A logo for a company

AI-generated content may be incorrect.

Master Thesis

**Evaluation of Quantized Large Language Models using LLM-as-a-Judge**

Submitted by: Riswan Basha Saleem Basha

First examiner: Prof. Dr. Andreas Pech

Second examiner: Prof. Dr. Armin Lehmann

Date of start: 25.10.2024

Date of submission: 25.03.2025

# Statement

##### I confirm that I, Riswan Basha Saleem Basha have written this thesis on my own. All sources used are duly referenced, and any content taken directly or indirectly from published or unpublished sources is properly cited. The figures and drawings included are my own creations or are appropriately referenced. While the core ideas and execution of this work are my own, I utilized tools such as ChatGPT for code debugging and troubleshooting, as well as Quillbot and Grammarly to enhance grammar and textual clarity.

##### Furthermore, I affirm that this work has not been submitted in the same or a similar form to any other examination board.

A blue text on a black background

AI-generated content may be incorrect.

##### 25.03.2025,

##### Date, signature of the student

# Table of Contents

[1. **Introduction** 5](#_Toc190560022)

[1.1 The Need for LLM Evaluation 5](#_Toc190560023)

[2. **Theoretical** **Background** 6](#_Toc190560024)

[2.1 Large Language Models (LLMs) 6](#_Toc190560025)

[2.1.1 Transformer Architecture Overview 6](#_Toc190560026)

[2.1.2 Decoder-Only Models (e.g., Mistral, Llama) 8](#_Toc190560027)

[2.2 Quantization Techniques 11](#_Toc190560029)

[2.2.1 Model Based Quantization 11](#_Toc190560030)

[2.2.2 Adapter Based Quantization 13](#_Toc190560034)

[2.2.3 Evaluation Metrics for Quantized Models 16](#_Toc190560039)

[2.3 LLM-as-a-Judge Framework 19](#_Toc190560045)

[2.3.1 Concept and Relevance 20](#_Toc190560046)

[2.3.2 ELO Score Evaluation Mechanism 22](#_Toc190560047)

[2.4 Streamlit for Visualization and Evaluation 24](#_Toc190560049)

[2.4.1 Key Features 24](#_Toc190560050)

[2.5 Literature Survey 24](#_Toc190560051)

[2.5.1 Literature 1 24](#_Toc190560052)

[2.5.2 Literature 2 25](#_Toc190560053)

[2.5.3 Literature 3 25](#_Toc190560054)

[3. **Requirement** **Analysis** 26](#_Toc190560055)

[3.1 General Objectives 26](#_Toc190560056)

[3.2 Software Requirements 26](#_Toc190560057)

[3.2.1 Gathering Requirements 26](#_Toc190560058)

[3.2.2 Architecture Design 27](#_Toc190560059)

[3.2.3 Dataset as a Query 28](#_Toc190560060)

[3.3 Time Frame 28](#_Toc190560061)

[3.3.1 Scrum Methodology 29](#_Toc190560062)

[3.3.2 Kanban Board 29](#_Toc190560063)

[4. **Realisation** 31](#_Toc190560064)

[4.1 Quantization Approach 31](#_Toc190560065)

[4.1.1 BitsandBytes Quantization 31](#_Toc190560066)

[4.1.2 Model Based Quantization 32](#_Toc190560067)

[4.2 Adapter Based Quantization 33](#_Toc190560068)

[4.2.1 Axolotl Fine-Tuning Approach 34](#_Toc190560069)

[4.2.2 Configuration of Llama 34](#_Toc190560070)

[4.2.3 Configuration of Mistral 37](#_Toc190560071)

[4.3 Model Based vs Adapter Based 38](#_Toc190560072)

[4.4 Output Generation Pipeline 39](#_Toc190560073)

[4.4.1 Generation Arguments 39](#_Toc190560074)

[4.4.2 Error Handling Mechanism 40](#_Toc190560075)

[4.5 Evaluation Metrics 41](#_Toc190560076)

[4.5.1 Perplexity Metric 41](#_Toc190560077)

[4.5.2 Coherence Metric 41](#_Toc190560078)

[4.5.3 Power Usage Metric 42](#_Toc190560079)

[4.5.4 Latency Metric 42](#_Toc190560080)

[4.5.5 Memory Usage Metric 43](#_Toc190560081)

[4.6 LLM-as-a-Judge 43](#_Toc190560082)

[4.6.1 Prompt Engineering and Judge Rules 43](#_Toc190560083)

[4.6.2 ELO Scoring Mechanism 45](#_Toc190560084)

[4.7 Front End Implementation 46](#_Toc190560085)

[4.8 MLOps Integration 47](#_Toc190560086)

[4.8.1 MLflow Integration 47](#_Toc190560087)

[4.8.2 LangSmith Integration 47](#_Toc190560088)

[4.9 Results 48](#_Toc190560089)

[5. **Test** **Inference** 59](#_Toc190560098)

[5.1 DeekSeek Integration 59](#_Toc190560099)

[6. **Summary** **and** **Perspectives** 65](#_Toc190560100)

[7. **Acknowledgment** 66](#_Toc190560101)

[8. **Abbreviations** 67](#_Toc190560102)

[9. **References** 69](#_Toc190560103)

[10. **Appendix** 71](#_Toc190560104)

[10.1 Table of Figures 71](#_Toc190560105)

[10.2 Tables 71](#_Toc190560106)

# 1. Introduction

The rapid advancement of Large Language Models (LLMs) has transformed the field of natural language processing (NLP), enabling remarkable progress in tasks such as text generation, summarization, and question answering. However, as these models grow in complexity and scale, evaluating their performance across diverse configurations and use cases remains a significant challenge. While substantial progress has been made in developing LLMs, the lack of standardized and comprehensive evaluation frameworks has created a gray area in assessing their true capabilities. This thesis aims to address this critical gap by proposing an application platform specifically designed for the evaluation of LLMs, integrating advanced quantization techniques and innovative evaluation methodologies.

## The Need for LLM Evaluation

Evaluating LLMs is a multifaceted challenge. Traditional metrics such as perplexity and BLEU scores often fail to capture nuanced aspects of model performance, such as coherence, relevance, and fluency [1]. Additionally, the computational demands of large-scale models make it difficult to deploy and test them in resource-constrained environments. As LLMs are increasingly applied in real-world scenarios—from healthcare to education—the need for robust evaluation frameworks becomes paramount. This thesis seeks to provide a systematic approach to LLM evaluation, enabling researchers and practitioners to assess model performance across varying configurations and tasks.

A key component of this project is the implementation and comparison of quantization techniques. Quantization reduces the size and computational requirements of LLMs by converting high-precision weights into lower-precision representations. This process enhances the efficiency of model deployment and makes it feasible to run LLMs on devices with limited resources, such as edge devices [2]. The platform will explore both adapter-based and model-based quantization approaches. Adapter-based methods involve inserting lightweight modules into pre-trained models, allowing for fine-tuning without modifying the entire architecture [3]. Model-based quantization, on the other hand, focuses on compressing the entire model while preserving its accuracy and fluency. By comparing these techniques, the platform will provide insights into the trade-offs between model size, computational efficiency, and performance.

The platform will also feature advanced querying and text generation capabilities, with a specific focus on summarization tasks. Summarization is a critical application of LLMs, enabling users to distill large volumes of text into concise and meaningful summaries [4]. The platform will allow users to interact with quantized models, generating summaries under various hyperparameter configurations. This functionality will not only demonstrate the practical utility of quantized models but also provide a testbed for evaluating their performance in real-world scenarios. By enabling users to experiment with different settings, the platform will facilitate a deeper understanding of how quantization impacts text generation quality.

A groundbreaking aspect of this project is the introduction of LLM-as-a-Judge, a novel evaluation methodology where an LLM serves as an objective evaluator of other LLMs. This approach leverages the inherent capabilities of LLMs to assess the quality of generated text, providing detailed metrics for coherence, relevance, and overall quality [5]. Unlike traditional evaluation methods, which rely on static benchmarks, LLM-as-a-Judge offers a dynamic and context-aware assessment framework. By scoring

generated texts and providing actionable feedback, this component will enable researchers to fine-tune models and improve their performance iteratively.

Through this work, we aim to address two key research questions: (1) How do different quantization techniques impact the performance of LLMs across quantitative and qualitative metrics? (2) Can the LLM-as-a-Judge framework provide a robust and scalable method for evaluating quantized models? By systematically investigating these questions, this research seeks to contribute to the development of efficient, scalable, and interpretable evaluation methodologies for LLMs.

# 2. Theoretical Background

This chapter provides a theoretical foundation for the core concepts explored in this thesis, focusing on the development, functioning, and evaluation of Large Language Models (LLMs). The chapter begins by introducing the fundamental architecture underlying modern LLMs, namely the Transformer architecture, and then explores decoder-only models such as Mistral and Llama, which play a central role in this research. Following this, the chapter delves into quantization techniques, which are essential for optimizing LLMs for resource-constrained environments and outlines the LLM-as-a-Judge framework, which serves as the evaluation methodology for this thesis.

Through this discussion, the chapter aims to establish a comprehensive understanding of the theoretical underpinnings required to appreciate the methodologies and results presented in subsequent sections. By situating this research within the broader context of LLM development and evaluation, the chapter lays the groundwork for analyzing how quantization impacts model performance and how innovative evaluation frameworks can address existing limitations.

## 2.1 Large Language Models (LLMs)

Large Language Models (LLMs) represent a significant leap in the field of natural language processing (NLP). Built upon advancements in deep learning, these models are capable of understanding and generating text with human-like fluency. Their applications span diverse areas, including automated translation, text summarization, question-answering systems, and conversational AI. The capabilities of LLMs are attributed to their vast parameter counts, extensive pretraining on diverse datasets, and the adoption of cutting-edge architectures like the Transformer.

Unlike traditional statistical or shallow learning models, LLMs leverage contextual embeddings to model relationships between words and sentences. This allows them to understand the nuances of language, such as idioms, metaphors, and contextually dependent meanings. The scalability of these models has led to groundbreaking achievements in NLP, but it also brings challenges, particularly concerning computational demands and resource efficiency.

### 2.1.1 Transformer Architecture Overview

The Transformer architecture, introduced by Vaswani et al. (2017) in the seminal paper "Attention Is All You Need" [6], has fundamentally reshaped the landscape of natural language processing (NLP) and modern large language models (LLMs). By leveraging the innovative self-attention mechanism, the Transformer eliminates the sequential bottleneck inherent in earlier sequence processing methods such as recurrent neural networks (RNNs). This architectural leap has enabled substantial improvements in scalability, efficiency, and the ability to capture long-range dependencies in text.

**Multi-Head Self-Attention Mechanism**

The Transformer's multi-head self-attention mechanism allows the model to simultaneously attend to different parts of an input sequence by generating three matrices: queries (Q), keys (K), and values (V). These matrices are created by linearly transforming the input embeddings, and the scaled dot product between Q and K is used to compute attention scores, which are then used to weight the values (V). This allows the model to learn multiple representation subspaces, capturing a variety of contextual relationships [6].

**Feedforward Neural Networks**

Each attention layer is followed by a position-wise feedforward neural network (FFN). These networks consist of two fully connected layers with a non-linear activation function, often ReLU, sandwiched in between. The FFNs boost the model's potential to learn complex transformations [6].

**Positional Encoding**

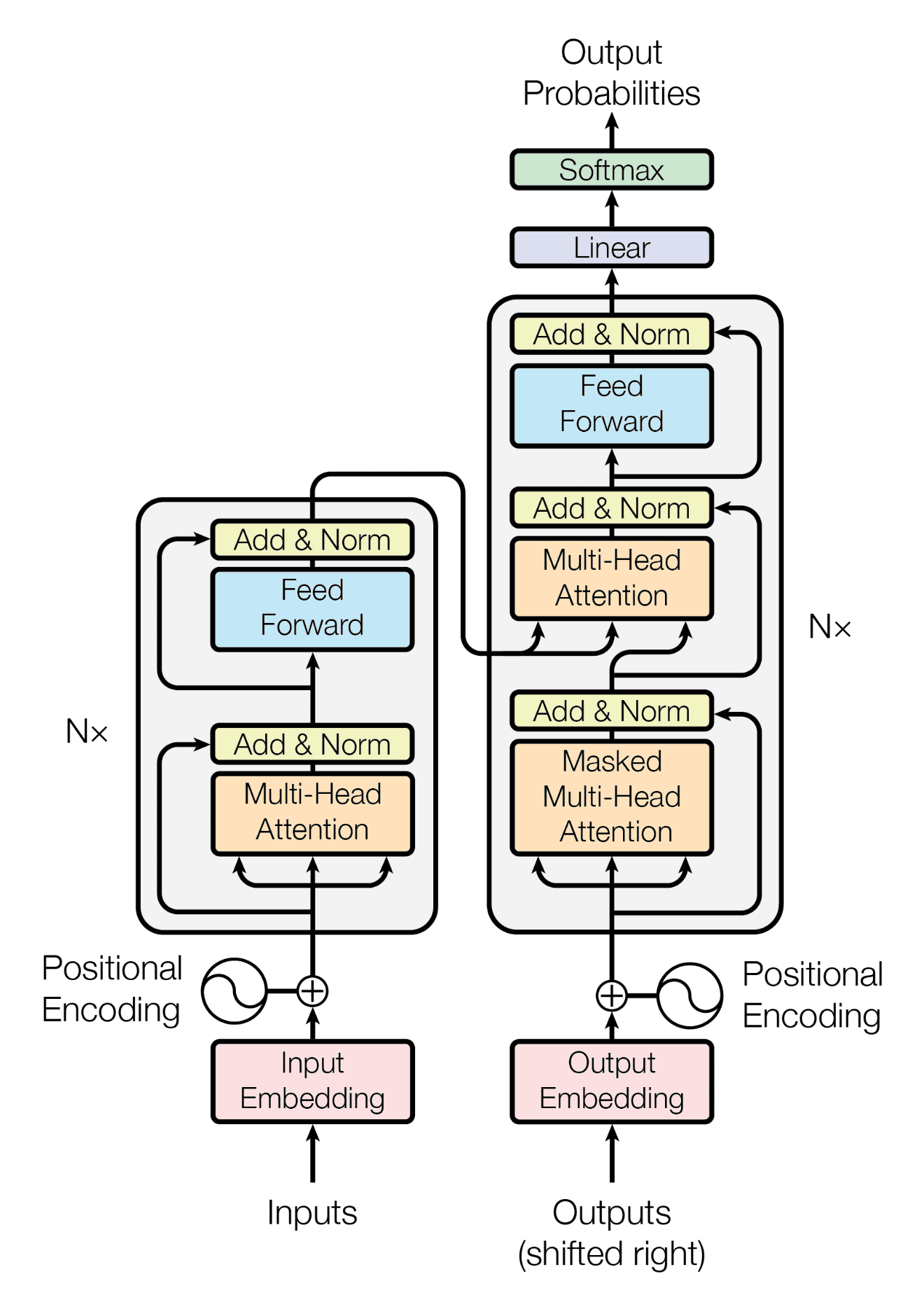
Unlike RNNs, which process input sequentially and inherently capture order information, the Transformer processes input tokens in parallel. To compensate for the lack of sequential modeling, positional encodings are added to the input embeddings. These encodings, often derived using sinusoidal functions, provide each token with a unique position-aware representation, enabling the model to distinguish between differently ordered sequences [6].

Figure 1 Architecture of Transformer Model

**Layer Normalization and Residual Connections**

To stabilize training and enable gradient flow in deep networks, the Transformer integrates residual connections and layer normalization. Residual connections add a layer's input to its output, preserving key features from previous layers, while layer normalization, applied after each sub-layer (feedforward or attention), normalizes the activations to lessen internal covariate shifts and improve convergence during training [6].

**Scalability and Parallelism**

A key advantage of the Transformer architecture is its ability to process sequences in parallel, as opposed to the inherently sequential nature of RNNs. This parallelism significantly reduces training times and makes the architecture highly scalable to large datasets and model sizes. As a result, Transformers have become the foundation for many state-of-the-art models, including BERT, GPT, and T5 [6].

**Encoder-Decoder vs. Decoder-Only Models**

The original Transformer design comprises both an encoder and a decoder component. The encoder processes input sequences to generate context-rich representations, while the decoder generates output sequences based on these representations. However, many LLMs, particularly those focused on text generation tasks, employ a decoder-only configuration. In such models, the decoder is optimized to predict the next token in a sequence, leveraging masked self-attention to ensure that predictions depend only on preceding tokens [6].

**Transformer's Impact on Modern LLMs**

The introduction of Transformers has had profound implications for NLP and beyond. By decoupling sequence processing from the constraints of recurrence, Transformers have enabled unprecedented scalability and performance. They have become the backbone for training massive LLMs, such as GPT-3, Llama, and PaLM, which have demonstrated remarkable capabilities in language understanding, generation, and even multi-modal tasks [6].

### 2.1.2 Decoder-Only Models (e.g., Mistral, Llama)

Decoder-only models are a specialized implementation of the Transformer architecture, designed exclusively for autoregressive tasks like text generation. Their architecture is optimized to predict the next token in a sequence by attending only to previous tokens. This unidirectional focus, combined with scalability and computational efficiency, has established these models as the backbone for many natural language generation applications.

### **Mathematical Intuition for Decoder-Only Models (e.g., Llama, Mistral)**

Decoder-only models, such as Llama and Mistral, are built on the Transformer decoder architecture, which employs causal self-attention and an autoregressive framework. The goal of these models is to predict the next token in a sequence given all preceding tokens . This process is modeled as a conditional probability distribution:

A diagram of a computer

Description automatically generatedwhere is the total number of tokens in the sequence. The causal attention mask ensures that only past tokens contribute to the prediction of , making the process strictly autoregressive [7].

Figure 2 Transformer vs Llama vs Mistral

#### **1. Input Embeddings and Positional Encoding**

The input tokens are first mapped to embeddings , where is the dimensionality of the model. To incorporate sequence order, positional encodings are added to the token embeddings:

where is the initial hidden state [7].

#### **2. Multi-Head Causal Self-Attention**

The core of the decoder-only model lies in its multi-head self-attention mechanism, which allows the model to attend to all previous tokens up to . For each attention head , the attention weights are computed using the scaled dot-product attention [7]:

where:

* : Queries for layer ,
* : Keys for layer ,
* : Values for layer ,
* : Dimensionality of each attention head,
* : Causal mask ensuring that token attends only to .

The output of the attention mechanism for each head is computed as:

The outputs from all heads are concatenated and projected back to the model dimension:

where is the number of attention heads and is a learnable projection matrix [7].

#### **3. Feedforward Networks**

After the attention mechanism, each token representation passes through a position-wise feedforward network (FFN):

where:

* : Weight matrices,
* : Bias terms.

This network introduces non-linearity and further refines the token representations [7].

#### **4. Final Output and Token Prediction**

The final output of the decoder is a sequence of token representations , where is the number of layers in the model. These representations are projected onto the vocabulary space using a linear layer :

The token with the highest probability is selected as the next token [7].

#### **5. Model-Specific Features for Llama and Mistral**

1. **Llama**:  
   Llama employs optimizations in pretraining, such as curated datasets and longer context lengths, to improve efficiency. Its mathematical intuition follows the general decoder-only model process, but Llama uses highly optimized training techniques, such as mixed-precision training with FP16 and bfloat16, to reduce memory usage while maintaining precision [8].
2. **Mistral**:  
   Mistral focuses on real-time inference, leveraging quantization techniques like INT8 and INT4 for parameter compression. The attention mechanism and FFN layers are carefully optimized for faster computation, ensuring minimal latency without compromising accuracy [9].

## 2.2 Quantization Techniques

Large Language Models (LLMs), such as GPT, Llama, and Mistral, are characterized by billions of parameters, which significantly contribute to their exceptional performance across a variety of NLP tasks. However, the memory and computational demands of these models make them challenge to deploy on resource-constrained environments such as mobile devices, edge systems, or energy-efficient servers. Model-based quantization emerges as an effective solution to this problem by compressing the model’s parameters and activations into a lower numerical precision format, thereby reducing memory usage and improving inference speed [10].

Quantization is not just about reducing numerical precision—it must balance computational efficiency with the preservation of model accuracy. Model-based quantization achieves this by quantizing the entire model, including weights and activations, into fixed-point representations (e.g., INT8 or INT4), without requiring modifications to the architecture or re-training from scratch [11].

### 2.2.1 Model Based Quantization

At its core, model-based quantization is the process of approximating high-precision floating-point parameters (e.g., 32-bit FP32) with low-precision representations (e.g., 8-bit INT8). The goal is to replace costly floating-point arithmetic with integer arithmetic, which is computationally more efficient and hardware-friendly [8].

#### **Quantization Process**

1. **Quantization Representation**: Let be a weight matrix in the model, where and are the dimensions of the layer. In the original FP32 representation, each element is represented by a 32-bit floating-point number.

* In model-based quantization, is mapped to a quantized version using a scaling factor and an offset , such that:
* where:
  + is the quantized integer value.
  + is the scaling factor that maps the FP32 range to the integer range.
  + is an optional zero-point offset used for symmetric quantization [12].

1. **Range Mapping**: The scaling factor is computed as:

* where is the bit-width of the quantized representation (e.g., for INT8).
* The weights are then clipped to lie within the quantization range :
* ensuring numerical stability during computation [12].

1. **Dequantization**: At runtime, the quantized weights are dequantized back to approximate the original floating-point values:

* This step allows the quantized model to operate on integer arithmetic during inference while maintaining compatibility with floating-point computations [12].

#### **Error Introduction**

Quantization introduces an approximation error defined as:

where is the original weight, and is its quantized approximation. This error must be minimized to preserve model accuracy [12].

### **Quantization Modes**

Model-based quantization can be categorized into the following modes:

1. **Uniform Quantization**: In this approach, the scaling factor is constant across all parameters in the model, ensuring a uniform mapping between floating-point and integer values. Uniform quantization is computationally efficient and well-suited for hardware implementations [13].
2. **Non-Uniform Quantization**: Here, the scaling factor varies across different layers or parameter groups. Non-uniform quantization is typically used for models with large variations in weight distributions, as it adapts better to the specific characteristics of each layer [3].
3. **Post-Training Quantization (PTQ)**: PTQ is applied to pre-trained models without modifying the training pipeline. It involves:
   * Quantizing weights and activations after training.
   * Evaluating the quantized model on a calibration dataset to fine-tune the scaling factors.
4. **Quantization-Aware Training (QAT)**: In QAT, quantization is simulated during training by applying quantization and dequantization operations in the forward pass. This enables the model to learn to compensate for quantization errors.

#### **Quantized Inference**

During inference, the computational cost of operations such as matrix multiplication is significantly reduced by using quantized weights and activations. Let represent the input matrix, and represent the weight matrix. The standard matrix multiplication in FP32 is given by:

In quantized inference, both and are quantized as and :

The scaling factors and are used to dequantize the result:

This enables efficient integer-based computation, with dequantization performed only at the output stage [12] .

#### **Error Propagation in Matrix Multiplication**

The error introduced in quantized matrix multiplication can be expressed as:

Minimizing is critical to ensure that the quantized model's predictions remain consistent with the original model [12].

### **Advantages of Model-Based Quantization**

1. **Memory Efficiency**: Quantized models require significantly less storage space. For example, converting a model from FP32 to INT8 reduces memory usage by 75%.
2. **Inference Speed**: Integer arithmetic is computationally faster than floating-point arithmetic, leading to reduced latency during inference.
3. **Hardware Compatibility**: Modern accelerators, such as NVIDIA TensorRT and Google’s Edge TPU, are optimized for low-precision computations, making quantized models highly deployable [14].

### **Challenges**

1. **Accuracy Drop**: Quantization may degrade model performance, particularly for sensitive tasks like text generation or translation.
2. **Quantization Granularity**: Determining the optimal granularity (e.g., per-layer or per-channel quantization) is non-trivial.
3. **Dynamic Ranges**: Activations in LLMs may have highly dynamic ranges, requiring careful calibration to avoid underflow or overflow errors [15].

### 2.2.2 Adapter Based Quantization

By focusing on quantizing only the adapters rather than the entire model, adapter-based quantization achieves a balance between efficiency and accuracy. This technique is particularly relevant in **Parameter-Efficient Fine-Tuning (PEFT)** approaches like **LoRA (Low-Rank Adaptation)**, where additional trainable parameters are introduced in a structured way.

### **Theory Behind Adapter-Based Quantization**

In adapter-based quantization, task-specific modifications are made by introducing small, quantized modules (adapters) into the pre-trained model while keeping the original weights frozen. These adapters are trained to capture the task-specific information required for fine-tuning.

#### **Adapter-Based Model Augmentation**

Let represent the hidden states of the model, where:

* is the sequence length,
* is the hidden dimension.

An adapter is introduced as an additional transformation applied to the hidden states:

where:

* : Down-projection matrix,
* : Up-projection matrix,
* : Rank of the low-dimensional space (where ),
* : Non-linear activation function, such as ReLU or GELU.

The use of and reduces the number of parameters, as is much smaller than . This makes the adapter lightweight and suitable for quantization [12].

#### **Quantization of Adapters**

The adapter weights and are quantized to lower-precision formats (e.g., INT8) to further reduce memory usage. Quantization involves:

1. **Mapping weights to a quantized range**:

* where:
  + : Scaling factor for quantization.

1. **Dequantization during inference**:

This quantization process applies separately to and , ensuring that the low-rank structure of the adapter is preserved [12].

### **PEFT with LoRA (Low-Rank Adaptation)**

**Low-Rank Adaptation (LoRA)** is a specific instance of PEFT designed to further reduce the number of trainable parameters by introducing low-rank matrices into the model's architecture [11]. Instead of modifying the entire model, LoRA injects trainable rank-decomposed updates directly into specific layers, such as attention or feedforward layers, without altering the base model’s weights.

Figure 3 PEFT LoRA Architecture

#### **LoRA Theory**

In LoRA, the weight matrix of a transformer layer is augmented with low-rank updates during training [13]:

where is decomposed as:

and:

* : Trainable low-rank matrix,
* : Trainable low-rank matrix,
* : The rank of the decomposition.

#### **Quantization in LoRA**

In quantized LoRA, the low-rank matrices and are quantized during inference. This reduces memory usage without compromising the efficiency of the rank decomposition:

1. Quantized matrices:
2. Dequantized inference:

By quantizing and , the memory and computation requirements of LoRA are further reduced, enabling its deployment on edge devices or GPUs with limited memory [8].

### **Advantages of Adapter-Based Quantization with LoRA**

1. **Parameter Efficiency**: Only the adapter parameters (or low-rank matrices in LoRA) are trained and quantized, significantly reducing the computational and memory requirements.
2. **Task-Specific Fine-Tuning**: Adapters enable customization of pre-trained models for specific tasks without retraining the entire model.
3. **Scalability**: LoRA’s low-rank decomposition ensures that even large LLMs can be fine-tuned with limited resources.
4. **Quantization-Friendly Design**: The smaller size of adapters and low-rank matrices makes them inherently suitable for quantization.

### **Challenges in Adapter-Based Quantization**

1. **Precision Loss**: Quantizing adapters can lead to a loss of precision in task-specific representations, affecting performance on sensitive tasks.
2. **Hardware Support**: Deploying quantized adapters require hardware and libraries optimized for low-precision computations.
3. **Rank Selection**: Determining the optimal rank for LoRA matrices involves trade-offs between accuracy and efficiency.

### 2.2.3 Evaluation Metrics for Quantized Models

Quantized models have revolutionized the deployment of machine learning systems by significantly reducing computational and memory requirements. However, evaluating the performance of quantized models is a critical step to ensure that the benefits of quantization do not come at the expense of unacceptable performance degradation. The evaluation process involves both **quantitative metrics** (e.g., perplexity, latency, memory usage) and qualitative metrics (e.g., coherence, hallucination rate). These metrics help quantify the trade-offs between computational efficiency and accuracy, guiding the optimization of quantized models for real-world applications [10].

Quantization can lead to performance degradation due to the approximation introduced during the conversion of high-precision weights and activations to lower precision. Evaluation metrics are, therefore, indispensable for understanding the impact of quantization on various tasks, particularly for Large Language Models (LLMs) such as GPT, Llama, and Mistral [11].

### LLM Evaluation Metrics: Benchmarks, Protocols & Best Practices

Figure 4 LLM Evaluation Metrics

### **Types of Evaluation Metrics**

Evaluation metrics for quantized models are broadly classified into **quantitative** and **qualitative** categories.

### **1. Quantitative Metrics**

#### **1.1 Perplexity ()**

Perplexity is a standard metric for evaluating language models. It measures how well the model predicts a sequence of tokens and is defined as the inverse probability of the test data normalized by the number of tokens. For a sequence of tokens , the perplexity is given by:

where:

* is the total number of tokens in the sequence,
* is the conditional probability of the -th token given its context.

Lower perplexity values indicate that the model is better at predicting the sequence. For quantized models, the perplexity is evaluated to ensure that the approximation error introduced by quantization does not significantly degrade the model’s predictive capabilities [8].

#### **1.2 Latency**

Latency measures the time taken by the model to generate a response or perform inference. For quantized models, the primary goal is to reduce latency without compromising accuracy. Latency is mathematically defined as:

where:

* Total Inference Time refers to the cumulative time taken for all predictions,
* Number of Inference Steps is the total number of sequences processed.

Quantized models typically achieve lower latency due to the use of efficient integer arithmetic. This makes them suitable for deployment in real-time systems where low response times are critical [6].

#### **1.3 Memory Usage**

Memory usage evaluates the storage requirements of a model. For quantized models, memory usage is significantly reduced due to the lower bit-width representation of weights and activations. The memory footprint of a quantized model is calculated as:

where:

* : Number of layers in the model,
* : Number of weights in the -th layer,
* : Dimensionality of the weights in the -th layer,
* : Bit-width of the quantized representation (e.g., for INT8).

Memory usage directly correlates with the quantization bit-width, making this metric a critical factor for resource-constrained environments [13].

#### **1.4 Energy Efficiency**

Energy efficiency measures the power consumption of a model during inference. Quantized models often consume less power due to the reduced complexity of integer computations compared to floating-point operations. Energy efficiency is mathematically expressed as:

where energy consumption is measured in joules. This metric is crucial for deploying LLMs in energy-sensitive applications, such as IoT devices and mobile systems [14].

### **2. Qualitative Metrics**

#### **2.1 Coherence**

Coherence evaluates the logical consistency of the model’s generated text. While quantitative metrics such as perplexity focus on probabilities, coherence assesses the semantic and contextual relevance of the output. Coherence is typically evaluated using human judgment or automated metrics like BLEU, ROUGE, or METEOR.

Mathematically, coherence can be approximated using embedding similarity:

where:

* : Embedding of the generated token,
* : Embedding of the reference token,
* : Cosine similarity between the embeddings.

Higher coherence values indicate better-quality output, which is critical for applications like summarization and dialogue generation [16].

#### **2.2 Hallucination Rate**

Hallucination rate measures the frequency of outputs containing fabricated or nonsensical information. For quantized models, hallucination can occur due to precision loss in critical parameters. Hallucination rate is defined as:

where incorrect tokens are those that deviate from the ground truth or reference output. Lower hallucination rates are desirable, especially for tasks requiring factual correctness, such as medical or legal document generation [4].

#### **2.3 ELO Score**

ELO scoring is inspired by the rating system used in competitive games like chess. In the context of quantized models, ELO scores are used to compare the performance of different models on the same task. Using an **LLM-as-a-Judge framework**, models are ranked based on pairwise comparisons of their outputs.

Let and be the ELO ratings of two models and . After evaluating a pairwise comparison, the ELO ratings are updated as:

where:

* : Scaling factor,
* and : Actual scores (e.g., 1 for a win, 0 for a loss, 0.5 for a tie),
* and : Expected scores, computed as:

The ELO scoring system provides a robust way to evaluate the qualitative performance of quantized models in a competitive framework [17].

### **Importance of Metrics for Quantized Models**

Evaluation metrics are indispensable for:

1. **Balancing Efficiency and Accuracy**: Ensuring that the performance degradation introduced by quantization is acceptable.
2. **Task-Specific Optimization**: Tailoring the evaluation process to the requirements of specific tasks, such as text generation or summarization.
3. **Hardware Integration**: Guiding the deployment of quantized models on hardware platforms optimized for low-precision arithmetic.

## 2.3 LLM-as-a-Judge Framework

As the size and complexity of Large Language Models (LLMs) grow, evaluating their performance across a variety of tasks becomes increasingly challenging. Traditional evaluation metrics such as perplexity and BLEU provide quantitative insights but often fail to capture the nuanced quality of model outputs. To address these limitations, the **LLM-as-a-Judge (LLM-J)** framework has emerged as a novel evaluation methodology. This framework leverages the capabilities of an LLM to act as an impartial evaluator, comparing and scoring outputs generated by other LLMs [8].

The LLM-as-a-Judge framework introduces a dynamic, task-agnostic evaluation process that assesses model outputs based on coherence, relevance, factual accuracy, and contextual understanding. Inspired by the **ELO rating system** commonly used in games like chess, this framework ranks LLMs based on pairwise comparisons, offering a robust and interpretable way to evaluate model performance [6].

### 2.3.1 Concept and Relevance

**Concept**

The **LLM-as-a-Judge (LLM-J)** framework is an innovative evaluation methodology designed to assess the performance of large language models (LLMs). Unlike traditional metrics such as perplexity, BLEU, and ROUGE, which focus on token-level matches or probabilities, the LLM-J framework leverages the reasoning and generative capabilities of a pre-trained LLM to act as a neutral evaluator. This approach enables the evaluation of higher-order text qualities, such as semantic coherence, contextual relevance, and fluency, which are often overlooked by traditional metrics.

In this framework:

1. **Input Prompts**: The evaluator LLM receives the same prompt as the candidate models, along with their respective outputs, to make a judgment.
2. **Evaluation Task**: The evaluator is tasked with selecting the better output or scoring both outputs based on predefined criteria such as accuracy, relevance, coherence, and overall quality.
3. **Decision Output**: The judge provides a decision, often in the form of a ranking, score, or qualitative explanation for its choice.

The LLM-as-a-Judge framework introduces flexibility and richness into evaluation, enabling more human-like assessments at scale.

**Relevance**

The relevance of the LLM-as-a-Judge framework lies in its ability to address critical challenges in LLM evaluation. Traditional metrics often fail to capture the nuanced qualities of text generated by LLMs, leading to evaluations that may not align with human judgment. The LLM-J framework bridges this gap by providing a scalable, dynamic, and task-agnostic approach to evaluating model outputs.

**Why Traditional Metrics Fall Short**

1. **Surface-Level Focus**:
   * Traditional metrics like BLEU and ROUGE measure similarity based on token overlap, which may not correlate with human perception of quality. For example:
     + BLEU may score high for "The cat sat on the mat" and "A cat was sitting on the rug" due to overlapping words, but it cannot determine which sentence is more coherent or contextually appropriate.
     + Conversely, BLEU may penalize creative rephrasing, even if the output aligns better with human expectations [16].
2. **Limited Task Adaptability**:
   * Metrics like perplexity evaluate language models based on their ability to predict token probabilities, which is more relevant for syntactic tasks (e.g., next-word prediction) but inadequate for evaluating semantic understanding or factual correctness in tasks like summarization or reasoning [4].
3. **High Human Involvement**:
   * Human evaluation, while considered the gold standard, is expensive, time-consuming, and infeasible at scale. The LLM-as-a-Judge framework replicates human evaluation by leveraging LLMs’ reasoning capabilities to perform judgments quickly and consistently [6].

**Key Advantages of LLM-as-a-Judge**

1. **Task Agnosticism**:
   * The LLM-as-a-Judge framework is not tied to a specific task. By modifying the prompts provided to the evaluator model, it can adapt to tasks such as text summarization, machine translation, and question answering. For example:
     + **Task**: Evaluate two summaries for the text "The Earth rotates on its axis, causing day and night."
     + **Candidate Outputs**:
       - Model A: "Day and night are caused by Earth's rotation."
       - Model B: "The Earth's rotation leads to day and night cycles."
     + **Judgment**: The LLM-J selects Model B as the better output because it is more complete and includes the term "cycles," which provides additional context.
2. **Dynamic Comparisons**:
   * The framework allows pairwise comparison of models, making it possible to rank multiple models based on their relative strengths and weaknesses.
3. **Semantic Evaluation**:
   * Unlike traditional metrics, LLM-J evaluates text for coherence, factual accuracy, and contextual alignment. For example:
     + **Input**: "Why is the sky blue?"
     + **Model Outputs**:
       - Model A: "The sky is blue because of light scattering."
       - Model B: "The blue color of the sky is due to Rayleigh scattering of sunlight in the atmosphere."
     + **Judgment**: The LLM-J would select Model B for its scientific precision and clarity.
4. **Scalability**:
   * The LLM-as-a-Judge framework scales effortlessly across datasets, tasks, and languages. This is particularly valuable in multilingual NLP systems, where traditional metrics are often biased toward specific languages [18].

**Example**

**Evaluating Summarization Models**

* **Task**: Summarize the following text: "Quantum mechanics explains the behavior of particles at microscopic scales, introducing concepts such as superposition and entanglement."
* **Outputs**:
  + Model A: "Quantum mechanics describes superposition and entanglement."
  + Model B: "Quantum mechanics explains particle behavior at microscopic scales."
* **LLM-J Prompt**: "Evaluate which summary better captures the essence of the original text. Provide a score for both outputs on a scale of 1-10."
* **Judgment**: The LLM-J assigns a score of 8 to Model B for including "particle behavior at microscopic scales," which is central to the source text, and 6 to Model A for its brevity.

**Machine Translation**

* **Task**: Translate the English sentence "She is studying physics at the university" into French.
* **Outputs**:
  + Model A: "Elle étudie la physique à l'université."
  + Model B: "Elle apprend la physique à l'université."
* **LLM-J Prompt**: "Compare the translations for grammatical correctness and fidelity to the English sentence."
* **Judgment**: The LLM-J selects Model A as more accurate, as "étudie" is a better translation for "studying" than "apprend" (learning).

### 2.3.2 ELO Score Evaluation Mechanism

The ELO score evaluation mechanism is a widely adopted comparative ranking system, originally developed for chess by Arpad Elo in 1960. Its applicability extends far beyond games, into ranking systems for competitive environments like online gaming, sports, and—more recently—machine learning models. In the context of Large Language Models (LLMs), the ELO score evaluation mechanism is an effective method for ranking models based on their relative performance in generating high-quality text outputs.

The ELO framework enables **dynamic comparative evaluation**, where models are ranked iteratively based on pairwise comparisons of their outputs. Instead of relying on absolute scores, the ELO system provides a robust way to evaluate models relative to one another, making it especially suited for scenarios where traditional metrics like BLEU or perplexity fail to capture nuanced differences in output quality [19].

**Concept of ELO Scoring**

The ELO scoring system is a method for ranking competing entities—in this case, Large Language Models (LLMs)—based on their relative performance in pairwise comparisons. Each model begins with an initial rating, and its score is dynamically adjusted based on the outcomes of evaluations against other models. The ELO system was originally developed for chess by Arpad Elo [19] and has since been adapted for various competitive environments, including the evaluation of LLMs.

#### **Steps in ELO Evaluation**

1. **Initialization**: Each model is assigned an initial ELO rating , typically 1000 for uniformity. This represents the baseline performance of the model.
2. **Pairwise Comparisons**: For a given input prompt, the outputs of two models and are compared. The comparison can be conducted either by:
   * **Human evaluators**, who judge which output is better, or
   * **LLM-as-a-Judge**, which evaluates the outputs based on predefined criteria like coherence, factual correctness, or fluency [8].
3. **Expected Scores**: The expected score for when compared to is calculated using the following formula [17] [19]:

* where:
  + : Current ELO rating of model ,
  + : Current ELO rating of model ,
  + : Probability that will win against ,
  + : Probability that will win against .

1. **Actual Scores**: After the comparison, the actual score for is assigned as [19] [17]:
   * : If wins,
   * : If loses,
   * : If the comparison results in a tie.
2. **Rating Update**: Based on the actual and expected scores, the ELO rating for is updated as:

* where:
  + : Updated ELO rating for ,
  + : Scaling factor (commonly set between 10 and 40) that determines the sensitivity of the rating system,
  + : Difference between actual and expected scores, reflecting the performance of relative to its expected outcome.

Similarly, the rating for is updated as [19] [17]:

1. **Iterative Process**: The pairwise comparisons and rating updates are repeated across multiple prompts and models. Over time, the rankings stabilize, providing a robust evaluation of the models' relative performance.

### **Mathematical Intuition**

#### **1. Expected Score**

The expected score is a sigmoid function of the difference in ratings between two models:

* When , , indicating an equal probability of winning.
* When , , meaning is more likely to win.
* When , , meaning is less likely to win.

The use of a logarithmic scale ensures that large rating differences result in proportionally smaller increases in expected probability, making the system resistant to extreme outliers [19].

#### **2. Rating Update**

The term determines the adjustment to the model's rating:

* **If**  (actual performance exceeds expectations), is increased.
* **If**  (actual performance falls below expectations), is decreased.
* **If**  (performance matches expectations), remains unchanged [19].

The scaling factor controls the magnitude of the rating change:

* A higher value allows ratings to adjust more quickly, useful in early iterations.
* A lower value stabilizes the ratings over time, making them less volatile.

## 2.4 Streamlit for Visualization and Evaluation

Streamlit is an open-source Python library designed to simplify the creation of interactive web applications for data visualization and machine learning tasks. Its ease of use, rapid prototyping capabilities, and seamless integration with Python-based data science workflows make it an ideal tool for visualizing and evaluating the performance of Large Language Models (LLMs) in this study. This section highlights the key features of Streamlit and its relevance to the evaluation framework.

### 2.4.1 Key Features

1. **Rapid Prototyping**

Streamlit allows developers to create interactive web applications with minimal code, making it highly efficient for prototyping. Its intuitive API enables the quick integration of data visualizations, user inputs, and dynamic updates, which are essential for visualizing LLM evaluation results.

1. **Interactive Widgets**

Streamlit provides a wide range of built-in widgets, such as sliders, dropdowns, and buttons, which enable users to interact with the application dynamically. These widgets can be used to adjust parameters, filter results, or compare model outputs, enhancing the user experience during LLM evaluation.

1. **Customizable Layouts**

Streamlit offers flexible layout options, including columns, expanders, and containers, which allow developers to organize content effectively. This is crucial for presenting complex evaluation metrics and visualizations in a structured and user-friendly manner.

1. **Deployment and Sharing**  
   Streamlit applications can be easily deployed and shared via the Streamlit Cloud platform or other hosting services. This facilitates collaboration among researchers and stakeholders, enabling them to access and interact with the evaluation framework remotely.

## 2.5 Literature Survey

### 2.5.1 Literature 1

In reviewing the paper **“A Comprehensive Evaluation of Quantization Strategies for Large Language Models”,** I found several relevant insights for my research on "Quantization Techniques for Transformer Decoder-Only Models: Evaluation and Application in NLP Tasks." The paper presents a structured evaluation framework that assesses quantized models across three critical dimensions: knowledge & capacity, alignment, and efficiency. This framework can be adapted to evaluate quantization in Transformer decoder-only models. Notably, the authors demonstrate that 4-bit quantization retains comparable performance to full-precision models, while further reductions (e.g., 2-bit quantization) introduce performance trade-offs. The use of perplexity as a performance indicator for quantized models also offers a metric applicable to my study of language generation tasks. Additionally, the paper highlights practical challenges, such as the balance between memory savings and inference speed, providing insights that can inform the deployment of quantized Transformer models in real-world NLP applications. This literature will serve as a foundation for evaluating and refining quantization techniques specific to decoder-only architectures [20].

### 2.5.2 Literature 2

In reviewing the paper **"Evaluating Quantized Large Language Models",** I found significant insights that directly support my research on "Quantization Techniques for Transformer Decoder-Only Models: Evaluation and Application in NLP Tasks." The paper provides a comprehensive evaluation of various post-training quantization (PTQ) methods, particularly focusing on weight-only, weight-activation, and KV cache quantization for large language models (LLMs). One of the key findings is that **4-bit quantization (W4, KV4)** strikes an optimal balance between memory efficiency and performance, with less than 2% degradation in accuracy across a wide range of NLP tasks. This is crucial for scaling transformer decoder models, such as Llama2, without compromising on performance [21].

Additionally, the analysis shows that more severe quantization, like W3 or KV3, significantly reduces performance, particularly for smaller devices as Llama2-13B. Even at W3, larger versions like Llama2-70B maintain competitive performance due to their superior tolerance for lower-bit quantization. This implies that a key factor in assessing the efficacy of quantization techniques is model size.

### 2.5.3 Literature 3

In reviewing the document **“Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”**, the findings highlight the limitations of traditional benchmarks, such as MMLU and HELM, in evaluating large language models' (LLMs) alignment with human preferences. The authors argue that these metrics fail to capture the nuances of user satisfaction in open-ended tasks. To address this, they propose using GPT-4 as an LLM judge for evaluating other LLMs in conversational tasks. With over 80% agreement between GPT-4 and human raters, the use of LLMs as surrogate judges is validated as a scalable, consistent, and effective method of evaluation. Furthermore, two new evaluation tools are introduced: MT-Bench, a benchmark with 80 high-quality multi-turn questions, and Chatbot Arena, a crowdsourced platform for head-to-head chatbot comparisons. These tools focus on real-world user preferences, complementing traditional metrics by capturing practical conversational performance [22].

The report also lists a number of biases that may compromise the credibility of LLM judges, including positional and verbosity biases. It has been demonstrated that mitigation techniques like location switching and few-shot learning increase evaluation consistency. High technical scores do not always transfer into user happiness, according to findings from MT-Bench and Chatbot Arena, which show differences between benchmark results and actual user preferences. In open-ended tasks, fine-tuned models such as Vicuna-13B perform better than base models, highlighting the necessity of user-aligned benchmarks to evaluate practical performance. In order to bridge the gap between LLM capabilities and practical usability, the authors ultimately support a hybrid assessment approach that blends conventional benchmarks with user-preference methods to thoroughly evaluate both technical skills and alignment with human expectations.

# 3. Requirement Analysis

## 3.1 General Objectives

This project aims to address one of the most critical challenges in the domain of Large Language Models: evaluation. While significant advancements have been made in the development of LLMs, assessing their performance across varying configurations remains a gray area. To bridge this gap, this project proposes building an application platform specifically designed for the evaluation of LLMs, incorporating cutting-edge quantization techniques and innovative evaluation frameworks.

The system is composed of three key components:

1. **Quantization:** Implementing and comparing quantization techniques, including both adapter-based and model-based approaches. This allows for optimized performance by reducing the size and computational requirements of LLMs while preserving accuracy and fluency.
2. **Querying and Text Generation for Summarization:** The platform enables efficient querying and text generation capabilities for quantized models, with a specific focus on summarization tasks. Users can interact with the models to generate concise and accurate summaries of input text, offering an opportunity to evaluate model behaviour under various hyperparameter configurations.
3. **Evaluation using LLM-as-a-Judge:** Introducing a novel evaluation methodology where an LLM acts as an objective evaluator (LLM-as-a-Judge) to score the performance of other LLMs. This component focuses on scoring generated texts and providing detailed metrics for assessing coherence, relevance, and overall quality.

## 3.2 Software Requirements

The platform requires a robust and scalable software stack to support quantization, querying, text generation, and evaluation functionalities efficiently. These requirements focus on selecting suitable frameworks, tools, and methodologies to achieve the project's goals.

### 3.2.1 Gathering Requirements

The requirements for the platform were determined through some of the references [20] [23],a review of the latest advancements in LLM evaluation, and the need for an efficient, scalable system. The following outlines the hardware and software requirements for the platform:

**Hardware Requirements:**

1. Linux GPU Server:
   * Model: LS40 with high-performance GPUs.
   * GPU Configuration: Minimum of 4 GPUs, each with 24GB VRAM, supporting CUDA-enabled acceleration for model quantization and inference tasks.
   * Memory: At least 256GB RAM to handle large datasets and multiple concurrent users efficiently.
   * Processor: Multi-core processors (e.g., AMD EPYC or Intel Xeon) for parallel processing and efficient workload management.

**Software Requirements:**

1. Operating System:
   * Linux Distribution: Ubuntu 20.04 LTS or higher for compatibility with CUDA and ML libraries.
2. Programming Environment:
   * Python: Version 3.8 or higher for implementing core logic and integration with libraries.
3. Machine Learning Libraries:
   * PyTorch: For managing model quantization, training, and inference tasks.
   * Hugging Face Transformers: For seamless integration with pre-trained LLMs like and task-specific pipelines.
   * Axolotl: For fine-tuning and managing quantized models efficiently.
   * LangChain: To handle LLM-based reasoning, workflows, and integrations for enhanced evaluation processes.
4. GPU Acceleration:
   * CUDA Toolkit: Version 11.3 or higher for utilizing GPU capabilities in model computations.
   * cuDNN: Compatible with the CUDA version to optimize deep learning operations.
5. Evaluation and Experimentation Tools:
   * MLflow: For tracking experiments, managing models, and logging evaluation metrics.
6. Web Frameworks:
   * Streamlit: For creating an interactive user interface to query models, view evaluation results, and adjust configurations dynamically.
7. Project Management:
   * Awork: To manage Scrum and Kanban boards, enabling effective task planning, sprint management, and collaboration among team members.

### 3.2.2 Architecture Design

*A diagram of a flowchart

Description automatically generated*

Figure 5 High Level Architecture of the Project

The architecture design of the platform is centred around a modular and scalable pipeline for evaluating large language models (LLMs) using both adapter-based and model-based quantization techniques. The workflow begins with user queries, which are processed through either adapter-based quantization, leveraging PEFT LoRA configurations at 16/8/4-bit precision, or model-based quantization applied directly to Llama3 and Mistral models. These quantized models are queried to generate outputs, which are monitored using MLflow and LangSmith for real-time tracking of performance and metrics.

The generated outputs are then evaluated through a dedicated Evaluation Metrics module, which measures key performance parameters such as coherence, memory usage, perplexity, power consumption, and query time. The evaluated results are subsequently passed to the LLM-as-a-Judge framework, where the outputs are scored based on their overall quality and task fulfilment using an ELO scoring system.

This iterative evaluation not only provides granular feedback for individual queries but also establishes a robust benchmarking system for LLM performance across various quantization techniques. The architecture ensures seamless integration of quantization, query processing, and evaluation, leveraging modern tools like Kedro pipelines, MLflow, and LangSmith to maintain scalability, efficiency, and real-time monitoring throughout the workflow.

### 3.2.3 Dataset as a Query

This project utilizes a specific task type, summarization, as a benchmark for evaluating large language models. The dataset employed is esdurmus/wiki\_lingua, a multilingual dataset tailored to evaluate summarization capabilities, with a specific focus on the German language. This dataset serves as the primary query input for the LLM models, enabling systematic evaluation under a robust testing framework.

Rather than querying the models one input at a time, this approach leverages the dataset as a whole, ensuring consistency, scalability, and the ability to test the models across a diverse range of scenarios. By employing a dataset-driven query style, the project creates a more comprehensive and resilient evaluation environment, mimicking real-world conditions and ensuring more reliable benchmarking results.

## 3.3 Time Frame

In this section, the time frames for the various segments of the projects are presented.

|  |  |  |
| --- | --- | --- |
| **Sprint** | **Month** | **Tasks to Do** |
| **1** | **October** | - Research preparation and literature collection. |
|  |  | - Summarize insights from the literature on quantization techniques and NLP models. |
|  |  | - Set up the basic environment with GPU-enabled configuration and pipeline framework. |
|  |  | - Create the Kedro pipeline setup and test its environment. |
| **2** | **November** | - Integrate the WikiLingua dataset as a query pipeline. |
|  |  | - Develop model-based quantization techniques. |
|  |  | - Implement adapter-based quantization. |
|  |  | - Fine-tune models for adapter-based quantization using PEFT LoRA. |
| **3** | **December** | - Present the progress to the supervisor and gather feedback. |
|  |  | - Build the evaluation framework, including ELO computation, LLM-as-a-Judge, and performance metrics. |
|  |  | - Begin designing the architecture for the frontend Streamlit integration. |
| **4** | **January** | - Perform testing in the console environment. |
|  |  | - Develop the frontend interface using Streamlit. |
|  |  | - Present the progress to the professor and collect feedback. |
| **5** | **February** | - Show the refined system to the professor and gather additional feedback. |
|  |  | - Work on the feedback provided. |
|  |  | - Begin drafting the thesis document. |
|  |  | - Continue thesis document writing. |
| **6** | **March** | - Conduct a final review of the system and polish its features. |
|  |  | - Finalize all sections of the thesis document. |
|  |  | - Submit the thesis and prepare for the defence. |

Table 1 Time Frame Plan

### 3.3.1 Scrum Methodology

The project adopts the Scrum methodology, an agile framework for managing and completing complex tasks efficiently. Scrum ensures iterative development, fostering collaboration, adaptability, and regular feedback. Each sprint spans one month, with clearly defined goals, tasks, and deliverables. Regular stand-up meetings are conducted to monitor progress, address challenges, and realign priorities. Key elements of the methodology include:

* **Sprint Planning:** Defining sprint goals and breaking down tasks into manageable units.
* **Bi-Weekly Stand-ups:** Short meetings to track progress and address blockers.
* **Sprint Reviews:** Demonstrating completed work to supervisor for feedback.
* **Sprint Retrospectives:** Reflecting on the sprint to identify areas for improvement.

### 3.3.2 Kanban Board

A Kanban board is utilized to visualize and manage the workflow of the project. The board provides a clear overview of tasks, organized into the following columns:

1. **Backlog:** Tasks that are yet to be started, including research activities, pipeline setup, and model development.
2. **To Do:** Tasks planned for the current sprint, ready to be executed.
3. **In Progress:** Tasks actively being worked on, such as integrating the dataset, implementing quantization, or building the evaluation framework.
4. **Review:** Completed tasks awaiting feedback from the supervisor or professor.
5. **A screenshot of a computer

   Description automatically generatedDone:** Finalized tasks that meet the project requirements.

Figure 6 Kanban Board of the Project

Using a tool like Awork, the Kanban board ensures that the team stays organized, tracks progress efficiently and adapts to changes as needed. This visual approach promotes transparency, accountability, and streamlined task management throughout the project lifecyc

# 4. Realisation

The project infrastructure, illustrated in Figure X, serves as the blueprint for implementing a scalable and modular evaluation platform for large language models (LLMs). The realisation of this project is primarily achieved using the Python programming language, leveraging its rich ecosystem of libraries and frameworks for machine learning, evaluation, and web development.

The architecture outlines a systematic workflow beginning with user queries, which are processed through either adapter-based or model-based quantization techniques. These techniques are implemented using PEFT LoRA configurations for adapter-based quantization and model-based quantization directly applied to Llama3 and Mistral models.

The Kedro pipeline framework is utilized to organize the data flow and processing stages efficiently. Each query is evaluated through quantized models, and the outputs are monitored using MLflow and LangSmith for real-time performance tracking and logging. The evaluation framework computes metrics such as coherence, perplexity, memory usage, power consumption, and query time, ensuring a comprehensive assessment of model behaviour.

Further, the LLM-as-a-Judge module evaluates the generated outputs using an ELO scoring system, providing a robust mechanism for benchmarking the models. Finally, all components of the project are integrated using the Streamlit framework, which ensures a highly scalable and user-friendly interface for querying models, visualizing results, and interacting with the evaluation metrics seamlessly. Further information of all realized components is discussed below:

## 4.1 Quantization Approach

Quantization plays a central role in this project by enabling efficient evaluation of large language models (LLMs) while significantly reducing their computational and memory requirements. It facilitates running models at reduced precision, optimizing their performance for resource-constrained environments. A critical element in this process is the BitsandBytes quantization configuration, which is specifically designed to manage precision-based operations. This section explores how BitsandBytes enables low-memory execution and highlights the parameters and configurations involved in quantization.

### 4.1.1 BitsandBytes Quantization

The BitsAndBytes Configuration is integral to this project, enabling efficient quantization of large language models (LLMs) to optimize memory usage and computational performance. This configuration provides support for multiple precision levels—16-bit, 8-bit, and 4-bit—allowing models to be deployed in resource-constrained environments without significant degradation in their performance.

1. **16-bit Precision (FP16):**

This is the highest precision level supported by BitsAndBytes, primarily used when accuracy is critical. It reduces memory usage compared to 32-bit full precision while maintaining high model fidelity. Models are loaded using torch.float16, enabling faster computation and reduced storage requirements.

1. **8-bit Precision (INT8):**

With load\_in\_8bit=True, the BitsAndBytes Configuration compresses model weights into 8-bit integers, achieving a 4x memory reduction compared to 32-bit precision. This is ideal for general-purpose tasks where resource optimization and inference speed are crucial.

1. **4-bit Precision (INT4):**

The most memory-efficient option, 4-bit precision further compresses model weights into 4-bit integers, reducing memory usage by 8x compared to 32-bit precision. Advanced features such as NF4 (Normal Float 4) and double quantization are employed to mitigate precision loss. Parameters like bnb\_4bit\_quant\_type="nf4", bnb\_4bit\_compute\_dtype=torch.bfloat16, and bnb\_4bit\_use\_double\_quant=True allow fine-tuning of 4-bit quantization for optimized accuracy and computation as well as for error correction.

The following snippet demonstrates how BitsAndBytes is configured for 8-bit and 4-bit precision:

quantization\_config = None

if precision == "8bit":

# 8-bit quantization configuration

quantization\_config = BitsAndBytesConfig(load\_in\_8bit=True)

elif precision == "4bit":

# 4-bit quantization configuration

quantization\_config = BitsAndBytesConfig(

load\_in\_4bit=True,

bnb\_4bit\_quant\_type="nf4", # Normal Float 4 for optimized quantization

bnb\_4bit\_use\_double\_quant=True, # Enable double quantization

bnb\_4bit\_compute\_dtype=torch.bfloat16 # Set computation to bfloat16

)

# Load the model using the defined quantization configuration

model = AutoModelForCausalLM.from\_pretrained(

model\_name,

quantization\_config=quantization\_config, # Apply the quantization configuration

device\_map="auto" # Automatically map model layers to available GPUs

)

### 4.1.2 Model Based Quantization

Model-based quantization focuses on directly reducing the precision of a pre-trained model's weights and activations to optimize memory usage and computational efficiency. In this project, BitsAndBytesConfig is utilized to implement model-based quantization for 16-bit, 8-bit, and 4-bit precision, and the initialize\_model\_and\_tokenizer method is used for implementing it. Below is the code snippet:

@st.cache\_resource

def initialize\_model\_and\_tokenizer(model\_name, precision):

# Free up unused GPU memory

gc.collect()

torch.cuda.empty\_cache()

device = "cuda" if torch.cuda.is\_available() else "cpu"

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

# Define quantization configuration

quantization\_config = None

if precision == "8bit":

quantization\_config = BitsAndBytesConfig(load\_in\_8bit=True)

elif precision == "4bit":

quantization\_config = BitsAndBytesConfig(

load\_in\_4bit=True,

bnb\_4bit\_quant\_type="nf4", # Optimized quantization type

bnb\_4bit\_use\_double\_quant=True, # Enable double quantization

bnb\_4bit\_compute\_dtype=torch.bfloat16 # Set computation type to bfloat16

)

# Load model with appropriate precision

if quantization\_config:

model = AutoModelForCausalLM.from\_pretrained(

model\_name,

quantization\_config=quantization\_config,

device\_map={"": "cuda:0"} # Automatically map model layers to GPU

)

else:

model = AutoModelForCausalLM.from\_pretrained(model\_name).to(torch.float16).to(device)

# Adjust tokenizer configuration

if tokenizer.pad\_token\_id is None:

tokenizer.pad\_token\_id = tokenizer.eos\_token\_id

print(f"Model loaded on device: {device}")

return model, tokenizer

## 4.2 Adapter Based Quantization

Adapter-based quantization works similarly to model-based quantization but introduces an additional fine-tuning mechanism using lightweight modules called adapters. In this project, fine-tuning is performed using Axolotl with two configurations: r=8, alpha=16 and r=16, alpha=32, tailored for both the Llama and Mistral models. These configurations allow efficient adaptation of pre-trained models to specific tasks without altering the original model parameters. The methods initialize\_model\_with\_adapters\_r\_8\_a\_16 and initialize\_model\_with\_adapters\_r\_16\_a\_32 are implemented to load fine-tuned models with these adapter setups. Below is an example code snippet:

@st.cache\_resource

def initialize\_model\_with\_adapters\_r\_8\_a\_16(precision, model\_name):

fine\_tuned\_model\_path = "./outputs/lora-out" if model\_name == "meta-llama/Llama-3.1-8B" else "./mistral\_outputs/lora-out"

tokenizer = AutoTokenizer.from\_pretrained(fine\_tuned\_model\_path)

if precision == "16bit":

model = AutoModelForCausalLM.from\_pretrained(

fine\_tuned\_model\_path,

torch\_dtype=torch.float16,

device\_map="auto"

)

elif precision == "8bit":

model = AutoModelForCausalLM.from\_pretrained(

fine\_tuned\_model\_path,

load\_in\_8bit=True,

device\_map="auto"

)

elif precision == "4bit":

bnb\_config = BitsAndBytesConfig(load\_in\_4bit=True)

model = AutoModelForCausalLM.from\_pretrained(

fine\_tuned\_model\_path,

quantization\_config=bnb\_config,

device\_map="auto"

)

else:

raise ValueError("Invalid precision. Please choose from '16bit', '8bit', or '4bit'.")

return model, tokenizer

### 4.2.1 Axolotl Fine-Tuning Approach

In this project, the Axolotl fine-tuning approach has been adopted to streamline the process of model adaptation, moving away from traditional coding techniques. Axolotl is an open-source library specifically designed for efficient fine-tuning of large language models (LLMs), offering flexibility in leveraging sub-techniques such as LoRA (Low-Rank Adaptation). This library simplifies the complex process of fine-tuning by providing predefined configurations and support for scalable operations across diverse models, including Llama and Mistral.

To achieve fine-tuning, Axolotl requires a structured approach involving datasets and configuration files in a JSON format, which define the parameters and settings for the fine-tuning process. In this project, the fine-tuning is conducted using the LoRA technique, which introduces lightweight, trainable adapters into the existing model architecture, reducing the need to modify the base model parameters. This method allows for efficient task-specific adaptations without extensive computational overhead or resource demands.

The fine-tuning process in this project explores two distinct configurations of LoRA hyperparameters to balance flexibility and performance. The first configuration sets the rank to 8 and alpha to 16, providing a lightweight yet effective adaptation suitable for most tasks. The second configuration increases the rank to 16 and alpha to 32, offering a more robust adaptation for tasks requiring higher capacity. These two variations allow for experimentation with the trade-offs between computational efficiency and task performance.

Axolotl ensures that these fine-tuning processes are streamlined and repeatable, enabling the use of pre-defined settings for dataset preparation, model configurations, and training pipelines. Through the use of Axolotl, this project achieves efficient fine-tuning of the Llama and Mistral models, ensuring scalability and flexibility in handling task-specific requirements. The configurations for these fine-tuning approaches are discussed in the subsequent sections.

### 4.2.2 Configuration of Llama

The configuration for fine-tuning the meta-llama/Llama-3.1-8B model is designed to optimize memory efficiency and task-specific adaptation using the LoRA (Low-Rank Adaptation) technique. Below is the detailed breakdown of the configuration, which highlights the parameters used for fine-tuning, dataset setup, and training. Two configurations are employed, varying the LoRA hyperparameters to achieve different trade-offs between performance and resource usage.

* **Configuration 1: Rank = 8, Alpha = 16**

This configuration sets the LoRA rank to 8 and alpha to 16, which is a lightweight setup suitable for efficient fine-tuning without significant resource demands. The detailed configuration is as follows:

* 1. Base Model:

The fine-tuning process uses meta-llama/Llama-3.1-8B as the base model, which is loaded in 8-bit precision to reduce memory consumption.

* 1. Dataset Configuration:

The dataset for fine-tuning is specified as summarization\_dataset.jsonl, formatted for summarization tasks. A validation set comprising 10% of the total data is used for evaluation.

* 1. LoRA-Specific Parameters:

The adapter type is set to lora, with:

* + 1. LoRA Rank (lora\_r): 8
    2. Alpha (lora\_alpha): 16
    3. Dropout (lora\_dropout): 0.05

The target modules for LoRA are q\_proj and v\_proj, ensuring that LoRA updates are applied to the query and value projections of the model.

* 1. Training and Optimization:
     1. Learning Rate: 0.0002
     2. Optimizer: adamw\_8bit, which is optimized for 8-bit quantized training.
     3. Scheduler: cosine for gradual learning rate decay.
     4. Batch Size: 1 (micro-batch size) with gradient accumulation over 4 steps.
     5. Sequence Length: 4096 tokens for faster processing.
  2. Mixed Precision and Checkpointing:

BF16 is enabled if supported by the hardware, and gradient checkpointing is used to reduce memory usage during backpropagation.

* 1. Regularization and Logging:

Minimal weight decay of 0.01 is applied for regularization. Logging occurs every 10 steps, with checkpoint saving every 100 steps and evaluations every 50 steps.

* **Configuration 2: Rank = 16, Alpha = 32**

The second configuration is more robust, with the LoRA rank set to 16 and alpha to 32, providing a greater capacity for task-specific adaptation. The overall structure remains consistent with Configuration 1, with the following key differences:

1. LoRA-Specific Parameters:
   * 1. LoRA Rank (lora\_r): 16
     2. Alpha (lora\_alpha): 32
     3. Dropout (lora\_dropout): 0.05

These parameters allow the model to capture more task-specific patterns by increasing the capacity of the LoRA matrices.

1. Output Directory:

Checkpoints are saved in a separate directory, such as ./outputs/lora-out-r16-a32, to distinguish it from the first configuration.

Both configurations are implemented using Axolotl, with the JSON configuration defining the parameters for each setup. Below is the sample json snippet of one the of configuration:

base\_model: meta-llama/Llama-3.1-8B  # Base model for fine-tuning

# Dataset configuration

datasets:

  - path: "summarization\_dataset.jsonl"  # Path to your dataset

    type: summarizetldr  # Dataset type (compatible with the instruction-based format)

# General settings

load\_in\_8bit: true  # Enable 8-bit loading for memory efficiency

load\_in\_4bit: false  # Disable 4-bit quantization for now

strict: false

dataset\_prepared\_path: last\_run\_prepared  # Cached dataset preparation path

val\_set\_size: 0.1  # 10% of data used for validation

output\_dir: ./outputs/lora-out  # Directory for saving LoRA checkpoints

# LoRA configuration

adapter: lora

lora\_model\_dir:

# LoRA-specific parameters

lora\_r: 8  # Reduced rank for LoRA matrices

lora\_alpha: 16  # Lower scaling factor

lora\_dropout: 0.05  # Dropout to regularize LoRA

lora\_target\_modules:

  - q\_proj  # Query projection

  - v\_proj  # Value projection

# Sequence configuration

sequence\_len: 4096  # Reduced sequence length for faster processing

pad\_to\_sequence\_len: true

# Training settings

gradient\_accumulation\_steps: 4  # Steps to accumulate gradients

micro\_batch\_size: 1  # Per-GPU batch size

num\_epochs: 1  # Reduce to 1 epoch for quicker experimentation

optimizer: adamw\_8bit  # Optimizer updated to a valid value

lr\_scheduler: cosine  # Learning rate scheduler

learning\_rate: 0.0002  # Lower learning rate for smaller updates

# Mixed precision and hardware

bf16: auto  # Enable BF16 if supported

fp16: null  # Make sure FP16 is not enabled when BF16 is used.

gradient\_checkpointing: true  # Disable checkpointing to save time

flash\_attention: false  # Use FlashAttention for faster training

# Logging and debugging

logging\_steps: 10  # Log less frequently

debug: false  # Disable debug mode unless troubleshooting

# Early stopping

early\_stopping\_patience: 2  # Stop training after 2 bad evaluations

save\_steps: 100  # Save checkpoints every 100 steps

eval\_steps: 50  # Evaluate every 50 steps

# Checkpoints and evaluation

warmup\_steps: 5  # Fewer warmup steps

resume\_from\_checkpoint:

# Regularization

weight\_decay: 0.01  # Minimal weight decay for regularization

# Special tokens

special\_tokens:

  pad\_token: "<|end\_of\_text|>"  # Padding token

### 4.2.3 Configuration of Mistral

The configuration for the mistralai/Mistral-7B-Instruct-v0.3 model is similar to that of the Llama model but includes a few key differences tailored to Mistral's architecture. Notably, the sequence length for Mistral is increased to 8192 tokens, accommodating longer contexts compared to Llama's 4096 tokens. Additionally, the output directory paths and base model are specific to Mistral, ensuring compatibility with its unique instruction-tuned design. The rest of the configuration, including LoRA parameters, dataset setup, and training settings, remains consistent with the Llama configuration.

Below is the snippet of one of the configurations:

base\_model: mistralai/Mistral-7B-Instruct-v0.3 # Base model for fine-tuning

# Dataset configuration

datasets:

  - path: "summarization\_dataset.jsonl"  # Path to your dataset

    type: summarizetldr  # Dataset type

# General settings

load\_in\_8bit: true  # Enable 8-bit loading for memory efficiency

load\_in\_4bit: false  # Disable 4-bit quantization for now

strict: false

dataset\_prepared\_path: last\_run\_prepared  # Cached dataset preparation path

val\_set\_size: 0.1  # 10% of data used for validation

output\_dir: ./mistral\_outputs/lora-out  # Directory for saving LoRA checkpoints

# LoRA configuration

adapter: lora

lora\_model\_dir:

# LoRA-specific parameters

lora\_r: 8  # Reduced rank for LoRA matrices

lora\_alpha: 16  # Lower scaling factor

lora\_dropout: 0.05  # Dropout to regularize LoRA

lora\_target\_modules:

  - q\_proj  # Query projection

  - v\_proj  # Value projection

# Sequence configuration

sequence\_len: 8192  # Reduced sequence length for faster processing

pad\_to\_sequence\_len: true

# Training settings

gradient\_accumulation\_steps: 4  # Steps to accumulate gradients

micro\_batch\_size: 1  # Per-GPU batch size

num\_epochs: 1  # Reduce to 1 epoch for quicker experimentation

optimizer: adamw\_8bit  # Optimizer updated to a valid value

lr\_scheduler: cosine  # Learning rate scheduler

learning\_rate: 0.0002  # Lower learning rate for smaller updates

# Mixed precision and hardware

bf16: auto  # Enable BF16 if supported

fp16: null  # Make sure FP16 is not enabled when BF16 is used.

gradient\_checkpointing: true  # Disable checkpointing to save time

flash\_attention: false  # Use FlashAttention for faster training

# Logging and debugging

logging\_steps: 10  # Log less frequently

debug: false  # Disable debug mode unless troubleshooting

# Early stopping

early\_stopping\_patience: 2  # Stop training after 2 bad evaluations

save\_steps: 100  # Save checkpoints every 100 steps

eval\_steps: 50  # Evaluate every 50 steps

# Checkpoints and evaluation

warmup\_steps: 5  # Fewer warmup steps

resume\_from\_checkpoint:

# Regularization

weight\_decay: 0.01  # Minimal weight decay for regularization

## 4.3 Model Based vs Adapter Based

The following table provides a comparison between the Model-Based Quantization and Adapter-Based Quantization approaches applied in this project, highlighting their key differences and similarities in implementation, efficiency, and configurations:

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Model-Based Quantization** | **Adapter-Based Quantization** |
| **Purpose** | Directly reduces the precision of the entire model's weights and activations. | Introduces lightweight LoRA modules to fine-tune specific layers of the model. |
| **Technique** | Uses **BitsAndBytesConfig** for 16-bit, 8-bit, and 4-bit precision. | Uses **BitsAndBytesConfig** and **LoRA (Low-Rank Adaptation)** for task-specific fine-tuning. |
| **Precision Levels** | Supports 16-bit, 8-bit (load\_in\_8bit=True), and 4-bit (load\_in\_4bit=True). | Primarily uses 8-bit quantization for memory efficiency as well as supports 16-bit, 8-bit, 4-bit quantization. |
| **Configuration** | Adjusts precision globally for the entire model. | Requires additional parameters like **rank (lora\_r)** and **alpha (lora\_alpha)** for adapter-specific configurations. |
| **Impact on Base Model** | Modifies the precision of the entire model (global). | Keeps the base model unchanged and only updates specific layers via adapters. |
| **Flexibility** | Less flexible; applies quantization uniformly to all layers. | More flexible; targets specific layers (q\_proj, v\_proj) for adaptation. |
| **Output Checkpoints** | Saves quantized models in memory (cache) without creating explicit checkpoints. | Saves LoRA checkpoints separately (e.g., ./outputs/lora-out). |
| **Use Cases** | Suitable for deploying models in memory-constrained environments. | Suitable for fine-tuning models on task-specific datasets. |
| **Sequence Length** | Not explicitly adjusted; depends on the model's default configuration. | Sequence length can be configured for task-specific requirements (e.g., 4096 or 8192 tokens). |
| **Training Steps** | No additional training steps; model is used directly post-quantization. | Requires additional fine-tuning steps to adapt to the specific task. |
| **Example Method** | initialize\_model\_and\_tokenizer for loading quantized models. | initialize\_model\_with\_adapters\_r\_8\_a\_16 and initialize\_model\_with\_adapters\_r\_16\_a\_32 for loading fine-tuned adapters. |
| **Advantages** | Simple and quick to deploy; effective for reducing resource requirements. | Allows task-specific customization while keeping the base model intact. |
| **Disadvantages** | May lose some task-specific performance due to global precision reduction. | Additional time and resources are required for fine-tuning and managing LoRA adapters. |

Table 2 Comparison Table

This comparison highlights the trade-offs between the two approaches: Model-Based Quantization focuses on global memory optimization and simplicity, with models being stored in cache during runtime, while Adapter-Based Quantization prioritizes task-specific customization through explicitly saved LoRA checkpoints. Both methods offer flexibility depending on the requirements of the task and resource availability.

## 4.4 Output Generation Pipeline

The Output Generation Pipeline is a critical component of this project, responsible for generating task-specific outputs, such as summaries, based on user inputs and fine-tuned models. It incorporates generation arguments to customize the output generation process and includes a robust error handling mechanism to ensure reliability and consistency in the outputs. Below, the two key aspects of this pipeline are discussed.

### 4.4.1 Generation Arguments

The pipeline leverages a set of configurable generation arguments to fine-tune the quality, diversity, and length of the generated outputs. These arguments, passed dynamically during the inference stage, allow the model to produce concise, high-quality summaries while minimizing repetitive or irrelevant content. Key arguments include:

* **max\_new\_tokens**: Limits the number of tokens in the generated output to ensure brevity.
* **num\_beams**: Uses beam search to explore multiple output possibilities and improve output quality.
* **top\_p and top\_k**: Control the diversity of the generated text using nucleus sampling and top-k sampling techniques.
* **temperature**: Adjusts the randomness of the model's predictions.
* **repetition\_penalty and no\_repeat\_ngram\_size**: Penalize repetitive sequences, enhancing output variety.
* **early\_stopping**: Ensures the generation stops when a complete and meaningful summary is formed.

These parameters ensure the generated text adheres to the desired characteristics, such as being concise, fluent, and contextually accurate. Below is the snippet showcasing how these arguments are incorporated into the pipeline:

generation\_args = {

"max\_new\_tokens": 300,

"min\_length": 50,

"return\_full\_text": False,

"num\_beams": 4,

"length\_penalty": 1.0,

"repetition\_penalty": 1.5,

"no\_repeat\_ngram\_size": 3,

"top\_p": top\_p,

"top\_k": top\_k,

"temperature": temperature,

"early\_stopping": True,

"do\_sample": True

}

output = pipe(prompt, \*\*generation\_args)

### 4.4.2 Error Handling Mechanism

A robust error handling mechanism is implemented to ensure the reliability of the output generation process. The pipeline addresses potential issues such as:

* **Improper Sentence Endings**: Ensures the generated text ends with proper punctuation (e.g., ., !, or ?), improving readability.
* **Language Detection and Translation**: Detects the language of each sentence in the generated text. If a sentence is not in German, it is translated to German using the Google Translator API.
* **Short Sentence Skipping**: Skips sentences that are too short (<3 characters) to maintain coherence in the output.
* **Exception Handling**: Handles exceptions (e.g., LangDetectException) during language detection and defaults the sentence to German if the language cannot be determined.

This mechanism ensures that the output text is fluent, contextually accurate, and presented entirely in German, as required by the project’s specifications. Below is the code snippet demonstrating how the pipeline handles errors:

try:

detected\_language = detect(sentence) # Detect language

if detected\_language != 'de': # Translate if not German

translated\_sentence = GoogleTranslator(source='auto', target='de').translate(sentence)

translated\_sentences.append(translated\_sentence)

else:

translated\_sentences.append(sentence)

except LangDetectException:

translated\_sentences.append(sentence) # Default to German

## 4.5 Evaluation Metrics

The Evaluation Metrics provide a systematic way to assess the quality of the generated outputs and measure the efficiency of the pipeline. This project integrates a range of metrics to evaluate the generated text, including its fluency, coherence, and length, while also measuring system-level factors like memory usage, latency, and GPU power consumption. These metrics ensure that the model's performance is comprehensively analyzed from both qualitative and resource-efficiency perspectives.

### 4.5.1 Perplexity Metric

The perplexity metric measures the fluency of the generated text by evaluating how well the model predicts the sequence of tokens. It is computed as the exponential of the average negative log-likelihood loss of the generated text, with lower values indicating better fluency and alignment with natural language norms. This metric is particularly useful for identifying issues with text generation at the token level. Below is the code snippet for calculating the perplexity metric:

def calculate\_perplexity(model, tokenizer, text):

    inputs = tokenizer(text, return\_tensors="pt").to(model.device)

    with torch.no\_grad():

        loss = model(\*\*inputs, labels=inputs["input\_ids"]).loss

    return torch.exp(loss).item()

### 4.5.2 Coherence Metric

The coherence metric evaluates the logical flow and semantic alignment between sentences in the generated text. Using SentenceTransformer embeddings, consecutive sentences are encoded into vectors, and their pairwise cosine similarity is calculated. A weighted average of these similarities, based on sentence lengths, provides a final coherence score. This metric is critical for ensuring that summaries and other generated outputs maintain a logical sequence of ideas.

Below is the code snippet for calculating the coherence metric.

def calculate\_coherence\_score(generated\_text):

    def split\_sentences(text):

        """Splits text into sentences using NLTK for robust sentence tokenization."""

        return nltk.tokenize.sent\_tokenize(text)

    # Load SentenceTransformer model

    sbert\_model = SentenceTransformer('all-MiniLM-L6-v2')

    # Split the text into sentences

    sentences = split\_sentences(generated\_text)

    # Handle cases with no valid sentences

    if len(sentences) == 0:

        return 0.0

    if len(sentences) == 1:

        return 1.0  # Default coherence for a single sentence

    # Encode sentences into embeddings

    embeddings = sbert\_model.encode(sentences)

    # Calculate pairwise cosine similarities

    similarities = [

        cosine\_similarity([embeddings[i]], [embeddings[i + 1]])[0, 0]

        for i in range(len(embeddings) - 1)

    ]

    # Calculate weights (based on sentence lengths)

    weights = [

        len(sentences[i]) + len(sentences[i + 1])

        for i in range(len(sentences) - 1)

    ]

    # Compute weighted average of similarities

    weighted\_similarity\_sum = np.dot(similarities, weights)

    total\_weight = sum(weights)

    # Return coherence score

    return weighted\_similarity\_sum / total\_weight if total\_weight > 0 else 0.0

### 4.5.3 Power Usage Metric

The power usage metric measures the energy efficiency of the GPU during the model's execution. By monitoring the GPU's power consumption before and after the inference process using the pynvml library, the average power usage in watts is calculated. This helps evaluate the energy efficiency of different model configurations, particularly when deploying quantized models. Below is the code snippet for monitoring GPU power usage:

def gpu\_power\_usage\_decorator(func):

    @functools.wraps(func)

    def wrapper(\*args, \*\*kwargs):

        pynvml.nvmlInit()

        handle = pynvml.nvmlDeviceGetHandleByIndex(0)

        initial\_power = pynvml.nvmlDeviceGetPowerUsage(handle)

        start\_time = time.time()

        result = func(\*args, \*\*kwargs)

        final\_power = pynvml.nvmlDeviceGetPowerUsage(handle)

        end\_time = time.time()

        avg\_power\_watts = ((initial\_power + final\_power) / 2) / 1000  # Convert to watts

        pynvml.nvmlShutdown()

        print(f"Average GPU Power Usage: {avg\_power\_watts:.2f} Watts")

        # Append avg\_power\_watts to the result

        if isinstance(result, tuple):

            return (\*result, avg\_power\_watts)

        return result, avg\_power\_watts

    return wrapper

### 4.5.4 Latency Metric

The latency metric tracks the time taken to process the input and generate an output. This metric is essential for assessing the responsiveness of the pipeline, especially in real-time applications. A timing decorator is used to accurately measure the elapsed time for model inference, ensuring consistent and reliable results across configurations. Below is the code snippet for measuring latency:

def timing\_decorator(func):

    @functools.wraps(func)

    def wrapper(\*args, \*\*kwargs):

        start\_time = time.time()

        result = func(\*args, \*\*kwargs)

        end\_time = time.time()

        elapsed\_time = end\_time - start\_time

        print(f"Function '{func.\_\_name\_\_}' completed in {elapsed\_time:.4f} seconds")

        # Append elapsed\_time to the result

        if isinstance(result, tuple):

            return (\*result, elapsed\_time)

        return result, elapsed\_time

    return wrapper

### 4.5.5 Memory Usage Metric

The memory usage metric evaluates the peak GPU memory utilization during the model's execution. By leveraging PyTorch's memory management utilities, the pipeline can record the maximum memory allocated during inference. This metric provides valuable insights into the resource demands of different quantization and fine-tuning configurations. Below is the code snippet for monitoring memory usage:

def log\_execution\_memory(func):

    @functools.wraps(func)

    def wrapper(\*args, \*\*kwargs):

        if torch.cuda.is\_available():

            torch.cuda.empty\_cache()

            torch.cuda.reset\_peak\_memory\_stats()

        result = func(\*args, \*\*kwargs)

        if torch.cuda.is\_available():

            peak\_memory = torch.cuda.max\_memory\_allocated() / (1024 \*\* 2)  # Convert to MB

        else:

            peak\_memory = 0

        print(f"Peak GPU Memory Usage: {peak\_memory:.2f} MB")

        # Append peak\_memory to the result

        if isinstance(result, tuple):

            return (\*result, peak\_memory)

        return result, peak\_memory

    return wrapper

## 4.6 LLM-as-a-Judge

The LLM-as-a-Judge framework provides a robust mechanism for evaluating the quality of generated outputs. By leveraging task-specific evaluation prompts and an ELO Scoring Mechanism, the framework assigns scores to the outputs based on predefined evaluation guidelines, ensuring structured and unbiased assessments. This section elaborates on the key components of the framework.

### 4.6.1 Prompt Engineering and Judge Rules

Prompt Engineering forms the core of this framework, enabling dynamic task-specific evaluation. Each task type is associated with detailed evaluation guidelines, focusing on critical quality parameters such as creativity, coherence, clarity, and relevance. The prompt is designed to include the following elements:

* **Task Type**: Specifies whether the task involves text generation, summarization, or question-answering.
* **Evaluation Guidelines**: Provides clear, task-specific instructions, focusing on aspects like logical flow, accuracy, and completeness.
* **Evaluation Focus**: A concise question prompting the LLM to evaluate the output based on the given criteria.

These prompts are dynamically adjusted based on the input data and task type, ensuring that the evaluation is consistent, flexible, and tailored to specific requirements.

Judge Rules govern the behavior of the evaluation framework, ensuring structured and unbiased assessments. The key rules include:

* **Numeric Scoring Only**: The LLM provides a numeric score on a scale of 1 to 10, avoiding subjective commentary or explanations.
* **Focus on Task-Specific Criteria**: The LLM evaluates outputs strictly based on the parameters outlined in the prompt, such as coherence, accuracy, and relevance.
* **Error Handling**: Any invalid responses or exceptions during the evaluation process are handled gracefully to ensure robustness.

Together, prompt engineering and judge rules form a reliable and repeatable framework for assessing the quality of outputs, ensuring that evaluations are both objective and comprehensive.

Below is the code snippet for task-specific prompt generation and implementing judge rules.

# Task-specific Evaluation Prompt Generator

def get\_task\_specific\_prompt(row):

    task\_type = row["task\_type"]

    if task\_type == "text-generation":

        task\_specific\_instruction = (

            "Evaluate the generated text with a comprehensive approach, focusing on its creativity, coherence, "

            "and overall readability. Assess whether the text demonstrates logical flow, maintains engagement, and "

            "addresses the given topic effectively. Minor stylistic deviations should be considered acceptable."

        )

        evaluation\_point = "Does the generated text adequately address the topic while maintaining creativity, coherence, and relevance?"

    elif task\_type == "summarization":

        task\_specific\_instruction = (

"### Task: Evaluate the Quality of a Summarization\n\n"

"You are an evaluator for text summarization tasks. Your objective is to assess whether the summary effectively "

"captures the key ideas of the original text while maintaining accuracy and clarity.\n\n"

"### Evaluation Criteria:\n"

"Completeness\n"

"- Ensure that the summary includes all essential information from the original text.\n"

"- Identify any missing key details that could alter the meaning or context.\n\n"

"Accuracy\n"

"- Verify that the summary does not introduce false or misleading information.\n"

"- Ensure that paraphrased content maintains the original meaning.\n\n"

"Conciseness and Relevance\n"

"- Assess whether the summary eliminates unnecessary details while preserving important points.\n"

"- Ensure that the summary is not excessively long or redundant.\n\n"

"Clarity and Readability\n"

"- The summary should be well-structured and easy to understand.\n"

"- Sentences should be grammatically correct and logically connected.\n\n"

"### Evaluation Outcome:\n"

"- Provide an assessment explaining strengths and weaknesses.\n"

"- If the summary is incomplete or inaccurate, specify what needs to be corrected.\n"

"- Ensure feedback is actionable, allowing for targeted improvements.\n"

)

evaluation\_point = (

"Does the summary effectively capture the main ideas of the input text while maintaining accuracy and conciseness?"

)    elif task\_type == "question-answering":

        task\_specific\_instruction = (

            "Evaluate the generated answer for its factual accuracy, relevance to the question, and clarity. Ensure that "

            "the response aligns with the question's intent and provides sufficient detail. Minor inaccuracies can be overlooked "

            "if the overall response remains satisfactory."

        )

        evaluation\_point = "Does the answer sufficiently address the question with accuracy, relevance, and clarity?"

    else:

        task\_specific\_instruction = (

            "Evaluate the generated text for its general coherence, clarity, and relevance to the task. Assess whether it meets "

            "standard expectations for quality and addresses the task requirements adequately."

        )

        evaluation\_point = "Does the generated text meet general standards for clarity, coherence, and relevance in addressing the task?"

    # Return the complete prompt

    return f"""

    Task Type: {task\_type}

    Evaluation Guidelines:

    {task\_specific\_instruction}

    Generated Text:

    {row['generated\_text']}

    Evaluation Focus:

    {evaluation\_point}

    Provide a quality rating on a scale of 1 to 10, where:

    - 1 represents a poor-quality response with significant issues

    - 10 represents an excellent response that fully meets or exceeds expectations

    Provide only the numeric rating as your response."""

### 4.6.2 ELO Scoring Mechanism

The ELO Scoring Mechanism is a dynamic system adapted from competitive gaming to evaluate and rank generated outputs based on their quality. In this project, the mechanism:

1. **Initializes ELO Scores**: Assigns a baseline score (e.g., 1500) to all outputs.
2. **Calculates Expected Scores**: Uses the difference between the current output's ELO score and the baseline to estimate the expected quality.
3. **Updates Scores**: Adjusts the ELO scores iteratively based on the actual quality ratings (scaled between 0 and 1).

This scoring mechanism captures improvements or regressions in output quality over successive iterations, providing an objective and evolving measure of performance. Below is the code snippet for calculating ELO scores.

def calculate\_elo(ratings, initial\_elo=1500, k=32):

    elo\_scores = np.zeros(len(ratings))

    elo\_scores[0] = initial\_elo  # Initialize ELO score

    for i in range(1, len(ratings)):

        prev\_elo = elo\_scores[i - 1]

        expected\_score = 1 / (1 + 10 \*\* ((prev\_elo - initial\_elo) / 400))

        # Normalize the current rating to [0, 1]

        actual\_score = ratings[i] / 10  # Adjust scaling as needed

        # Update the ELO score

        elo\_scores[i] = prev\_elo + k \* (actual\_score - expected\_score)

    return elo\_scores

## 4.7 Front End Implementation

The front-end for this project is implemented using Streamlit, an open-source Python framework designed for building interactive and user-friendly web applications. Streamlit provides a lightweight and efficient platform for creating custom dashboards, making it ideal for presenting the evaluation pipeline and its configurations.

The interface allows users to interact with the model evaluation pipeline seamlessly by adjusting key parameters such as:

* **Evaluation Type**: Users can select between adapter\_based and model\_based evaluation approaches.
* **Model Selection**: Options include llama or other supported models.
* **Dataset Limit**: Allows the user to control the number of data samples to evaluate.
* **Hyperparameters**: Parameters like Top P (nucleus sampling), Top K (sampling), and Temperature are adjustable through intuitive sliders to fine-tune the text generation process.
* **Adapter Configuration**: Users can choose specific adapter setups for fine-tuned evaluation.

The UI displays real-time results, facilitating experimentation and evaluation of quantized models. As seen in Figure 4.x, the application provides an elegant and accessible way to interact with the pipeline, showcasing configurations and results while adhering to the project's objectives.

A screenshot of a computer

Description automatically generated

Figure 7 Front-end interface of the project

## 4.8 MLOps Integration

The project incorporates MLOps principles to ensure seamless experimentation, tracking, and monitoring of the evaluation pipeline. By integrating tools such as MLflow and LangSmith, the pipeline achieves robust tracking of metrics, parameters, and model performance, enabling a scalable and reproducible development workflow.

### 4.8.1 MLflow Integration

**MLflow** is integrated into the project to streamline the experiment tracking and results management. Through MLflow, the project tracks:

* **Parameters**: Key configurations such as evaluation\_type, model\_name, top\_p, top\_k, temperature, and the dataset limit are logged for each run.
* **Metrics**: Aggregated evaluation metrics like avg\_perplexity, avg\_coherence\_score, and avg\_memory\_usage\_mb are recorded to analyze the model's performance over different configurations.
* **Artifacts**: Output files containing results and logs are saved for easy access and reproducibility.

This integration allows for detailed comparisons of different runs, providing insights into how parameter variations impact the model's performance. Below is the code snippet for logging parameters, metrics, and artifacts in MLflow.

# Start MLflow Experiment with mlflow.start\_run(run\_name=unique\_name): # Log evaluation parameters mlflow.log\_param("evaluation\_type", evaluation\_type) mlflow.log\_param("model\_name", model\_name) mlflow.log\_param("top\_p", top\_p) mlflow.log\_param("top\_k", top\_k) mlflow.log\_param("temperature", temperature) mlflow.log\_param("dataset\_limit", len(dataset)) mlflow.log\_metric("avg\_perplexity", avg\_metrics["perplexity"]) mlflow.log\_metric("avg\_coherence\_score", avg\_metrics["coherence\_score"]) mlflow.log\_metric("avg\_memory\_usage\_mb", avg\_metrics["memory\_usage\_mb"]) # Log the results file mlflow.log\_artifact(output\_file)

### 4.8.2 LangSmith Integration

**LangSmith** is used for real-time tracing and monitoring of the pipeline. It enables tracking the execution flow of the language models, ensuring that all steps, from input prompts to output generation, are captured for analysis. Key features include:

* **Tracing**: Captures the internal flow of LLM interactions, ensuring traceability for debugging and optimization.
* **API Integration**: The LangSmith API is configured with the credentials provided, and environment variables are set for project-specific monitoring.
* **Project Management**: All traces are associated with the project identifier master-thesis-rb, allowing organized monitoring and historical analysis.

LangSmith provides advanced insights into the behaviour of the models during evaluation, making it a valuable addition to the pipeline's MLOps stack. Below is the code snippet demonstrating the LangSmith API integration and configuration.

langchain\_api\_key = credentials['langchain']['api\_key'] os.environ["LANGCHAIN\_TRACING\_V2"] = "true" os.environ["LANGCHAIN\_ENDPOINT"] = "https://api.smith.langchain.com" os.environ["LANGCHAIN\_API\_KEY"] = langchain\_api\_key os.environ["LANGCHAIN\_PROJECT"] = "master-thesis-rb"

## 4.9 Results

The primary focus of our platform’s development has been to integrate quantization techniques with rigorous evaluation methodologies. The evaluation is conducted using the LLM-as-a-Judge framework to assess response quality. The following sections present a summary of our findings with relevant visualizations.

### 4.9.1 Model Based and Llama

Applied with multiple precision levels to the Llama-8B model and evaluated its performance. The results demonstrate efficiency trade-offs across different bit precisions. Below are the key outcomes visualized through images.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 8 ELO Score - Llama-Model-Based

Below Table 3, presents aggregated metadata from experiments conducted on Llama model based, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 17719 | 21 | 128 | 4 | 51 | 7 |
| 16 | 17041 | 17 | 145 | 4 | 47 | 3 |
| 16 | 18224 | 18 | 165 | 4 | 31 | 3 |
| 16 | 15844 | 10 | 162 | 3 | 39 | 4 |
| 16 | 19150 | 31 | 173 | 4 | 32 | 4 |
| 8 | 27002 | 33 | 127 | 3 | 30 | 7 |
| 8 | 26132 | 1 | 153 | 32 | 100 | 1 |
| 8 | 27696 | 26 | 116 | 5 | 52 | 5 |
| 8 | 24618 | 22 | 114 | 4 | 39 | 4 |
| 8 | 28771 | 42 | 135 | 4 | 42 | 5 |
| 4 | 31919 | 40 | 138 | 5 | 50 | 5 |
| 4 | 31217 | 19 | 159 | 5 | 34 | 4 |
| 4 | 32437 | 20 | 141 | 5 | 33 | 4 |
| 4 | 30130 | 28 | 187 | 3 | 44 | 7 |
| 4 | 33483 | 49 | 198 | 5 | 36 | 6 |

Table 3 Meta Data - Llama-Model-Based

A screenshot of a computer

AI-generated content may be incorrect.The ELO score evaluation for Llama-3.1-8B shows a decline as quantization precision decreases. While 16-bit precision maintains higher ELO scores, performance drops significantly at 4-bit due to response degradation. The results suggest that 8-bit quantization offers a balanced trade-off between efficiency and quality. However, if we also consider efficiency, the 4-bit model stands out as the most memory-efficient while achieving a higher average rating. This indicates that 4-bit precision can be an optimal choice when resource constraints are prioritized over minor quality trade-offs.

Figure 9 Judge Statement - Llama-Model-Based

### 4.9.2 Model Based and Mistral

Applied with multiple precision levels to the Mistral-7B model and evaluated its performance. The results demonstrate efficiency trade-offs across different bit precisions. Below are the key outcomes visualized through images.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 10 ELO Score - Mistral-Model-Based

Below Table 4, presents aggregated metadata from experiments conducted on Mistral model based, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 16,521.83 | 21.50 | 123.30 | 4.92 | 0.37 | 6 |
| 16 | 15,698.11 | 17.00 | 134.58 | 7.65 | 0.52 | 5 |
| 16 | 17,113.87 | 27.54 | 126.73 | 6.91 | 0.34 | 6 |
| 16 | 14,391.63 | 8.58 | 150.01 | 5.70 | 0.36 | 7 |
| 16 | 18,121.94 | 34.66 | 138.30 | 7.01 | 0.44 | 7 |
| 8 | 29,180.27 | 41.99 | 126.02 | 4.90 | 0.35 | 6 |
| 8 | 27,745.18 | 35.44 | 123.42 | 7.68 | 0.50 | 7 |
| 8 | 29,823.24 | 46.23 | 117.11 | 8.39 | 0.32 | 5 |
| 8 | 25,816.38 | 28.42 | 126.37 | 6.31 | 0.35 | 4 |
| 8 | 31,258.74 | 53.89 | 117.54 | 5.03 | 0.35 | 6 |
| 4 | 31,318.29 | 41.51 | 187.16 | 10.04 | 0.32 | 5 |
| 4 | 30,465.14 | 35.44 | 177.94 | 7.76 | 0.49 | 7 |
| 4 | 31,927.25 | 45.59 | 151.97 | 4.56 | 0.44 | 8 |
| 4 | 29,267.03 | 27.99 | 198.74 | 7.22 | 0.41 | 6 |
| 4 | 32,970.16 | 51.38 | 152.40 | 6.34 | 0.40 | 5 |

Table 4 Meta Data - Mistal-Model-Based

The evaluation of Mistral-7B across different quantization levels highlights that the 4-bit model achieves the highest efficiency while maintaining an average rating of 6.2, comparable to its 16-bit counterpart. The ELO score trends show that as precision decreases, the model's performance initially improves before stabilizing at 4-bit. This suggests that lower-bit quantization can optimize resource usage without significant quality degradation. The efficiency metric further confirms that the 4-bit model is the most scalable, making it a strong candidate for deployment in constrained environments.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 11 Judge Statement - Mistral-Model-Based

### 4.9.3 Adapter Based and Llama with Setup 1

Applied with multiple precision levels to the Llama-8B model with adapter-based configurations with rank 8 and alpha 16 and evaluated its performance. The results demonstrate efficiency trade-offs across different bit precisions. Below are the key outcomes visualized through images.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 12 ELO Score - Llama-Adapter-Based (R-8, A-16)

Below Table 5, presents aggregated metadata from experiments conducted on Llama adapter based with rank 8 and alpha 16, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 17725 | 10 | 120 | 7 | 0.39 | 5 |
| 16 | 17047 | 1 | 175 | 34 | 1.00 | 1 |
| 16 | 18318 | 3 | 133 | 25971 | 1.00 | 1 |
| 16 | 15851 | 3 | 152 | 4 | 0.26 | 5 |
| 16 | 19158 | 6 | 137 | 13 | 1.00 | 3 |
| 8 | 27021 | 10 | 126 | 6 | 0.42 | 5 |
| 8 | 26149 | 4 | 123 | 14 | 1.00 | 2 |
| 8 | 27629 | 4 | 128 | 31 | 1.00 | 1 |
| 8 | 24636 | 11 | 111 | 3 | 0.51 | 5 |
| 8 | 28879 | 6 | 135 | 9 | 1.00 | 1 |
| 4 | 32635 | 30 | 177 | 7 | 0.48 | 4 |
| 4 | 31860 | 5 | 138 | 492 | 1.00 | 1 |
| 4 | 33212 | 15 | 194 | 7 | 0.68 | 6 |
| 4 | 30651 | 5 | 176 | 17 | 1.00 | 5 |
| 4 | 34397 | 10 | 190 | 55 | 1.00 | 1 |

Table 5 Meta Data - Llama-Adapter-Based (R-8, A-16)

The evaluation of the Llama-8B model with adapter-based configurations (rank 8, alpha 16) highlights the efficiency trade-offs across different quantization levels. The 4-bit model emerges as the best overall choice, achieving the highest average rating of 3.4 while consuming the least memory (5518.13 MB) and offering the best efficiency (0.000616). The ELO score trends indicate a decline as precision decreases, but the 4-bit model maintains a balance between quality and resource usage. While 16-bit precision ensures higher accuracy, its computational cost makes it impractical for deployment. The results suggest that the 4-bit model is optimal for applications requiring both efficiency and acceptable response quality.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 13 Judge Statement - Llama-Model-Based (R-8, A-16)

### 4.9.4 Adapter Based and Mistral with Setup 1

Applied with multiple precision levels to the Mistral-7B model with adapter-based configurations with rank 8 and alpha 16 and evaluated its performance. The results demonstrate efficiency trade-offs across different bit precisions. Below are the key outcomes visualized through images.

The evaluation of the Mistral-7B model with adapter-based configurations (rank 8, alpha 16) shows that the 16-bit precision model achieves the highest average rating of 2.8, indicating superior quality and performance. However, it consumes the most memory (18128.94 MB), making it less scalable for resource-constrained environments. The ELO score trends indicate a decline as precision decreases, but the 4-bit model remains the most efficient, achieving the highest efficiency (0.001974) while maintaining acceptable quality. The trade-off analysis suggests that while 16-bit precision provides the best quality, the 4-bit model offers the best balance between efficiency and performance.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 14 ELO Score - Mistral-Adapter-Based (R-8, A-16)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 15 Judge Statement – Mistral-Model-Based (R-8, A-16)

Below Table 6, presents aggregated metadata from experiments conducted on Mistral adapter based with rank 8 and alpha 16, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 16528 | 13 | 120 | 8 | 0.42 | 6 |
| 16 | 15705 | 2 | 158 | 31096 | 1.00 | 1 |
| 16 | 17208 | 4 | 140 | 31096 | 1.00 | 1 |
| 16 | 14398 | 1 | 132 | 26 | 1.00 | 1 |
| 16 | 18129 | 7 | 165 | 6 | 0.37 | 5 |
| 8 | 29195 | 40 | 128 | 5 | 0.45 | 3 |
| 8 | 27776 | 16 | 130 | 33530 | 1.00 | 1 |
| 8 | 29858 | 6 | 132 | 33530 | 1.00 | 1 |
| 8 | 25755 | 3 | 121 | 10 | 1.00 | 1 |
| 8 | 31291 | 11 | 116 | 14 | 1.00 | 3 |
| 4 | 30349 | 9 | 178 | 10 | 1.00 | 1 |
| 4 | 29379 | 7 | 190 | 18 | 1.00 | 1 |
| 4 | 31032 | 10 | 186 | 24232 | 1.00 | 1 |
| 4 | 28062 | 25 | 178 | 4 | 0.50 | 5 |
| 4 | 32304 | 12 | 180 | 52 | 1.00 | 2 |

Table 6 Meta Data - Mistal-Adapter-Model (R-8, A-16)

### 4.9.5 Adapter Based and Llama with Setup 2

Applied with multiple precision levels to the Llama-8B model with adapter-based configurations with rank 16 and alpha 32 and evaluated its performance. The results demonstrate efficiency trade-offs across different bit precisions. Below are the key outcomes visualized through images.

The evaluation of the Llama-3.1-8B model with adapter-based configurations (rank 16, alpha 32) demonstrates that the 8-bit model provides the best balance between quality and efficiency. It achieves the highest average rating of 5.2 while maintaining moderate memory consumption (9846.14 MB), making it a practical choice. The 4-bit model, although more efficient with lower memory usage (5389.26 MB), shows a slight decline in quality with an average rating of 3.6. On the other hand, the 16-bit model offers higher accuracy but consumes significantly more memory (19177.25 MB), making it less viable for deployment. The results confirm that the 8-bit quantization achieves an optimal trade-off between model performance and resource efficiency.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 16 ELO Score – Llama-Adapter-Based (R-16, A-32)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 17 Judge Statement – Llama-Model-Based (R-16, A-32)

Below Table 7, presents aggregated metadata from experiments conducted on Llama adapter based with rank 16 and alpha 32, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 17745 | 4 | 115 | 5.93 | 0.54 | 3 |
| 16 | 17067 | 2 | 158 | 14.76 | 1.00 | 4 |
| 16 | 18337 | 25 | 166 | 5.47 | 0.43 | 6 |
| 16 | 15871 | 2 | 150 | 4.20 | 0.52 | 3 |
| 16 | 19177 | 11 | 163 | 13.15 | 1.00 | 4 |
| 8 | 27080 | 22 | 125 | 5.62 | 0.32 | 6 |
| 8 | 26210 | 5 | 106 | 8.17 | 1.00 | 4 |
| 8 | 27687 | 4 | 128 | 13.27 | 1.00 | 3 |
| 8 | 24783 | 5 | 111 | 3.59 | 0.37 | 7 |
| 8 | 29023 | 16 | 133 | 6.35 | 0.36 | 6 |
| 4 | 32738 | 17 | 174 | 9.05 | 1.00 | 4 |
| 4 | 31963 | 5 | 190 | 432.09 | 1.00 | 1 |
| 4 | 33316 | 12 | 187 | 12.40 | 0.36 | 7 |
| 4 | 30754 | 4 | 186 | 12.73 | 1.00 | 3 |
| 4 | 34413 | 16 | 186 | 13.90 | 0.37 | 3 |

Table 7 Meta Data - Llama-Model-Based (R-16, A-32)

### 4.9.6 Adapter Based and Mistral with Setup 2

Applied with multiple precision levels to the Mistral-7B model with adapter-based configurations with rank 16 and alpha 32 and evaluated its performance. The results demonstrate efficiency trade-offs across different bit precisions. Below are the key outcomes visualized through images.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 18 ELO Score – Mistral-Adapter-Based (R-16, A-32)

The evaluation of the Mistral-7B model with adapter-based configurations (rank 16, alpha 32) highlights the trade-offs between memory efficiency and performance. The 8-bit model emerges as the most balanced choice, achieving the highest average rating of 2.8 while maintaining a reasonable memory footprint (13203.05 MB). Although the 4-bit model is the most efficient (efficiency = 0.001190) with the lowest memory consumption (1344.78 MB), it suffers from reduced output quality, as seen in its lower average rating of 1.6. On the other hand, the 16-bit model offers slightly better quality but consumes excessive memory (18236.34 MB), making it less practical. The results confirm that the 8-bit model provides the optimal balance.

Below Table 8, presents aggregated metadata from experiments conducted on Mistal adapter based with rank 16 and alpha 32, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 16548 | 3 | 125 | 9.17 | 1.00 | 3 |
| 16 | 15724 | 5 | 124 | 29087 | 1.00 | 1 |
| 16 | 17140 | 3 | 168 | 29087 | 1.00 | 1 |
| 16 | 14502 | 1 | 141 | 11.69 | 1.00 | 1 |
| 16 | 18236 | 4 | 162 | 13.25 | 1.00 | 2 |
| 8 | 29260 | 12 | 125 | 8.08 | 0.33 | 4 |
| 8 | 27843 | 4 | 120 | 30974 | 1.00 | 1 |
| 8 | 30026 | 6 | 127 | 30974 | 1.00 | 1 |
| 8 | 25816 | 3 | 114 | 11.44 | 1.00 | 3 |
| 8 | 31439 | 10 | 131 | 9.74 | 1.00 | 5 |
| 4 | 30913 | 10 | 179 | 23260 | 1.00 | 1 |
| 4 | 29945 | 8 | 187 | 13.24 | 1.00 | 1 |
| 4 | 31598 | 11 | 176 | 23260 | 1.00 | 1 |
| 4 | 28628 | 5 | 183 | 9.75 | 1.00 | 1 |
| 4 | 32784 | 15 | 142 | 20.05 | 1.00 | 4 |

Table 8 Meta Data - Mistral-Adapter-Based (R-16, A-32)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 19 Judge Statement – Mistral-Model-Based (R-16, A-32)

### 4.9.7 MlFlow Monitoring

MLflow was utilized for tracking model-based evaluations, capturing key metrics such as memory usage, coherence scores, and perplexity. The platform provided an efficient way to log, compare, and analyze various model configurations. Our experiments demonstrated how different quantization levels impact performance, highlighting trade-offs in efficiency and accuracy. The results stored in MLflow facilitate reproducibility and provide valuable insights for model optimization. Below are visualizations depicting tracked runs and logged metrics.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 20 MLFlow Monitoring

A screenshot of a computer

AI-generated content may be incorrect.

Figure 21 MLFlow Monitoring – Individual Results

### 4.9.8 LangSmith Monitoring

LangSmith was used to trace and monitor model evaluations, particularly focusing on latency, token usage, and error rates. The platform provided detailed insights into each model's execution, helping to identify performance bottlenecks and optimize model selection. Through LangSmith’s tracking, we evaluated the trade-offs between model latency and response accuracy across different configurations. The monitoring framework ensured transparency in benchmarking quantized models. Below are visualizations illustrating the monitored runs and their associated metrics.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 22 LangSmith Monitori

# 5. Test Inference

## 5.1 DeekSeek Integration

DeepSeek, a cutting-edge addition to the LLM landscape, was integrated and thoroughly tested within our evaluation framework using the DeepSeek-R1-Distill-Llama-8B model. The evaluation encompassed both model-based and adapter-based quantization techniques, assessing the model’s performance across various precision levels. Our tests confirmed that DeepSeek performs robustly in different quantization settings while maintaining efficiency in inference tasks.

The successful integration of DeepSeek highlights our platform's ability to accommodate and analyze a wide range of decoder-only models. By leveraging our standardized evaluation framework, we demonstrated that DeepSeek maintains competitive coherence and response quality while optimizing memory usage and latency. This reinforces the adaptability of our system in benchmarking quantized models, ensuring it can be utilized for diverse LLM architectures. The results validate the efficacy of DeepSeek within our experimental setup, paving the way for broader applications in LLM evaluation and deployment.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 23 ELO Score – DeepSeek-Model-Based

A screenshot of a computer

AI-generated content may be incorrect.

Figure 24 Judge Statement – DeepSeek-Model-Based

The evaluation of DeepSeek-R1-Distill-Llama-8B within our framework demonstrated its capability to balance efficiency and performance across different quantization levels. The 8-bit model emerged as the most optimal choice, achieving a high average rating of 6.0 while maintaining significantly lower memory consumption compared to the 16-bit variant. Although the 4-bit model exhibited higher efficiency, its lower quality rating made the 8-bit version a preferable option for practical use. The results reinforce the flexibility of our platform in handling diverse decoder-only models while maintaining a robust evaluation framework.

Below Table 9, presents aggregated metadata from experiments conducted on DeepSe ek model based, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 17719 | 22 | 92 | 7.54 | 0.37 | 7 |
| 16 | 17041 | 18 | 89 | 13.05 | 0.44 | 6 |
| 16 | 18224 | 25 | 93 | 10.25 | 0.34 | 4 |
| 16 | 15844 | 12 | 90 | 8.50 | 0.39 | 5 |
| 16 | 19237 | 31 | 122 | 7.36 | 0.39 | 6 |
| 8 | 27000 | 33 | 88 | 10.07 | 0.33 | 8 |
| 8 | 26128 | 30 | 82 | 13.60 | 0.42 | 5 |
| 8 | 27606 | 36 | 93 | 11.00 | 0.35 | 7 |
| 8 | 24619 | 23 | 93 | 8.63 | 0.33 | 6 |
| 8 | 28764 | 45 | 89 | 8.66 | 0.41 | 4 |
| 4 | 31920 | 41 | 83 | 8.48 | 0.39 | 7 |
| 4 | 31219 | 36 | 102 | 11.61 | 0.41 | 6 |
| 4 | 32439 | 44 | 93 | 8.73 | 0.39 | 6 |
| 4 | 30128 | 29 | 163 | 5.61 | 0.40 | 3 |
| 4 | 33397 | 50 | 105 | 8.61 | 0.41 | 5 |

Table 9 Meta Data - DeepSeek-Model-Based

The evaluation of DeepSeek-R1-Distill-Llama-8B under the adapter-based configuration (r=8, alpha=16) highlights the trade-offs between precision levels in terms of quality, memory consumption, and efficiency. The 8-bit model emerges as the most balanced choice, achieving the highest average rating of 5.0 while maintaining reasonable memory consumption at 9629.89 MB. Compared to the 16-bit model, which consumes significantly more memory (19157.75 MB) but delivers a lower rating (4.2), the 8-bit configuration provides a scalable alternative without sacrificing quality. Although the 4-bit model demonstrates higher efficiency, its slightly lower rating (4.6) indicates some compromise in response quality. These findings reinforce the flexibility of our platform in supporting adapter-based quantization while ensuring a well-optimized balance between performance and resource constraints.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 25 Judge Statement – DeepSeek-Adapter-Based (R-8 A-16)

The evaluation of DeepSeek-R1-Distill-Llama-8B under the adapter-based configuration (r=16, alpha=32) highlights the trade-offs between memory efficiency and model performance. The 4-bit precision model emerges as the best overall choice, offering the highest efficiency (0.000742) while consuming the least memory (5657.13 MB). Despite having a slightly lower average rating (4.2) compared to the 16-bit model (4.6), it significantly outperforms the 8-bit model (3.6), demonstrating its ability to maintain quality while optimizing resource utilization. This result underscores the practicality and scalability of the 4-bit model, making it a well-rounded option for scenarios where both performance and memory efficiency are critical considerations.

Below Table 10, presents aggregated metadata from experiments conducted on DeepSeek Adapter based with rank 8 and alpha 16, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 17725 | 11 | 157 | 6.38 | 0.36 | 3 |
| 16 | 17047 | 3 | 161 | 16.82 | 0.60 | 5 |
| 16 | 18231 | 7 | 171 | 16.17 | 0.31 | 3 |
| 16 | 15850 | 11 | 151 | 2.88 | 0.39 | 5 |
| 16 | 19158 | 10 | 142 | 8.92 | 0.44 | 5 |
| 8 | 27020 | 26 | 130 | 7.75 | 0.39 | 6 |
| 8 | 26146 | 29 | 111 | 4.35 | 0.44 | 4 |
| 8 | 27628 | 21 | 117 | 5.26 | 0.45 | 6 |
| 8 | 24637 | 23 | 113 | 3.43 | 0.43 | 3 |
| 8 | 28788 | 27 | 134 | 5.03 | 0.30 | 6 |
| 4 | 32650 | 19 | 175 | 9.64 | 0.66 | 5 |
| 4 | 31875 | 7 | 189 | 33.23 | 1.00 | 3 |
| 4 | 33314 | 20 | 147 | 4.61 | 0.47 | 4 |
| 4 | 30665 | 32 | 187 | 3.04 | 0.38 | 4 |
| 4 | 34325 | 36 | 177 | 10.03 | 0.50 | 7 |

Table 10 Meta Data - DeepSeek-Adapter-Model (R-8, A-16)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 26 Judge Statement – DeepSeek-Adapter-Based (R-16 A-32)

Below Table 11, presents aggregated metadata from experiments conducted on DeepSeek Adapter based with rank 16 and alpha 32, highlighting key performance metrics across different parametric configurations and task complexities. The data reveals critical insights into the trade-offs between computational efficiency, accuracy, and energy consumption in large-scale language models.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **precision** | **memory\_usage\_mb** | **query\_time** | **power\_usage** | **perplexity** | **coherence\_score** | **rating** |
| 16 | 17745 | 14 | 157 | 3.61 | 0.32 | 6 |
| 16 | 17067 | 2 | 158 | 23.87 | 1.00 | 5 |
| 16 | 18251 | 15 | 173 | 7.69 | 0.65 | 3 |
| 16 | 15871 | 11 | 156 | 2.92 | 0.46 | 5 |
| 16 | 19177 | 16 | 133 | 4.12 | 0.31 | 4 |
| 8 | 27078 | 4 | 127 | 35.22 | 1.00 | 1 |
| 8 | 26205 | 12 | 118 | 6.78 | 0.46 | 4 |
| 8 | 27688 | 40 | 129 | 4.38 | 0.44 | 4 |
| 8 | 24698 | 26 | 115 | 12.91 | 1.00 | 4 |
| 8 | 28845 | 27 | 134 | 5.00 | 0.41 | 5 |
| 4 | 32740 | 29 | 177 | 15.33 | 0.30 | 5 |
| 4 | 31965 | 10 | 179 | 15.80 | 0.37 | 1 |
| 4 | 33318 | 60 | 176 | 4.19 | 0.48 | 5 |
| 4 | 30755 | 43 | 177 | 3.75 | 0.40 | 5 |
| 4 | 34502 | 69 | 187 | 5.42 | 0.43 | 5 |

Table 11 Meta Data - DeepSeek-Adapter-Based (R-16 A-32)

From the analysis presented in Figure 27, it is evident that mistral-model-based achieved the highest rating (6.0) among the model-based approaches, closely followed by deepseek-model-based (5.67) and llama-model-based (5.66). This suggests that Mistral’s model-based variant is the most effective in handling queries, demonstrating strong overall performance across different precision levels.

In contrast, for adapter-based techniques, deepseek-adapter-based (R-8, A-16) received the highest average rating of 4.6, outperforming all other adapter-based configurations. This indicates that DeepSeek’s adapter-based method at (R-8, A-16) provides the best balance between efficiency and response quality compared to other adapter-based setups.

One interesting observation is that DeepSeek consistently provides a tradeoff between model-based and adapter-based approaches, maintaining stable average ratings across both categories. This highlights DeepSeek’s adaptability in delivering competitive results across different quantization strategies.

The overall trend suggests that model-based approaches consistently achieve higher ratings compared to adapter-based approaches, reinforcing their robustness in handling diverse queries. However, among adapter-based techniques, DeepSeek's configuration at R-8, A-16 stands out as the most effective option. This makes DeepSeek a versatile choice for scenarios where both model-based and adapter-based solutions are being considered.

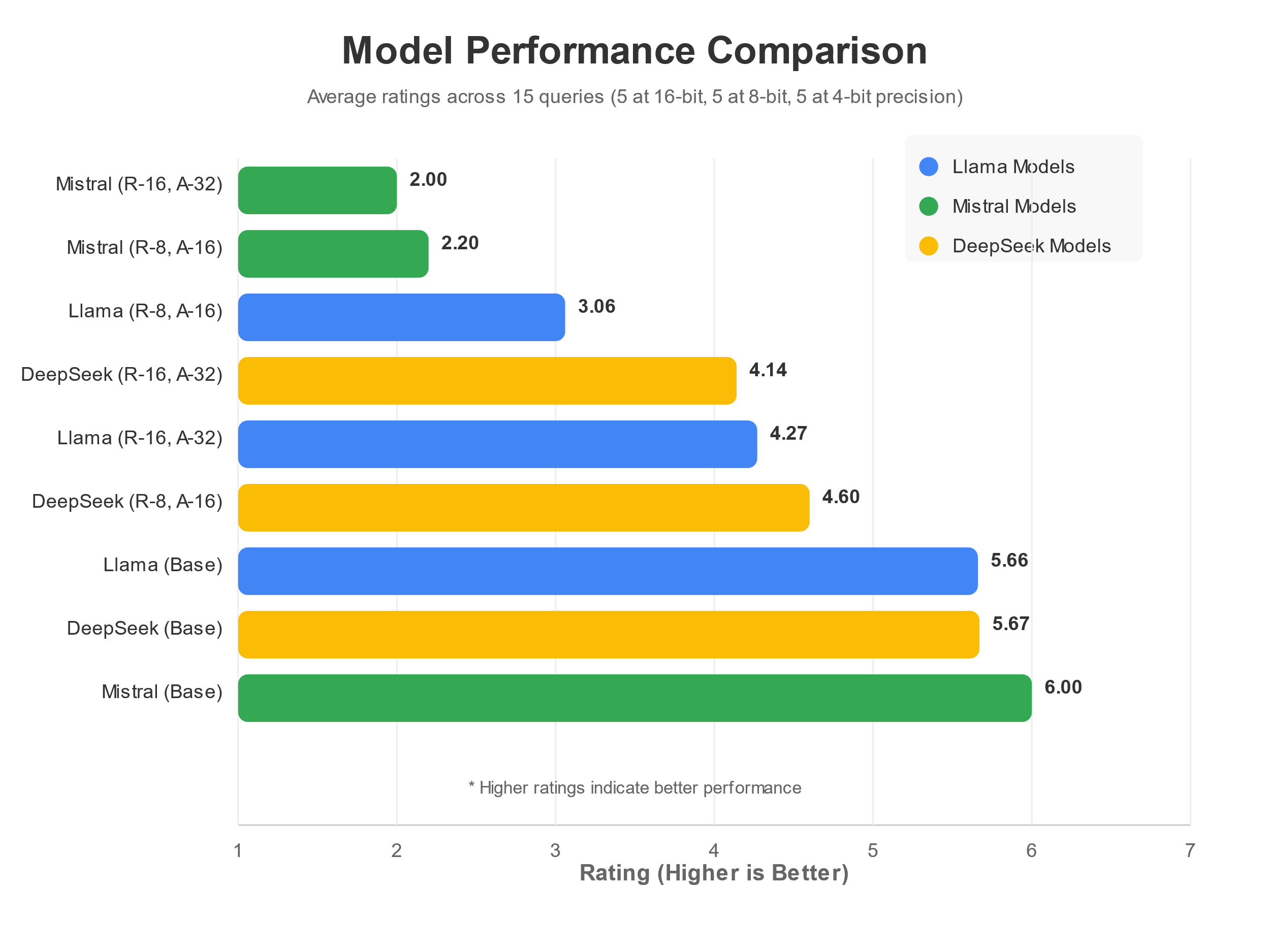


Figure 27 Model Performance Comparison via Ratings

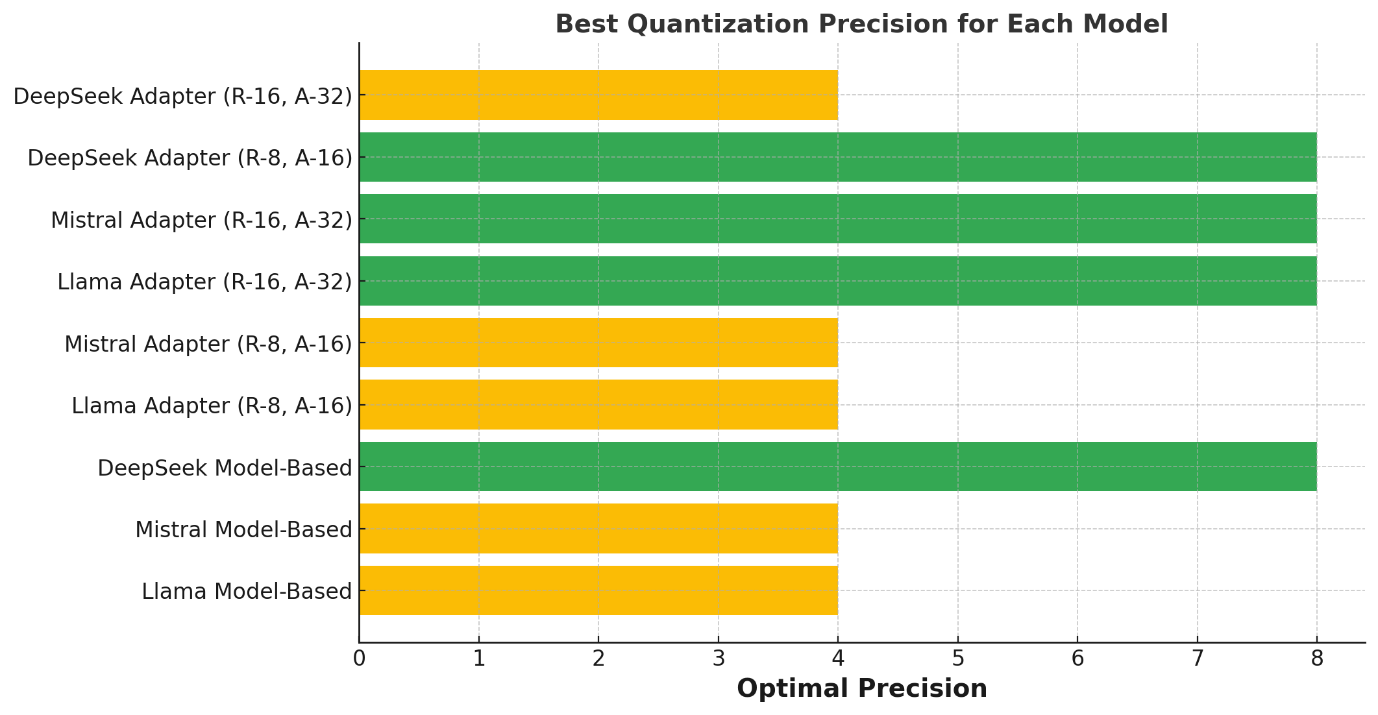


Figure 28 Best Quantization Precision for Each Model

Overall, 8-bit quantization emerged as the most balanced choice across models, offering a trade-off between performance and efficiency. However, 4-bit precision proved optimal for specific setups, particularly for Mistral (R-8, A-16) and DeepSeek (R-16, A-32), demonstrating that lower-bit quantization can be viable without significant quality loss in constrained environments.

# 6. Summary and Perspectives

The research conducted in this thesis focuses on the in-depth evaluation of quantized large language models (LLMs) using an LLM-as-a-Judge framework to benchmark various quantization techniques, including model-based and adapter-based quantization. The study meticulously investigates the trade-offs between precision, efficiency, and memory consumption, providing comprehensive insights into the performance of multiple decoder-only models, including Llama-3.1-8B, Mistral-7B, and DeepSeek-R1-Distill-Llama-8B. The core objective was to establish a reliable framework for evaluating different quantization levels (16-bit, 8-bit, and 4-bit) under different configurations, particularly adapter-based setups with varying rank (r) and alpha values. By leveraging continuous evaluation methods, incorporating ELO rating as a performance metric, and integrating tracking tools such as MLflow and LangSmith, this research provides a structured methodology for quantized LLM assessment.

Throughout this study, various configurations of model-based and adapter-based quantization were tested, enabling a comparative analysis of their performance under different memory constraints. Conversely, lower-bit quantized models, particularly the 8-bit and 4-bit configurations, demonstrate remarkable efficiency improvements while maintaining a reasonable quality of inference, making them more viable for real-world deployment scenarios where memory constraints are critical. The integration of DeepSeek, a newer entrant in the LLM space, further validated the robustness of the evaluation framework, proving that the developed methodology is flexible and adaptable across various decoder-only architectures.

Despite its contributions, this research acknowledges several limitations. One key constraint lies in the computational restrictions associated with running multiple quantized models at different precision levels. Although GPU-based execution was ideal, the lack of high-VRAM accelerators (e.g., GPUs, TPUs) limited testing of models beyond 8B parameters. This constraint affects the generalizability of the findings when applied to enterprise-scale deployments where LLMs often exceed 30B parameters. Furthermore, while the LLM-as-a-Judge framework proved to be effective in evaluating responses based on ELO scores, the subjectivity involved in defining qualitative measures such as coherence and fluency introduces a degree of uncertainty in final judgments. Future work could benefit from the integration of additional evaluation metrics, including human-in-the-loop assessments, to refine the scoring methodology further.

A key avenue for future work involves integrating retrieval-augmented generation (RAG) techniques within the quantized model evaluation pipeline. Given that many real-world applications require factual consistency and long-context reasoning, combining RAG with quantized LLMs could provide enhanced performance while reducing computational overhead. Furthermore, the integration of sparse fine-tuning methodologies, such as prefix tuning or soft prompts, could be exploredto determine whether they can mitigate the impact of lower-bit quantization on response quality.

Another aspect of scalability is the potential application of quantized models in multilingual and code-based tasks. While this research primarily evaluated models using German Wikipedia and news datasets, future expansions could involve benchmarking quantized models on other high-resource and low-resource languages. Additionally, the applicability of quantized models for code generation and reasoning tasks remains an open question. Given the growing importance of code-focused LLMs, such as DeepSeek-Coder and CodeLlama, future research could analyze how quantization impacts the syntactic and semantic correctness of generated code.

# 7. Acknowledgment

I would like to thank my supervisor, Prof. Andreas Pech, and my Company Supervisor, Dr. Simonas Cerniauskas for his excellent direction, ongoing support, and insightful input during this project. His guidance and support were invaluable in managing the difficulties I encountered during the study process. I sincerely appreciate his patience in offering clear clarifications when necessary and for always making time for talks, despite his hectic schedule. This thesis would not have been possible without his devotion and conviction in my abilities, for which I am very grateful. It has been an honour to work under his direction.

# 8. Abbreviations

**A**

API – Application Programming Interface

**B**

BERT – Bidirectional Encoder Representations from Transformers

BLEU – Bilingual Evaluation Understudy

**C**

CUDA – Compute Unified Device Architecture

**F**

FFN – Feed-Forward Network

FP – Floating Point

**G**

GPU – Graphics Processing Unit

GPT – Generative Pre-trained Transformer

**H**

HELM – Holistic Evaluation of Language Models

**J**

JSON – JavaScript Object Notation

**L**

LLM – Large Language Model

LoRA – Low-Rank Adaptation

**M**

METEOR – Metric for Evaluation of Translation with Explicit ORdering

MLOps – Machine Learning Operations

MMLU – Massive Multitask Language Understanding

**N**

NLP – Natural Language Processing

**P**

PaLM – Pathways Language Model

PEFT – Parameter-Efficient Fine-Tuning

PPL – Perplexity

PTQ – Post-Training Quantization

**Q**

QAT – Quantization-Aware Training

**R**

ReLU – Rectified Linear Unit

RNN – Recurrent Neural Network

ROUGE – Recall-Oriented Understudy for Gisting Evaluation

**T**

T5 – Text-To-Text Transfer Transformers

# 9. References

|  |  |
| --- | --- |
| [1] | Zhang, et al., “On the Evaluation of Large Language Models,” *arXiv.org,* 2022. |
| [2] | Zafrir, et al., “Q8BERT: Quantized 8Bit BERT,” *arXiv.org,* 2019. |
| [3] | Houlsby, et al., “Parameter-Efficient Transfer Learning for NLP,” *Proceedings of the 36th International Conference on Machine Learning,* 2019. |
| [4] | C. Y. Lin, “ROUGE: A Package for Automatic Evaluation of Summaries.,” 2004. |
| [5] | Brown, et al., “Language Models are Few-Shot Learners,” *arXiv.org,* 2020. |
| [6] | Vaswani, et al., “Attention is all you need. In Advances in Neural Information Processing Systems,” 2017. |
| [7] | Carolin, “A mathematician's introduction to transformers and large language models,” x-dev.pages.jsc.fz-juelich.de, 22 July 2022. [Online]. Available: https://x-dev.pages.jsc.fz-juelich.de//2022/07/13/transformers-matmul.html. |
| [8] | Touvron, et al., “Llama: Open and Efficient Foundation Language Models.,” *arXiv.org,* 2023. |
| [9] | T. Mistral, “Mistral: High-Performance Open Source Language Models.,” 2023. |
| [10] | Jacob, et al., “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference.,” 2018. |
| [11] | Hubara, et al., “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations,” *Journal of Machine Learning Research,* 2017. |
| [12] | Jiedong Lang, et al., “A Comprehensive Study on Quantization Techniques for Large Language Models,” *https://arxiv.org/,* 2024. |
| [13] | Han, et al., “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.,” 2016. |
| [14] | NVIDIA , “NVIDIA TensorRT Documentation,” [Online]. Available: https://developer.nvidia.com/tensorrt. |
| [15] | Google Edge, “ Google Edge TPU Documentation,” [Online]. Available: https://coral.ai/docs/. |
| [16] | Papineni, et al., “BLEU: A Method for Automatic Evaluation of Machine Translation,” 2002. |
| [17] | chess.com, “ELO Rating System,” [Online]. Available: https://www.chess.com/terms/elo-rating. |
| [18] | Devlin, et al., “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* , 2019. |
| [19] | A. ELO, “The Rating of Chessplayers, Past and Present,” 1978. |
| [20] | Renren Jin, et al., “A Comprehensive Evaluation of Quantization Strategies for Large Language Models,” *arXiv.org,* 2024. |
| [21] | Li, et al., “Evaluating Quantized Large Language Models,” 2024. |
| [22] | Zheng, et al., “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena,” *arXiv.org,* 2023. |
| [23] | Yixiao Li, et al., “LoftQ: LoRA-Fine-Tuning-Aware Quantization for Large,” *arXiv.org,* 2023. |
| [24] | Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, Deyi Xiong, “A Comprehensive Evaluation of Quantization Strategies for Large Language Models,” *arXiv.org,* 2024. |
| [25] | Hu, et al., “LoRA: Low-Rank Adaptation of Large Language Models,” *arXiv.org,* 2021. |

# 10. Appendix

## 10.1 Table of Figures

[Figure 1 Architecture of Transformer Model 7](#_Toc192337242)

[Figure 2 Transformer vs Llama vs Mistral 9](#_Toc192337243)

[Figure 3 PEFT LoRA Architecture 14](#_Toc192337244)

[Figure 4 LLM Evaluation Metrics 16](#_Toc192337245)

[Figure 5 High Level Architecture of the Project 27](#_Toc192337246)

[Figure 6 Kanban Board of the Project 30](#_Toc192337247)

[Figure 7 Front-end interface of the project 46](#_Toc192337248)

[Figure 8 ELO Score - Llama-Model-Based 48](#_Toc192337249)

[Figure 9 Judge Statement - Llama-Model-Based 49](#_Toc192337250)

[Figure 10 ELO Score - Mistral-Model-Based 49](#_Toc192337251)

[Figure 11 Judge Statement - Mistral-Model-Based 50](#_Toc192337252)

[Figure 12 ELO Score - Llama-Adapter-Based (R-8, A-16) 51](#_Toc192337253)

[Figure 13 Judge Statement - Llama-Model-Based (R-8, A-16) 52](#_Toc192337254)

[Figure 14 ELO Score - Mistral-Adapter-Based (R-8, A-16) 52](#_Toc192337255)

[Figure 15 Judge Statement – Mistral-Model-Based (R-8, A-16) 53](#_Toc192337256)

[Figure 16 ELO Score – Llama-Adapter-Based (R-16, A-32) 54](#_Toc192337257)

[Figure 17 Judge Statement – Llama-Model-Based (R-16, A-32) 54](#_Toc192337258)

[Figure 18 ELO Score – Mistral-Adapter-Based (R-16, A-32) 55](#_Toc192337259)

[Figure 19 Judge Statement – Mistral-Model-Based (R-16, A-32) 56](#_Toc192337260)

[Figure 20 MLFlow Monitoring 57](#_Toc192337261)

[Figure 21 MLFlow Monitoring – Individual Results 57](#_Toc192337262)

[Figure 22 LangSmith Monitoring 58](#_Toc192337263)

[Figure 23 ELO Score – DeepSeek-Model-Based 59](#_Toc192337264)

[Figure 24 Judge Statement – DeepSeek-Model-Based 59](#_Toc192337265)

[Figure 25 Judge Statement – DeepSeek-Adapter-Based (R-8 A-16) 61](#_Toc192337266)

[Figure 26 Judge Statement – DeepSeek-Adapter-Based (R-16 A-32) 62](#_Toc192337267)

[Figure 27 Model Performance Comparison via Ratings 63](#_Toc192337268)

[Figure 28 Best Quantization Precision for Each Model 64](#_Toc192337269)

## 10.2 Tables

[Table 1 Time Frame Plan 29](#_Toc192338790)

[Table 2 Comparison Table 39](#_Toc192338791)

[Table 3 Meta Data - Llama-Model-Based 49](#_Toc192338792)

[Table 4 Meta Data - Mistal-Model-Based 50](#_Toc192338793)

[Table 5 Meta Data - Llama-Adapter-Based (R-8, A-16) 51](#_Toc192338794)

[Table 6 Meta Data - Mistal-Adapter-Model (R-8, A-16) 53](#_Toc192338795)

[Table 7 Meta Data - Llama-Model-Based (R-16, A-32) 55](#_Toc192338796)

[Table 8 Meta Data - Mistral-Adapter-Based (R-16, A-32) 56](#_Toc192338797)

[Table 9 Meta Data - DeepSeek-Model-Based 60](#_Toc192338798)

[Table 10 Meta Data - DeepSeek-Adapter-Model (R-8, A-16) 62](#_Toc192338799)

[Table 11 Meta Data - DeepSeek-Adapter-Based (R-16 A-32) 63](#_Toc192338800)