university of groningen

faculty of science and engineering

The effect of four-bit quantization on multi-agent LLM coding performance

Thijs Lukkien

university of groningen

faculty of science and engineering

University of Groningen

The effect of four-bit quantization on multi-agent LLM coding performance

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of

Prof. T. M. Tashu (University of Groningen)
and
Dr. S. D. Scheffer (University of Groningen)

Thijs Lukkien (s3978389)

September 29, 2025

Contents

List of Figures

List of Tables

Abstract

The aim of this research was to investigate the effect of post-training four-bit quantization on collaborative large language model-based multi-agent systems (LLM-MAs). Collaborative LLM-MAs have shown impressive performance in many domains such as gaming operation or coding. However, their computational costs are high, as their iterative collaboration requires multiple prompts. This can result in greater economic cost and environmental impact. Quantization is a technique that maps data, such as model parameters, onto lower-precision formats, thereby compressing the model. This can increase inference speed and reduce VRAM requirements, thereby lowering economical and ecological barriers. However, the effect of quantization on LLM-MAs has not yet been explored. In this thesis, four models are quantized and retrained using QLoRA to regain performance. The four base models and the four quantized versions were then evaluated using the Mostly Basic Python Problems (MBPP) dataset and pass@k metric. It was hypothesized that quantization would reduce collaborative performance, due to its lossy nature and cascading error through iterative collaboration. The findings show that quantization can be employed with a minimal effect on LLM-MA programming performance. However, more research is needed to investigate the significance and implications of these findings.

1 Introduction

Large language models (LLMs) have achieved remarkable performance in recent years, showing human-level performance in a variety of domains. Despite these advancements, LLMs still faced many challenges when used for high-complexity or long-horizon tasks [1].

Such tasks require an LLM to maintain coherence and accuracy, spanning over a large or difficult problem. Many have suggested LLM collaboration as a promising approach to address these problems. There have been several methods of improvement using this idea such as output blending or iterative output improvement [2].

However, the iterative and multi-agent nature of these systems inherently increases the number of LLM prompts. Given that singular non-iterative LLMs are already known to consume large amounts of energy [3, 4], resulting in a substantial carbon footprint and cost of usage. This can be particularly challenging to manage for small organizations or people in developing countries. Since the financial expense and ecological footprint both arise from the computational demands of LLMs, they will be collectively referred to as 'computational costs', or 'costs' throughout this thesis. Beyond accessibility, the high computational costs of LLM-based multi-agent systems (LLM-MAs) may outweigh performance gains, possibly reducing their practical applicability and potential.

Meanwhile, LLM-MAs could have the potential to improve a variety of domains such as healthcare, and programming. As iterative collaboration may increase the quality and scope of any output, LLM-MAs could be used to iteratively produce a comprehensive and accurate diagnosis, or to create entirely independent programming teams.

Therefore, improving LLM-MA efficiency could directly improve its applicability and democratization, while also opening the doors to impact a wide variety of domains.

One method to reduce the cost associated with large language models is quantization: a technique that compresses models by mapping weights and activations to lower-precision numerical formats [5]. This reduces the model storage size and may increase inference speed. Quantizing a model from 16 bits to 4 bits has been found to decrease the energy consumption of some models by 50.7% without substantial performance loss [6, 7]. While the effect of quantization on model performance has been researched, its effect has not yet been investigated with regard to multi-model collaboration performance.

This gap limits the potential efficiency of deployed collaborative LLM-MAs, as well as the overall accessibility of such systems in resource-constrained settings.

This project aims to investigate the effect of four-bit quantization on LLM collaborative functional coding performance. It will provide the first insights into the trade-off between coding performance and efficiency through post-training quantization.

To achieve this, the collaborative coding performance of four models will be compared to their four-bit quantized formats. Using a coding dataset, the base and quantized models will be prompted in natural language for a python function. This will be done in individual and collaborative settings such that the effect of quantization can be clearly understood. The generated python functions will be compiled and evaluated on functional correctness using input-output pairs.

An account of previous literature regarding this subject and further motivation for this research approach will be given in the next section. After that, the technical details will be discussed in Section 3. The results will be presented in Section 4 and discussed in Section 5. Moreover, the LLM-MA issues regarding computational costs will be reflected upon in the final section.

2 Theoretical Framework

2.1 Transformers

A key development that enabled LLMs was the transformer structure [8]. This is a sequence model that uses an encoder-decoder architecture in combination with self-attention and positional encoding. It works as follows: an input sequence, such as a sentence, is converted to a numerical representation called the input embedding. Numerical similarity between these vectors represents similarity in meaning between the input words. This allows the model to learn semantic meaning. Then, the positional encoding is added to each vector. This is a numerical representation of each word's position. In other words: a cosine or sine value is computed with the index of each word, and the resulting vector is added to the embedding vector. This allows the model to learn syntactic meaning. The combined vectors are used as input to the encoder layer.

The encoder layer aims to create a sequence of vectors that capture the semantic and syntactic meaning of the input sentence. The encoder consists of blocks, each consisting of a multi-headed attention layer and feed-forward layer, both with residual connections and layer normalizations. The multi-headed attention layer aims to extract how words correlate to one another throughout the input. It consists of multiple self-attention heads, which are mechanisms that match each word pair's co-dependencies to determine how strongly they influence each other. It then combines this information from each word, to give a representation of contextual importance in the input token. Each self-attention head produces the attention weights for all input vectors, and captures a different dependency between vectors. These dependencies are then returned, so that the overall system can aggregate the meaning of each input section, thereby creating a comprehensive representation of the entire input. Lastly, the feed-forward layer applies non-linear transformations to find more intricate connections within the data.

The contextually-rich and abstract sequence of vectors from the encoder is used as input in the decoder. The decoder takes this sequence and aims to predict the token that fits best after it. It uses a similar structure to the encoder, with some key differences. Firstly, it uses masks within self-attention mechanisms to prevent contextual information from reaching tokens earlier in the sequence. Secondly, the second multi-headed attention block takes the queries from the previous decoder layer and the key and value vectors from the encoder. This allows the decoder to determine which parts of the original input are important to predict the next token throughout the generation process. This is important as the decoder is autoregressive, meaning it will take its own output as input until the sequence is finished.

This system was a major improvement over previous sequential data models such as long short-term memory networks (LSTMs) or recurrent neural networks (RNNs). These systems used recurrence, which suffered from short contextual windows and an inability to parallelize well.

2.2 Large Language Models

The introduction of transformers meant that attention-based language models could be trained in parallel. This greatly increased efficiency, however training still required large amounts of labeled training data. Bi-directional Encoder Representations for Transformers (BERT) [9] is based on the encoder section of the transformer architecture and aimed to solve the issue of requiring large large labeled datasets. This was done through a new and unique training method using self-supervision. Instead of labeled data, this method used parts of a sentence as input, and part as label. More precisely, the prediction tasks included predicting masked words, and predicting which sentences logically followed from a given context sentence. The innovative thought here was that one must understand the meaning from sentences if it is able to correctly predict parts of sentences. Furthermore, BERT used bi-directional attention, instead of the causal masking used in the original transformer, in order to improve context learning.

BERT was a breakthrough especially because it managed to prove that the previously mentioned methods were able to distill substantial amounts of knowledge into network parameters. By using the main model parameters with a specific fine-tuned final layer, also known as transfer learning, BERT was able to improve 11 state-of-the-art benchmark scores. This showcased efficient learning, broad capacities, and a basic 'language foundation' captured in a model.

After BERT, the next large innovation came from generative pre-trained transformer 2 (GPT-2) [10]. This model is based on the decoder architecture from transformers. Similar to the decoder, this means that GPTs are auto-regressive. This means that GPTs have the ability to generate a sequence of outputs of varying lengths. While BERT showed that that language knowledge could be encoded in a model's parameters, GPT-2 showed it could be extracted using only prompts instead of a fine-tuned final layer.

This made the model much more flexible, though the performance was still lacking. This was later improved by GPT-3 [11], which scaled the model to 175 billion (B) parameters. As was first shown by GPT-2, and later confirmed by GPT-3, was that an increase in the number of parameters could result in an increase in model performance. Due to the now-proven importance of model parameter size, this is considered as the paper to initiate the field (or term) of large language models (LLMs).

These models still required prompts to be specific, with performance quickly degrading if prompts were not aligned with the model. In other words; the models performed poorly on user alignment. The solution came in the form of instruction tuning; supervised fine-tuning on natural language task-response pairs. Later reinforcement learning from human feedback (RLHF) [12] became the new method for supervised fine-tuning.

2.3 Multi-agent LLM systems

Many studies have proposed unique approaches to LLM-MA collaboration. A short summarization of different approaches to multi-agent LLM systems is given below.

Firstly, wisdom-of-the-crowd principles that were applicable to MLP collaboration were applied to LLMs, such as ensembles. An example of this is majority voting where multiple models produce an output, of which the most frequent answer is chosen as the final output. This has also been done using LLMs, giving binary output [13], or by blending textual output [14]. This is a method of collaboration through model output.

Secondly, it was shown that the output from one LLM could be used as input for another prompt, improving overall performance. This was done by having one LLM produce a problem analysis and propose a high-level solution. A second LLM could then use that planning as guide to perform better on long-horizon and complex tasks [15]. This has also been done using self-planning, prompting a model to produce a planning which is then used to iteratively prompt the model to complete its self-proposed steps [16].

Collaboration through planning improved performance on long-horizon tasks, but output accuracy at each step could still be lacking. Currently, many papers propose to resolve this issue using iterative improvement. An example of this is self-improvement [17]. Here, an LLM is alternately prompted to improve its output and write test reports for the next iteration. Only once the test report indicated that the code sufficed, was it passed as final output. This technique was shown to improve results on low-level tasks.

Communication between models has been shown to be an important aspect for collaboration. For example, it was found that prompts involving role-playing improved LLM agents' ability to perform their roles [18]. Another study suggested standardized communication structures to prevent models from hallucinating, thereby breaking their roles and the iterative collaboration loop [19].

Lastly, planning, communication structures, and iterative (correction) prompting were combined, resulting in the current state-of-the-art collaborative systems. These systems are often inspired by human teams, combining wisdom-of-the-crowd benefits with planning and collaboration. Examples of this are 'AgentVerse' [20] where a 'manager' LLM agent iteratively recomposes LLM teams. The teams collaborate to follow a step-by-step plan from a planner agent. The planner agent receives feedback in the form of progress on the plan, after which it revises and redistributes the plan. This system completed complex objectives in open-world game play. Meanwhile, similar systems have been used in software development [18] and (multi-)robot systems [21] to achieve state of the art performance. Across these studies, it has been shown that LLM-MAs outperform their composing LLMs. However, it must be noted that these systems have achieved their performance with models that are too large to run on consumer hardware.

2.4 LLM cost reduction

The cost from LLM usage stems from its large number of parameters. As mentioned previously, commonly used chat LLMs have between 13 billion and 70 billion parameters. This results in a storage size between 26 and 140 gigabytes and a VRAM usage between 24 and 129 gigabytes. Moreover, a large number of parameters leads to a large number of floating-point operations that need to be computed. The usage cost of LLMs can be reduced by decreasing the number of these operations, or reducing the cost per calculation. There are multiple methods that can be used to reduce the cost of using LLMs. The most commonly used methods are pruning, knowledge distillation, and quantization [22]. These methods were originally designed for DNNs, but have now been adapted for LLMs. It is important to note that this thesis will focus on the models, hence prompting techniques for inference cost reduction are out of scope.

Pruning [23] refers to the removal of weights in a DNN to increase model efficiency. There are two types of pruning; structured and unstructured pruning. Unstructured pruning aims to increase the sparsity of a model by setting weights to zero. This reduces the storage size of the model. As the number of parameters remains the same, it does not have an effect on inference cost on non-specialized hardware. Structured pruning aims to remove neurons. This can be done in blocks, layers, or attention heads in LLMs. Structured pruning increases inference efficiency but often decreases performance substantially. One study [24] achieved 20% structured sparsity with a 10.8% decrease in NLP task performance despite task-specific fine-tuning.

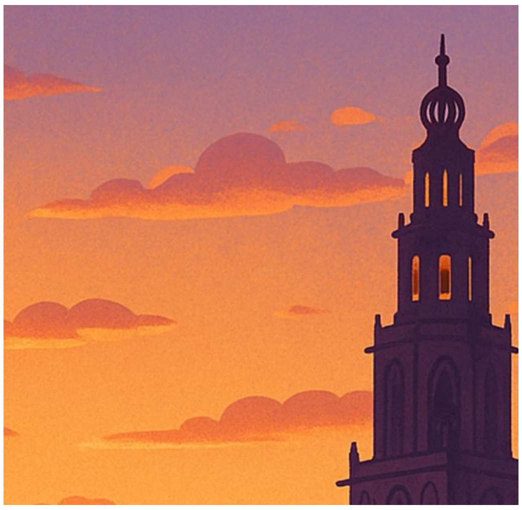
Knowledge distillation [25] is a method that aims to transfer the knowledge from a larger teacher model to a smaller student model. This can be done by combining the ground truth and output logits from the larger model, in the loss for the student model. This requires a lot of data and heavy re-training, though there are studies that aim to make this method more accessible using smaller datasets [26]. This is achieved by training the student model using LoRA, decreasing the required amount of data by 20%.

Quantization allows one to decrease the number of bits required for the model parameters. Quantization maps parameters onto smaller-precision formats. This means that fewer bits are required for mathematical operations. It was popularized after being applied to CNNs [27] and has since been applied to LLMs [28]. This also introduced the NF4 format. We will first examine the effect of quantization in general, before discussing the effect of this new format.

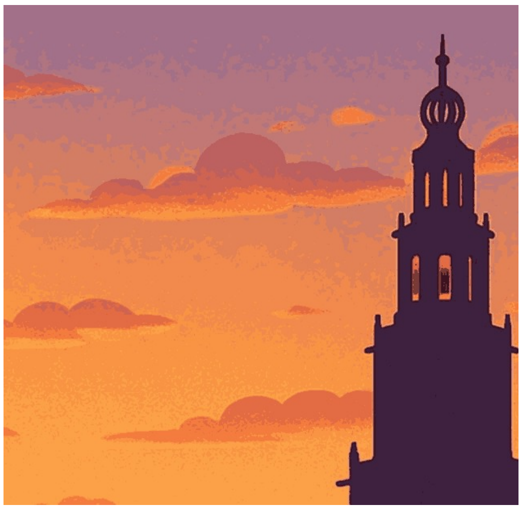
The effect of quantization can be intuitively illustrated using images, as can be seen in Figure 1. In the images, colors are binned to 16 values to simulate the effect of mapping to a four-bit format. A crucial insight is that, while the overall picture remains the same, some information is always lost. Especially when one inspects areas where colors gradually fade into one another, it becomes apparent that the transitions from one color to the next have become less fine-grained.

(a) ‘Full-precision’ grayscale image

(b) ‘Four-bit’ grayscale image



(c) ‘full-precision’ colored image

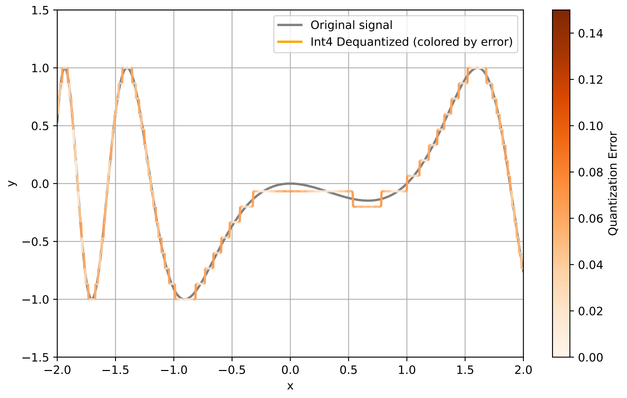


(d) ‘Four-bit’ colored image

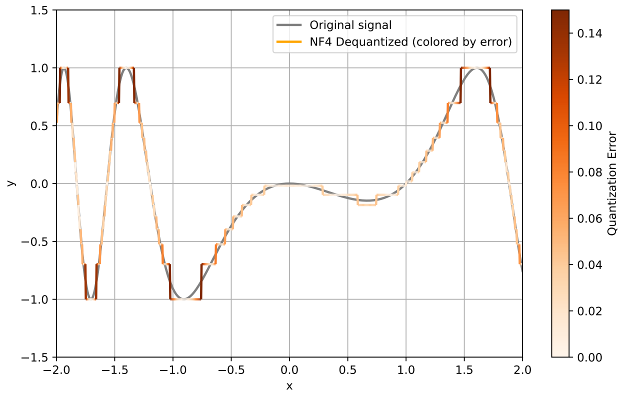
Figure 1: Visualization of ‘quantization’ effects using color bins

However, what if some colors were more important than others? In neural networks, weights have a normal distribution. The precision becomes more important near zero, as the difference between one weight and the next are much smaller. This must be taken into account when quantizing.

In this thesis, weights are mapped from half-precision (bf16, 16 bits) to four-bit integer precision (NF4 [28]). Bf16 was chosen over full precision (fp32) due to memory limitations. Employing models with bf16 has been shown to have equal performance to full-precision (fp32) methods [29]. Quantization to NF4 has three key elements: non-linear mapping, quantization blocks and double quantization. The non-linear mapping refers to the fact that weights of neural networks are normally distributed. Therefore, more integers are dedicated to near-zero floating point values. This alleviates the precision problem mentioned in the previous paragraph. The impact of this mapping technique can be seen in Figure 2.



(a) Mapped to Int4 and back



(b) Mapped to NF4 and back

Figure 2: Comparison of quantization errors between Int4 and NF4

In Figure 2, it can be seen how an arbitrary full-precision signal is converted to two four-bit formats and back. The color indicates the error between the converted signal and the original signal. Note how the NF4 signal results in a larger error near $y = 1$ and, crucially, a smaller error near $y = 0$. Note that the mapping in Figure 2 did not include quantization blocks or double quantization.

The quantization blocks refer to a process that divides the original weight distribution into partitions, called blocks. Each block can then be individually mapped to a quantization constant. As the mapping precision is mainly focused on near-zero values, this may have a negative impact on mapping precision of outliers. This could also be seen in Figure 2. By dividing the original weight distributions into blocks, each can be quantized separately with fewer outliers.
The result is multiple vectors called quantization constants. Double quantization can then be used to quantize the obtained constants, further reducing memory usage.

2.5 LLM-MA cost reduction

Previously, we mentioned methods of cost-reduction for LLMs, including structured pruning, knowledge distillation, and quantization. Most studies address inference costs by utilizing the smaller versions of state-of-the-art models, such as GPT-Turbo, have some efficiency without losing too much performance (which is their main focus).

Although there exists some literature [30] investigating (inference) cost-reduction of LLM-MAs, its focus lies on prompt or pipeline efficiency, using heuristics to maximize token efficiency in prompts, using cheaper models, or reducing the number of API calls in pipelines.
However, to the best of our knowledge, there is currently no literature that investigates the cost-reduction for LLM-MAs through model compression techniques such as the previously mentioned pruning, knowledge distillation, or quantization.

2.6 Functional performance

Before functional performance metrics, metrics from the natural language processing field were usually employed, such as BLEU [31]. This approach measured textual similarities by counting word occurrence. Specifically, it compared groups of words, referred to as N-grams, between the output and label text.

However, this metric has several limitations. Firstly, it is unable to capture more intricate dependencies that can exist in code. Whether Python code is successful, for example, depends largely on signs and indentations that BLEU does not evaluate. Second, it does not account for functional equivalence. Generating code that is exactly the same as the label, while only changing the variable names will result in a low BLEU score, while the code is functionally equivalent. The same critique applies for coding structures and methods.

To solve these issues, the pass@k metric was introduced [32]. To compute the metric, generated code is compiled, run, and tested with input-output pairs. If the code compiles and passes all input-output tests, it is denoted with a boolean True. The pass@1 score for a model can then be computed as the percentage of questions that compiled and passed all their tests.

Pass@1 intuitively informs a user how many questions can be ‘perfectly’ solved by the model at its best performance. Moreover, it is much closer to how humans evaluate code.

Besides a model’s best performance, it might be able to solve more questions if it was given a larger number of attempts. In other words, more attempts might allow a model to show more of its capacity. This is what k values higher than one are used for.

The original pass@k with $k > 1$ would be computed over k runs. A question is counted as correct if any of the runs resulted in a correct code. This would give the user an indication of the maximal performance of a model.

Later, it was argued that this method of computing pass@k was biased as the results could vary largely between runs [33]. To solve this, an unbiased variant was proposed in which the number of runs $n >> k$.

Both pass@k variants have become a standard evaluation metric for code generation [34] and both are used in this thesis. The technical details will be further discussed in Section 3.10.

2.7 Motivation

As described earlier, LLMs have demonstrated remarkable capabilities across various applications. However, this comes at a significant computational cost. LLMs are resource-intensive, posing challenges both financially, for individuals and organizations with limited budgets, and environmentally.

Multi-agent LLM systems (LLM-MAs) have demonstrated an ability to further enhance LLM functionality. However, these systems require iterative LLM prompting and are even more resource-intensive as a result. Furthermore, such systems often rely on models that are too large to be feasibly used on consumer-available hardware.

Previous efforts to reduce LLM-MA cost have focused on prompt engineering and pipeline optimizations, which improve efficiency by reducing token usage or minimizing API calls. While valuable, these approaches do not address the structural cost associated with model size and inference complexity.

Model compression techniques such as pruning, knowledge distillation, and quantization have demonstrated substantial efficiency gains for individual LLMs. Yet, their potential for LLM-MAs remains largely unexplored.

Among these methods, structured pruning may compromise performance, and knowledge distillation requires substantial retraining using large datasets. Quantization, however, may reduce inference costs while maintaining performance and typically requires minimal retraining.

This thesis aims to fill this research gap by examining the impact of four-bit post-training quantization on the functional performance of LLM-MA systems, implemented with consumer-usable models.

2.8 Research Question

This thesis aims to investigate the impact of four-bit post-training quantization on the functional performance of multi-agent large language model (LLM-MA) systems. Using task success rate as the primary metric, it explores how compression through quantization affects collaborative effectiveness. The central research question is: 'How does 4-bit quantization influence functional performance in collaborate LLM-MA coding systems?'. With this, we aim to provide a first insight into efficiency-performance trade-offs in the design of LLM-MAs.

We hypothesize that post-training four-bit quantization will have a noticeably negative impact on collaboration performance as quantization is inherently a lossy process. Since collaboration requires iterative prompting using previous output, this may introduce cascading error.

3 Methodology

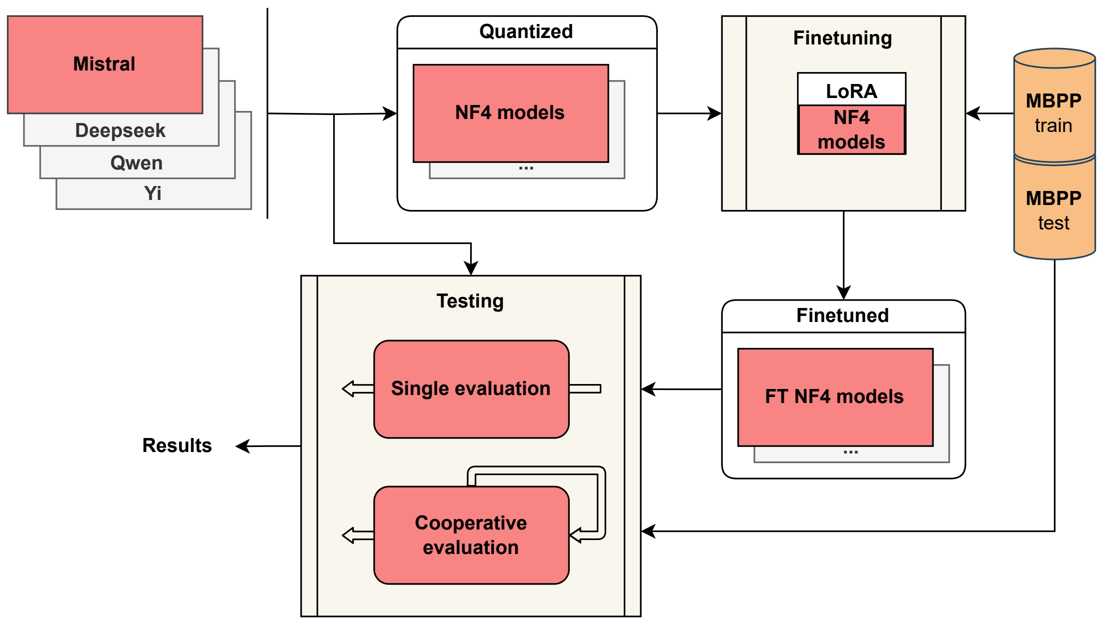


Figure 3: Overview of the pipeline. In quantization and fine-tuning, each model is processed once, resulting in four models that are quantized and fine-tuned. The four base models and their processed variants were tested both individually and collaborative.

An overview of the pipeline used in this thesis can be seen in Figure 3. Four models of varying size and performance were selected and quantized. To regain performance after quantization, the quantized models were fine-tuned using QLoRA. The models were tested on MBPP, in a 1-shot environment. Evaluation was done using the 'pass@k' metric, considering both individual and collaborative environments.

3.1 Dataset

To train and evaluate the models, the Mostly Basic Python Problems (MBPP) [35] dataset was used. This is a benchmark that is widely used for (multi-agent) code generation tasks [34]. It consists of 974 entry-level python problems, an example of which can be seen in Table 1.

Task	Write a python function to check whether the two numbers differ at one bit position only or not.
Test 1	$13,9 \Rightarrow \mathit{True}$
Test 2	$15,8 \Rightarrow \mathit{False}$
Test 3	$2,4 \Rightarrow \mathit{False}$

Code	`def is_Power_Of_Two (x):`
	`    return x and (not(x & (x - 1)))`
	`def differ_At_One_Bit_Pos(a,b):`
	`    return is_Power_Of_Two(a ^ b)`

Table 1: MBPP sample

As shown, each problem consists of a description in natural language, three input-output test cases, and a Python code solution. The dataset is split into a train partition containing 474 problems and a test partition containing 500 problems.
There exist some alterations for MBPP, namely the sanitized or plus versions. The sanitized version removes ill-phrased questions are removed, resulting in fewer entries. The plus-versions contain more test cases, focusing on edge-cases.
The regular MBPP dataset contains the largest number of entries, which is preferable for fine-tuning, and is the most regularly used [34]. For these reasons, the regular MBPP dataset was chosen instead of its alterations.

3.2 Model Selection

For this study, we selected four models that are suitable for coding and collaboration. Previous literature has shown that 'instruct-versions' of CodeLlama consistently outperform their base version when used in self-collaboration [18]. Moreover, it was shown that self-collaboration capacities increase with general model capacities, and start around 7 billion parameters. Hardware limitations constrained the largest model in this thesis to 9 billion parameters. Therefore, we opted to use only instruct- and chat-tuned models to maximize the potential collaborative performance available to us.

Considering this, the following models were selected:

1. Mistralai/Mistral-7B-Instruct-v0.3 [36] is a decoder-only model with 7.3B parameters. It consists of 32 transformer blocks and utilizes a sliding window attention mechanism.

2. Deepseek-ai/deepseek-coder-6.7b-instruct [37] has a similar architecture to Mistral-7B-Instruct-v0.3 but does not use a sliding window attention mechanism. While Mistral is a model with general capabilities, Deepseek is focussed on coding, being trained from start on a mixture of natural-language and code repositories.

3. Qwen/Qwen2.5-Coder-3B-Instruct [38] is a much smaller model than the previous two. While it has the same number of attention heads, the hidden size of each head is smaller, making this a lighter model. Qwen 2.5 Coder was built off the Qwen 2.5 series. It emphasized the importance of architectural choices that optimize coding behavior such as SwigLU activation functions, QKV bias, and RMS normalization functions to achieve state-of-the-art performance.

4. Yi-Coder-9B-Chat [39] is the largest of the four selected models. While it has 3B to 6B more parameters than the other models, its storage size is only slightly larger than Deepseek or Mistral, as can be seen in Table 2. The 9B-variant of the Yi model family was an extension from the 6B variant, where twelve middle transformer layers were duplicated before pretraining the resulting 9B model. While the other models are instruct-tuned, Yi was chat-tuned, which allowed for more flexible prompting.

All four models are based on the Llama architectures, hence they share similarities. Although the number and size of hidden layers may differ, as well as specific mechanisms such as attention computation or pretraining tactics.
With this selection, we aimed to incorporate models of varying levels of recency and size.

3.3 Quantization

As mentioned in Section 2.4, four selected models were quantized post-training from bf16 to the NF4 format. This was done using the BitsAndBytes (BnB) library [5]. Quantizing the models effectively reduced their VRAM usage by 64.5-72%, as can be seen in Table 2. Note that the base models are not loaded in full-precision (fp32), but in half-precision due to resource constraints for the larger models. It has been found that utilizing bf16 instead of fp32 does not reduce performance [29]. The precise settings for quantization can be found in the Appendix.

	Qwen	Deepseek	Mistral	Yi
Base	5.75	12.56	13.50	16.45
Quant + LoRA	1.88	3.52	3.85	5.84
	-67.4%	-72.0%	-71.5%	-64.5%

Table 2: VRAM usage (GBs) of models before (‘Base’) and after the pipeline (‘Quant + LoRA’)

3.4 Fine-tuning

As quantization is inherently lossy, the models were fine-tuned back to original performance. This was done using Low-Rank Adapters (LoRA). LoRA is a parameter-efficient fine-tuning (PEFT) method, meaning it requires few parameters to be fine-tuned for full model retraining effects. A visual explanation is given in Figure 4.

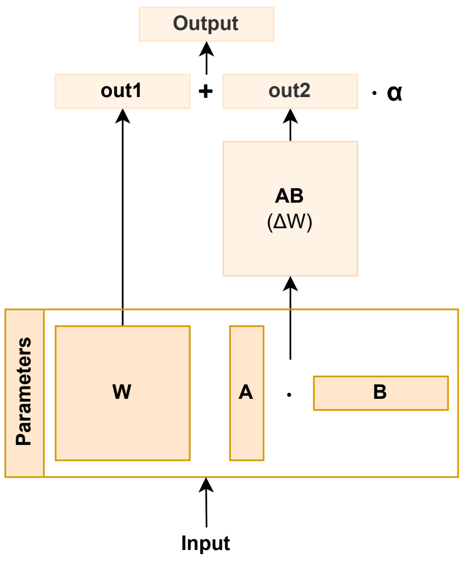


Figure 4: A visualization of how low-rank adapters are applied to change output. *A* represents a module, or block of parameters, from the model. The vectors at *C* are the learned adapters. *D* is the result of multiplying the adapters. Input to the LoRA-applied model will pass through *A* and *D*, resulting in vectors *B* and *E*, respectively. Vector *E* is scaled by *alpha* and added to the original output.

Here, model parameters (or weights) *W* are frozen, meaning that gradients are not computed such that the weights *W* will not be updated. Instead, these parameter changes are approximated. This is

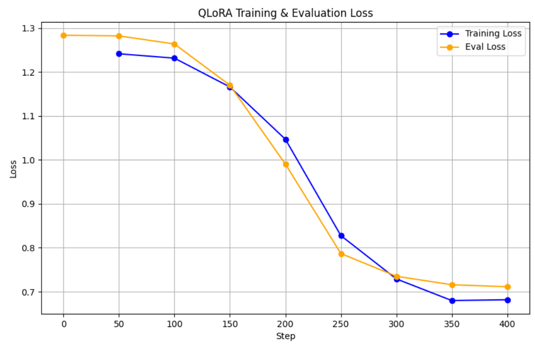
done by matrix vectors (called coefficients) A and B. The product of A and B is a matrix ΔW—also labeled as AB in the figure—which represents the actual model's parameter changes. This way, there is a double reduction on the number of parameters that are updated.
When combined with quantization, this method is known as QLoRA [28] (see Eq. 1).

$$\hat{W} = W_{NF4} + \Delta W_{fp32}, \quad \text{where} \quad \Delta W = AB \tag{1}$$

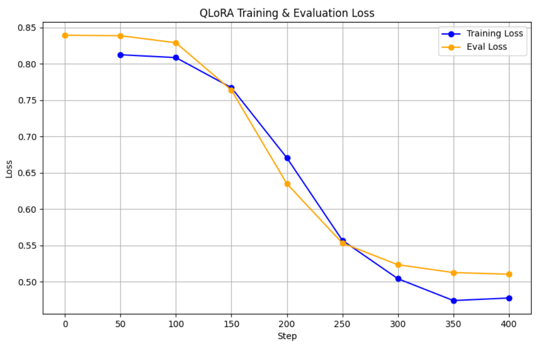
QLoRA allows for efficient fine-tuning of models by quantizing the frozen base model and only training an additional set of non-quantized parameters. This set of parameters consists of two matrices A and B that, when multiplied, should approximate the desired changes in the original modules (ΔW). When producing output, the two matrices are multiplied and the product of their output and the model input is added to the output of the base model with a factor of α.
As coding problems with pass@k are a precise task, fine-tuning a model must be done carefully to prevent degrading the original model's performance. This will be further discussed in Section 3.6.

3.5 Training

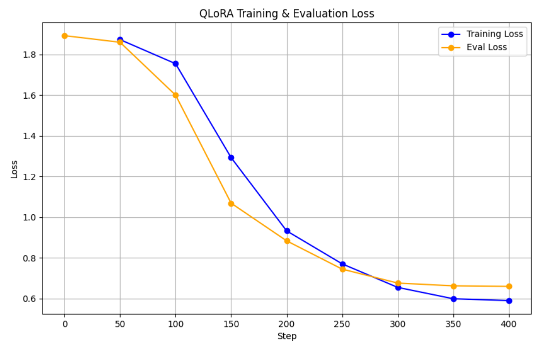
We trained each model for 20 epochs with early stopping if the evaluation loss did not decrease in 3 evaluations. Input batches were padded dynamically to multiples of 8. This reduces the quality decrease due to padding. As can be seen in Figure 5, all models were trained for the full 20 epochs.



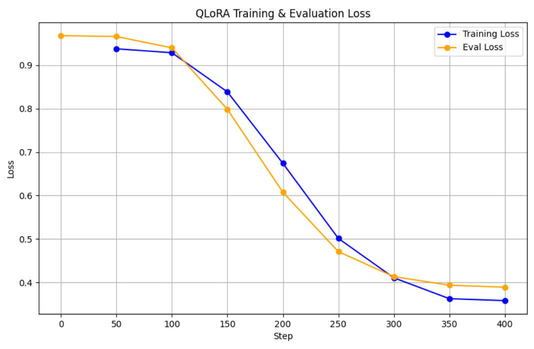
(a) Qwen (400)



(b) Deepseek (400)



(c) Mistral (400)



(d) Yi (400)

Figure 5: Loss plots for QLoRA fine-tuning of selected models

For evaluation, we now have the four base models and four quantized and fine-tuned models. These eight models will be evaluated in further sections.

3.6 Hyperparameters

The batch size was optimized to maximize VRAM utilization, resulting in a size of 8. To achieve smoother gradients and potentially better generalization, a gradient accumulation step of two was implemented. An initial learning rate of 2×10^{-5} was used in combination with a cosine scheduler, the Adam optimizer and 200 warm-up steps. The learning rate, smaller than typically employed, was chosen to ensure smoother gradient updates. During training, a LoRA dropout of 0.05 was used to reduce the risk of overfitting.

While we explored higher LoRA ranks, these necessitated larger datasets to mitigate overfitting, as larger changes to the model could be approximated. This was tested using a curated dataset containing 6.000 instruction strings with code labels. The training outcome was evaluated on the training partition of MBPP. As can be seen from from Table 3, there was a small improvement in pass@1 with $r = 254$, however this is likely by chance as $r = 64$ resulted in a lower pass@1. Using $r = 8$, the final results from Table 4 were produced.

Lora rank	r=16	r=64	r=254
Pass@1	29.4%	28.8%	30.1%

Table 3: Rank tuning using Mistral

As a rank of 8 sufficed, a larger dataset was not required. Hence, the models were fine-tuned on the train partition of MBPP. To maintain comparability across experiments and models, uniform hyper-parameters were adopted for all models. While hyper parameter tuning for each model could enhance their overall performance, it may introduce variability in results due to different tuning strategies. Given the scope of this study, the focus on quantization, and resource limitations, we chose to use uniform hyper parameters across all models. This was sufficient to regain comparable or improved performance for all models, as can be seen in Section 4.

3.7 Prompting

Both individual and collaborative evaluations employed a 1-shot prompting technique, using the initial input-output pair for each problem. This approach was designed to provide the systems with an illustrative example while withholding comprehensive details of the test cases. The objective was to strike a balance between directing the systems and allowing for inference and enhancement opportunities. This is similar to an experiment from previous literature [40], where the coder had access to one test case.

3.8 Individual evaluation

To assess the effect of collaboration, we first establish a baseline by evaluating each model individually. This is done by prompting for the user requirement, with a style prompt to only respond in code, without test cases or explanations. Individual prompting was done greedily for one sample, and with temperature $T = 0.8$ for 10 samples. This will be discussed further in Section 3.10.

3.9 collaborative evaluation

The collaborative environment structure is based on previous self-collaboration studies [18, 17]. It consists of a loop in which the coder aims to generate and improve code, while the tester aims to evaluate the code and provide feedback. An overview of the collaborative test cycle can be seen in Figure 6

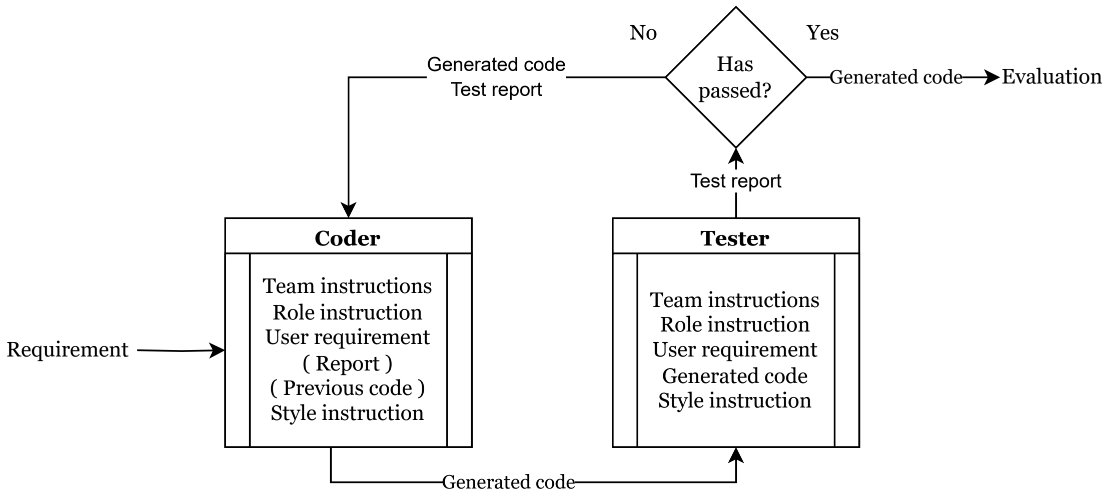


Figure 6: Workflow of the collaborative environment. Items in brackets (e.g. ‘report’) are optional: only after receiving feedback from the tester is this included in the coder prompt.

Whether the code has been passed, is determined by string matching ”Code test passed” or ”Code test failed” against the tester output. When the tester does not pass the code after five iterations of improvement, the code is automatically moved to evaluation. Code is automatically counted as incorrect when compiling and running takes longer than 5 seconds. The coder is allowed a maximum of 5 corrections per question. Previous literature [18] allowed for 10 tries, and 5 seconds run-time per code snippet. Considering the relative simplicity of the dataset, and the limited resources, the number of allowed attempts and duration of run-time were halved. While these reductions may cause a slight decrease in results, it affects all models equally. Therefore it still allows us to evaluate the relative performance that is discussed in the research question.
It is important to note that the code is only compiled and tested after it is approved by the tester. Moreover, the tester does not have access to a terminal, it can only evaluate the code based on its own inspection.
The collaborative structure is based on previous literature [18], with some alterations. Originally the structure contained three roles, which was now reduced to two. This made a large part of the input prompts redundant, hence the prompts were compacted. Furthermore, a previous study underlined the importance of error localization in collaborative feedback, without which the feedback would have a 33% chance of being destructive instead of helpful [17]. The original collaborative structure can be found in the Appendix. Here follows the employed collaborative structure.

Prompting structure

Coder := C. instructions + User requirement [+ Test report] [+ Previous code] + C. format
Tester := T. instructions + User requirement + Previous code + T. format

Collaborative system segments

Tester

Instructions:
You are a tester in a code-review team. You will receive:
- A natural language description of what the function is supposed to do
- One or more input/output test cases
- The code written by a developer

Format:
Write a test report using the following structure:
{
"Code compilation": Evaluate the structure, syntax, and logic correctness of the code. Will the code compile correctly?
"Input/output": Does the function take the correct number of variables? Are they in the correct format? How about the output?
"Improvements": If there are problems with the code, what should the coder improve? Be specific.
"Localization": Where in the code does this fix apply?
"Conclusion": "Code Test Passed" or "Code Test Failed". If you are even slightly unsure, choose "Code Test Failed" and explain.
}

Coder

Instructions:
I want you to act as a developer on our development team. Your job is to write or improve Python code that meets the requirements. Ensure that the code you write is efficient, readable, and follows best practices.

Format:
Remember, only provide code: do not explain the code and do not give test cases.

User Requirement

This is the user requirement:
Write a function to find sequences of lowercase letters joined with an underscore.

Example input: "aab_cbbbc"
Example output: ('Found a match!')

3.10 Evaluation metric

To evaluate the generated code, the pass@k metric [41] was used. This metric evaluates whether the generated code passes a set of predefined tests, thereby assessing the functional correctness of the

code. It has become a standard benchmark for code generation evaluation in recent studies [34], as also discussed in Section 2.6.
In this section, we will formally define the two metrics. With pass@1, “a problem is considered solved if any of the samples pass the tests; and the total fraction of problems solved is reported ” [32]. Let $Q = \{q_1, q_2, \dots, q_N\}$ be the set of N evaluation problems. For each problem q_i, the model generates a single solution s_i. This can be defined with an indicator function:

$$\delta(s_i) = \begin{cases} 1 & \text{if } s_i \text{ passes all predefined tests,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{pass@}1 = \frac{1}{N} \sum_{i=1}^{N} \delta(s_i). \tag{2}$$

As previously discussed, there is a distinction between pass@1 and pass@k where $k > 1$. Pass@1 aims to specifically capture a model’s optimal performance, by performing greedy decoding. This means the temperature is set to zero, and tokens are not sampled besides the token with maximal logit. Pass@1 is therefore deterministic.
However, pass@k for $k > 1$ does sample from output logits during generation. Computing pass@2 where the number of samples $n == k$ would therefore be biased as the outcome would differ strongly per run. For this reason, pass@k where $k > 1$ is computed using a number of samples $n >> k$ such that Equation 3 can be used.

$$\text{Pass@}k = \mathbb{E}\left(1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right) \tag{3}$$

Here, n is the number of generated code attempts per question, c is the number of correct attempts and k is an arbitrarily chosen integer. For a whole dataset, the average estimate is taken. In this thesis pass@k was computed using $k = 2$ and $n = 10$.

3.11 Reproducibility

In order to ensure reproducibility, the random seed for *torch*, *random*, and *numpy* was set to 333 during evaluation. Moreover, all data, output reports and results will be made available on the Github of this project.

4 Results

The goal of this study was to investigate the effect of four-bit quantization on the functional coding performance of LLM-based multi-agent systems. We hypothesized that quantization would have a negative effect on collaborative performance, as quantization is a lossy process, and collaboration could introduce a cascading effect on the error.

To investigate this, we created a pipeline in which models were quantized, fine-tuned and tested in two settings. By evaluating the performance of each model in the collaborative setting, and in an individual setting, a comparison can be made with regard to its collaboration capacities. A total of four models, and their four four-bit versions were used in this experiment. Each model was tested in two settings, where the first setting (individual) was evaluated using both pass@k metrics. The main results of the experiment can be found in Table 4.

Model	Quant	Pass@1	Pass@2	Collaboration	$\uparrow$ over pass@1
Qwen	Base	42.4	86.8	33.8	-20.3%
	Four-bit	45.4	84.6	42.4	-6.6%
Deepseek	Base	55.4	88.0	54.6	-1.4%
	Four-bit	53.4	78.8	54.8	+2.6%
Mistral	Base	33.6	77.7	31.6	-6.0%
	Four-bit	32.2	64.4	33.0	+2.5%
Yi	Base	40.20	71.3	59.2	+47.3%
	Four-bit	48.8	76.0	57.0	+16.8%

Table 4: Main results

As mentioned before, pass@1 refers to optimal behavior using greedy generation, whereas pass@2 (with $k = 2$ and $n = 10$) was computed to give an approximation of model capacity without greedy generation. In the 'Quant' column, 'Base' refers to the four models before being processed in the pipeline and 'Four-bit' refers to the QLoRA fine-tuned version, as described in in Figure 3. Both 'pass' columns refer to the individual testing environment. The 'collaboration' column refers to the pass@1 score in the collaborative testing environment. Lastly, the right-most column describes the proportional relation of the collaboration.

The results can be interpreted in multiple ways. Firstly, one can inspect the right-most column to gain an understanding of the overall effect of collaboration on performance. Secondly, one can compare the results of each model to its quantized version to gain an understanding of the effect of quantization on model performance. Lastly, one can inspect the adapted collaboration column to answer the research question, comparing the base collaborative performance with quantized collaborative performance. To give further context to these results, the error sources will be briefly discussed. After that, relations within the results will be highlighted towards the end of this section.

Besides boolean correctness, the source of incorrect responses can also yield valuable insights. Here, we differentiate between compiling errors and output errors. The first errors relate to general code generation mistakes, such as invalid variable name usage, logic faults or indentation errors. Output errors occur when there is a mismatch between the generated output and output label. This may occur when faulty logic, or erroneous output formats were used. For an output error to occur, it must have first compiled correctly. It does not have the basic coding mistakes that faultily-compiling code contains. Code with errors of the output category is, in that sense, of higher quality than the first category. These results can be seen in Table 5.

		Individual			Collaborative		
Model	Quant	Compile	Output	Output %	Compile	Output	Output %
Qwen	Base	129	159	55.2%	193	138	41.7%
	Four-bit	85	188	68.9%	119	169	58.7%
Deepseek	Base	48	175	78.5%	38	189	83.3%
	Four-bit	68	165	70.8%	39	187	82.7%
Mistral	Base	102	203	66.6%	128	214	62.6%
	Four-bit	87	252	74.3%	103	232	69.3%
Yi	Base	172	127	42.5%	45	159	77.9%
	Four-bit	115	141	55.1%	52	163	75.8%

Table 5: Error source

Here, 'compile' and 'output' are absolute counts of errors whereas 'output %' is the proportional percentage between compilation and output errors. A higher percentage indicates a higher proportion of code that compiled correctly yet gave incorrect output, which we interpreted as better performance.

For Qwen, Table 4 previously showed that QLoRA slightly improved the performance. This is confirmed by Table 5 as the quality of code that was incorrect is also higher for QLoRA compared to the base model; both in the individual environment (68.9% to 55.2%) as in the collaborative environment (58.7% to 41.7%). Focusing on collaboration, Table 4 showed that collaboration decreased Qwen's performance. Here too, it is shown that collaboration decreased Qwen's performance in output quality for both the base model (55.2% to 41.7%) and four-bit model (68.9% to 58.7%).

For Deepseek, it was shown in Table 4 that QLoRA slightly decreased its performance, which is also shown in Table 5 for the individual setting (78.5% to 70.8%) and the collaborative setting (83.3% to 82.7%). Moreover, collaboration was previously shown to produce in similar results as individual prompting. However, Table 5 shows collaboration to slightly improve output quality in the base model (78.5% to 83.3%) and the four-bit model (70.8% to 82.7%).

For Mistral, QLoRA was shown to decrease performance in Table 4. However, Table 5 shows an improvement in output quality in the individual setting (66.6% to 74.3%) and the collaborative setting (62.6% to 69.3%). Moreover, Table 4 showed mixed results with regard to the effect of collaboration while Table 5 shows a decline in performance for both the base model (66.6% to 62.6%) and four-bit model (74.3% to 69.3%).

For Yi, Table 4 showed a clear increase in performance for the QLoRA model in the individual setting, and a slight decrease in the collaborative setting. Both Yi models scored considerably higher in the collaborative setting than in the individual setting. This aligns with the error source, as can be seen in Table 5.

5 Conclusion and discussion

This thesis aimed to investigate the effect of four-bit post-training quantization on the collaborative performance of LLMs in coding tasks. It was hypothesized that quantization would decrease the collaborative capacities of models, by introducing a lossy parameter space compression.

The findings of this study are best interpreted in two parts. First, the baseline performance will be analyzed, after which the effect of quantization on that performance will be discussed.

5.1 Model performance

The results of Table 4 show the boolean 'correctness' of each model. The results from the individual setting were consistent with expectations, given the 1-shot testing, and the non-filtered nature of the dataset. Although the models achieve a smaller pass@1 in the individual setting compared to their original studies, the pass@2 scores show that the models still retained similar coding capacity.

In the collaborative setting, Qwen exhibited a noticeable decline in performance when compared to its individual performance. Larger models, such as Deepseek and Mistral, show similar performance in collaboration as individually. Only the largest model, Yi, displayed a substantial improvement due to collaboration. This aligns with the findings of previous literature, where it was found that smaller models displayed less capacity for collaboration [17] or that models start to show collaborative improvement from around 7B parameters [18].

5.2 Effect of quantization

The effect of quantization was mixed, as it can be seen to both improve and decrease model performance if one looks at the pass@1 scores in the individual test setting. However, the pass@2 scores reveal that coding capacity does often decrease. One possible explanation for the increase in individual pass@1 performance could be that fine-tuning on the training dataset has increased optimal performance while quantization has decreased overall model capacity.

In the collaborative setting, quantization showed similar results. For the first three models, the quantized versions performed better, albeit slightly, than their base models. Notably, Yi performed slightly worse in collaborative setting after quantization despite performing better in both pass@1 and pass@2.

It was hypothesized that quantization would lead to a strong deprecation of collaborative performance. However, our findings indicate that the effect of quantization on collaborative performance is minimal and may sometimes be positive.

While these results seem promising, some limitations must be acknowledged.

5.3 Limitations

As mentioned in previous literature [18] and observed in our results, collaboration performance appears to emerge when model parameter counts exceed approximately 7 billion parameters. As such, some of the models used in this thesis may not have been suitable for studying the effect of quantization on collaboration, as there is not yet any collaborative performance increase. Consequently, findings derived from these smaller-scale models should be interpreted with caution.

In Section 3.1, it was discussed that this study utilized one dataset, potentially limiting the generalizability of our findings. Specifically, the dataset employed was the standard MBPP dataset. As discussed earlier, the regular MBPP contains shortcomings such as ambiguously posed questions. Examples of such questions can be found in Tables 6 to 8. Retaining these questions can reduce overall performance scores and make comparisons less straightforward. As a result, the interpretability of our findings may be reduced.

As mentioned in 3.6, hyperparameters were shared across models after selection. While this allows for a more consistent comparisons between models, it does limit each model's optimal performance. This makes our results less realistic, as practical applications of any model typically aim to optimize the performance of each individual model.

Pass@k presents several challenges as an evaluation metric. Firstly, loss performance could improve while the pass@1 metric decreases. This makes it difficult to interpret model performance, as interpretations of the loss and pass@1 could lead to different conclusions. Secondly, the collaborative setting required a large number of samples to be generated. As our computational resources were limited, only the pass@1 could be computed. While this can be intuitively compared to the individual pass@1, an improvement could be made by using the non-biased pass@2 in order to achieve more stable results.

Finally, this study implemented a 1-shot prompting strategy. The potential effect of few-shot prompting, especially the widely implemented 3-shot prompting, has remained unexplored. As LLMs are few-shot learners [11], this could have a large impact on our findings.

5.4 Implications

The findings of this thesis imply that models can be quantized to lower-precision formats, such as NF4, without substantial collaboration performance loss in coding tasks. It has given a first insights into the trade-off between coding performance and efficiency through post-training quantization. If this effect is generalizable, it could lower financial and economic barriers to employing LLM-MAs. The resulting democratization could open the door to innovation in a variety of domains.

Firstly, a direct implication is the potential for LLM-MAs to be employed on consumer-grade hardware. Decreasing the hardware requirements lowers the economical barriers, making LLM-MAs accessible to small organizations, individuals, or educational institutions with limited resources. In turn, this could potentially increase its functional application and accelerate its research.

Secondly, it could lead to the a broader application of LLM-MAs. If LLM-MAs can be deployed in NF4 format with minimal effect on performance, then it could be deployed more often in wide variety of domains. Earlier examples included healthcare, where LLM-MAs could independently simulate teams of doctors generating a comprehensive diagnosis, and a team of developers programming an extensive program.

As research continues to increase model performance, our findings may change. Perhaps smaller models will achieve similar capacities that is currently only observed in models with a parameter count that exceeds 7 billion parameters. This could result in an exciting research opportunity. If models can be employed collaboratively using a smaller amount of VRAM, more models can be employed. This could potentially lead to larger systems using a larger variety of LLMs. In this study, self-collaboration was used as it required only one model to be loaded at any time. If a larger number of models can be loaded simultaneously, specialized models can be used for different roles, potentially resulting in a mixture-of-experts collaborative system.

5.5 Future research

In previous sections, several potential directions for future work were identified. Here, these points are summarized and expanded upon.

This study quantized the models using the Bits and Bytes library. While this decreases VRAM requirements, and thereby indirectly accelerates inference through batch size, this could be taken further with fused kernels. Quantization with fused kernels reduces model size by decreasing the number of parameters, similar to structured pruning. Previous literature has suggested that models with fused kernels retain similar performance to their base models [42]. This could further increase the efficiency of LLM-MAs.

As quantization is a lossy process, the models in this study were retrained using QLoRA. While this had the desired effect of restoring performance with limited retraining, this also meant that the quantization process was no longer isolated. Future research could investigate the effect of LoRA on LLM-MAs, so that the effect of compression could potentially be better isolated.

Furthermore, a broader study could compare different methods of compression, such as distillation, pruning, and quantization. This could result in a systematic comparison between compression methods, potentially yielding insights regarding the effect of compression on model behavior in general.

Another point that was discussed was the model selection. Our findings aligned with previous literature regarding parameter count and collaboration capacities, highlighting a possibility for future research. While this study used models of 3 to 9 billion parameters, larger models are expected to display stronger collaborative capacities. Future research could therefore further investigate this topic using larger models, in order to gain more definitive insights into the effect of quantization on collaborative performance.

Alternatively, the origin of collaborative performance could be investigated. Future research could investigate the model performance as a predictor of collaborative performance. This would shift the emphasis from sheer parameter count to model quality, which may increase in importance as LLMs continue to improve. Additionally, the impact of different tuning styles, such as instruction or chat tuning, could be investigated. Such investigations could reveal insights into optimization methods that are best suited for LLM-MAs.

Results indicate that larger models may achieve higher (collaborative) performance. If performance increases, it could be tested against higher standards, using the extended test-cases from MBPP+, for example. Moreover, the generalizability of our findings could be investigated using a larger variety of datasets.

Finally, as highlighted in the introduction, iterative collaboration is argued to be especially suitable for long-horizon tasks. A possible avenue for future research could be to investigate the effect of quantization on LLM-MAs, focusing on task coherency instead of coding accuracy. This focus would be especially relevant for many real-world applications, such as web or app development, where sustained coherence and coding performance across multiple steps is essential.

6 Acknowledgments

While LLMs were used to assist during the research and writing of this thesis, all original ideas, analyses, and final contents are my own.

Bibliography

[1] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," 2023.

[2] J. Lu, Z. Pang, M. Xiao, Y. Zhu, R. Xia, and J. Zhang, "Merge, ensemble, and cooperate! a survey on collaborative strategies in the era of large language models," 2024.

[3] E. J. Husom, A. Goknil, L. K. Shar, and S. Sen, "The price of prompting: Profiling energy use in large language models inference," 2024.

[4] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big? ," in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, (New York, NY, USA), p. 610–623, Association for Computing Machinery, 2021.

[5] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017.

[6] E. J. Husom, A. Goknil, M. Astekin, L. K. Shar, A. Kåsen, S. Sen, B. A. Mithassel, and A. Soylu, "Sustainable llm inference for edge ai: Evaluating quantized llms for energy efficiency, output accuracy, and inference latency," *ACM Trans. Internet Things*, Sept. 2025. Just Accepted.

[7] N. Alizadeh, B. Belchev, N. Saurabh, P. Kelbert, and F. Castor, "Language models in software development tasks: An experimental analysis of energy and accuracy," 2025.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 6000–6010, Curran Associates Inc., 2017.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (J. Burstein, C. Doran, and T. Solorio, eds.), (Minneapolis, Minnesota), pp. 4171–4186, Association for Computational Linguistics, June 2019.

[10] A. Radford and J. Wu, "Rewon child, david luan, dario amodei, and ilya sutskever. 2019," *Language models are unsupervised multitask learners. OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020.

[12] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022.

[13] P. Schoenegger, I. Tuminauskaite, P. S. Park, R. V. S. Bastos, and P. E. Tetlock, "Wisdom of the silicon crowd: Llm ensemble prediction capabilities rival human crowd accuracy," *Science Advances*, vol. 10, no. 45, p. eadp1528, 2024.

[14] D. Jiang, X. Ren, and B. Y. Lin, "LLM-blender: Ensembling large language models with pairwise ranking and generative fusion," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (A. Rogers, J. Boyd-Graber, and N. Okazaki, eds.), (Toronto, Canada), pp. 14165–14178, Association for Computational Linguistics, July 2023.

[15] C. H. Song, B. M. Sadler, J. Wu, W.-L. Chao, C. Washington, and Y. Su, " LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models ," in *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, (Los Alamitos, CA, USA), pp. 2986–2997, IEEE Computer Society, Oct. 2023.

[16] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, Sept. 2024.

[17] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, "Self-refine: iterative refinement with self-feedback," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[18] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, 06 2024.

[19] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for a multi-agent collaborative framework," 2024.

[20] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C.-M. Chan, H. Yu, Y. Lu, Y.-H. Hung, C. Qian, Y. Qin, X. Cong, R. Xie, Z. Liu, M. Sun, and J. Zhou, "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors," in *International Conference on Representation Learning* (B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, eds.), pp. 20094–20136, 2024.

[21] Z. Mandi, S. Jain, and S. Song, "Roco: Dialectic multi-robot collaboration with large language models," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 286–299, 2024.

[22] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," 2024.

[23] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[24] X. Ma, G. Fang, and X. Wang, "Llm-pruner: on the structural pruning of large language models," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[25] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015.

[26] R. Yang, T. Wu, J. Wang, P. Hu, Y.-C. Wu, N. Wong, and Y. Yang, "Llm-neo: Parameter efficient knowledge distillation for large language models," 2025.

[27] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018.

[28] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: efficient finetuning of quantized llms," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[29] NVIDIA Corporation, "Nvidia a100 tensor core gpu architecture." https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020. Accessed: 13-08-2025.

[30] S. Gandhi, M. Patwardhan, L. Vig, and G. Shroff, "Budgetmlagent: A cost-effective llm multi-agent system for automating machine learning tasks," in *Proceedings of the 4th International Conference on AI-ML Systems*, AIMLSystems '24, (New York, NY, USA), Association for Computing Machinery, 2025.

[31] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (USA), p. 311–318, Association for Computational Linguistics, 2002.

[32] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, *SPoC: search-based pseudocode to code*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[33] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[34] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, July 2025. Just Accepted.

[35] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[36] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023.

[37] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[38] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, “Qwen2.5-coder technical report,” 2024.

[39] . AI, :, A. Young, B. Chen, C. Li, C. Huang, G. Zhang, G. Zhang, G. Wang, H. Li, J. Zhu, J. Chen, J. Chang, K. Yu, P. Liu, Q. Liu, S. Yue, S. Yang, S. Yang, W. Xie, W. Huang, X. Hu, X. Ren, X. Niu, P. Nie, Y. Li, Y. Xu, Y. Liu, Y. Wang, Y. Cai, Z. Gu, Z. Liu, and Z. Dai, “Yi: Open foundation models by 01.ai,” 2025.

[40] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” 2023.

[41] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, “Spoc: Search-based pseudocode to code,” 2019.

[42] H. Guo, W. Brandon, R. Cholakov, J. Ragan-Kelley, E. P. Xing, and Y. Kim, “Fast matrix multiplications for lookup table-quantized llms,” 2025.

7 Appendix A

7.1 Ambiguous MBPP examples

Task (13)	Write a function to count the most common words in a dictionary.
Test 1	['red','green',[..],'black'] ⇒ [('pink',6),('black',5),('white',5),('red',4)]
Test 2	['one','two',[..],'four'] ⇒ [('one',4),('two',2),('three',2),('four',1)]
Test 3	['Facebook','Apple',[..],'Netflix'] ⇒ [('Apple',2),('Amazon',2),('Netflix',2),('Facebook',1)]

Code

```
from collections import Counter
def count_common(words):
    word_counts = Counter(words)
    top_four = word_counts.most_common(4)
    return (top_four)
```

Table 6: Only the four most common words were expected

Task (28)	Write a python function to find binomial co-efficient.
Test 1	$4,2 \Rightarrow 10$
Test 2	$4,3 \Rightarrow 4$
Test 3	$3,2 \Rightarrow 3$

Code

```
def binomial_Coeff(n,k):
    if k > n :
        return 0
    if k==0 or k ==n :
        return 1
    return binomial_Coeff(n-1,k-1) + binomial_Coeff(n-1,k)
```

Table 7: A difficult description, requiring recursion.

Task (147)	Write a function to find the maximum total path sum in the given triangle.
Test 1	[[1,0,0],[4,8,0],[1,5,3]],2,2 ⇒ 14
Test 2	[[13,0,0],[7,4,0],[2,4,6]],2,2) ⇒ 24
Test 3	[[2,0,0],[11,18,0],[21,25,33]],2,2) ⇒ 53

Code

```
def max_path_sum(tri, m, n):
    for i in range(m-1, -1, -1):
        for j in range(i+1):
            if (tri[i+1][j] > tri[i+1][j+1]):
                tri[i][j] += tri[i+1][j]
            else:
            tri[i][j] += tri[i+1][j+1]
    return tri[0][0]
```

Table 8: An ambiguous path

7.2 Original collaborative system

Old collaborative system [18] - coder

Team Instructions

There is a development team that includes a requirements analyst, a developer, and a quality assurance reviewer. The team needs to develop programs that satisfy the requirements of the users. The different roles have different divisions of labor and need to cooperate with each other.

Coder Instructions

I want you to act as a developer on our development team. You will receive plans from a requirements analyst or test reports from a reviewer. Your job is split into two parts:
1. If you receive a plan from a requirements analyst, write code in Python that meets the requirements following the plan. Ensure that the code you write is efficient, readable, and follows best practices.
2. If you receive a test report from a reviewer, fix or improve the code based on the content of the report. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code.

User Requirement

This is the user requirement:
Write a function to find sequences of lowercase letters joined with an underscore.

Example input: ”aab_cbbbc”
Example output: (’Found a match!’)

Test report

{
”Code Review”: [...]
”Code Description”: [...]
”Satisfying the requirements”: Bool
”Edge cases”: [...] ”Conclusion”: ”Code Test Passed” or ”Code Test Failed”.
}

Previous code

This is the previous code to improve:
“‘python
def [...]
”’

Old collaborative system [18] - tester

Team Instructions

There is a development team that includes a requirements analyst, a developer, and a quality assurance reviewer. The team needs to develop programs that satisfy the requirements of the users. The different roles have different divisions of labor and need to cooperate with each other.

Tester Instructions

I want you to act as a tester in the team. You will receive the code written by the developer, and your job is to complete a report as follows:

”Code Review”: Evaluate the structure and syntax of the code to ensure that it conforms to the specifications of Python, that the APIs used are correct, and that the code does not contain syntax errors or logic holes.
”Code Description”: Briefly describe what the code is supposed to do. This helps identify differences between the code implementation and the requirement.
”Satisfying the requirements”: ”True” or ”False” - This indicates whether the code satisfies the requirement.
”Edge cases”: Edge cases are scenarios where the code might not behave as expected or where inputs are at the extreme ends of what the code should handle.
”Conclusion”: ”Code Test Passed” or ”Code Test Failed” - This is a summary of the test results.

’’’

User Requirement

This is the user requirement:
Write a function to find sequences of lowercase letters joined with an underscore.

Example input: ”aab_cbbbc”
Example output: (’Found a match!’)

Previous code

This is the previous code to improve:
‘‘‘python
def [...]
’’’

7.3 Qwen individual prompt

Example prompt:

```
Write a python function to remove first and last occurrence of a given character
```

```
from the string.

Example input: "hello","l"
Example output: "heo"

Remember, only provide code: do NOT explain the code and do NOT give test cases.
```

Cleaned output:

~~~
```python
def remove_first_and_last_occurrence(s, char):
    # Find the first occurrence of the character
    first_index = s.find(char)
    # Find all occurrences of the character and store their indices
    all_indices = [i for i, c in enumerate(s) if c == char]
    # Remove the first and last occurrences
    if first_index != -1:
        s = s[:first_index] + s[first_index + 1:]
    if all_indices:
        last_index = all_indices[-1]
        s = ''.join([s[:last_index], s[last_index + 2:]])
    return s
```
~~~

Testing:

```
input: "hello","l"
output: "hel"
label: "heo"
```

7.4 Qwen collaborative prompts

```
You are a tester in a code-review team. You will receive:
- A natural language description of what the function is supposed to do
- One or more input/output test cases
- The code written by a developer

This is the user requirement:
Write a python function to remove first and last occurrence of a given character
from the string.

Example input: "hello","l"
Example output: "heo"

The code provided by developer is as follows:
def remove_first_and_last_occurrence(s, char):
    # Find the first occurrence of the character
    first_index = s.find(char)
    # Find all occurrences of the character and store their indices
```

```
    all_indices = [i for i, c in enumerate(s) if c == char]

    # If the character is not found, return the original string
    if first_index == -1:
        return s

    # Remove the first occurrence
    s = s[:first_index] + s[first_index + 1:]

    # Remove all occurrences except the last one
    for index in all_indices[:-1]:
        s = s.replace(s[index], '', 1)

    return s
```

```
Write a test report using the following structure:
{
  "Code compilation": Evaluate the structure, syntax, and logic correctness of
  the code. Will the code compile correctly?
  "Input/output": Does the function take the correct number of variables?
  Are they in the correct format? How about the output?
  "Improvements": If there are problems with the code, what should the coder improve?
  "Conclusion": "Code Test Passed" or "Code Test Failed".
  If you are even slightly unsure, choose "Code Test Failed" and explain.
}
```

Output:

```
{
  "Test Report": {
    "Code compilation":
      "The code provided compiles correctly without any syntax errors. The function
      `remove_first_and_last_ocurrence` is defined with the correct parameters
       and uses appropriate Python functions and constructs.",
    "Input/output":
      "For the given input-output pair, the function correctly removes
       the first and last occurrences of the specified character. The function
       takes the correct number and format of variables and returns the
       expected output.",
    "Improvements":
      "There are no significant improvements needed for the code. The logic is clear
       and efficient for the task. However, the function could be optimized slightly
       by using a single loop to remove all occurrences except for the last one,
       which would reduce the number of iterations over the string.",
    "Conclusion":
      "Code Test passed."
  }
```

}

7.5 Hyperparameters

load_in_4bit	Quant type	Double quant	Compute dtype
True	nf4	True	bf16

Table 9: Quantiztaion hyperparameters

Target modules	Rank	Alpha	Dropout	Bias	Task
q_proj, v_proj	8	4	0.05	none	causal

Table 10: LoRA hyperparameters

Temperature	Top_p	Top_k
0.8	0.95	20

Table 11: Prompting hyperparameters

Learning rate	Warmup steps	Optimizer	Scheduler	Accumulated batch size	Epochs	Early stopping patience
2e-5	200	Adam	cosine	8x2	20	3

Table 12: Training hyperparameters