Contents

[Pre-Requisite for Implementing QA bot 2](#_Toc172059998)

[System Architecture 4](#_Toc172059999)

[Deployment Instructions 6](#_Toc172060000)

[Scalability of QA Bot 7](#_Toc172060001)

[1. Efficient Model Architecture 7](#_Toc172060002)

[2. Distributed Computing 7](#_Toc172060003)

[3. Load Balancing 7](#_Toc172060004)

[4. Caching and Batch Processing 7](#_Toc172060005)

[5. Monitoring and Maintenance 8](#_Toc172060006)

[6. Ethical Considerations and Compliance 8](#_Toc172060007)

[Performance Evaluation 9](#_Toc172060008)

[1. Response Evaluation: 9](#_Toc172060009)

[2. Retrieval Evaluation: 9](#_Toc172060010)

# Pre-Requisite for Implementing QA bot

Below are the pre-requisites for implementing QA bot using LLM:

1. **LLM:** We have selected **TinyLlama1.1B** which takes 2.3 GB disk space because of resource crunch. Also it provides API\_KEY for using this LLM free of cost.
2. **Embedding Model:** We have selected Sentence Transformer
3. **Vector Database:** We have selected Pinecone Vector Database as it allows to have one free index created.

**Creating API Key for Pinecone:**

Register on Pinecone site and create API Key for using Pinceone in this solution

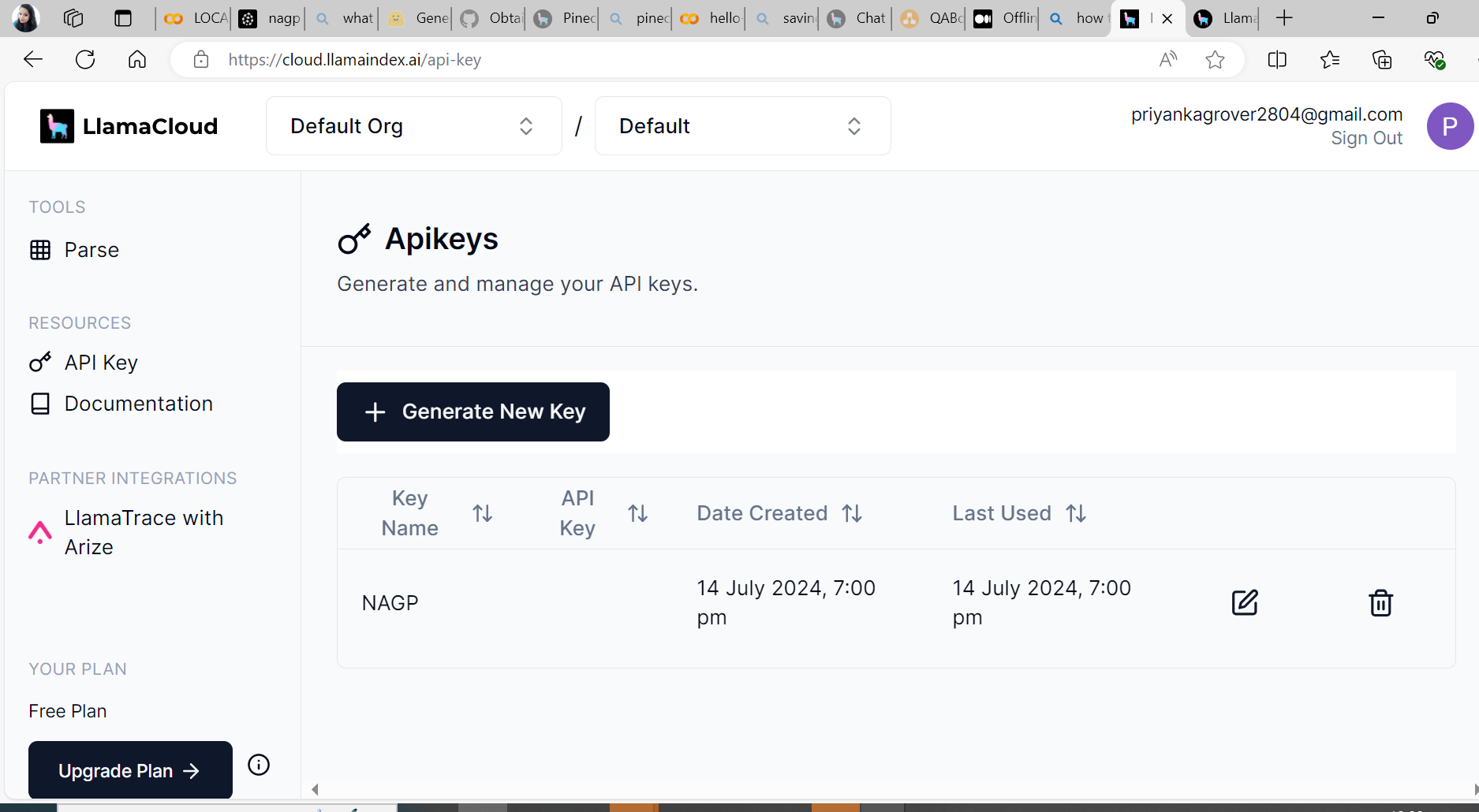
A screenshot of a computer

Description automatically generated

**Getting LLAMA\_CLOUD\_API\_KEY:**

Register to LLAMA Cloud Portal and API Key in user profile as shown below:

[LlamaCloud (llamaindex.ai)](https://cloud.llamaindex.ai/)



# System Architecture

Below is System Architecture for QA bot which will answer the user queries using Supporting Document and Tiny Llama as LLM.

We have divided it into 2 phases:

1. **Ingestion Phase:** In which we load supporting document, extract text out of it and do chunking and embedding of chunks and lastly storing it in Vector Database
2. **Retrieval Phase**: In which, user will query the QA bot. This query will be converted into embeddings using same embedding model used while storing the data in ingestion process. System will query Vector DB based on Similarity search and top 5 matching records will be retrieved. Afterwards, query will be sent to LLM along with Retrieved context and response is generated for the user query

A diagram of a computer

Description automatically generated

This implementation involves below steps:

1. **Loading and Parsing Document:** This step involves loading PDF file and parsing using Llama Parser to extract text from PDF file. *In our scenario, we have uploaded “Assignment Support Document ” in Google Colab and installed LLAMA parser for parsing and extracting the text from PDF file.*
2. **Document chunking:** This process involves breaking down large document into manageable chunks or nodes. *We have used Sentence Splitter to split the document into nodes.*
3. **Embedding extraction**: This step involves capturing the essential features of each document chunk. *We have used Sentence Transformers to generate embedding of each chunk.*
4. **Vector database storage:** This step involves storing these embeddings in a vector database for efficient retrieval. *We have used Pinecone Vector Database to store these Vectors*
5. **Relevant context retrieval:** This step involves retrieving context pieces relevant to the user’s query from the stored database.
6. **Querying the LLM:** This step involves presenting the refined context and user query to the LLM for an accurate response. We have used Tiny Llama as LLM here

# Deployment Instructions

Below are the deployment instructions for running the solution in Google Colab

1. Entire code is hosted on Google Colab. Kindly browse to below link:

[LOCAL\_CODE.ipynb - Colab (google.com)](https://colab.research.google.com/drive/1CJbxmCfDPGhc-Hbz168Lsi7muW9T7ufd#scrollTo=qjxhUs5_IFLR)

1. If above link does not work, take Jupyter Notebook from zip file (QuestionAnswerBot) and upload in Google Colab
2. Connect to runtime with T4 GPU configuration
3. Now upload Assignment Support Document in Google Colab

A screenshot of a computer

Description automatically generated

1. Run each cell one by one. It firstly include all the installation that needs to be done, then importing the modules. After that it has code for implementation.

# Scalability of QA Bot

For scalability of QA bot, we can deploy the LLM application on Cloud e.g. AWS instance and expose It as an endpoint under load balancer.

Below are key strategies to consider for scaling the LLM :

1. Efficient Model Architecture

Before deployment, it's essential to optimize the architecture of the LLM to balance performance with computational efficiency. Techniques such as model pruning, quantization, and knowledge distillation can reduce the model size and computational requirements, making them more manageable to deploy at scale.

**Model Pruning:** Reducing the number of parameters in the model that contribute minimally to its performance.

**Quantization:** Converting the model from floating point to integer formats to reduce the size and increase the inference speed.

**Knowledge Distillation:** KD is a **technique for reducing the high computational demand** of large language models (LLMs). It involves including the outputs of larger, pre-trained model into the training process of a smaller model

## 2. Distributed Computing

Utilizing distributed computing frameworks allows LLMs to handle larger volumes of requests and datasets. Frameworks like Apache Spark or Dask enable parallel processing and data management at scale, distributing the workload across multiple machines.

**Implementation:** Deploy the LLM across a cluster of servers using a tool like Kubernetes, which can manage the containers and scale them according to the load.

## 3. Load Balancing

Effective load balancing ensures that computational resources are utilized efficiently and can respond dynamically to fluctuations in demand. Techniques include **horizontal scaling** (adding more machines) or **vertical scaling** (adding more power to existing machines).

**Use Case:** Using a load balancer to distribute user requests evenly across a pool of servers, each running an instance of the LLM.

4. Caching and Batch Processing

Caching frequent queries can significantly reduce the need to repeatedly process the same requests, thereby saving computational resources. Batch processing can also optimize resource usage by grouping similar tasks together.

**Example:** Implement a Redis cache to store the results of common queries, reducing the latency and load on the LLM.

***We have used In-Memory Cache in our solution to save the context of chat.***

## 5. Monitoring and Maintenance

Continuous monitoring of system performance and regular maintenance are crucial for sustaining the LLM’s effectiveness at scale. Monitoring tools can help detect and address issues like performance bottlenecks, unusual patterns, or system failures in real-time.

**Tools:** Use Prometheus for monitoring metrics and Grafana for visualizing them in real-time, enabling quick identification and resolution of issues.

***We have used MRR and Hit Rate Metrics to evaluate the performance of model***

## 6. Ethical Considerations and Compliance

As LLMs are scaled, ensuring they adhere to ethical guidelines and regulatory compliance, particularly regarding data privacy and bias, is essential. Regular audits and updates to the models should be conducted to align with these standards.

# Performance Evaluation

Evaluating the performance of a LLM is crucial for optimizing LLM applications. There are two key aspects to consider:

## Response Evaluation:

This module assesses whether the generated response aligns with the retrieved context and query. For example, it checks if the response is faithful to the source from which it was generated.

In the solution, we have below section for evaluating the response of LLM model.

A screenshot of a computer

Description automatically generated

## Retrieval Evaluation:

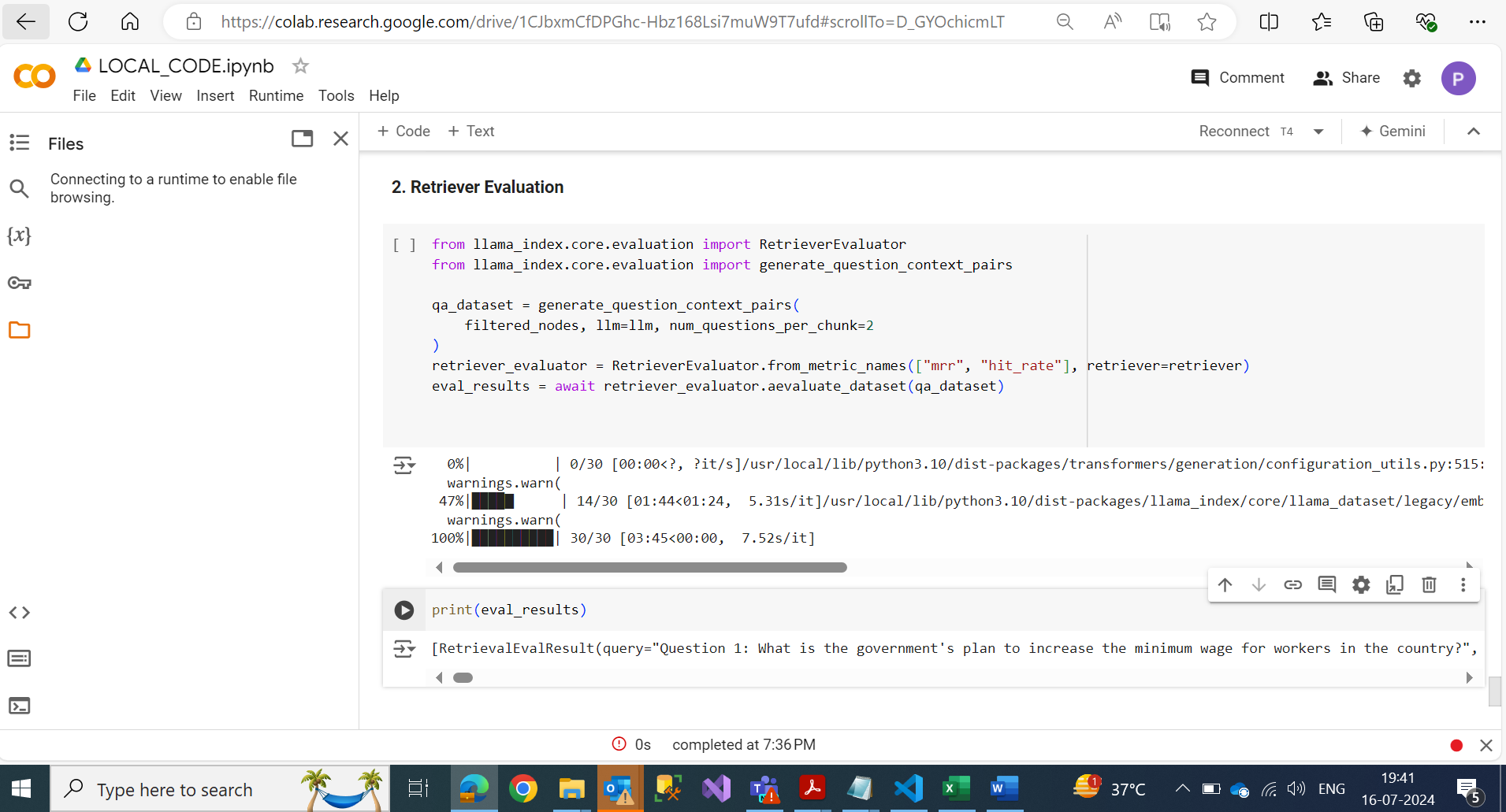
This module assesses the relevance of retrieved sources to the input query. It compares what was retrieved to the expected nodes.

For Retrieval Evaluation : We have 2 key metrics:

1. **Hit Rate**: Hit Rate tells us if the LLM found any relevant information.
2. **MRR** **(Mean Reciprocal Rank):** MRR tells us how high it ranked the most relevant information.

Both metrics are crucial for building effective LLM retrievers and shaping the future of information access.

Code for the Evaluation is as follow:



Below snippet shows some questions which are created by generate\_question\_context\_pairs() function and got expected answers with Hit Rate = 1 and MRR= 1

