI GPT-3.5 Turbo, Claude Haiku, Claude Sonnet):** The paper thouroughly examined the performance of several closed-source LLMs to include benchmarks for inference quality on code review tasks. These models could produce high-quality and context-sensitive feedback. But the research opted not to build on them due to their closed-source nature, which entails limited transparency, controllability, as well as usage restrictions especially when it comes to fine-tuning and offline deployment.

Final Model Choice: LLaMA 3.2 3B To explore the possibility of using lightweight LLMs for automating code review in big code domain, Meta's LLaMA 3.2 3B model was selected as the foundation for all the experiments. The reasons are as follows:

- **Parameter Efficiency:** LLaMA 3.2 3B has a mere 3 billion parameters, making it one of the smallest in the LLaMA 3.2 model family. Experimenting with this model allows to witness how small models can be made to perform well through efficient fine-tuning
- **Open-Source and Fully Controllable:** LLaMA 3.2 3B is open-source, license-friendly, and fully controllable, including fine-tuning, quantization, and offline deployment—something that cannot be achieved with closed-source LLMs
- **Challenge-Driven Research Motivation:** The paper intentionally chose a smaller model size to evaluate and find out whether state-of-the-art approaches like QLoRA and RAG can break through model size constraints. The target was also to demonstrate that even the lowest-parameter models can be competitive in some domains when paired with the right optimization strategies.

3.6 Training Infrastructure

- **Platform:** RunPod
- **GPU:** 1xH100 SXM (80GB VRAM). While these experiments can run on personal computers, the training time becomes long, often exceeding practical limits for experimentation. This research therefore used a high-performance GPU (H100) to conduct evaluations more efficiently. This also highlights an important trade-off that users must consider: accept longer training times on low-resource devices or invest in higher GPU costs to enable faster iteration.
- **RAM:** 287GB
- **Environment:** PyTorch 2.1.0, CUDA 11.8.0, Ubuntu 22.04

3.7 Fine-tuning the chosen Large Language Model

Several comprehensive experiments were conducted on two parameter-efficient fine-tuning methods:

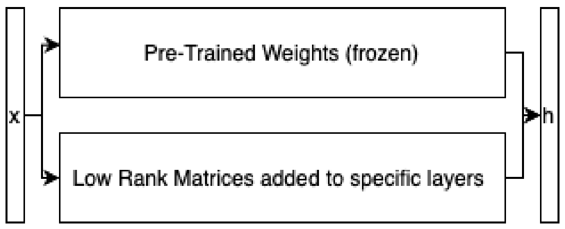
Pre-Trained Weights (frozen)
x
h
Low Rank Matrices added to specific layers

Fig. 4. General LoRA architecture

LoRA Low-Rank Adaptation (LoRA) is a fine-tuning approach that uses certain number of parameters to adapt large pre-trained language models to downstream tasks without having to update the entire set of model parameters as shown in Figure 4. Instead of updating all the weights of the model, LoRA uses low-rank adapters that are small learnable matrices in certain components of the model, usually in the attention mechanism (e.g., the query and value projection layers). The adapters estimate weight updates in a more memory-efficient manner. The core idea behind LoRA lies in the idea that the update needed to map a pre-trained model to perform a new task usually resides within a low-dimensional subspace. Hence, instead of learning the full-rank weight matrices, LoRA keeps the weights fixed and learns only the low-rank adapters, resulting in a significant reduction in the number of trainable parameters.

In the LoRA evaluation phase, an experiment was conducted to evaluate the use of keying in low-rank trainable matrices into the attention layers of the Llama 3.2 3B parameter model. These were applied specifically to the `q_proj`, `k_proj`, `v_proj`, and `o_proj` modules.

The configuration parameters used were:

LoRA Rank (r): 16 ; **Alpha:** 32 ; **Dropout:** 0.1 ; **Task Type:** CAUSAL_LM **Bias:** none

This method allowed to freeze the base model weights of the out-of-the-box Llama 3.2 model and update only a small number of parameters introduced by LoRA. As a result, training or fine-tuning this model required lesser GPU memory, fewer compute cycles, and could be conducted on a single H100 GPU without any batching constraints. All monitored results have been discussed in detail in the results and discussion section of this paper.

QLoRA Following LoRA, the experiment extended the understanding on these fine-tuning methods by working with QLoRA to further compress the model and enable even more memory-efficient training. QLoRA is an enhanced version of LoRA and is built on top of LoRA. It adds another layer of optimization by allowing fine-tuning of quantized models. QLoRA allows training LLMs in 4-bit precision with NormalFloat (NF4) quantization, without a loss of accuracy. QLoRA aims to make it possible to fine-tune extremely large models on consumer-grade or resource-constrained hardware.

The innovation of QLoRA lies in:

- 4-bit quantization of the base model to greatly reduce memory footprint
- Double quantization to preserve numerical stability
- LoRA adapters for task-adaptive adaptation
- Paged optimizers to facilitate dynamic memory training

QLoRA has achieved state-of-the-art performance in all domains, and can fine-tune up to 65B parameter models on a single 48 GB GPU, thus being a significant leap in LLM fine-tuning democratization.

The same low-rank adapters (as used in LoRA) were applied on top of this quantized model.

Quantization settings:

- Load in 4-bit: True
- Quant Type: nf4
- Compute Dtype: bfloat16
- Double Quantization: Enabled

LoRA Settings on Quantized Model:

- Rank: 16
- Alpha: 32
- Dropout: 0.1

This training was performed using Hugging Face's `Trainer` class, extended with a custom trainer `MetricsTrainer` to include details of each step in the evaluation. It was found that QLoRA offered an ideal trade-off between the model's memory and yet its inference performance was comparable if not better than LoRA.

3.8 Retrieval Augmented Generation

Retrieval-Augmented Generation (RAG) [7] is a hybrid model that enhances the effectiveness of generative language models by incorporating external, non-parametric sources of knowledge at inference time. Unlike typical LLMs that are limited to relying solely on parametric memory (i.e., the knowledge stored in their weights), RAG models dynamically retrieve pertinent information from external knowledge bases or datasets to supplement their responses. Therefore, they are good with factual accuracy, contextual grounding, or domain-specific tasks such as code review automation.

RAG has two main stages:

1. Retrieval Stage A retriever model — usually a dense vector similarity model that uses sentence embeddings which takes user input (e.g., code snippet or query) and returns the top-k most relevant documents, passages, or code-review cases from an indexed knowledge base

2. Generation Phase The retrieved documents are then passed as additional context to a generative model (e.g., a fine-tuned transformer), which conditions its output based on both the original query and the retrieved documents. This helps the model generate more informed, accurate, and contextually sensitive responses.

3.9 QLoRA with RAG

To reduce hallucination and enhance inference quality, the next goal was to combine Retrieval-Augmented Generation (RAG) [7] with the QLoRA fine-tuned model. The pipeline uses semantic embeddings to get review context that is suitable for a (query) code snippet. This context is appended to the prompt and passed through the QLoRA model, enhancing its knowledge of code style and review expectations. RAG Pipeline:

– **Embedding Model:** all-MiniLM-L6-v2
– **Retriever:** FAISS dense vector index built from Code-Review pairs
– **Augmentation:** Top-K similar Code-Review samples appended to input query
– **Model:** QLoRA fine-tuned LLaMA 3.2 3 Billion parameters

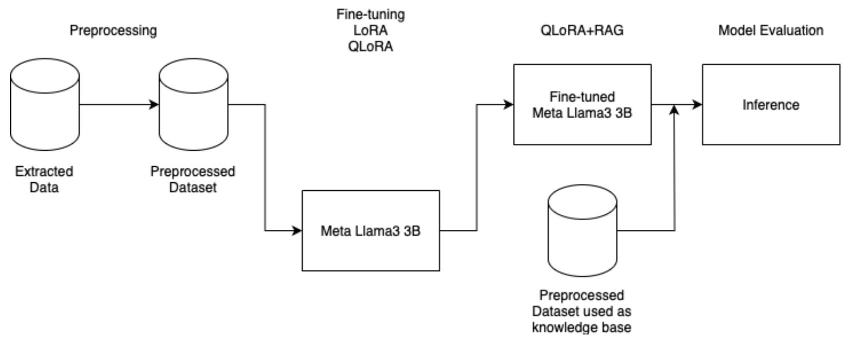

Fig. 5. Overall system architecture combining QLoRA fine-tuning with RAG-based inference

3.10 Evaluation metrics

To compare the efficiency of different fine-tuning methods in code review automation, the research focused on a combination of system-level and inference quality metrics. Training time was recorded to measure the computational cost taken in completing one epoch for each method. CPU and GPU usage during training were also tracked to measure how resource-consuming each method was. In particular, this was necessary to verify the claims of efficiency by QLoRA and QLoRA+RAG over the baseline LoRA. Entropy at inference time was also evaluated to determine how random the outputs were - the lower the entropy the more deterministic and stable the output is. This helped observe the benefit of including Retrieval-Augmented Generation (RAG) as it lowered output entropy by grounding model responses in the proper retrieved context. While traditional NLP performance measures such as F1 Score and BLEU may have been used, this paper focused on performance and stability under memory-constrained conditions along with a focus on resource utilization and inference quality instead of typical token-level accuracy.

4 Results and Discussion

The goal of this research has been focused on trying to find scalable memory-efficient solutions to fine-tuning LLMs for code review automation tasks while making sure inference quality is high even when large codebases are used as an input to the model. In this section, the paper presents the results obtained from several experiments that were conducted. The main goal was to implement both existing and hybrid methods to analyze their performance - training time, memory utilization, and inference quality under resource and infrastructure constraints that are common in Big Data environments. This paper has structured the experiments across three stages - re-implementation of baseline systems from literature, comparative evaluation of both closed and open-source language models, and fine-tuned/hybrid methodology combining QLoRA with Retrieval-Augmented Generation (RAG).

4.1 Implementing T5-based Code Review from [10]

To assess capabilities of large language models in the code review automation domain, the first implemention was the approach from paper [10]. This uses a fine-tuned T5 model from an open-source repository. All necessary datasets have been uploaded to Zenodo for reproducibility. This served as a baseline to evaluate the approach. The model was initially fine-tuned for three tasks - code-to-code transformation, code-to-comment generation, and code-comment-to-code refinement. The focus was on using the fine-tuned model's code-to-comment module for testing its performance on custom input samples. The very first installation required a significant amount of tweaking to get it to work locally. Once dependencies were downloaded, the fine-tuned T5 model checkpoint in TensorFlow format was converted to PyTorch. Configuration files, model weights, and tokenizer files were fetched from Zenodo, formatted, and organized so that the transformers library requires. Several issues were addressed concerning the file paths and unsupported types such as .safetensors by appropriately rearranging the directories and setting up the dependencies needed for it. For the purpose of inference, a custom dataset text.csv was created containing badly written snippets of Python code within the source columns. The expected review comments in the target column was intentionally left blank to test the model's inference capabilities. The generate_predictions.py script was also altered to feed in the pre-processed input data. The generated predictions were manually analyzed in terms of relevance and quality in the code review automation domain.

4.2 Evaluating different closed and open-source models

Next, several closed and open-source LLMs were evaluated for their effectiveness in automating code reviews. Each model was given a sample of flawed code along with two prompts: (a) a general request for a formatted code review, and (b) a detailed checklist including code style, error detection, structure, security, task completion, and technical debt. These prompts were tested across four closed-source models namely OpenAI's ChatGPT 4, GPT-4o, GPT-4o-mini, and Anthropic's Claude Haiku—and two open-source models Meta's LLaMA 3 8B and LLaMA3.2 3B. The outputs varied in depth and quality, with Claude subjectively producing the most comprehensive reviews under identical conditions.

4.3 Implementing the Dual-Encoder Transformer from [1]

In the next experiment, approach from [1] was replicated. Here, a fine-tuned 2-encoder transformer model using OpenNMT for generating code reviews was evaluated. Two models were tested - 1-encoder which took a pair of abstracted code versions, and 2-encoder transformer which used a triplet consisting of abstracted code, natural language review of the code and the code that has the review implemented. The experiment trained the model on a MacBook Pro for 100,000 steps (as per paper), completing in about 4 hours. The model was trained successfully, but as for the inference results, that was not the case. A considerable amount of work was put into adjusting inputs, verifying dataset consistency, and refining inference configurations. While the outcome wasn't as predicted, the experiment exposed key issues in the process of designing LLMs for the purposes of code review within high-usable, task-centric areas. Such findings show the importance of using large datasets and task-directed models, showing the merit of using retrieval-based systems for large-scale, Big Data-based code review automation tasks.

4.4 Custom Fine-Tuning with LoRA, QLoRA, and QLoRA+RAG

The main contribution began when the first implemention of LoRA was run on a MacBook Pro (16GB RAM, M2 Pro). This ran for more than 28 hours and still was not able to reach completion. Considering the underwhelming performance in terms of memory usage and time taken for training, an upgrade to an L40S instance (16 vCPUs, 62GB RAM) on RunPod was made. After observing that similar scalability issues persisted, all experiments were run on a RunPod H100SXM (27 vCPU, 287GB RAM) GPU on RunPod, which enabled stable execution of LoRA, QLoRA, and QLoRA+RAG experiments.

From the results, observations were that:

– **LoRA** consumed the most memory and had the longest train time, though it gave valuable benchmark
– **QLoRA** significantly reduced memory usage and training time by applying the 4-bit quantization, though its inference results varied across runs
– **QLoRA+RAG** was the best solution that showed consistent inference (low entropy - reduced randomness), low memory utilization, and low training time. The integration of RAG allowed the model to ground its responses well which boosted the output relevance

Table 1. CPU and GPU Utilization Across Fine-Tuning Methods

	Run 1		**Run 2**		**Run 3**	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
LoRA	12%	100%	11.2%	100%	11.5%	100%
QLoRA	10%	85%	10%	87.7%	10.1%	87%
Reduction	**16.67%**	**15%**	10.71%	12.3%	12.17%	13%

The results in Table 1 clearly shows the reduction in memory utilization achieved by utilizing QLoRA fine-tuning on the code review dataset. A best reduction of 16.67% was obtained, as opposed to its predecessor LoRA. These results when cumulated over thousands of epochs in a real training scenario would help save significant amounts of resources.

Table 2. Training Configuration and Time Comparison

Method	**Samples**	**Batching Strategy**	**Time (s)**
LoRA	147,981	Full set at 8 rows/batch = 18,498	8476.30
QLoRA	147,981	90% train set at 64 rows/batch = 2,081	11248.29

Table 2 provides details on the training time taken by different fine-tuning methods under consideration. This experiment utilized a full set of 147,981 samples of the code review dataset [16] for LoRA and used 90% of the set for QLoRA. There is difference in time taken to complete one epoch of fine-tuning with QLoRA taking several minutes longer than LoRA. Though the training time is longer, the amount of memory utilization saved acts as a reasonable trade-off to consider QLoRA to be generally a better choice.

Table 3. Entropy Comparison (Inference Randomness)

Method	Entropy
LoRA	3.570
QLoRA	3.477
QLoRA+RAG	2.943

The entropy values obtained from the runs reflect the model's ability to generate more deterministic and grounded outputs when paired with the RAG mechanism. This achieves what was initially set out to get out of the research - to figure out efficient ways to get good (deterministic) inference by making sure memory utilization is as low as possible.

5 Conclusion

This paper demonstrates how memory-constrained fine-tuning methods - specifically LoRA and QLoRA, can be effectively applied in fine-tuning open-sourced large-scale language models like Meta's LLaMA 3.2 3B for automated code review generation. By utilizing this approach, capable LLMs were trained using much fewer resources, which makes it feasible and handy in a lot of real-world scenarios. This is highly relevant in Big Data and cloud environments, where infrastructure cost and efficiency are key constraints. QLoRA also demonstrated its capabilities by utilizing 4-bit quantization and low-rank adapters, reducing memory usage without compromising the quality of outputs. Further refinement was achieved by incorporating Retrieval-Augmented Generation (RAG) to enable context-aware inference and to promote generated review consistency and relevance as measured by improvements in training time, GPU utilization and entropy metrics. One of the possible future areas of improvement of this research involves inclusion of a human-in-loop feedback system to further understand the quality of model inference.

The findings indicate that even small LLMs, when supplemented with smart fine-tuning and retrieval capabilities, can be as good as, if not better than, larger, hardware-hungry models. Though there are modest performance gains, this research provides an analysis of trade-offs that show that inference quality can be preserved while significantly reducing GPU memory usage. This highlights the importance of infrastructure-aware AI systems that can balance performance and scalability in real-time data-intensive software systems. This only adds to the argument for lean, open-source, yet controllable AI systems for critical applications like software quality assurance. With strong performance, scalability, and reduced hardware demands, the approach sets out a blueprint for developing AI tools that are both efficient and adaptable to Big Data-scale software engineering challenges. In the upcoming research, the authors plan to expand on these results, understanding, evaluating a dynamic QLoRA fine-tuning technique, and complementing it with a contrastive version of RAG in a similar code review automation domain.

References

1. Tufano, Rosalia, et al. "Towards automating code review activities." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.
2. Lu, Junyi, et al. "LLaMA-Reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning." 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2023.
3. Yu, Jiaxin, et al. "Security code review by llms: A deep dive into responses." arXiv preprint arXiv:2401.16310 (2024).
4. Luo, Qinyu, et al. "Repoagent: An llm-powered open-source framework for repository-level code documentation generation." arXiv preprint arXiv:2402.16667 (2024).
5. Li, Ziyang, Saikat Dutta, and Mayur Naik. "LLM-Assisted Static Analysis for Detecting Security Vulnerabilities." arXiv preprint arXiv:2405.17238 (2024).
6. Jensen, Rasmus Ingemann Tuffveson, Vali Tawosi, and Salwa Alamir. "Software vulnerability and functionality assessment using llms." 2024 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE). IEEE, 2024.
7. Lewis, Patrick, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks." Advances in Neural Information Processing Systems 33 (2020): 9459-9474.
8. Rasheed, Zeeshan, et al. "AI-powered Code Review with LLMs: Early Results." arXiv preprint arXiv:2404.18496 (2024).
9. Tufano, Rosalia, et al. "Code review automation: strengths and weaknesses of the state of the art." IEEE Transactions on Software Engineering (2024).
10. Tufano, Rosalia, et al. "Using pre-trained models to boost code review automation." Proceedings of the 44th international conference on software engineering. 2022.
11. Choudhury, Dikshya, and Deepa Gupta. "Investigation on Integration of Machine Learning Techniques into LC/NC Platforms for Code Review, Quality Assessment, and Error Detection Automation." 2024 11th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO). IEEE, 2024.
12. Kim, Hyungjin, et al. "A unified code review automation for large-scale industry with diverse development environments." Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. 2022.
13. Dettmers, Tim, et al. "Qlora: Efficient finetuning of quantized llms." Advances in Neural Information Processing Systems 36 (2024).
14. Zhou, Xin, et al. "Generation-based code review automation: how far are wef." 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC). IEEE, 2023.
15. Rajabzadeh, Hossein, et al. "Qdylora: Quantized dynamic low-rank adaptation for efficient large language model tuning." arXiv preprint arXiv:2402.10462 (2024).
16. https://github.com/sumukh-aradhya/LLM_RAG_code_review_automation