# DocuQuest: Optimizing Large Language Model Inference with a Hybrid Cloud-Edge System for Document Summarization

**Arunim Samudra**
UIN: 234009318
arunim_samudra@tamu.edu

## Abstract

This project investigates a hybrid system for optimizing Large Language Model (LLM) inference, focusing on document summarization and analysis tasks. The system leverages both local and cloud-based LLMs to balance computational load, resource efficiency, and task complexity. By employing LLaMA 2 models, we designed an architecture that incorporates a decision module to dynamically route tasks based on complexity, demonstrating the feasibility of deploying LLMs on resource-constrained edge devices like a MacBook Air. The system was evaluated using metrics such as ROUGE scores, BERTScore F1 and latency, validating its ability to achieve high-quality outputs while optimizing resource utilization.The system serves as a foundation for further research into inference optimization and hybrid cloud-edge architectures. This work underscores the potential for such architectures to improve the accessibility and efficiency of LLMs in real-world applications. GitHub Project Video

## 1 Introduction

The advent of large language models (LLMs) has revolutionized natural language processing (NLP), enabling sophisticated applications such as text summarization, question answering, and content generation. However, deploying LLMs in real-world scenarios remains challenging due to the high computational cost, memory requirements, and latency issues associated with inference. These challenges are particularly pronounced in resource-constrained environments, where balancing performance, cost, and latency is critical.

This project aims to address these challenges by developing a hybrid system for optimizing LLM inference in document summarization tasks. The system leverages a dual-model approach: a smaller, quantized LLM deployed locally for lightweight summarization tasks and a larger, cloud-hosted LLM for more complex analyses. By intelligently switching between models based on task complexity, the system achieves an optimal trade-off between resource utilization and performance quality. The system's performance is evaluated on key metrics such as latency and summarization quality (measured using ROUGE scores).

This report details the design, implementation, and evaluation of the hybrid system. The frontend, built using Streamlit, provides an intuitive interface for user interaction, while the backend, developed with FastAPI, efficiently manages local and cloud-based LLMs using GPU resources. The ultimate goal of this project is to demonstrate a scalable and efficient approach to deploying LLMs for document summarization in resource-constrained settings, offering valuable insights into the trade-offs and potential of hybrid LLM systems.

Preprint. Under review.

## 2 Methodology

This project is a desktop application designed to summarize documents and answer user queries. The application is specifically tailored for macOS systems due to the utilization of the MLX library, which enables GPU acceleration on Mac devices. This library was essential as PyTorch does not natively support GPU usage on macOS, making MLX a critical component for achieving efficient local inference.

### 2.1 System Design

The proposed system leverages both local and cloud-based resources to optimize the performance of document summarization and analysis tasks. The key components of the system are:

1. **User Interface (UI)**: Developed using Streamlit, the UI provides an interactive and user-friendly platform for users to upload documents, request summaries, or pose questions about the document content.
2. **Backend with Local LLM**: The backend processes user requests, makes decisions about task allocation, and ensures the efficient use of local and cloud resources. A lightweight LLaMA 3.2 1B Instruct model is deployed locally, leveraging GPU capabilities for faster inference on simpler tasks.
3. **Cloud Server with Bigger LLM**: For more complex tasks, a LLaMA 3.1 8B Instruct model is deployed on the cloud to handle heavy workloads, ensuring better performance for high-complexity documents.

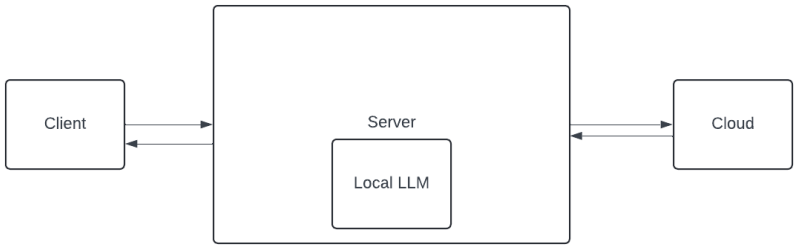



Figure 1: Different Components in the System

### 2.2 Tech Stack

The system incorporates various technologies to ensure scalability, efficiency, and ease of deployment:

1. **User Interface**:
   - **Streamlit**: Chosen for its simplicity and ability to integrate seamlessly with Python-based backends.
2. **Backend**:
   - **FastAPI**: Enables the development of a fast, asynchronous, and lightweight API.
   - **MLX**: Optimizes GPU utilization on Apple Silicon, enabling efficient local deployment on resource-constrained devices.
   - **LangChain**: Implements retrieval-augmented generation (RAG) workflows, enhancing the system's ability to handle document-specific queries.
   - **Local LLM**: **LLaMA 3.2 1B Instruct**, optimized for lower latency and efficient resource usage.
3. **Cloud Server**:
   - **PyTorch**: Used for deploying the cloud-based LLM.
   - **Cloud LLM**: LLaMA 3.1 8B Instruct, chosen for its superior performance on complex tasks.

- **Cloud Resources**: The primary objective was to leverage popular cloud platforms like AWS and GCP for deploying the cloud LLM. After evaluating options, the initial focus was on setting up the application on Google Cloud Platform (GCP) using Vertex AI.
  (a) **Google Cloud Platform (GCP)**: Deploying an ML application on Google Cloud Platform (GCP) involves several components working together to enable efficient cloud-based inference. The deployment process for this project included the following steps:
    – Vertex AI: Used to host the LLaMA 3.1 8B model, providing a managed environment for inference tasks.
    – Model Storage: The model artifacts, including weights and tokenizer files, were stored in Google Cloud Storage (GCS) to ensure quick and efficient access during container initialization.
    – Docker Container: The application code, runtime dependencies (e.g., transformers, torch), and inference logic were packaged into a Docker container. This container was deployed on Vertex AI, ensuring that all required components were encapsulated and consistently executed in the cloud environment.
    – Vertex AI Endpoint: A managed REST API endpoint was created to host the Docker container. This endpoint facilitated seamless interaction with the model, allowing users to send requests
    – **Challenges**: I wrote my python script and created all the setup needed to deploy my app on GCP but while configuring the Docker container, multiple dependency issues arose during the build process. After spending a lot of time debugging and not finding a solution, I looked for another solution.
  (b) **Beam.cloud**:
    – As an alternative to GCP, **Beam.cloud**, a serverless GPU platform, was explored for hosting the cloud LLM. It is fast when operational but I encountered way too many failures due to resource overload or memory constraints, hence I find this option very unreliable in general. But ultimately, I decided stick to this for this project.

### 2.3 Datasets Used

The system was evaluated using three datasets of varying complexities:

1. **XSum (Extreme Summarization)**: Contains short, simple summaries of news articles.
2. **ArXiv-Summarization**: Features scientific papers, requiring structured summaries of abstracts and key content.
3. **GovReport**: Contains lengthy and complex government reports.

### 2.4 Low Level Backend Design

The backend is the core of the system, responsible for routing requests, retrieving documents, and selecting the appropriate model. The backend offers two primary endpoints:

1. **POST /summarize** (For summarization) (Refer Figure2)
   - app.py receives the request.
   - Request handler parses input data.
   - The decision module determines whether to use the local or cloud LLM.
   - The doc_retriever indexes and chunks the document in a non-blocking manner.
   - The appropriate LLM processes the input and generates the summary.
   - Response is sent back to the user.
2. **POST /ask** (for asking questions on the document (RAG)) (Refer Figure3)
   - app.py receives the query.
   - Request handler parses input data.
   - The doc_retriever searches for relevant document chunks.

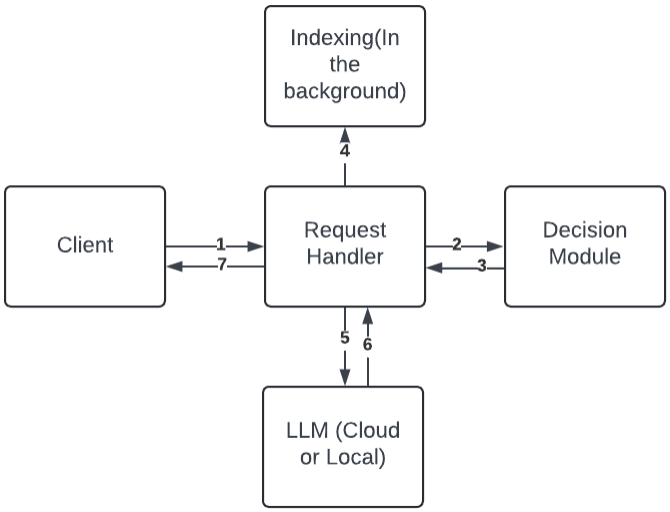



Figure 2: Summarization Workflow

- The decision module determines the appropriate LLM.
- The selected LLM processes the query and generates a response.
- Response is sent back to the user.

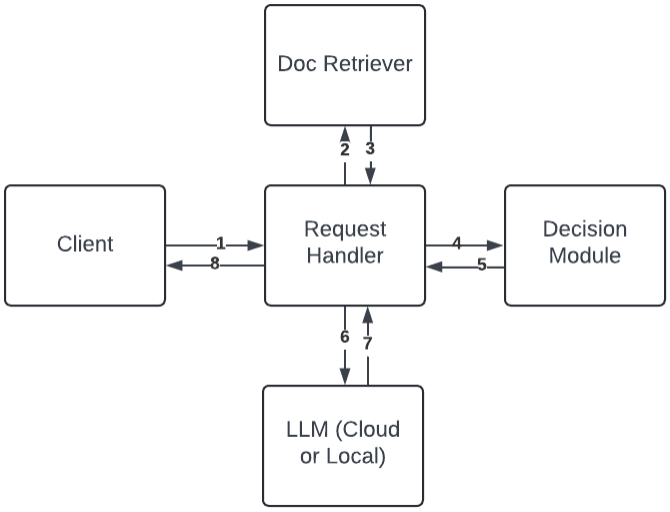



Figure 3: RAG workflow

### 2.5 How Does Decision Moduel which LLM to route to?

The Decision Module is the most critical component of the system, responsible for intelligently determining whether a task should be processed locally or routed to the cloud. This decision is based on two key factors: System Health and Document Complexity.

### 2.5.1 System Health Evaluation

The decision module continuously monitors local system resources to ensure the local LLM (LLaMA 3.2 1B) operates smoothly without overloading the system. Tasks are routed to the cloud if any of the following thresholds are exceeded:

- CPU Usage: >70
- Available Memory: <3000 MB

**Rationale Behind These Thresholds**: The LLaMA 3.2 1B model, although lightweight, requires sufficient system resources to ensure efficient inference. The model, along with its tokenizer and runtime dependencies, typically consumes around 2-3 GB of RAM during execution. Reserving at least 3 GB of free memory ensures there is enough headroom for other system processes to run without degradation. Also, while the GPU (via MLX) performs most of the computation, it is good to keep a threshold on CPU usage as well to not cause bottlenecks or system instability.

### 2.5.2 Document Complexity Prediction

To determine the complexity of a document, the decision module uses a logistic regression classifier trained on key linguistic and structural metrics. These metrics include:

- **Average Word Length**: Measures the average length of words in the document. Longer words typically indicate more complex vocabulary and content.
- **Type-Token Ratio (TTR)**: Ratio of unique words (types) to the total number of words (tokens) in the document. A higher TTR suggests a greater variety of vocabulary, often associated with higher complexity.
- **Average Sentence Length**: Captures the average number of words per sentence. Longer sentences usually indicate more complex sentence structures, which are harder to process.
- **Flesch-Kincaid (FK) Score**: A readability metric that combines sentence length and word syllable count to estimate reading difficulty. Higher FK scores indicate easier readability, while lower scores denote more complex content.

These metrics provide a well-rounded view of document complexity by evaluating both linguistic features (e.g., vocabulary diversity) and structural features (e.g., sentence construction). They were chosen based on their proven correlation with text difficulty across various domains.

#### *Classifier Training*

To ensure reliable predictions, the classifier was meticulously trained on a dataset derived from the three primary datasets: X-Sum, ArXiv, and GovReport—representing varying levels of document complexity.

The first step towards training such a classifier was creating the dataset. For each document in the datasets, we calculated the metrics mentioned previously. Using these metrics served as feature columns, and the document complexity level (easy, medium, difficult) as the target column, a new dataset was made.

Dataset metrics is shown in the Tables 1, 2 and 3.

| Metric | Value |
|---|---|
| Average word length | 4.8924 |
| Average TTR | 0.6572 |
| Average sentence length | 17.1714 |
| Average FK Score | 63.5466 |

Table 1: Statistics for the X-Sum dataset

#### *Classification Results*

A logistic regression classifier was then trained on this dataset. The trained classifier demonstrated excellent performance with an accuracy of **98.3%**, as shown in the classification report4:

| Metric | Value |
|---|---|
| Average word length | 4.5082 |
| Average TTR | 0.2263 |
| Average sentence length | 18.6518 |
| Average FK Score | 49.5321 |

Table 2: Statistics for the ArXiv dataset

| Metric | Value |
|---|---|
| Average word length | 5.6382 |
| Average TTR | 0.2560 |
| Average sentence length | 22.5551 |
| Average FK Score | 32.9416 |

Table 3: Statistics for the Gov-Report dataset

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| 0 | 0.99 | 1.00 | 0.99 | 1788 |
| 1 | 0.99 | 0.97 | 0.98 | 1738 |
| 2 | 0.97 | 0.98 | 0.98 | 1730 |
| **Accuracy** | 0.98 (on 5256 samples) | | | |
| **Macro Avg** | 0.98 | 0.98 | 0.98 | 5256 |
| **Weighted Avg** | 0.98 | 0.98 | 0.98 | 5256 |

Table 4: Classification Report Summary

***Decision Logic in the Decision Module***

This trained model is used inside the decision module to predict the document complexity. It then uses a structured logic flow to allocate tasks based on document complexity:

1. Easy Documents: Processed locally using the LLaMA 3.2 1B model since it is optimized for lightweight tasks, ensuring low latency and efficient resource use.
2. Medium Documents:
   - <=5000 characters: Processed locally.
   - >5000 characters: Routed to the cloud model (The reason for this threshold is explained in the **Results and Analysis** section).
3. Difficult Documents: Always routed to the cloud LLM.
4. Fallback Mechanism: If the cloud LLM is unavailable, tasks default to the local LLM.

## 3 Experimental Setup

### 3.1 Evaluation Metrics

To assess the quality of summarization, the following metrics were used:

1. **ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation)**: Measures the overlap of longest common subsequences (LCS) between the generated summary and the reference summary. ROUGE-L evaluates both content coverage and coherence, emphasizing sentence-level fluency.
2. **BERTScore F1**:Leverages contextual embeddings from pre-trained BERT models to compute semantic similarity between the generated and reference summaries. Unlike ROUGE, which relies on exact token matching, BERTScore evaluates semantic understanding, making it better suited for abstractive summarization.

### 3.2 Experiment Design

The experiment compared the performance of the system under two configurations:

- Using only the local LLM (LLaMA 3.2 1B): Tasks were processed entirely on the local machine, leveraging the MLX library for GPU acceleration.
- Using only the cloud LLM (LLaMA 3.1 8B): Tasks were offloaded to the cloud, utilizing the larger model's enhanced capacity.

### 3.3 Data Collected

Three datasets of varying complexity levels were used: XSum (easy), ArXiv (medium), and Gov-Report (difficult) and 100 random rows were selected from each dataset, resulting in a total of 300 evaluation samples.
For each configuration, the following details were recorded:

- Time Taken: Measured the duration (in seconds) to generate summaries for each sample.
- Length of Text to Be Summarized: Captured the number of characters in each document.

This experimental setup ensures a thorough evaluation of the system's performance under different configurations and document complexities.

## 4 Results and Analysis

### 4.1 Statistical Summary

Table 5, 6 and 7 represent a summarized version of the metrics for the three datasets.

| Metric | Local | Cloud |
|---|---|---|
| Time taken(s) | 7.721500 | **2.349000** |
| rougeL | **0.137550** | 0.116912 |
| BERTScore F1 | 0.855441 | **0.858433** |

Table 5: Mean values for Local and Cloud-based summarization for the X-Sum dataset

| Metric | Local | Cloud |
|---|---|---|
| Time taken(s) | 34.276100 | **4.770200** |
| rougeL | **0.193011** | 0.182547 |
| BERTScore F1 | 0.827140 | **0.835522** |

Table 6: Mean values for Local and Cloud summarization for the ARXIV dataset

| Metric | Local | Cloud |
|---|---|---|
| Time taken (s) | 31.98 | **4.599000** |
| rougeL | 0.122849 | **0.165567** |
| BERTScore F1 | 0.833495 | **0.855592** |

Table 7: Mean values for Local and Cloud summarization for the GOV-REPORT dataset

**Observations:**

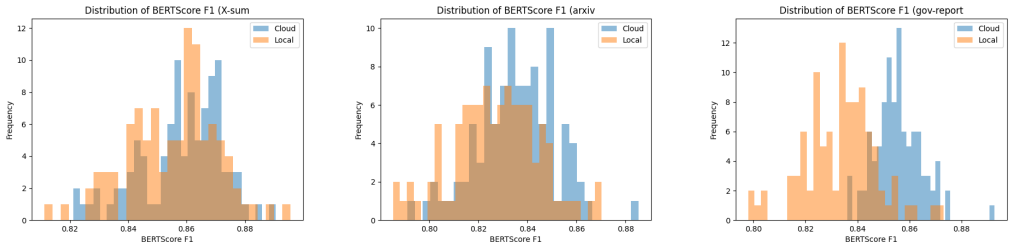
- Efficiency: Cloud systems consistently outperform local systems in terms of time taken, making them ideal for applications requiring rapid processing.
- Content Alignment (ROUGE-L): Local systems often perform better or comparably on structural metrics (ROUGE-L) for datasets like X-Sum and ARXIV but fall behind for GOV-REPORT.

- Semantic Similarity (BERTScore F1): Cloud systems generally achieve higher BERTScore F1 across datasets, indicating their superiority in capturing semantic nuances.

- Another thing to note here is that overall, the local model still performs as good as cloud model. But using it for harder texts is impractical as there is a lot of latency.

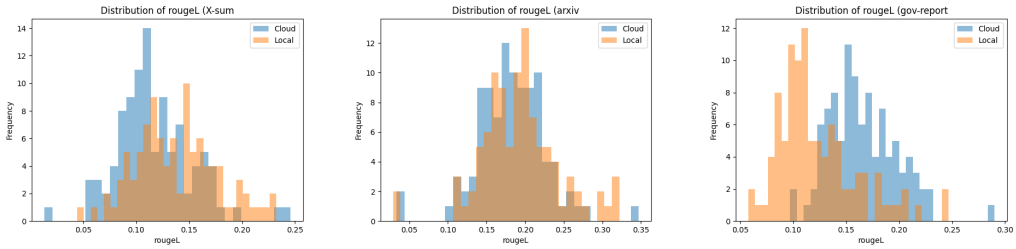
### 4.2 Visual Analysis

Figures 4 and 5 shows the distribution of BERTScore F1 and rougeL value for all three datasets.




(a) X-Sum (b) Arxiv (c) Gov-report

Figure 4: Distribution of BERTScore F1 for different datasets



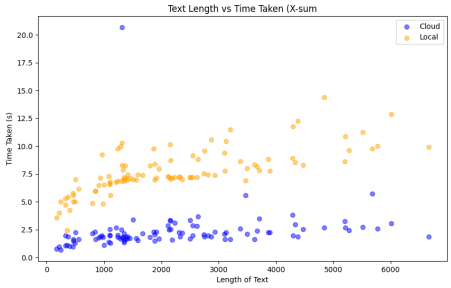

(a) X-Sum (b) Arxiv (c) Gov-report

Figure 5: Distribution of rougeL for different datasets

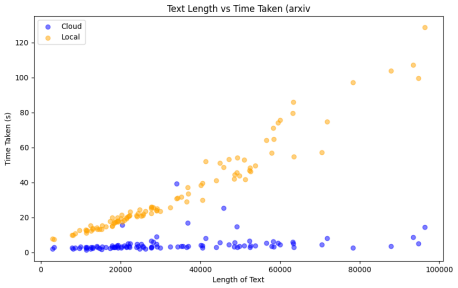
The main observation here is that for ARXIV and X-Sum datasets, the distributions of time taken and performance metrics are fairly similar between Cloud and Local systems. However, for GOV-REPORT, there is significant variability, particularly in the performance of Local systems. This highlights that Cloud systems are superior for consistent, high-quality summarization across diverse datasets, whereas Local systems may perform well in specific contexts but lack the reliability and scalability necessary for more complex or structured datasets.

From the analysis of both models across the metrics, it is evident that for medium-to-hard tasks, the cloud model is better suited due to its significantly faster processing times.

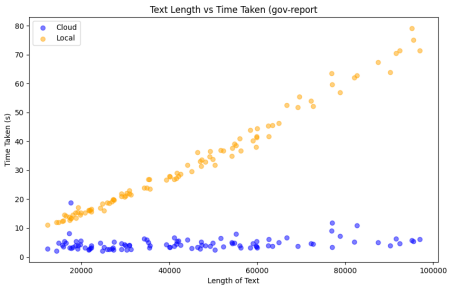
Next, let's examine how the models perform over time as the length of the text to be summarized increases 6




(a) X-Sum




(b) Arxiv




(c) Gov-report

Figure 6: Time taken vs length of text to summarize

**Observation**:

- Cloud models consistently demonstrate low processing times, even for longer texts, showcasing excellent scalability and efficiency. In contrast, local models exhibit a sharp increase in time taken as the text length grows, highlighting poor scalability for larger inputs. However, for shorter texts, performance gap between Local and Cloud is narrower.

Referring to the earlier discussion on determining the threshold for switching to the cloud model, the figures clearly show that the performance of the local model significantly deteriorates after processing texts around 5000–6000 characters in length. Based on this observation, a threshold of **5000 characters** can be set as the limit for switching to the cloud model to maintain efficiency and scalability.

## 5 Weaknesses and Future Work

These results went against my eariler assumptions when I decided to work on this as a course project. I initially assumed that using a local model could be faster than a cloud model, depending on the system's resources. However, this assumption did not hold true. With an Apple M2 Air (16GB RAM), the local model struggled to compete with the speed of the cloud model. It is possible that better-equipped machines, such as high-end MacBooks or desktops with more powerful hardware, could improve local performance. Also, it could still hold true for domains other than document summarization since for summarization, generally longer texts are fed as input and naturally they will take more time. But if the task is relatively simple, such as operating as a conversational bot, this approach proves effective. For instance, in this application, once a document is uploaded, we index it in the background and store the resulting chunks in a vector database. When a user asks a question, we search through these chunks and determine, based on the relevant chunk, whether to use the local or cloud-based LLM. In most cases, the local LLM is sufficient, reducing the reliance on the cloud LLM and saving resources while maintaining efficiency.

**Why Not Use a Cloud API Instead of Deployment?**
A valid question arises: why not use a cloud API instead of deploying the model?

My approach was designed to simulate scenarios where a company or individual might have a fine-tuned LLM trained on a specific downstream task and wishes to deploy it locally. This could be due to privacy concerns—avoiding exposing sensitive data to companies like OpenAI or Google—or to reduce reliance on external services. Additionally, I believe Small Language Models (SLMs) may gain popularity in the future due to their efficiency, reduced computational requirements, and growing advancements in distillation and quantization techniques. These models could leverage end-user devices for local computation, reducing dependency on cloud infrastructure.

Other weaknesses I believe exist in my project:

- **Speculative Decoding (Stretch Goal)**: Speculative decoding was planned as a stretch goal, and I managed to implement it. However, due to time and resource constraints, I could not analyze its performance. Ideally, I would have liked to use a draft model such as LLaMA 3.2 (1B parameters) with a target model like LLaMA 3.1 (8B parameters), or use LLaMA 3.1 (8B) as a draft and LLaMA 3.1 (70B) as the target. Unfortunately, resource limitations, including my system's storage capacity (256GB SSD) and limited free-tier memory on cloud platforms like Beam Cloud, made this infeasible.
- **Mac-Specific Limitations**: The implementation is constrained to macOS and needs to be portable. This limits broader applicability and makes it less versatile for deployment across different platforms.
- **Hardware Metrics Analysis**: I could not analyze GPU or RAM utilization metrics due to macOS-specific limitations. Tools like pynvml are not compatible with macOS, restricting my ability to gather and evaluate system-level hardware performance data.
- **Analyzing Decision Module Accuracy**: Test the decision module under varying system loads to evaluate its robustness and accuracy. Additionally, experiment with a larger and more diverse set of documents to determine if the module classifies tasks correctly and consistently under different scenarios.

## 6 Conclusion

In this project, we successfully designed and implemented a hybrid system for optimizing Large Language Model (LLM) inference in the context of document summarization and analysis tasks. The system leveraged both local and cloud-based LLMs to strike a balance between performance, latency, and resource usage.

The project demonstrated the viability of deploying resource-efficient LLMs on edge devices, such as a MacBook Air, for simpler tasks while offloading complex tasks to cloud-based LLMs. Our architecture integrated a decision module to dynamically route tasks based on their complexity, ensuring an optimal balance of computational load and latency. We evaluated the system using metrics such as latency, and ROUGE scores, confirming its effectiveness in addressing the trade-offs inherent in hybrid LLM deployment. Another key outcome of this work was the development of an end-to-end pipeline that streamlined the summarization and question-answering workflow while optimizing resource utilization.

Overall, this project highlighted the potential for hybrid cloud-edge architectures to make LLMs more accessible and efficient, paving the way for their integration into a wide range of real-world applications. The methodologies and findings from this project can serve as a foundation for further research into deploying large-scale LLMs in resource-constrained environments.