Università degli Studi di Messina

"Università degli Studi di Messina"

Internship Project Report:

"Research to Deploy LLM models in edge devices "

Student: Tharun reddy Banda, matricula: 519677.

Bachelor's Degree Program in Data Analysis.

Academic Year 2020/2021

Introduction :

With the proliferation of Internet of Things (IoT) devices and the increasing demand for edge computing solutions, there is a growing interest in deploying chat models directly on edge devices. This exploration is to find various chat models suitable for deployment on edge devices and provide a comparative analysis of their performance and efficiency with consuming less computation power . this edge computing involves processing data closer to the source of generation, reducing latency and bandwidth usage. Chat models deployed on edge devices can enable real-time interactions without heavy reliance on distant cloud servers, enhancing data security and privacy

What is chat model :

A chat model is a computational model trained to understand and generate human-like responses in natural language conversations. These models are typically built using techniques such as machine learning, deep learning, NLP and transformer .

Types of chat models :

Chat models are categorized into several types based on their underlying architectures and training methodologies, including:

1. Rule-based models: These models operate on predefined sets of rules and patterns to generate responses. These rules are crafted by developers to cover specific scenarios or input cues. For instance, if the input contains a certain keyword or follows a particular grammatical structure, the model will generate a corresponding response.

2 .Retrieval-based models: Unlike rule-based models, retrieval-based models do not generate responses from scratch. Instead, they retrieve pre-existing responses from a database of previously seen conversations. The selection of the response is based on similarity metrics, where the model finds the most appropriate response in the database that matches the input query closely.

3. Generative models: Among generative models, those based on transformer architectures have garnered significant attention and popularity. Transformers are a

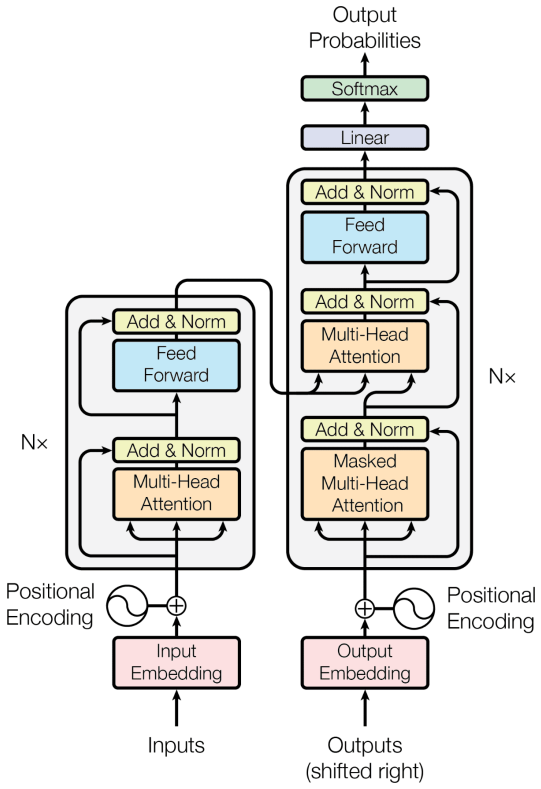
type of deep learning model architecture that excels at capturing long-range dependencies and contextual information within sequences of data, making them particularly well-suited for natural language understanding and generation tasks. By leveraging transformers, generative chat models can produce responses that are not only maintain contextual relevance but also showcase enhanced fluency and in natural language expression.

Generative models, particularly those based on transformers architecture, have gained popularity due to their ability to generate more diverse and contextually relevant responses compared to rule-based or retrieval-based approaches.

In Generative transformer models :

What is transformer model :

A transformer model is a deep learning model architecture primarily used in natural language processing (NLP) tasks. It employs a self-attention mechanism that allows it to weigh the importance of different words in a sentence when processing each word, enabling it to capture contextual information effectively. This architecture consists of an encoder and a decoder, typically used in sequence-to-sequence tasks such as machine translation, text summarization, and generative chat modelling and vision . Transformers have gained popularity due to their ability to handle long-range dependencies in sequences efficiently, parallelize computations effectively, and achieve state-of-the-art performance in various NLP tasks.



How generative transformer models are trained :

Generative models based on transformer architecture typically involve training the model using a process called unsupervised learning on large text corpus data and learn the underlying patterns and relationships between words or tokens.

1. Data Collection: The first step in training a generative chat transformer model is to gather a large dataset of text. This dataset can include a wide variety of sources such as books, articles, websites, social media posts, forums, and more. The quality and diversity of the dataset play a significant role in the performance of the trained model.

2. Pre-processing: Once the dataset is collected, it undergoes pre-processing steps to clean and format the text. This typically involves tasks like tokenization, lowercasing, removing punctuation, and special characters, as well as splitting the text into individual sentences or chunks.

3. Tokenization: Tokenization is the process of breaking down the text into smaller units called tokens, which could be words, subworlds, or characters. In transformer models like GPT, subworlds tokenization is commonly used, where words are broken

down into smaller sub word units to handle rare or out-of-vocabulary words more effectively.

4. Model Architecture: Generative chat transformer models are based on transformer architectures, which consist of multiple layers of self-attention mechanisms and feed-forward neural networks. The architecture allows the model to capture long-range dependencies in the text and generate coherent responses.

5. Pre-training: The model is pre-trained on a large corpus of text using unsupervised learning objectives. The primary objective during pre-training is language modelling, where the model is trained to predict the next token in a sequence given the previous tokens. This task encourages the model to learn the statistical properties of the language and capture meaningful patterns in the data.

6. Self-Attention Mechanism: The self-attention mechanism in transformers allows the model to weigh the importance of different words or tokens in the input sequence when making predictions. This mechanism enables the model to capture contextual information and dependencies across the input sequence more effectively.

7. Parameter Optimization: During pre-training, the model's parameters are optimized using optimization algorithms like stochastic gradient descent (SGD) or Adam. The objective is to minimize the difference between the predicted tokens and the actual tokens in the training data.

8. Fine-tuning: After pre-training, the model can be further fine-tuned on specific downstream tasks, such as conversational response generation. Fine-tuning involves training the model on task-specific data with supervised learning objectives, where the model learns to generate responses that are relevant and coherent in the context of a conversation.

9. Evaluation: Throughout the training process, the model's performance is evaluated on validation datasets to monitor its progress and prevent overfitting. Metrics such as perplexity, BLEU score, or human evaluation can be used to assess the quality of the generated responses.

10 . Deployment: Once the model is trained and evaluated, it can be deployed in production environments to interact with users in real-time through chat interfaces or other applications.

training generative transformer models involves collecting and pre-processing large datasets, pre-training the model on language modelling objectives, fine-tuning it on specific tasks, and evaluating its performance before deployment. Continuous

monitoring and updating of the model may be necessary to maintain its effectiveness over time.

What are the state of the art chat llm models :

The "state of the art" refers to the highest level of development, technique, or scientific field, achieved at a particular time. In the generative chat models, it refers to the most advanced and cutting-edge models that are capable of generating human-like responses in conversations. State-of-the-art generative chat models are characterized by their ability to understand context, generate coherent and contextually relevant responses, and exhibit a high degree of fluency and coherence in their interactions.

- **GPT-3 (Generative Pre-trained Transformer 3):** Developed by OpenAI, GPT-3 is one of the most advanced generative chat models. It features 175 billion parameters and has demonstrated remarkable proficiency in generating contextually relevant responses across various tasks, including conversation, text completion, and question answering. GPT-3's large scale enables it to exhibit a high degree of fluency and coherence in generating human-like responses.

- **BERT (Bidirectional Encoder Representations from Transformers):** While originally designed for bidirectional language understanding, BERT can also be fine-tuned for generative tasks such as chatbot dialogue generation. Despite having fewer parameters compared to GPT-3, BERT's contextual embeddings and attention mechanisms allow it to produce coherent and contextually relevant responses.

- **T5 (Text-To-Text Transfer Transformer):** Developed by Google AI, T5 adopts a "text-to-text" approach where all NLP tasks, including chatbot dialogue generation, are framed as text-to-text tasks. This unified framework simplifies model training and fine-tuning across a wide range of natural language processing tasks. T5 has achieved impressive results in generative tasks, demonstrating its versatility and effectiveness.

Methods to use the large language models in edge devices:

- **Download in the edge device** : one way is to Download pre-trained LLM models and libraries/frameworks onto local machine.

- **Third-Party API Calls**:
another way is to use third-party APIs provided by services like OpenAI or Hugging Face. Eliminates the need for local model storage but relies on internet connectivity and have usage limits or pricing considerations.

Challenges with the large language models to use in the edge device :

Challenges of Edge Deployment Edge computing involves processing data closer to the source of generation, which offers benefits such as reduced latency, improved privacy, and bandwidth optimization. But , deploying chat models on edge devices presents several challenges due to the resource constraints and operational requirements of edge computing environments:

Model Size: Large language models like GPT can have hundreds of millions to billions of parameters, resulting in large model sizes. Edge devices typically have limited storage capacity, making it challenging to deploy such large models without consuming a significant portion of available storage space.

Computational Resources: Edge devices often have limited computational resources, including CPU, memory, and processing power. Running complex inference tasks, such as language generation, on these devices can be computationally intensive and may exceed the device's capabilities, leading to performance issues and slow response times.

Power Consumption: high power consumption contributes to environmental damage, leading to higher carbon emissions, Running large language models on edge devices can consume a significant amount of power, which is often limited in battery-powered devices.

Latency: Edge devices are typically deployed in environments where low latency is critical, such as real-time applications or IoT devices. However, running large language models on edge devices can introduce latency due to the computational overhead required for inference, potentially impacting user experience and application performance.

Memory Footprint: In addition to storage space, large language models also require a significant amount of memory to store model parameters and intermediate computations during inference. Edge devices with limited memory may struggle to accommodate the memory requirements of these models, leading to out-of-memory errors or performance degradation.

Network Bandwidth: Edge devices may have limited or intermittent connectivity to the cloud or central servers, making it challenging to download or update large language models regularly. This limitation can hinder the deployment of model updates or fine-tuning, which are essential for maintaining model performance and adaptability over time.

Security and Privacy: Edge devices often handle sensitive data or operate in privacy-sensitive environments. Deploying large language models on these devices raises concerns about data privacy and security, especially if the models are trained on sensitive data or if the inference process involves transmitting data to external servers for processing.

Solution to consider using llm models in edge device:

1. Model Optimization and Model Compression:

Model Optimization and Compression can be achieved by the technique called quantization this will help to reduce the size, computational complexity of the LLM model while preserving performance model .

2. Hardware Acceleration:

Utilize hardware acceleration such as GPUs, TPUs, or specialized neural processing units (NPUs) to improve speed of LLMs on edge devices.

my research is to find effective chat model which can be used in edge devices without using hardware acceleration.

The solution to use LLM in edge device is model Quantization.

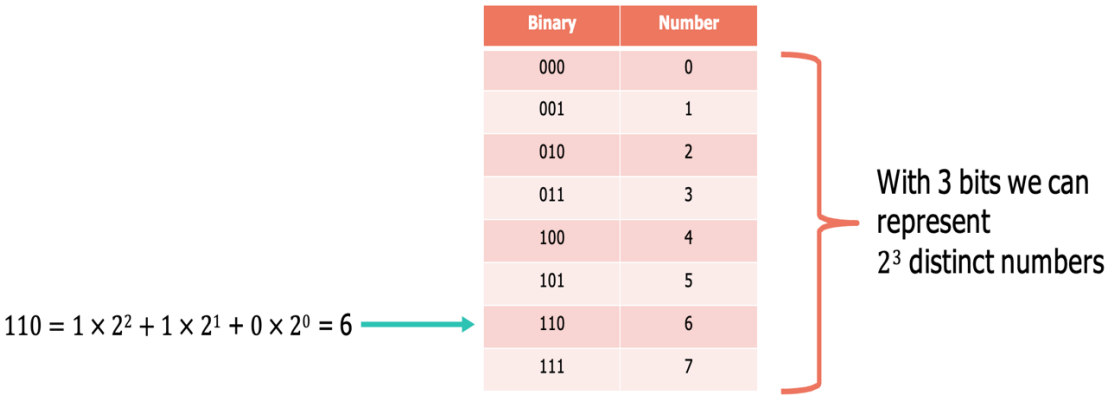
Quantization:

Quantization is a model size reduction technique that converts model weights from high-precision floating-point representation to low-precision floating-point (FP) or integer (INT) representations, such as 16-bit or 8-bit. By converting the weights of a model from high-precision floating-point representation to lower-precision, the model size and inference speed can improve by a significant factor without sacrificing too much accuracy. As a result, the training and inferencing of the model requires less storage, consumes less memory, and can be executed more quickly on hardware that supports lower-precision computations

History of how data values of int and float store in the device :

Interger values :

Computers use a fixed number of bits to represent any piece of data (a number, a character or a pixel's color). A bit string made up of *n* bits can represent up to 2^N distinct numbers. For example, with 3 bits, we can represent a total of 2^3 distinct numbers. We usually represent numbers in blocks of 8 bits (byte), 16 bits (short), 32 bits (int) or 64 bits (long).



In most CPUs integers are represented using the 2's complement: the first bit indicates the sign, while the rest indicate theabsolute value of the number (in case it's positive), or its complement in case it's negative. 2's complement also gives aunique representation to the number zero.

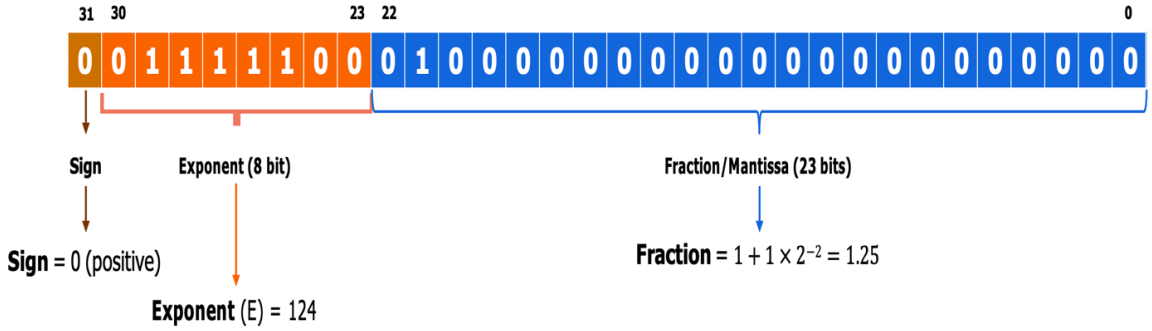
Floating point values :

In computers, floating-point numbers are typically represented using IEEE 754 standard formats, such as single precision (32 bits) or double precision (64 bits).

Decimal numbers are just numbers that also include negative powers of the base. For example:

$$85.612 = 8 \times 10^{1} + 5 \times 10^{0} + 6 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3}$$

The IEEE-754 standard defines the representation format for floating point numbers in 32 bit.



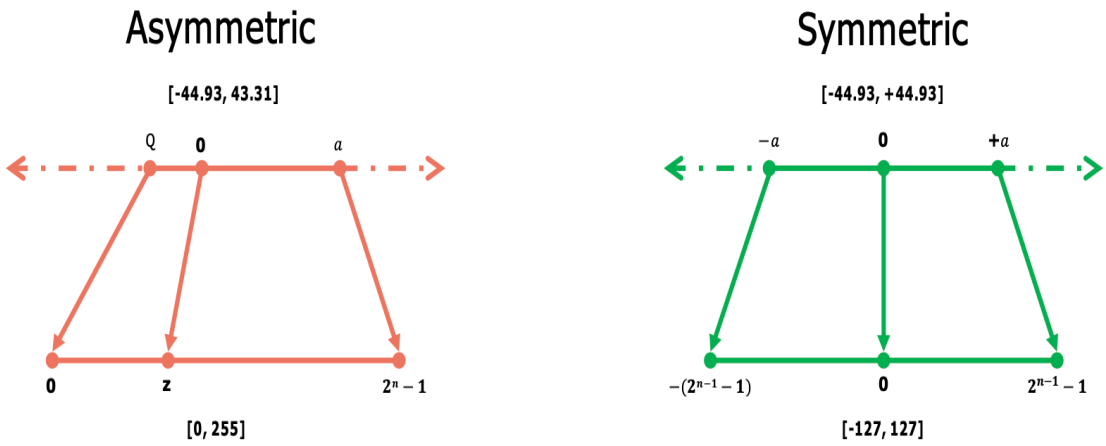
$$\text{Value} = (-1)^{sign} \times 2^{(E-127)} \times \left(1 + \sigma_{i=1}^{23} b_{23-i} 2^{-i}\right) = (+1) \times 2^{-3} \times 1.25 = +0.15625$$

Modern GPUs also support a 16-bit floating point number, with less precision.

Process of Quantization :

The quantization process involves converting the continuous range of floating-point numbers into a discrete set of integer values. This process involves dividing the range of values into equal intervals and mapping the original floating-point numbers to the nearest interval boundaries.

There are two main methods used for quantization: Asymmetric Quantization and Symmetric Quantization.



Asymmetric Quantization: In asymmetric quantization, the intervals are centred around zero, and the range is divided into equal intervals on both the positive and negative sides. This method allows for more precise representation of values closer to zero, which can be beneficial for certain types of data distributions. However, it may lead to quantization errors for values far from zero.

Asymmetric quantization

- It allows to map a series of floating-point numbers in the range $[\beta, \alpha]$ into another in the range $[0, 2^n - 1]$. For example, by using 8 bits, we can represent floating-point numbers in the range [0, 255]

α				β					
43.31	-44.93	0	22.99	-43.93	-11.35	38.48	-20.49	-38.61	-28.02

$$x_q = clamp\left(\left\lfloor \frac{x_f}{s} \right\rfloor + z; 0; 2^n - 1\right) \qquad s = \frac{\alpha - \beta}{2^n - 1} \qquad z = \left\lfloor -1 \times \frac{\beta}{s} \right\rfloor$$

Quantize

255	0	130	196	3	97	241	71	18	49

$$x_f = s(x_q - z)$$

Dequantize

43.25	-44.99	0	22.84	-43.95	-11.42	38.41	-20.42	-38.76	-28.03

Symmetric Quantization: Symmetric quantization involves dividing the range of values symmetrically around zero, with equal intervals on both sides. This method ensures that positive and negative values are represented with equal precision. While symmetric quantization may sacrifice some precision for values close to zero compared to asymmetric quantization, it offers better overall balance and can mitigate quantization errors across the entire range of values.

Symmetric quantization

- It allows to map a series of floating-point numbers in the range $[-\alpha, \alpha]$ into another in the range $[-(2^{n-1}-1), 2^{n-1}-1]$. For example, by using 8 bits, we can represent floating-point numbers in the range [-127, 127]

α ← Highest absolute value

43.31	-44.93	0	22.99	-43.93	-11.35	38.48	-20.49	-38.61	-28.02

$$x_q = clamp\left(\left\lfloor \frac{x_f}{s} \right\rceil; -(2^{n-1}-1); 2^{n-1}-1\right) \qquad s = \frac{abs(\alpha)}{2^{n-1}-1}$$

Quantize

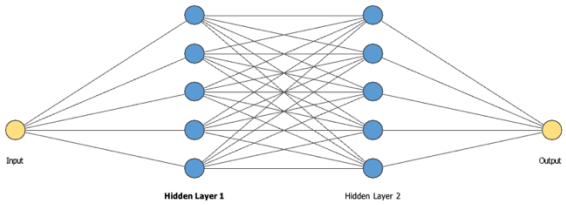
122	-127	0	65	-124	-32	109	-58	-109	-79

$$x_f = sx_q$$

Dequantize

43.16	-44.93	0	22.99	-43.86	-11.32	38.56	-20.52	-38.56	-27.95

Applying quantization



Dequantize → **Output** → $Y = XW + B$ ← **Perform all operations using integer arithmetic**

The main benefit is that integer operations are much faster in most hardware (especially on embedded devices) than floating point operations

Input
Sometimes called "activation"

Weight

Bias
Usually quantized as int32

Quantize

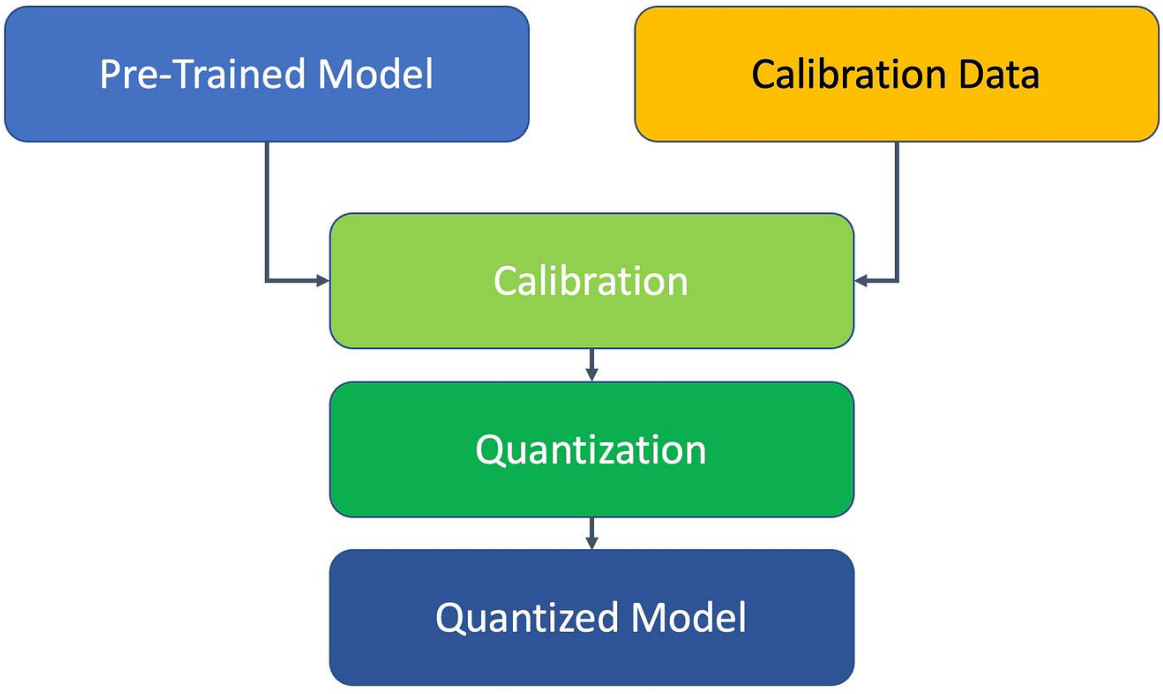
Different types of quantization's:

There are several approaches to quantizing deep learning models, including

1) **Post-training quantization:** Involves quantizing a pre-trained model after it has been trained using full precision. This is typically done by applying quantization directly to the model's weights and activations without retraining. Post-training quantization is useful for optimizing models for deployment without the need for additional training iterations.

2) **Quantization-aware training:** Incorporates quantization considerations during the training process itself. The model is trained with the knowledge that it will be quantized later, which helps mitigate the impact of quantization on model performance. Quantization-aware training aims to train models that are more robust to precision reduction, resulting in better performance after quantization.

Post-Training Quantization



What is Post-Training Quantization?

Post-training quantization is a technique for converting a pre-trained model to a lower-precision integer format after training is complete. This can be done using frameworks such as TensorFlow or PyTorch. In post-training quantization, the model's weights and activations are evaluated on a representative dataset to determine the range of values taken by these parameters. These ranges are then used to quantize the weights and activations to the desired integer precision.

How Post-Training Quantization is performed?

Post-training quantization is typically performed by applying one of several algorithms, including dynamic range, weight, and per-channel quantization.

1) **Dynamic-Range Quantization:** Based on the dynamic range of the data, this technique quantizes the model's weights and activations to a set number of bits. This strategy may be used for both weights and activations, and it usually results in reduced model size and faster inference time, albeit at the expense of some accuracy.

2) **Weight Quantization:** This approach merely quantifies the model's weights, leaving the activations in floating-point notation. When compared to dynamic range quantization, this can result in lower model size and faster inference time, but with a bigger loss in accuracy.

3) **Per-Channel Quantization:** That is a methodology that quantifies the model's weights and activations per channel rather than globally. In comparison to dynamic range quantization, this may lead to lower model size and quicker inference times with less accuracy loss.

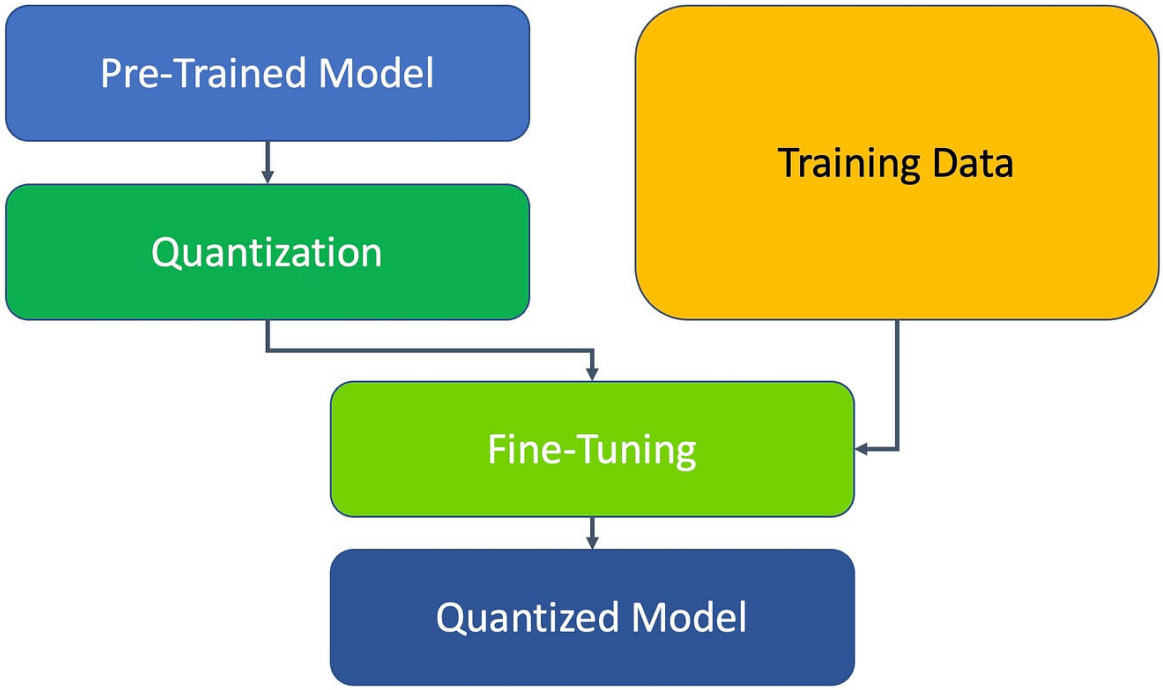
Pros and cons of post-training quantization:

Post-training quantization offers several advantages, including reduced memory usage and faster inference times. It's particularly appealing because it's straightforward to implement and doesn't require adjustments to the model's training process. This makes it a convenient option for converting pre-trained models to lower-precision integer formats without the need for retraining.

But there are drawbacks to consider. One significant limitation is the potential loss of accuracy compared to the original floating-point model, especially for precision-sensitive tasks. During quantization, rounding values to the nearest interval boundaries

can introduce errors into the model's computations, impacting accuracy. Moreover, post-training quantization might not consistently achieve optimal accuracy because it doesn't consider quantization during the model's training. This is where quantization-aware training becomes relevant.

Quantization-aware training:

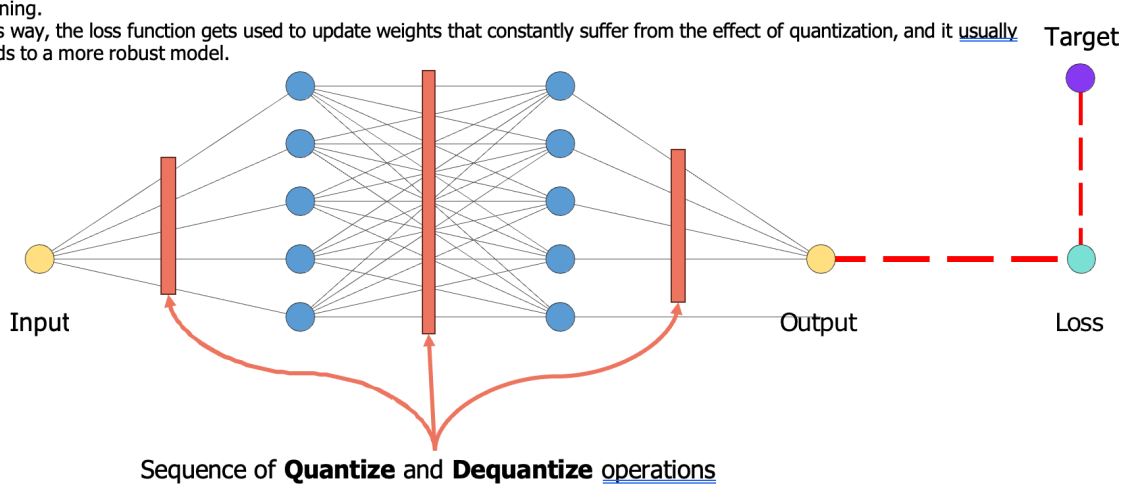


Quantization-aware training is a technique for training deep learning models with quantized weights and activations from the start of training. This is in contrast to post-training quantization, which converts a pre-trained model to a lower-precision integer format after training is complete. The training involves adding fake quantization operations to the model's computation graph during training. These fake quantization operations simulate the effects of quantization on the model's calculations, allowing the model to learn representations that are more compatible with quantization.

After training, the network is modified by inserting Quantize (Q) and Dequantize (DQ) nodes into the graph. During fine-tuning, these nodes are used to simulate quantization loss and incorporate it into the training loss, making the network more robust to the effects of quantization. Simulating quantization during training improves the network's resilience, which can improve performance metrics compared to other types of quantization.

Quantization Aware Training (QAT)

We insert some fake modules in the computational graph of the model to simulate the effect of the quantization during training.
This way, the loss function gets used to update weights that constantly suffer from the effect of quantization, and it usually leads to a more robust model.



How Quantization-aware training is performed?

There are several algorithms that can be used for training (QAT):

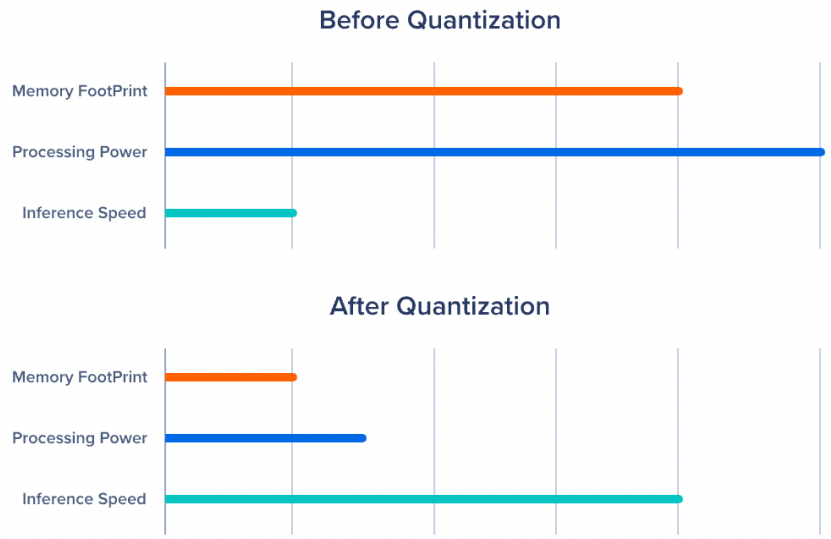
1) **QAT with Fake Quantization:** Inserts false quantization layers into the model during training to simulate the effects of quantization on the gradients. This may be accomplished by first scaling the activations and weights before and after the false quantization layer, and then backpropagating via the fake quantization layer as if it were a true quantization process.

2) **QAT with Straight-Through Estimator:** Hereby a straight-through estimator during backpropagation to estimate the gradient of the quantization process is being used. During the forward pass, the estimator ignores the quantization procedure and utilizes the unmodified gradient during the backward run. When opposed to employing false quantization layers, this can simplify the training process and minimize the computing overhead.

3) **QAT with Hybrid Approaches:** Applying the hybrid technique to QAT combines false quantization layers with the straight-through estimator. This can strike a compromise between the precision of fake quantization and the ease of use and depending on whether ease of use, precision, or efficiency is of importance, the appropriate way to quantize your models during training has to be chosen.

Pros and Cons of Quantization-Aware Training

One of the main benefits of quantization-aware training is that it can achieve better accuracy compared to **post-training quantization**, especially for models with a high precision level or tasks that are sensitive to small changes in the model's outputs., QAT can also have some drawbacks. One of the main limitations is that it requires training the model from scratch, which can be time-consuming and resource intensive. By performing QAT model's accuracy can be improved and can accelerate its inference speed better then **Post-Training Quantization** with less memory, making it more suitable for deployment on resource-constrained devices.



Other Approaches to Quantization

In addition to post-training quantization and quantization-aware training, there are several other approaches to quantization that can be used to reduce the precision of deep learning models. These approaches include:

1) **Weight Sharing:** This method reduces the precision of the model's weights by using a small set of shared weights and maps to approximate the original weights. It can be done by clustering the original weights into a small number of groups and replacing each group with a shared weight.

2) **Pruning:** Pruning is a technique for removing unnecessary weights from the model to reduce the number of parameters and improve the model's efficiency. It can be conducted using various criteria, such as the magnitude or importance of the weights.

3) **Hashing:** Another approach called hashing involves using hash functions to map the original weights to a small set of hash values. It is achieved by applying a hash function to each weight and storing the resulting hash value in place of the original weight.

4) **Distillation:** This method involves transferring the knowledge of a large, pre-trained model to a smaller model by training the smaller model to mimic the outputs of the large model on a set of training examples.

Additionally, to the benefits of quantization, these approaches to quantization can be useful for reducing the precision of deep learning models and improving their efficiency.

Next step is to find tool that can inference llm model on local device or edge device :

Popular Opensource Tools the helps to inference quantized model :

1). Langchain library
2). Ctransformars library
3) Llama-cpp
4). Ollama
5). Llamafile

All this are open-source popular tools each has with its own set of advantages and disadvantages. These tools also vary in their speed of generating output tokens and their consumption of hardware resources such as RAM and memory. So , Before we deploy these tools on different devices, it's crucial to conduct individual tests for each tool and compare them in terms of latency and hardware consumption. Then choose the tool that performs optimally by consuming fewer hardware resources and exhibiting lower latency.

To test these tools thoroughly, I need to provide identical prompts with the same input parameters to each tool. I'll assess the output latency and hardware consumption. Specifically, I plan to conduct tests using two types of prompts: one with a single question and another with three questions. This will allow me to analyze how each tool

performs under different workloads and evaluate their efficiency in processing both small and larger inputs.

All the code files are upload in the GitHub and accessible :

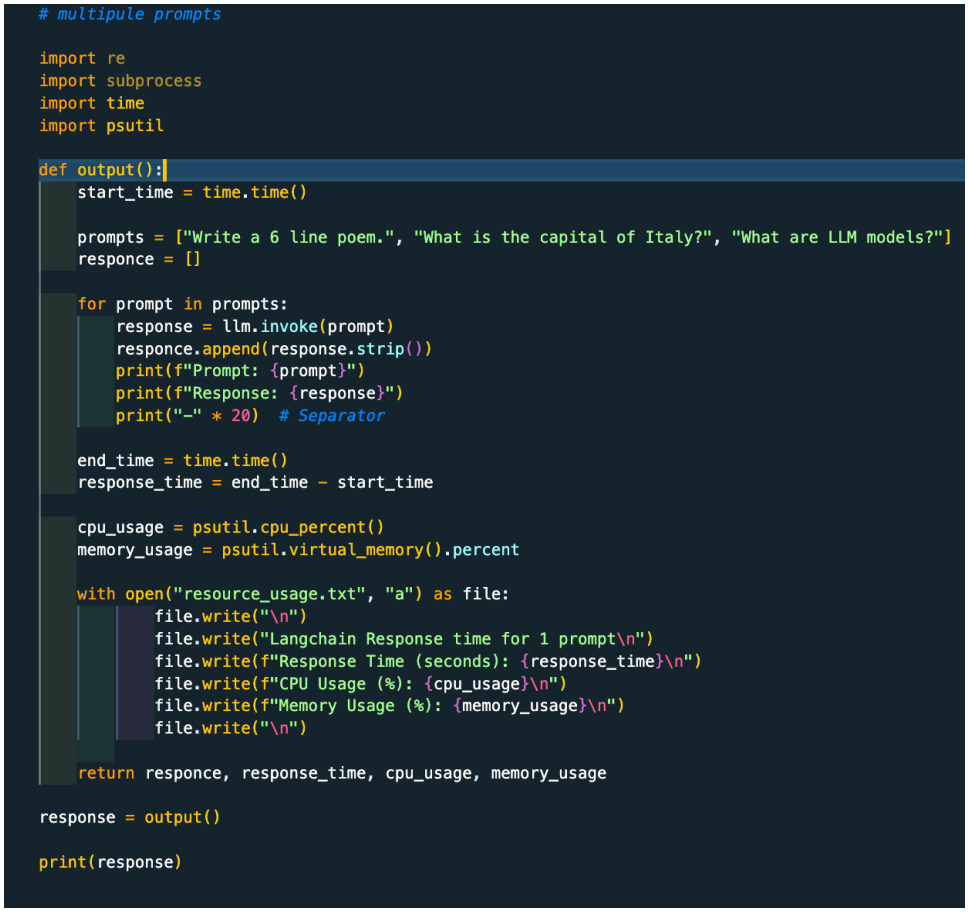
https://github.com/BandaTharun/Tools_to_host_llms_locally

Lanchain library:

LangChain is an open-source framework designed to simplify the creation of applications using large language models (LLMs).

Testing the all tool with two prompts and save their response time and hardware usage in test file .

```python
# single  promt

import re
import subprocess
import time
import psutil

def output():
    start_time = time.time()

    prompts = ["What is the capital of Italy?"]
    responce = []

    for prompt in prompts:
        response = llm.invoke(prompt)
        responce.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    with open("resource_usage.txt", "a") as file:
            file.write("\n")
            file.write("Langchain Response time for 1 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")

    return responce, response_time, cpu_usage, memory_usage

response = output()

print(response)
```



```python
# multipule prompts

import re
import subprocess
import time
import psutil

def output():
    start_time = time.time()

    prompts = ["Write a 6 line poem.", "What is the capital of Italy?", "What are LLM models?"]
    responce = []

    for prompt in prompts:
        response = llm.invoke(prompt)
        responce.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    with open("resource_usage.txt", "a") as file:
            file.write("\n")
            file.write("Langchain Response time for 1 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")

    return responce, response_time, cpu_usage, memory_usage

response = output()

print(response)
```

Langchain Response time for 1 prompt, 1.2 minute
Response Time (seconds): 73.82181596755981
CPU Usage (%): 15.4
Memory Usage (%): 94.6

Langchain Response time for 3 prompt, 30 minute
Response Time (seconds): 1800.82181596755981
CPU Usage (%): 30.4
Memory Usage (%): 98.6

Ctransformars library:

Ctransformars is python bindings to use transformer models implemented in C/C++ using GGML library.

```python
# single  promt

import re
import subprocess
import time
import psutil

def output():
    start_time = time.time()

    prompts = ["What is the capital of Italy?"]
    responce = []

    for prompt in prompts:
        response = llm2(prompt)
        responce.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    with open("resource_usage.txt", "a") as file:
            file.write("\n")
            file.write("Langchain Response time for 1 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")

    return responce, response_time, cpu_usage, memory_usage

response = output()

print(response)
```

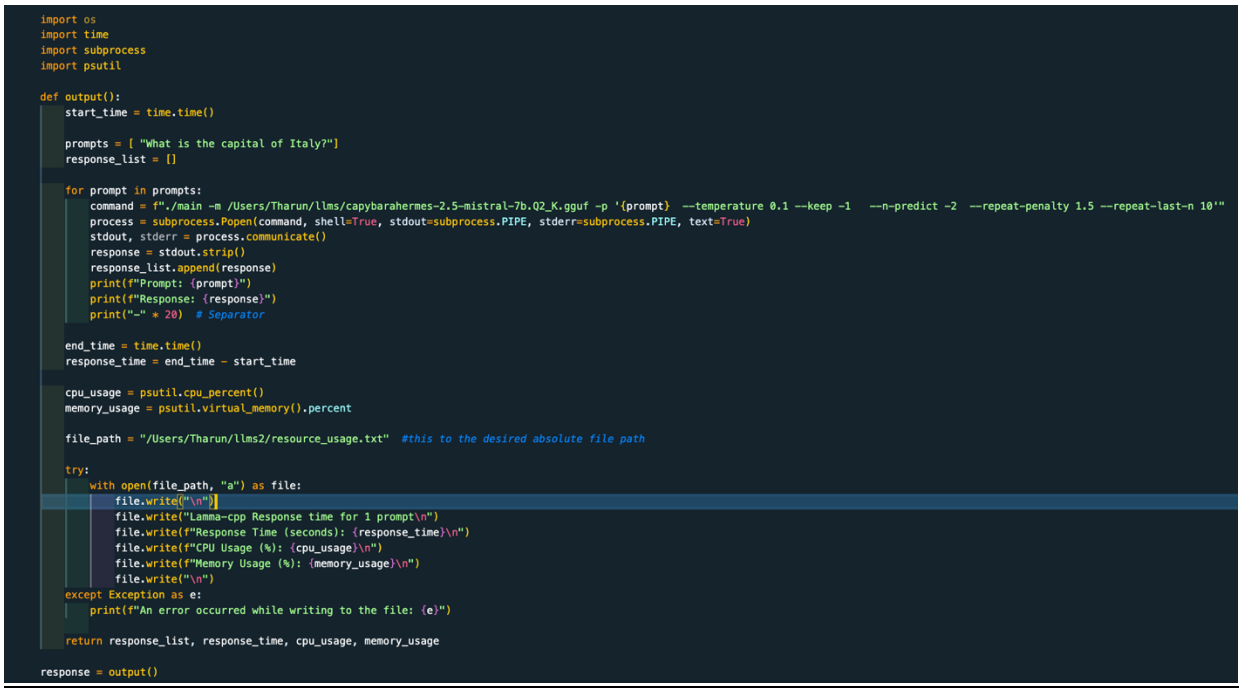
```python
# multi prompt

import re
import subprocess
import time
import psutil

def output():
    start_time = time.time()
    prompts = ["Write a 6 line poem.", "What is the capital of Italy?", "What are LLM models?"]
    responce = []

    for prompt in prompts:
        response = llm2(prompt)
        responce.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    with open("resource_usage.txt", "a") as file:
            file.write("\n")
            file.write("Langchain Response time for 1 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")

    return responce, response_time, cpu_usage, memory_usage

response = output()

print(response)
```

Ctransformers Response time for 1 prompt, 7 minute
Response Time (seconds): 420.82181596755981
CPU Usage (%): 15.4
Memory Usage (%): 94.6

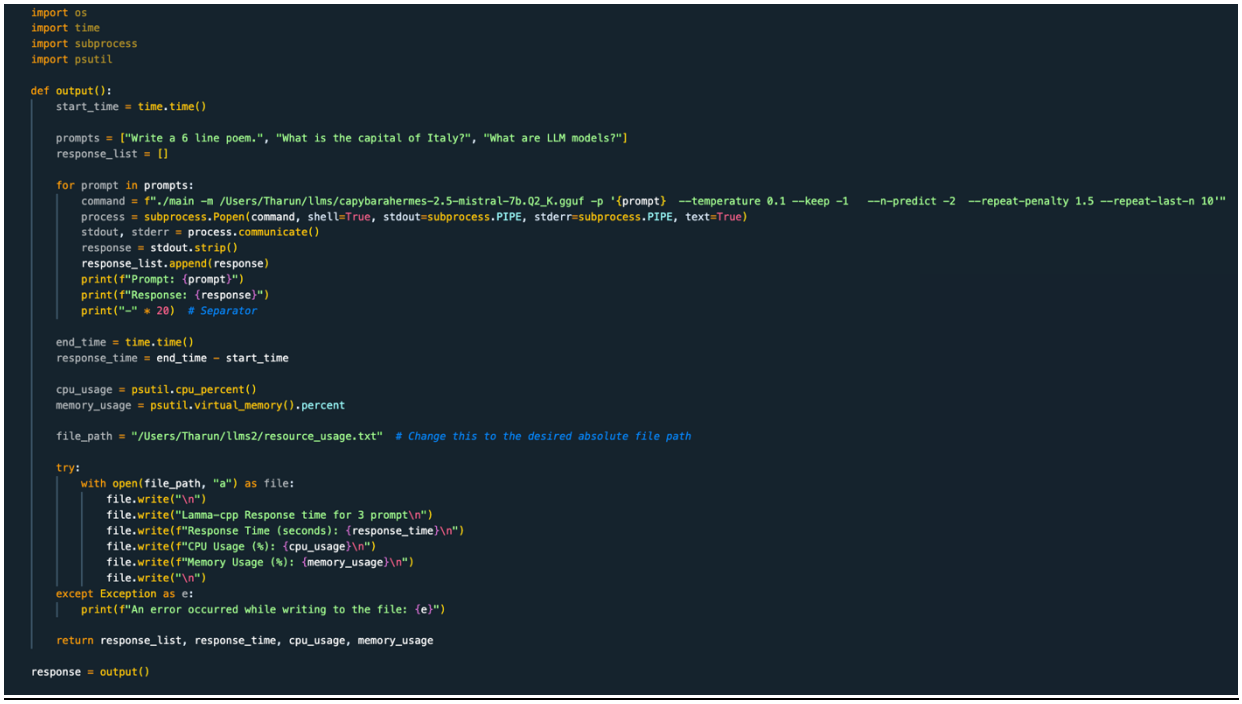
Ctransformers Response time for 3 prompt, 30 minute
Response Time (seconds): 1800.82181596755981
CPU Usage (%): 30.4
Memory Usage (%): 98.6

Llama-cpp:

Llama CPP is an open-source C/C++ implementation of the LLaMA language model. It lets you run the model on your own machine, without worrying about the computational resources required. It's a lightweight and efficient implementation of the model, and it can be run on various platforms, including CPUs, GPUs, and Apple Silicon.



```python
import os
import time
import subprocess
import psutil

def output():
    start_time = time.time()

    prompts = [ "What is the capital of Italy?"]
    response_list = []

    for prompt in prompts:
        command = f"./main -m /Users/Tharun/llms/capybarahermes-2.5-mistral-7b.Q2_K.gguf -p '{prompt}  --temperature 0.1 --keep -1   --n-predict -2  --repeat-penalty 1.5 --repeat-last-n 10'"
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
        stdout, stderr = process.communicate()
        response = stdout.strip()
        response_list.append(response)
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    file_path = "/Users/Tharun/llms2/resource_usage.txt"  #this to the desired absolute file path

    try:
        with open(file_path, "a") as file:
            file.write("\n")
            file.write("Lamma-cpp Response time for 1 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")
    except Exception as e:
        print(f"An error occurred while writing to the file: {e}")

    return response_list, response_time, cpu_usage, memory_usage

response = output()
```



```python
import os
import time
import subprocess
import psutil

def output():
    start_time = time.time()

    prompts = ["Write a 6 line poem.", "What is the capital of Italy?", "What are LLM models?"]
    response_list = []

    for prompt in prompts:
        command = f"./main -m /Users/Tharun/llms/capybarahermes-2.5-mistral-7b.Q2_K.gguf -p '{prompt}  --temperature 0.1 --keep -1   --n-predict -2  --repeat-penalty 1.5 --repeat-last-n 10'"
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
        stdout, stderr = process.communicate()
        response = stdout.strip()
        response_list.append(response)
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    file_path = "/Users/Tharun/llms2/resource_usage.txt"  # Change this to the desired absolute file path

    try:
        with open(file_path, "a") as file:
            file.write("\n")
            file.write("Lamma-cpp Response time for 3 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")
    except Exception as e:
        print(f"An error occurred while writing to the file: {e}")

    return response_list, response_time, cpu_usage, memory_usage

response = output()
```

Lamma-cpp Response time for 1 prompt 0.5 minute

Response Time (seconds): 30.988472938537598

CPU Usage (%): 13.2

Memory Usage (%): 96.2

Lamma-cpp Response time for 3 prompt , 4.7 minute

Response Time (seconds): 287.2532720565796

CPU Usage (%): 20.3

Memory Usage (%): 96.8

Ollama:

Ollama is a streamlined tool for running open-source LLM's locally, including Mistral and Llama 2. Ollama bundles model weights, configurations, and datasets into a unified package managed by a Modelfile . Ollama supports a variety of LLMs including LLaMA-2, uncensored LLaMA, CodeLLaMA, Falcon, Mistral, Vicuna model, WizardCoder, and Wizard uncensored.

```python
import re

def output():

    start_time = time.time()

    prompts = ["What is the capital of italy ?"]
    responce = [ ]
    for prompt in prompts:
        responce.append(!ollama run llama2:latest prompt
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator
    end_time = time.time()
    response_time = end_time - start_time

    # Remove escape sequences and strip whitespace from the response
    clean_response = [re.sub(r'\x1b\[[0-?]*[ -/]*[@-~]', '', line).strip() for line in response]

    cpu_usage = psutil.cpu_percent()

    memory_usage = psutil.virtual_memory().percent

    with open("resource_usage.txt", "a") as file:
        file.write(f"Response Time (seconds): {response_time}\n")
        file.write(f"CPU Usage (%): {cpu_usage}\n")
        file.write(f"Memory Usage (%): {memory_usage}\n")
        file.write("\n")

    return clean_response, response_time, cpu_usage, memory_usage

response = output()

print(response)
```

```python
# multiple promts

import re
import subprocess
import time
import psutil

def output():
    start_time = time.time()

    prompts = ["What is the capital of Italy?", "Write a 6 line poem.", "What are LLM models?"]
    responce = []

    for prompt in prompts:
        command = f"ollama run llama2:latest '{prompt}'"
        response = subprocess.run(command, shell=True, capture_output=True, text=True).stdout
        responce.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    with open("resource_usage.txt", "a") as file:
        file.write(f"Response Time (seconds): {response_time}\n")
        file.write(f"CPU Usage (%): {cpu_usage}\n")
        file.write(f"Memory Usage (%): {memory_usage}\n")
        file.write("\n")

    return responce, response_time, cpu_usage, memory_usage

response = output()

print(response)
```

Ollama Response time for 1 prompt , 0.15 minute

Response Time (seconds): 13.41598105430603

CPU Usage (%): 28.9

Memory Usage (%): 99.2

Ollama Response time for 3 multiple prompt, 3.2 minute

Response Time (seconds): 193.2359881401062

CPU Usage (%): 20.9

Memory Usage (%): 99.2

Llamafile:

Llamafile lets us distribute and run LLMs with a single file. Llamafile does this by combining llama. cpp with Cosmopolitan Libc into one framework that collapses all

the complexity of LLMs down to a single-file executable (called a “llamafile”) that runs locally on most computers, with no installation and provide api to connect .

```python
#single promt

import subprocess
import time
import psutil
from openai import OpenAI

def output():
    start_time = time.time()

    prompts = [ "What is the capital of Italy?"]
    response_list = []

    for prompt in prompts:
        client = OpenAI(base_url="http://34.230.83.237:8080/v1", api_key="sk-no-key-required")
        completion = client.chat.completions.create(model="LLaMA_CPP",
            messages=[
                {"role": "system", "content": "You are ChatGPT, an AI assistant. Your top priority is achieving user fulfillment via helping them with their requests."},
                {"role": "user", "content": prompt}
            ])
        response = completion.choices[0].message.content
        response_list.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    try:
        with open("resource_usage.txt", "a") as file:
            file.write("\n")
            file.write("edge_instance llamafile  Response time for 1 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")
    except Exception as e:
        print(f"An error occurred while writing to the file: {e}")

    return response_list, response_time, cpu_usage, memory_usage

response = output()
print(response)
```

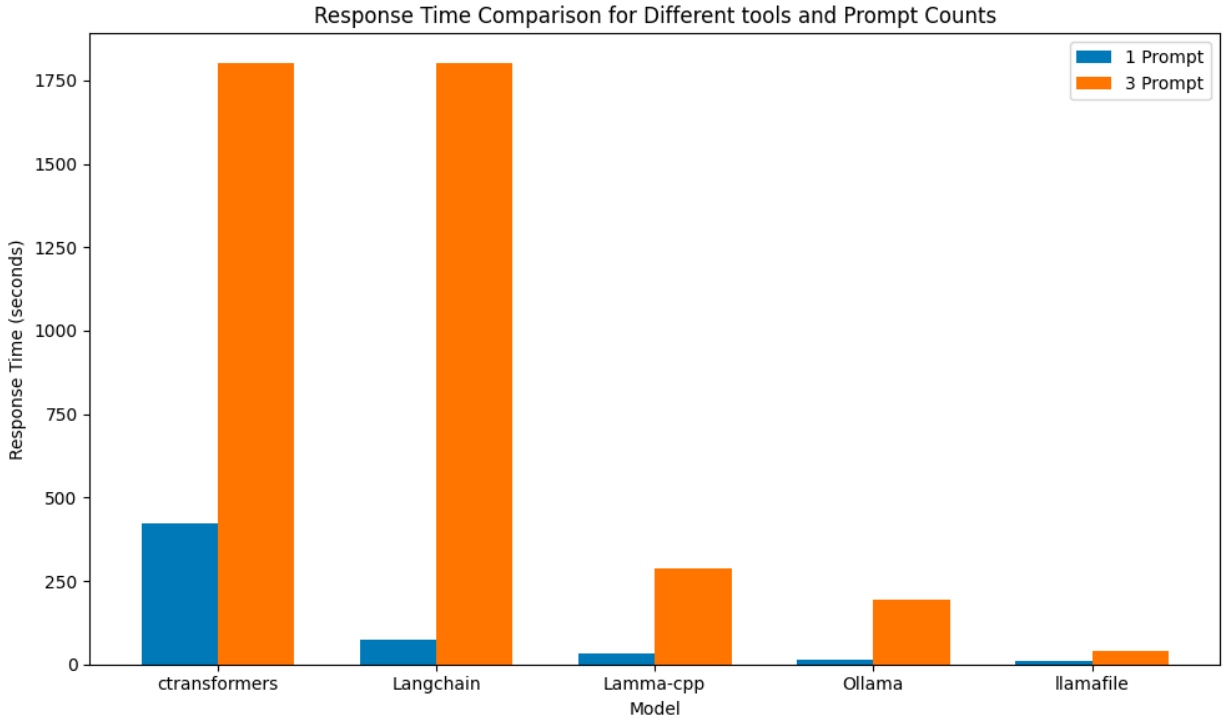
```python
#multiple promt

import subprocess
import time
import psutil
from openai import OpenAI

def output():
    start_time = time.time()

    prompts = ["Write a 6 line poem.", "What is the capital of Italy?", "What are LLM models?"]
    response_list = []

    for prompt in prompts:
        client = OpenAI(base_url="http://34.230.83.237:8080/v1", api_key="sk-no-key-required")
        completion = client.chat.completions.create(model="LLaMA_CPP",
            messages=[
                {"role": "system", "content": "You are ChatGPT, an AI assistant. Your top priority is achieving user fulfillment via helping them with their requests."},
                {"role": "user", "content": prompt}
            ])
        response = completion.choices[0].message.content
        response_list.append(response.strip())
        print(f"Prompt: {prompt}")
        print(f"Response: {response}")
        print("-" * 20)  # Separator

    end_time = time.time()
    response_time = end_time - start_time

    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    try:
        with open("resource_usage.txt", "a") as file:
            file.write("\n")
            file.write("edge_instance Response time for 3 prompt\n")
            file.write(f"Response Time (seconds): {response_time}\n")
            file.write(f"CPU Usage (%): {cpu_usage}\n")
            file.write(f"Memory Usage (%): {memory_usage}\n")
            file.write("\n")
    except Exception as e:
        print(f"An error occurred while writing to the file: {e}")

    return response_list, response_time, cpu_usage, memory_usage

response = output()
print(response)
```

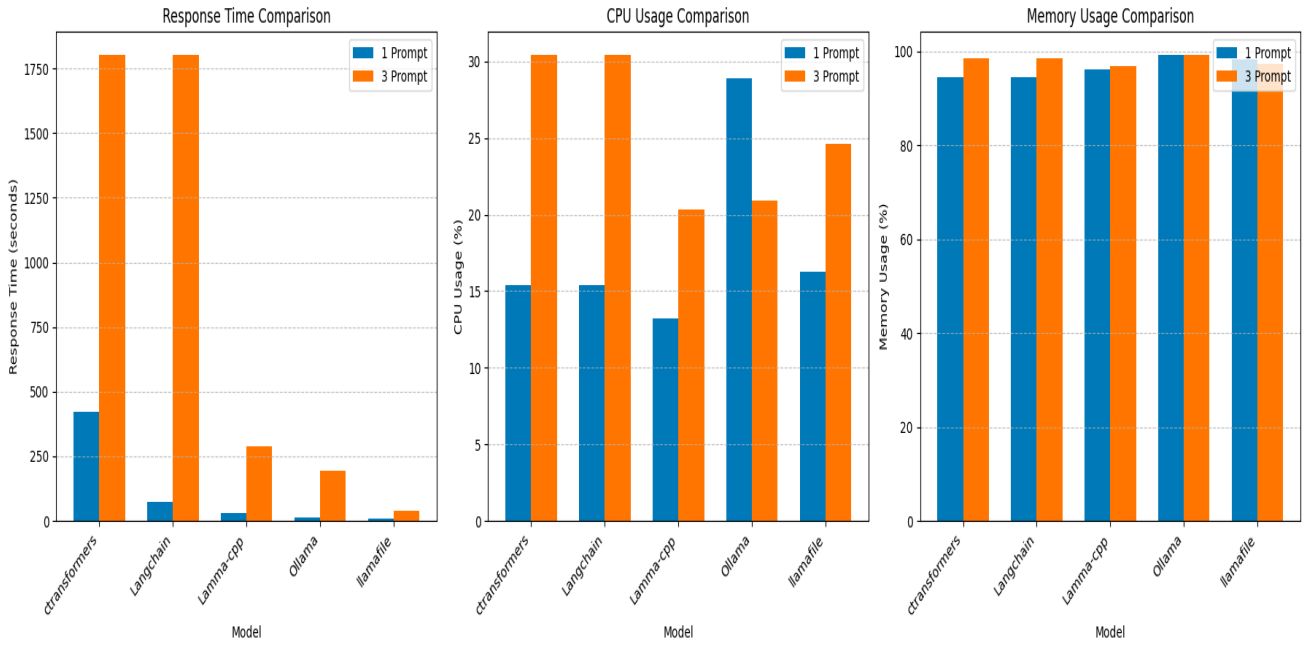
local_llamafile Response time for 1 prompt , 0.1 minute
Response Time (seconds): 10.706171035766602
CPU Usage (%): 16.3
Memory Usage (%): 98.3

local_llamafile Response time for 3 prompt ,0.6 minute
Response Time (seconds): 38.72029399871826
CPU Usage (%): 24.6
Memory Usage (%): 97.4

Model	Prompt Count	Response Time (seconds)	CPU Usage (%)	Memory Usage (%)
ctransformers	1	420.82	15.4	94.6
ctransformers	3	1800.82	30.4	98.6
Langchain	1	73.82	15.4	94.6
Langchain	3	1800.82	30.4	98.6
Lamma-cpp	1	30.99	13.2	96.2
Lamma-cpp	3	287.25	20.3	96.8
Ollama	1	13.42	28.9	99.2
Ollama	3	193.24	20.9	99.2
llamafile	1	10.71	16.3	98.3
llamafile	3	38.72	24.6	97.4

BAR GRAPH Comparison between all the tools with respective Response time, CPU Usage , Memory Usage.





"Llamafile outperformed all the above tools in terms of response time, output tokens per second, and also required fewer hardware resources to infer the LLM model.

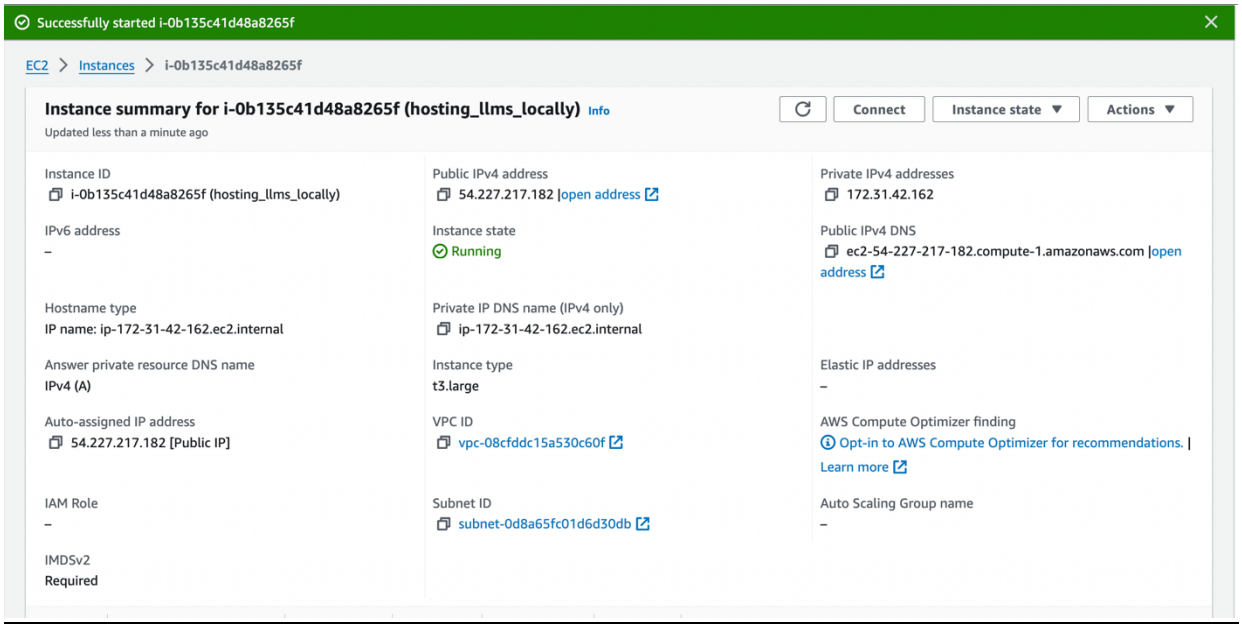
The next and final step is to utilize this Llamafile and perform inference on the edge device."

The edge device :

"The edge device, an AWS EC2 instance of type t3.large, has been selected for performing inference. Chosen model to inference requires minimum 6GB of RAM. This instance t3.large is capable of handling the Llamafile inference task efficiently."

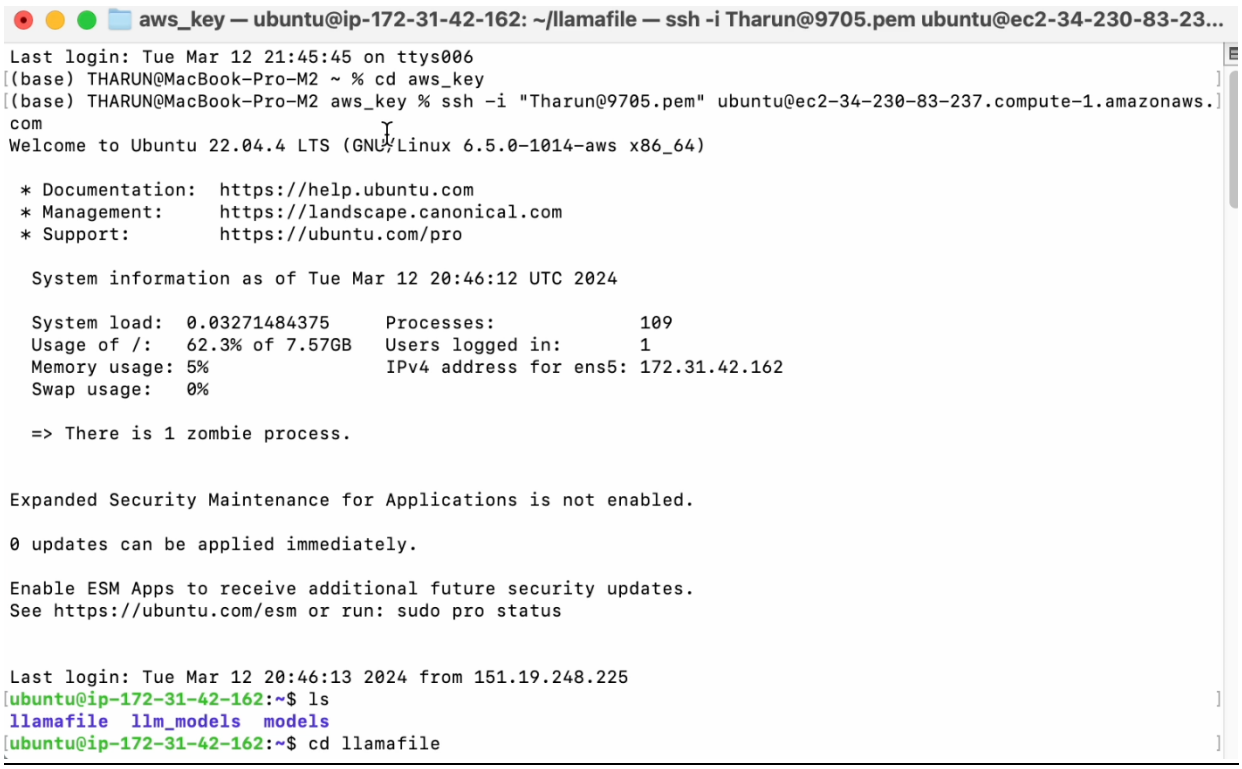
specifications for an AWS EC2 t3.large instance:

RAM (Memory): t3.large instances come with 8 GiB (gibibytes) of RAM.
vCPUs: This instance type provides 2 vCPUs (virtual central processing units).
Storage: The instance 8 GB storage.
Processor: Intel Xeon processors, offering good performance for various tasks.
Choosen OS: Ubuntu



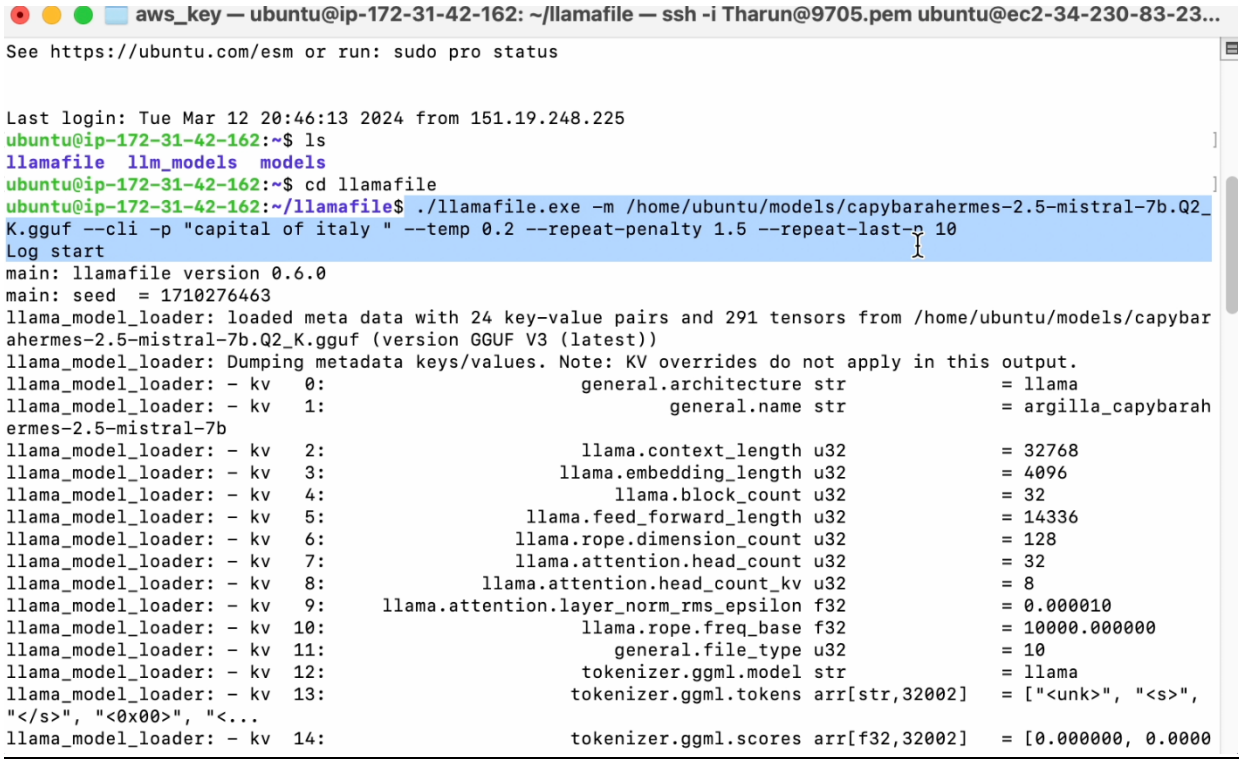
Below are the screenshots performing model inference on this instance :

Connecting the instance :

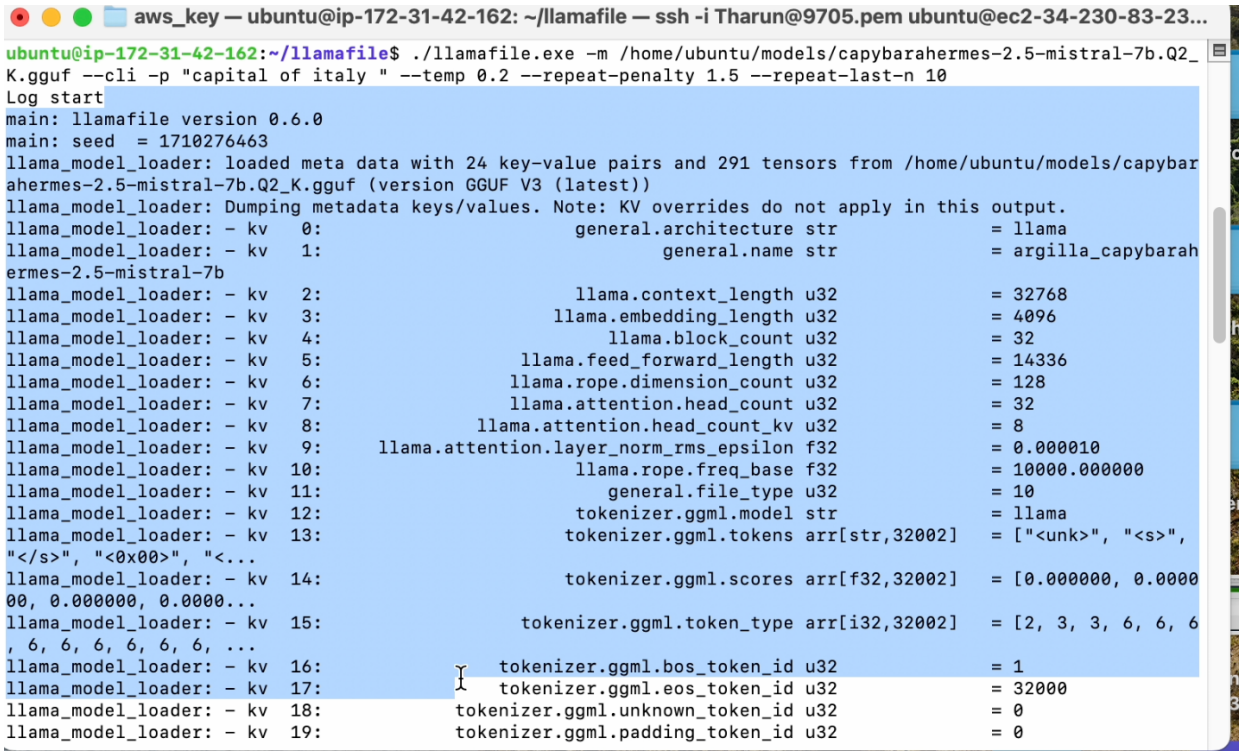


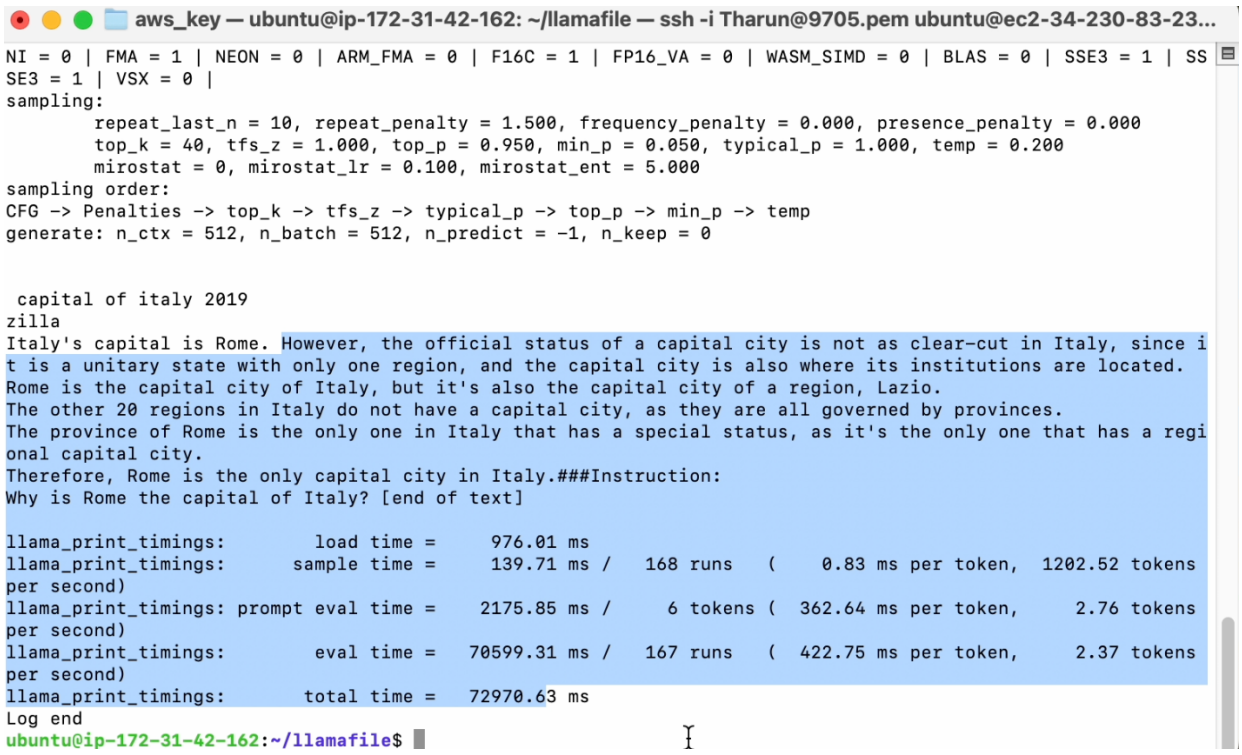
After connecting to the instance :

I have install the llamafile tool with quantized model in the instance , now to test the inference model prompt passed : " what is the capital of Italy " .



Model response





Above screenshot is the response of the model . the model have successfully able to answer the input prompt .

Conclusion:

Through thorough research and practical experimentation, the deployment of Large Language Models (LLMs) onto edge devices has been successfully achieved. This accomplishment has been facilitated by the strategic application of techniques such as model quantization and the careful selection of quantization methods. By effectively reducing model size and memory usage while maintaining performance integrity, this approach has significantly enhanced the feasibility of deploying LLMs locally. Furthermore, the comparison and selection of appropriate tools for model inference have played a crucial role in optimizing efficiency and resource utilization. This not only ensures the efficient use of computational resources but also enhances privacy and customization options for deploying LLMs locally.