wipro engineering edge

Optimizing Information Retrieval

Exploring Google Cloud's RAG API for Advanced RAG Pipelines

February 2025

Steps to implement RAG application using the RAG API with Vertex AI Vector Search

Let's use an illustrative example of a multi-cloud enterprise environment, where a substantial team of engineers required effective technical support to address diverse queries and issues. To meet this demand, a Retrieval-Augmented Generation (RAG) pipeline can be implemented, leveraging the RAG API to provide contextual technical answers by referencing a knowledge corpus encompassing historical incidents, FAQ documentation, video transcripts, and StackOverflow.

Step 1: Choose the Vector Database: The RAG Engine offers `RagManagedDb` as the default choice of a vector database, with the option to add additional Vector Databases. Experimenting with Vector AI Search, based on vector search technology developed by Google Research, was chosen for this implementation.

Step 2: Set up Vertex AI SDK: Before proceeding, install and configure the Vertex AI SDK to enable seamless integration with the RAG API and Vertex AI Vector Search:

```
pip install google-cloud-aiplatform

# Set up the SDK

aiplatform.init(project="your-project-id", location="your-region")
```

Ensure that you've set up authentication with Google Cloud and have the necessary permissions for Vertex AI and Matching Engine services.

Step 3: Create a Vector Search Index: A Vector Search index is where vector embeddings will be stored and queried. Criteria such as an empty Vector Search index, `IndexUpdateMethod`, distance measure type, and vector dimension must be met to ensure compatibility for similarity comparison.

To create a Vector Search index, following criteria must be met:

`IndexUpdateMethod` should be set to `STREAM_UPDATE` that allows to update and query the index in real time.

The distance measure type must be either `DOT_PRODUCT_DISTANCE or COSINE_DISTANCE.`

The vector dimension should match that of the embedding model used for the RAG corpus. This ensures the embeddings are compatible for similarity comparison. The code snippet is as follows:

```python
 def vector_search_create_streaming_index(
    project: str, location: str, display_name: str, gcs_uri: Optional[str] = None
) -> aiplatform.MatchingEngineIndex:
    """Create a vector search index.
    Args:
        project (str): Required. Project ID
        location (str): Required. The region name
        display_name (str): Required. The index display name
        gcs_uri (str): Optional. The Google Cloud Storage uri for index content
    Returns:
        The created MatchingEngineIndex.
    """
    # Initialize the Vertex AI client
    aiplatform.init(project=project, location=location)
    # Create Index
    index = aiplatform.MatchingEngineIndex.create_tree_ah_index(
        display_name=display_name,
        contents_delta_uri=gcs_uri,
        description="Matching Engine Index",
        dimensions=100,
        approximate_neighbors_count=150,
        leaf_node_embedding_count=500,
        leaf_nodes_to_search_percent=7,
        index_update_method="STREAM_UPDATE",  # Options: STREAM_UPDATE, BATCH_UPDATE

distance_measure_type=aiplatform.matching_engine.matching_engine_index_config.DistanceMeasureType.DOT_PRODUCT_DISTANCE,
    )
    return index
```

Note: RAG API uses `text-embedding-004` model as a default embedding model.

Step 4: Create a Vector Search Index Endpoint: The endpoint provides a public-facing access point to your vector search index, allowing for easy retrieval of relevant contexts based on user queries. The code snippet is as follows:

```python
def vector_search_create_index_endpoint(
    project: str, location: str, display_name: str
) -> None:
    """"Create a vector search index endpoint.

    Args:
        project (str): Required. Project ID
        location (str): Required. The region name
        display_name (str): Required. The index endpoint display name
    """
    # Initialize the Vertex AI client
    aiplatform.init(project=project, location=location)

    # Create Index Endpoint
    index_endpoint = aiplatform.MatchingEngineIndexEndpoint.create(
        display_name=display_name,
        public_endpoint_enabled=True,
        description="Matching Engine Index Endpoint",
    ) I don't think we want, as a company/team, to start listing individual team leaders on the
website beyond the Board and Executive Leadership. We'll open up the floodgates to every group
wanting photos and titles (WEF, Consulting, Cloud.....) and we'll constantly be changing them,
updating photos, etc etc

    print(index_endpoint.name)
```

Note: Use a descriptive display_name to help identify your endpoint in the Vertex AI console. Once the index is created, deploy it to an endpoint for public-facing access.

Step 5: Deploy the Index to the Endpoint: To enable nearest neighbour search, the index must be deployed to the endpoint for public-facing access, enabling efficient retrieval of relevant contexts.

```python
def vector_search_deploy_index(
    project: str,
    location: str,
    index_name: str,
    index_endpoint_name: str,
    deployed_index_id: str,
) -> None:
    """"Deploy a vector search index to a vector search index endpoint.

    Args:
        project (str): Required. Project ID
        location (str): Required. The region name
        index_name (str): Required. The index to update. A fully-qualified index
          resource name or a index ID.  Example:
          "projects/123/locations/us-central1/indexes/my_index_id" or
          "my_index_id".
        index_endpoint_name (str): Required. Index endpoint to deploy the index
          to.
        deployed_index_id (str): Required. The user specified ID of the
          DeployedIndex.
    """
    # Initialize the Vertex AI client
    aiplatform.init(project=project, location=location)

    # Create the index instance from an existing index
    index = aiplatform.MatchingEngineIndex(index_name=index_name)

    # Create the index endpoint instance from an existing endpoint.
    index_endpoint = aiplatform.MatchingEngineIndexEndpoint(
        index_endpoint_name=index_endpoint_name
    )
```

```
    # Deploy Index to Endpoint
index_endpoint = index_endpoint.deploy_index(
    index=index, deployed_index_id=deployed_index_id
)

    print(index_endpoint.deployed_indexes)
```

Note: The first deployment may take up to 30 minutes.

Step 6: Create the RAG Corpus: Create the RAG corpus and associate it with the Vector Search instance as the Vector database, ensuring the necessary validations are performed to ensure correct setup.

Specify INDEX_ENDPOINT_NAME and INDEX_NAME while creating the corpus.

```python
from vertexai.preview import rag
import vertexai

# TODO(developer): Update and un-comment below lines
# PROJECT_ID = "your-project-id"
# vector_search_index_name =
"projects/{PROJECT_ID}/locations/{LOCATION}/indexes/{INDEX_ID}"
# vector_search_index_endpoint_name =
"projects/{PROJECT_ID}/locations/{LOCATION}/indexEndpoints/{INDEX_ENDPOINT_ID}"
# display_name = "test_corpus"
# description = "Corpus Description"

# Initialize Vertex AI API once per session
vertexai.init(project=PROJECT_ID, location="us-central1")

# Configure embedding model (Optional)
embedding_model_config = rag.EmbeddingModelConfig(
    publisher_model="publishers/google/models/text-embedding-004"
)

# Configure Vector DB
vector_db = rag.VertexVectorSearch(
    index=vector_search_index_name, index_endpoint=vector_search_index_endpoint_name
)

corpus = rag.create_corpus(
    display_name=display_name,
    description=description,
    embedding_model_config=embedding_model_config,
    vector_db=vector_db,
)
print(corpus)
# Example response:
# RagCorpus(name='projects/1234567890/locations/us-central1/ragCorpora/1234567890',
# display_name='test_corpus', description='Corpus Description',
embedding_model_config=...
# ...
```

Step 7: Import Files into the RAG Corpus: Using the `ImportRagFiles` API, import documents from Cloud Storage or Google Drive into the Vector Search index that will form the corpus, allowing for easy retrieval and embedding of documents. The code snippet is as follows:

```
from vertexai.preview import rag
import vertexai

# TODO(developer): Update and un-comment below lines
# PROJECT_ID = "your-project-id"
# corpus_name = "projects/{PROJECT_ID}/locations/us-
central1/ragCorpora/{rag_corpus_id}"
# paths = ["https://drive.google.com/file/123", "gs://my_bucket/my_files_dir"]  #
Supports Google Cloud Storage and Google Drive Links

# Initialize Vertex AI API once per session
vertexai.init(project=PROJECT_ID, location="us-central1")

response = rag.import_files(
    corpus_name=corpus_name,
    paths=paths,
    chunk_size=512,  # Optional
    chunk_overlap=100,  # Optional
    max_embedding_requests_per_min=900,  # Optional
)
print(f"Imported {response.imported_rag_files_count} files.")
# Example response:
# Imported 2 files.
```

Step 8: Retrieve Relevant Contexts using RAG API: After importing the documents, relevant contexts can be retrieved using the `RetrieveContexts` API, allowing the RAG application to access semantic information stored in the index for answering specific queries. The code snippet is as follows:

```
from vertexai.preview import rag
import vertexai

# TODO(developer): Update and un-comment below lines
# PROJECT_ID = "your-project-id"
# corpus_name = "projects/[PROJECT_ID]/locations/us-
central1/ragCorpora/[rag_corpus_id]"

# Initialize Vertex AI API once per session
vertexai.init(project=PROJECT_ID, location="us-central1")

response = rag.retrieval_query(
    rag_resources=[
        rag.RagResource(
            rag_corpus=corpus_name,
            # Optional: supply IDs from `rag.list_files()`.
            # rag_file_ids=["rag-file-1", "rag-file-2", ...],
        )
    ],
    text="Hello World!",
    similarity_top_k=10,  # Optional
    vector_distance_threshold=0.5,  # Optional
)
print(response)
# Example response:
# contexts {
#   contexts {
#     source_uri: "gs://your-bucket-name/file.txt"
#     text: "....
#   ....
```

Step 9: Generate Content Using Vertex AI Gemini models: Make a call to the Vertex AI `GenerateContent` API, specifying the `RAG_CORPUS_RESOURCE` in the request to generate content using Generative AI models like Gemini.

```python
From vertexai.preview import rag
from vertexai.preview.generative_models import GenerativeModel, Tool
import vertexai

# TODO(developer): Update and un-comment below lines
# PROJECT_ID = "your-project-id"
# corpus_name = "projects/{PROJECT_ID}/locations/us-
central1/ragCorpora/{rag_corpus_id}"

# Initialize Vertex AI API once per session
vertexai.init(project=PROJECT_ID, location="us-central1")

rag_retrieval_tool = Tool.from_retrieval(
    retrieval=rag.Retrieval(
        source=rag.VertexRagStore(
            rag_resources=[
                rag.RagResource(
                    rag_corpus=corpus_name,
                    # Optional: supply IDs from `rag.list_files()`.
                    # rag_file_ids=["rag-file-1", "rag-file-2", …],
                )
            ],
            similarity_top_k=3,  # Optional
            vector_distance_threshold=0.5,  # Optional
        ),
    )
)

rag_model = GenerativeModel(
    model_name="gemini-1.5-flash-001", tools=[rag_retrieval_tool]
)
response = rag_model.generate_content("Why is the sky blue?")
print(response.text)
# Example response:
#   The sky appears blue due to a phenomenon called Rayleigh scattering.
#   Sunlight, which contains all colors of the rainbow, is scattered
#   by the tiny particles in the Earth's atmosphere….
#   …
```

The process of implementing Retrieval-Augmented Generation (RAG) pipelines with Google Cloud's RAG API demonstrates the utilization of advanced AI capabilities. By integrating the Vertex AI SDK and establishing a Streaming Vector Search index, organizations can effectively engage with the RAG Engine to store and retrieve vector embeddings. Deploying the index to a public-facing endpoint enables seamless access to the vector search index, facilitating the retrieval of relevant contexts. Additionally, the RAG API's capability to deliver detailed and contextually relevant responses to user queries, along with its robust management and embedding of content for retrieval, showcases its advanced functionality and potential to enhance information retrieval and content generation processes through generative models like Gemini.

Sample query and response from RAG API

Question: Why do I get a "curl: (6) Could not resolve host" or "ERR_NAME_NOT_RESOLVED" error when I try to access my website on Lightsail?

Model Response:

"""The errors "curl: (6) Could not resolve host" or "ERR_NAME_NOT_RESOLVED" mean your browser can't find your website's IP address. Here's how to troubleshoot:

1. **Check Domain Registration and Nameservers:**

    -   Use `whois your_domain` to check if your domain is registered.

    -   Use `dig your_domain NS` or `whois your_domain | grep "Name Server"` to verify the nameservers are correct. Update them with your registrar if needed.

2. **Check if the Issue Is Local or Global:**

    -   Try accessing your website from another network or device. If it works, the problem is with your local network or machine.

    -   **Verify resolver configuration:** Check your local DNS resolver settings. On Windows, use `ipconfig /all`. On macOS or Linux, check `/etc/resolv.conf`. Ensure the DNS servers listed are correct and responsive.

3. **Check Hosts File:**

    -   Incorrect entries in your hosts file (Windows: `C:\Windows\System32\drivers\etc\hosts`, macOS/Linux: `/etc/hosts`) can cause this error. Verify the entry for your domain has the correct IP address, or remove/comment it out if incorrect or unnecessary.

4. **Update Nameserver or Register Domain:**

    -   If the domain is unregistered, register it and enter the nameservers provided by your hosting provider or DNS service.

    -   If the nameservers are incorrect, update them with your domain registrar.

5. **Configure A Record:**

    -   If the `dig +trace` command reaches your nameserver but doesn't return an A record (IP address), create or update the A record in your DNS settings (e.g. Lightsail DNS zone) to point to your instance's public IP address.

6. **Check Firewall Rules:**

    -   Ensure your firewall allows inbound traffic on ports 80 (HTTP) and 443 (HTTPS). Check both Lightsail firewall rules and operating system-level firewalls (like UFW or iptables).

7. **Check Instance Health:**

    -   If you get "connection timed out" errors, check your Lightsail instance's health and performance metrics to rule out server-side issues.

Advancing RAG Performance: A Comparative Evaluation of RAG API and Traditional RAG Approach

The evaluation of the RAG pipeline implementation using the RAG API and the traditional RAG approach revealed that the RAG API provided better Retriever and Generation metric scores, showcasing its robustness and improved performance. The Retriever functionality is made robust in RAG API that brings the google-quality search through Vertex AI Vector Search that delivers more relevant results from the content provided. This is made available in a configurable way wherein; we can operationalize the repeatability of RAG implementation for successive use cases.

RAG API	Traditional RAG

<table>
<tr>
<td>

```
Output:
RAGResults
(
  4 RAG results,
  Metrics:
  {
    "overall_metrics": {
      "precision": 62.0,
      "recall": 47.8,
      "f1": 51.7
    },
    "retriever_metrics": {
      "claim_recall": 73.4,
      "context_precision": 90.0
    },
    "generator_metrics": {
      "context_utilization": 61.7,
      "noise_sensitivity_in_relevant":
24.0,
      "noise_sensitivity_in_irrelevant":
0.0,
      "hallucination": 14.1,
      "self_knowledge": 6.2,
      "faithfulness": 79.7
    }
  }
)
```

</td>
<td>

```
Output:
RAGResults
(
  4 RAG results,
  Metrics:
  {
    "overall_metrics": {
      "precision": 67.1,
      "recall": 43.5,
      "f1": 52.2
    },
    "retriever_metrics": {
      "claim_recall": 26.2,
      "context_precision": 60.0
    },
    "generator_metrics": {
      "context_utilization": 93.9,
      "noise_sensitivity_in_relevant":
5.6,
      "noise_sensitivity_in_irrelevant":
6.2,
      "hallucination": 21.1,
      "self_knowledge": 36.9,
      "faithfulness": 42.0
    }
  }
)
```

</td>
</tr>
</table>