Build your own Production ready RAG-System at Zero Cost.

Gathnex

.

Follow

10 min read

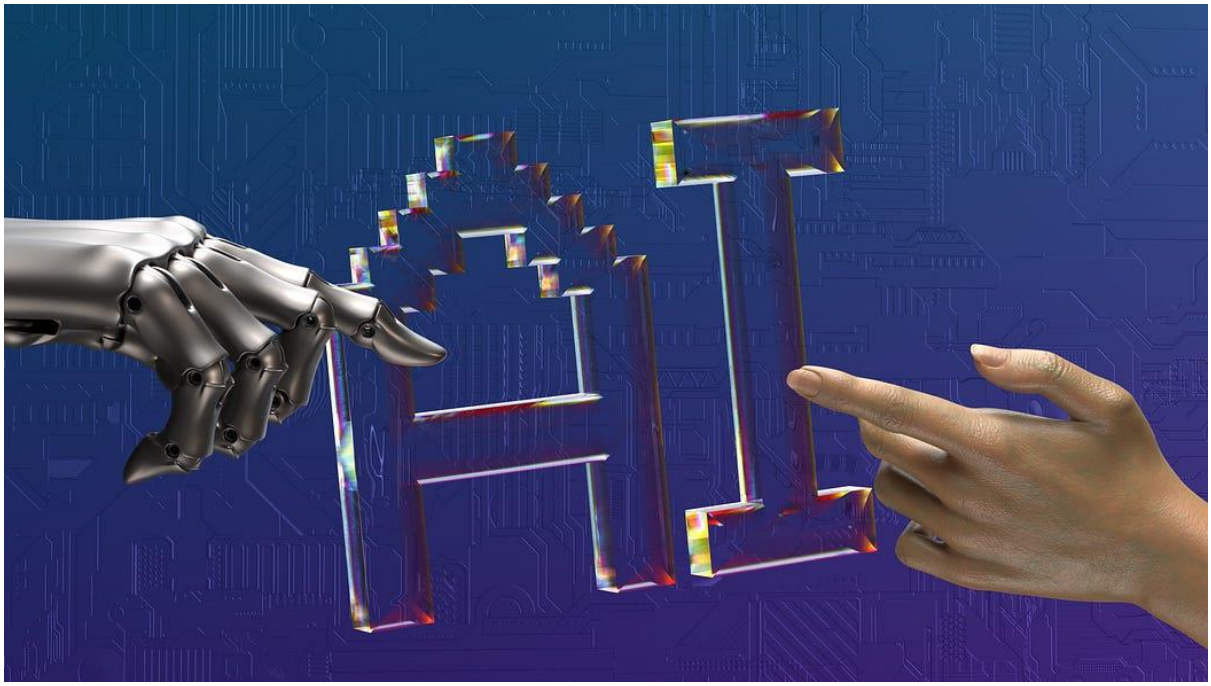
.

Nov 27, 2023

317

1

In this article, we have implemented a production-ready Advanced RAG System or Vector Search, at zero cost😎. This is the same system we have deployed in our production environment using Azure-based products at $200/month. Now, we are going to demonstrate how to implement it for free of cost. Sounds interesting, right? Let's deep dive into RAG 🚀 .



What is RAG?

Retrieval-Augmented Generation (RAG) is a popular technique that makes Large Language Model (LLM) generation more effective and accurate by incorporating an external knowledge base.

we are going build this system and deploy it on production level using the free recourses that are available.

Table of Contents:

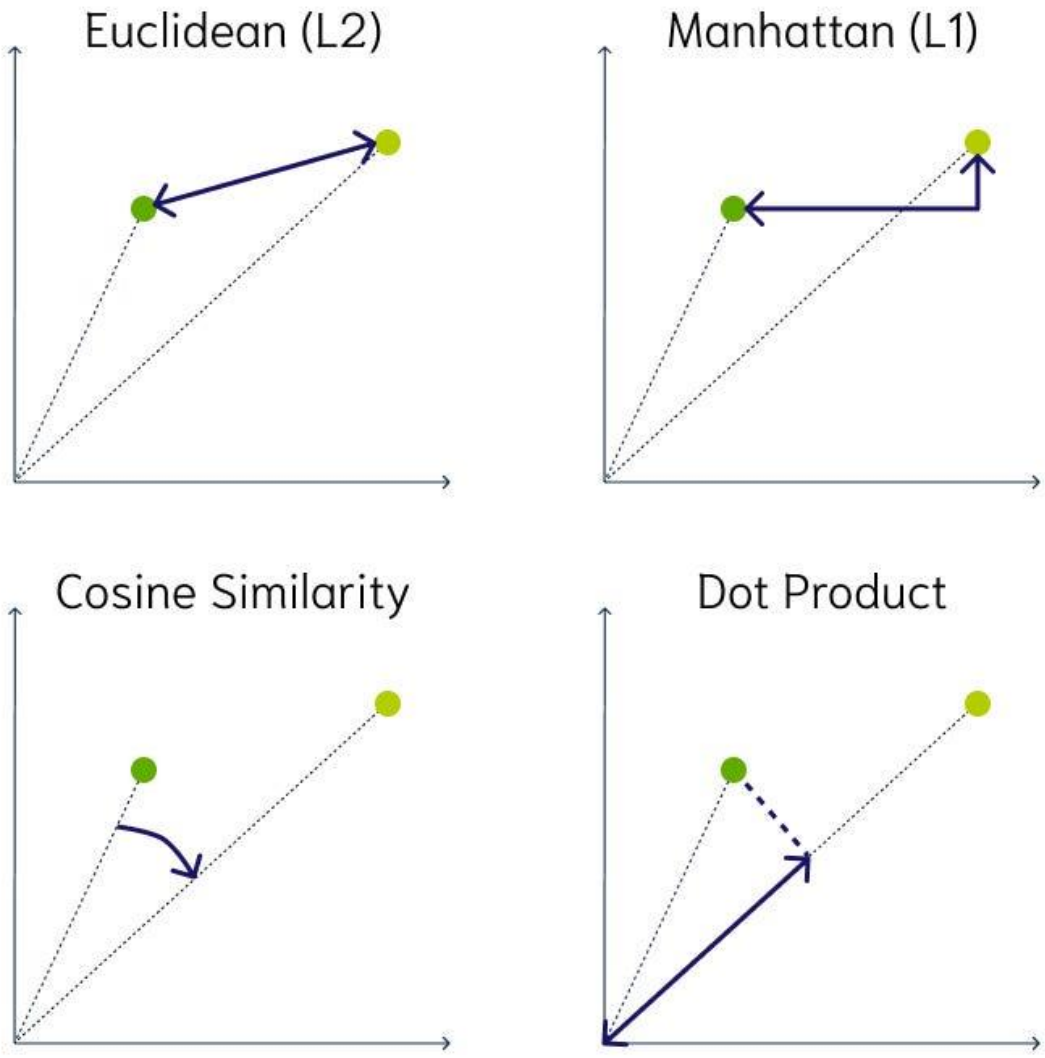
1. Retrieval Augmented Generation.

RAG provides a way to optimize the output of an LLM with targeted information without modifying the underlying model itself; that targeted information can be more up-to-date than the LLM as well as specific to a particular organization and industry. That means the generative AI system can provide more contextually appropriate answers to prompts as well as base those answers on extremely current data.

It’s basically works on two stages:

Retrieval Stage

When user send a query, the RAG system first scours its connected database to find relevant information. where the data’s are converted into vector embeddings and stored in higher dimension.



there are many types of methods are used to find relevant information from the database which are Cosine similarity, Dot Product similarity, Euclidean distance, etc. we'll cover this working in separate article.

Generation Stage

Once the relevant data is retrieved, the generative component of the model synthesizes this information, integrating it with what it has learned during its training to create coherent, contextually rich responses.

2. Tech Stacks

The tech stacks we are going to using for this end-to-end deployment is:

Pinecone: Pinecone vector database is a vector-based database that offers high-performance search and similarity matching. Pinecone offers a basic free version of vector databases, so we plan to utilize that for our vector storage.

OpenAI: We are going to use the free tier of OpenAI GPT models for completion and embedding models for converting the data into vectors. you can also use any other open source models or api.

We highly suggest using Cohere's free APIs (Chat, text generation, embedding, etc.) because they're way better than OpenAI's free APIs. Refer to our previous article and get more insights on Cohere's APIs.

Breaking Chains: Cohere's Free LLM API's Shakes OpenAI's Foundation.

In this article, we are going to test Cohere's free LLM APIs and assess their potential significant impact on open…

gathnex.medium.com

Huggingface Space: we can build, host, and share your ML apps using Spaces. they are providing 16GB RAM, 2 vCPU cores and 50GB of disk space hardware for Free. we are going use it as deployment space.

FastAPI and Docker: FastAPI is a Python framework for building APIs and Docker is used for containerize our AI Application for deployment in huggingface space.

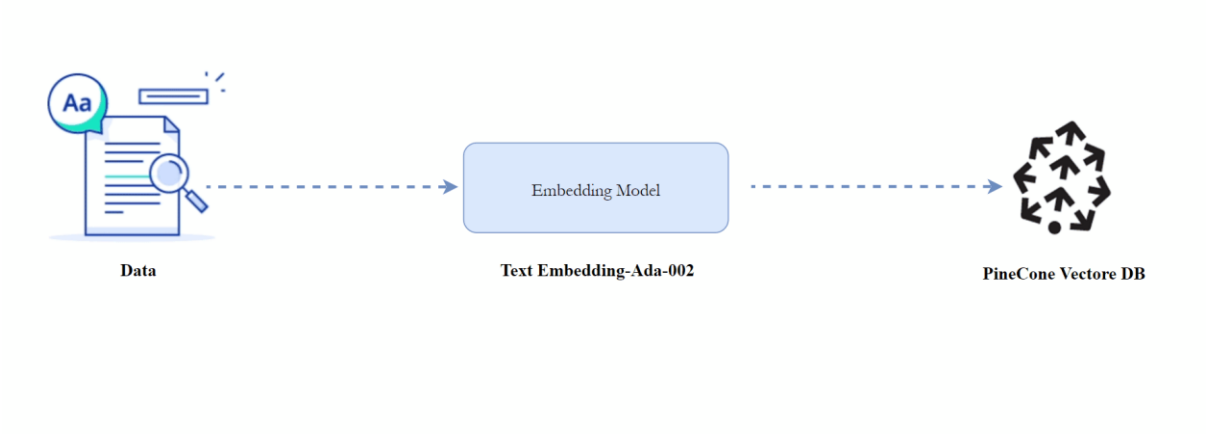
Above mentioned tech stacks are open source and closed source free tier versions.

3. Running RAG with Your Own Data (with code)

There are two phases to the development of this RAG system. The data will be transformed into embeddings and stored in a vector database in phase 1. phase 2 will be deploying the retrieval with completion setup in production.

3.1 Embed and index

- Use the OpenAI Embedding API to generate vector embeddings of your documents (or any text data).
- Upload those vector embeddings into Pinecone, which can store and index millions/billions of these vector embeddings, and search through them at ultra-low latencies.



Data Storage

Code

```
# Environment Setup
!pip install -q pinecone-client openai datasets
```

To create embeddings we must first initialize our connection to OpenAI Embeddings, we sign up for an API key at OpenAI.

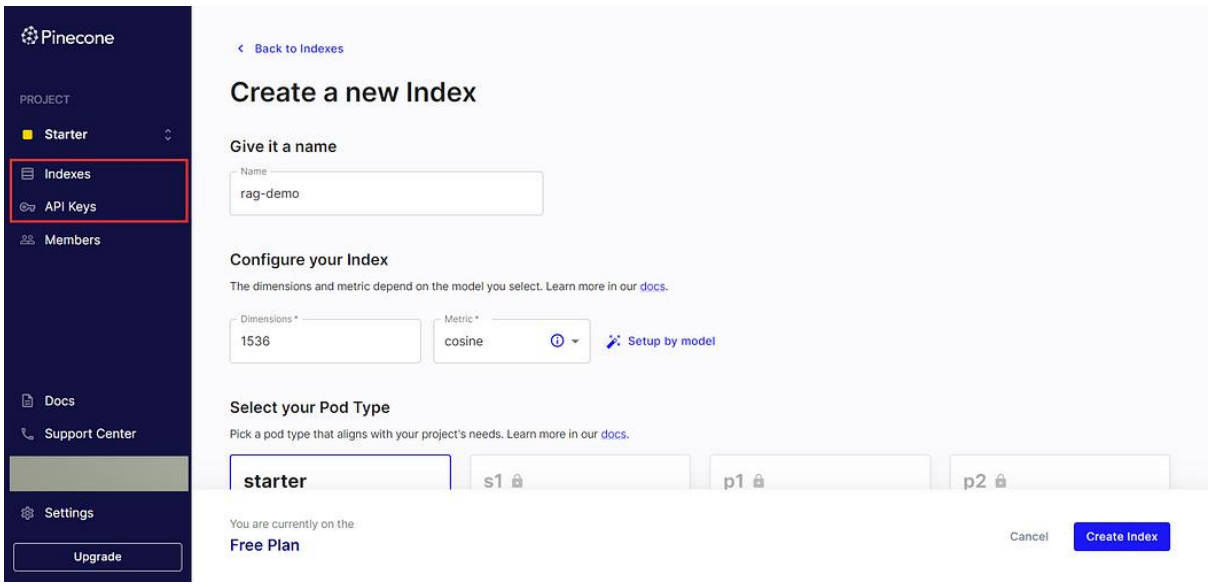
```
from openai import OpenAI
from google.colab import userdata
import os, pinecone

client = OpenAI(api_key= "Openai_api_key")
```

We can now create embeddings with the OpenAI Ada similarity model like so:

```
res = client.embeddings.create(
    input=[
        "The Gathnex AI community is a group of individuals who are interested in learning about and using AI. The community is open to anyone who wants to participate, and there are no prerequisites for membership. The community is a great place to learn about AI, share your experiences, and connect with other like-minded individuals.",
    ],
    model="text-embedding-ada-002"
)
print(res.data[0].embedding)
```

Initializing a Pinecone Index

Next, we initialize an index to store the vector embeddings. For this we need a Pinecone API key, sign up for one here.



Step 1: Create an index by entering the index name, dimension (assuming the dimension is 1536), and metric (choose a vector search method). We are going to use the starter pod because it's free. Then, hit the 'Create Index' button.

Step 2: Navigate to the API Key tab and copy the Pinecone API for connection.

```
import pinecone

index_name = 'rag-demo'

# initialize connection to pinecone (get API key at app.pinecone.io)
pinecone.init(
    api_key =  "Pinecone_api_key",
    environment = "gcp-starter"  # find next to api key in console
)
# check if 'openai' index already exists (only create index if not)
if index_name not in pinecone.list_indexes():
    pinecone.create_index(index_name, dimension=len(res.data[0].embedding))
# connect to index
index = pinecone.Index(index_name)
```

Now we'll load our sample dataset

```
from datasets import load_dataset

# load the first 150 rows of the indian law dataset
data = load_dataset('nisaar/Lawyer_GPT_India', split='train')
data
```

Then we create a vector embedding for each phrase using OpenAI, and upsert the ID, vector embedding, and original text for each phrase to Pinecone.

Note: We highly recommend using the Cohere embedding model due to some limitations in the OpenAI API during the conversion of text into vectors. Refer to our previous article about Cohere's free API access for more information.

Breaking Chains: Cohere's Free LLM API's Shakes OpenAI's Foundation.

In this article, we are going to test Cohere's free LLM APIs and assess their potential significant impact on open…

gathnex.medium.com

```
from tqdm.auto import tqdm
import time

count = 0  # we'll use the count to create unique IDs
batch_size = 1
for i in tqdm(range(0, len(data['answer']), batch_size)):
    # set end position of batch
    i_end = min(i+batch_size, len(data['answer']))
    # get batch of lines and IDs
    lines_batch = data['answer'][i: i+batch_size]
```

```
    ids_batch = [str(n) for n in range(i, i_end)]
    # create embeddings
    res = client.embeddings.create(input=lines_batch,model="text-embedding-ada-002")
    time.sleep(20)
    embeds = [res.data[0].embedding]
    # prep metadata and upsert batch
    meta = [{'text': line} for line in lines_batch]
    to_upsert = zip(ids_batch, embeds, meta)
    # upsert to Pinecone
    index.upsert(vectors=list(to_upsert))
```

After the data has been successfully uploaded, it's time for testing.

Query

With our data indexed, we're now ready to move onto performing searches. This follows a similar process to indexing. We start with a text query, that we would like to use to find similar sentences. As before we encode this with OpenAI's text similarity ada model to create a *query vector* xq. We then use xq to query the Pinecone index.

```
query = "transportation law"
xq = client.embeddings.create(input=query,model="text-embedding-ada-002")
#Retrieving relevent data from pinecone DB
res = index.query([xq.data[0].embedding], top_k=2, include_metadata=True)
res

#Output
{'matches': [{'id': '78',
              'metadata': {'text': 'The Motor Vehicles (Amendment) Act, 2019 '
                                   'has substantial implications for road '
                                   'safety. It increases penalties for traffic '
                                   'violations, introduces provisions for '
                                   'electronic enforcement, and provides for a '
                                   'Motor Vehicle Accident Fund. It also '
                                   'mandates the central government to '
                                   'regulate taxi aggregators and establishes '
                                   'a National Road Safety Board. This Act is '
                                   'expected to enhance compliance with '
                                   'traffic rules and reduce road accidents.'},
              'score': 0.805287659,
              'values': []},
             {'id': '8',
              'metadata': {'text': 'The Motor Vehicles (Amendment) Act 2019 '
                                   'introduces stricter penalties for traffic '
                                   'offenses, improving road safety. It '
                                   'establishes a Motor Vehicle Accident Fund '
                                   'and mandates insurance. However, it also '
                                   'raises concerns about state autonomy, as '
                                   'it encroaches upon their legislative '
                                   'purview.'},
```

```
        'score': 0.800310254,
        'values': []}],
 'namespace': ''}
```

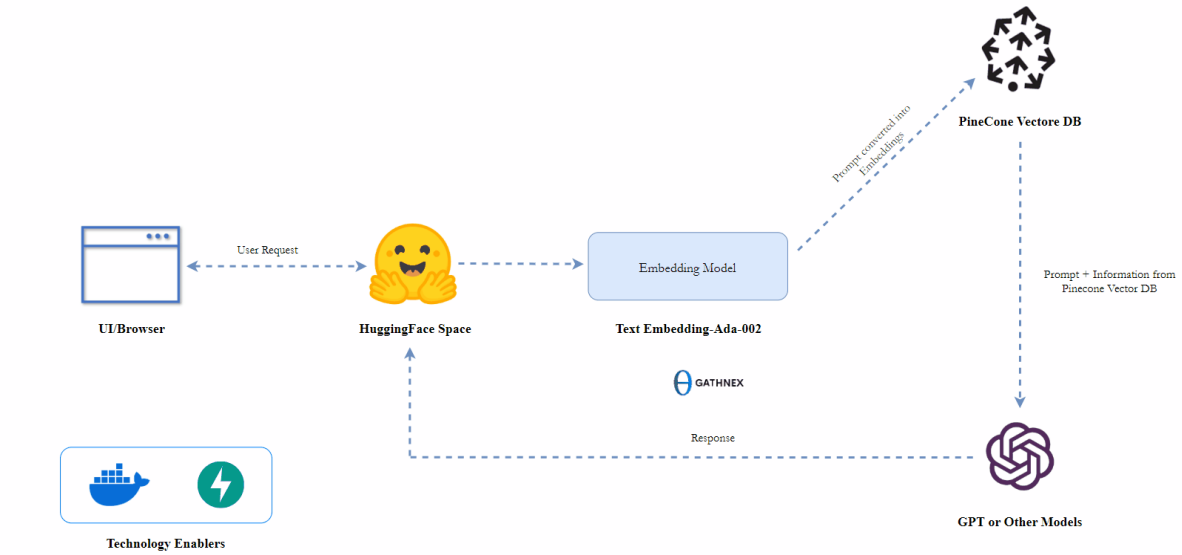
It's clear from this example that the semantic search pipeline is clearly able to identify the meaning between each of our queries. Using these embeddings with Pinecone allows us to return the most semantically similar document from the already indexed Indian law dataset.

Refer the code
: https://colab.research.google.com/drive/1U9SBwbGLtAhsTwPgWO5rTCiiJir2yYNY?usp=sharing

3.2 Vector Search with completion and deployment

Here is a detailed description of our RAG system.

1. We are going to deploy the structured files below in Hugging Face and create an API.
2. The initial user request is passed to the Hugging Face space, which contains our container.
3. After that, it's passed through the embedding model and converted into vectors.
4. Take the resulting vector embedding and send it as a query to Pinecone.
5. Get back semantically similar documents, even if they don't share any keywords with the query.
6. Then, it's passed to GPT-3.5-turbo, which efficiently completes the user query using the retrieved documents. Next, The user gets a response.



RAG Architecture

The file structure for deployment will be as follows:

Rag_Demo/
| — __init__.py
| — credentials.env
| — Dockerfile
| — main.py

| — rag_retriver.py
| — requirements.txt

Please refer to this GitHub repository for a better understanding of the structure : https://github.com/gathnexadmin/Production-Ready-RAG-System-at-Zero-Cost

requirements.txt

This file contains a list of packages or libraries needed to work on a project, all of which can be installed using the file.

```
openai
python-multipart
fastapi
pydantic
uvicorn
python-dotenv
pinecone-client
```

credentials.env

Generally, .env files are used to store sensitive variables or credentials.

```
openai_api = "openai_api_key"
Pinecone_api_key = "api key"
Pinecone_environment = "gcp-starter"
index_name = "index_name"
```

rag_retriver.py

The rag_retriever.py file contains GPT completion, vector search, and retriever functionalities from the Pinecone database.

```
import pinecone
from openai import OpenAI
from dotenv import dotenv_values

#Loading Credentials
env_name = "credentials.env"
config = dotenv_values(env_name)
client = OpenAI(api_key= config["openai_api"])


#Connection
index_name = config["index_name"]
# initialize connection to pinecone (get API key at app.pinecone.io)
pinecone.init(
    api_key =  config["Pinecone_api_key"],
    environment =  config["Pinecone_environment"]
)
index = pinecone.Index(index_name)
```

```
#Vector Search
def Vector_search(query):
  Rag_data = ""
  xq = client.embeddings.create(input=query,model="text-embedding-ada-002")
  res = index.query([xq.data[0].embedding], top_k=2, include_metadata=True)
  for match in res['matches']:
     if match['score'] < 0.80:
       continue
     Rag_data += match['metadata']['text']
  return Rag_data

#GPT Completion
def GPT_completion_with_vector_search(prompt, rag):
   DEFAULT_SYSTEM_PROMPT = '''You are a helpful, respectful and honest INTP-T AI Assistant named Gathnex AI. You are talking to a human User.
   Always answer as helpfully and logically as possible, while being safe. Your answers should not include any harmful, political, religious, unethical, racist, sexist, toxic, dangerous, or illegal content. Please ensure that your responses are socially unbiased and positive in nature.
   If a question does not make any sense, or is not factually coherent, explain why instead of answering something not correct. If you don't know the answer to a question, please don't share false information.
   You also have access to RAG vectore database access which has Indian Law data. Be careful when giving response, sometime irrelevent Rag content will be there so give response effectivly to user based on the prompt.
   You can speak fluently in English.
   Note: Sometimes the Context is not relevant to Question, so give Answer according to that sutiation.
   '''
   response = client.chat.completions.create(
   model="gpt-3.5-turbo-1106",
   messages=[
     {f"role": "system", "content": DEFAULT_SYSTEM_PROMPT},
     {f"role": "user", "content": rag +", Prompt: "+ prompt},
   ]
   )

   return response.choices[0].message.content
```

main.py

The main.py file contains a FastAPI function that returns a RAG completion.

```
from Rag_Retriver import Vector_search, GPT_completion_with_vector_search
from fastapi import FastAPI
from pydantic import BaseModel

#Pydantic object
class validation(BaseModel):
```

```
    prompt: str

#Fast API
app = FastAPI()


@app.post("/Gathnex_Rag_System")
async def retrival_augmented_generation(item: validation):
    rag = Vector_search(item.prompt)
    completion = GPT_completion_with_vector_search(item.prompt, rag)
    return completion
```

Dockerfile

Finally, the Dockerfile was used to containerize our application for deployment.

```
FROM python:3.9

WORKDIR /code

COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./__init__.py /code/__init__.py
COPY ./credentials.env /code/credentials.env
COPY ./rag_retriver.py /code/rag_retriver.py
COPY ./main.py /code/main.py

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "7860"]
```

Now, our files are ready for deployment.

Deployment

Huggingface space

Step 1: Create a new **Hugging Face space.**

Step 2: Enter the space name and choose the license type.

Step 3: Select the Space SDK-Docker (Blank) option.

Step 4: Choose ‘Public’ initially to copy our deployment endpoint, and then hit ‘Create Space.

Refer to our Hugging Face space files structure to get an idea : https://huggingface.co/spaces/gathnex/rag-demo/tree/main

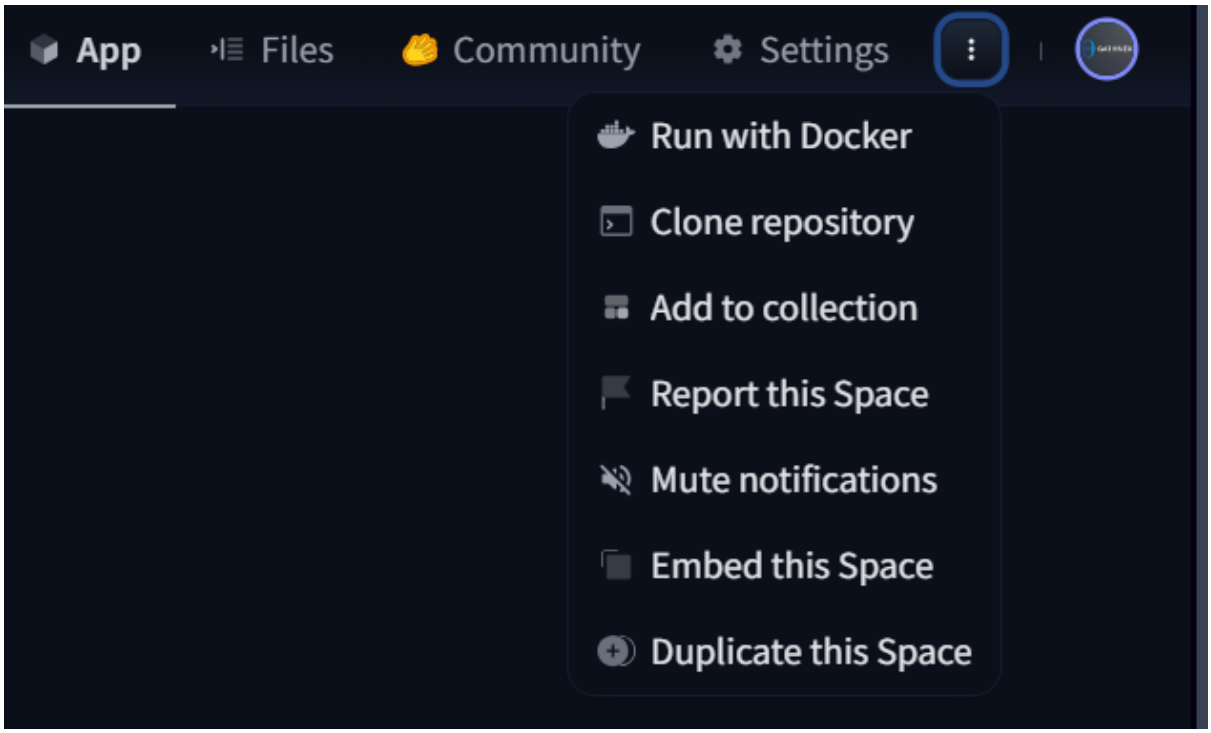
Step 5: Upload the files we have created above.

Rag_Demo/
| — __init__.py
| — credentials.env
| — Dockerfile
| — main.py
| — rag_retriver.py
| — requirements.txt

Congratulations! Your production-ready application has been deployed successfully.

If there are no errors in the code, you’ll see that it’s running, and you can check the logs if an error occurs.

Step 6: Now the space created, then click on “Embed this space” and copy the space link (This serves as your API key for production).



Copy the direct URL.

Embed this Space

Iframe

```
<iframe
    src="https://gathnex-rag-demo.hf.space"
    frameborder="0"
    width="850"
    height="450"
></iframe>
```

Copy

Direct URL

https://gathnex-rag-demo.hf.space

Copy

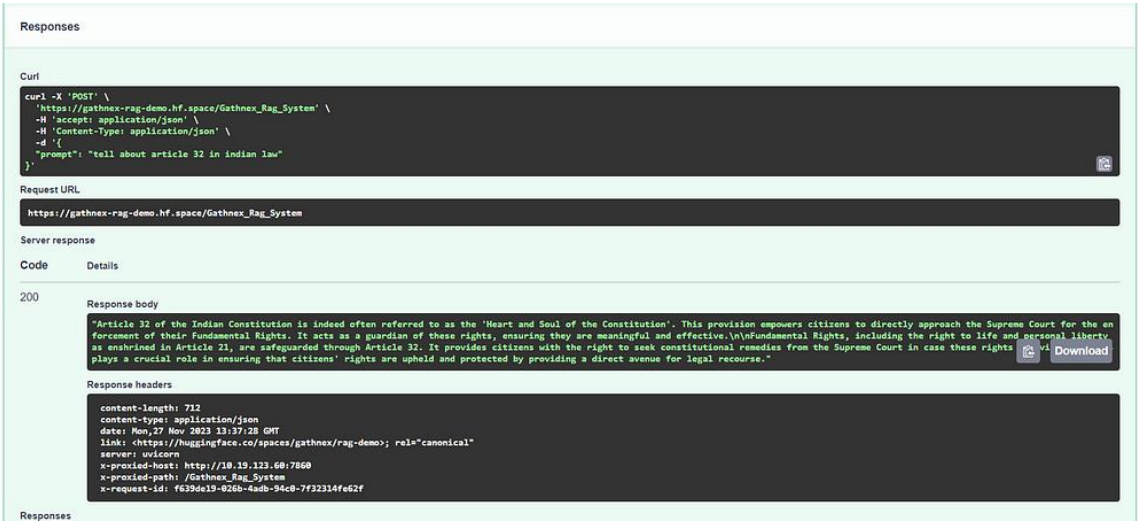
Step 7: Now make space visibility private ⚠️ in setting (Note: Your data and credentials will be exposed in a public space).

Everything is now completed, and our RAG system has been deployed successfully. Let's proceed to test our API.

This is our Space link : https://gathnex-rag-demo.hf.space

Add /docs to the URL to enable Swagger UI and test our API.

Fast API swagger : https://gathnex-rag-demo.hf.space/docs



Endpoint : https://gathnex-rag-demo.hf.space/Gathnex_Rag_System

We have implemented the RAG system with an API endpoint at the production level, utilizing open-source tools and other free-tier resources. This project can be employed for high school projects, and it is suitable for production use as well, as there is minimal latency.

Follow us on medium for more update.

Follow us on LinkedIn : https://www.linkedin.com/company/gathnex/

If you have any questions, feel free to reach out to us on LinkedIn.