Building RAG solutions

In this module, you learn to ...

01 Create solution architectures for different customer problems referencing the RAG pattern

02 Use and choose the right embedding technology for creation, storage and serving

03 Automate and improve workflows and RAG solutions

Topics

01	Introducing the case study
02	Ingestion
03	Serving
04	Workflow and improvements

Customer problem

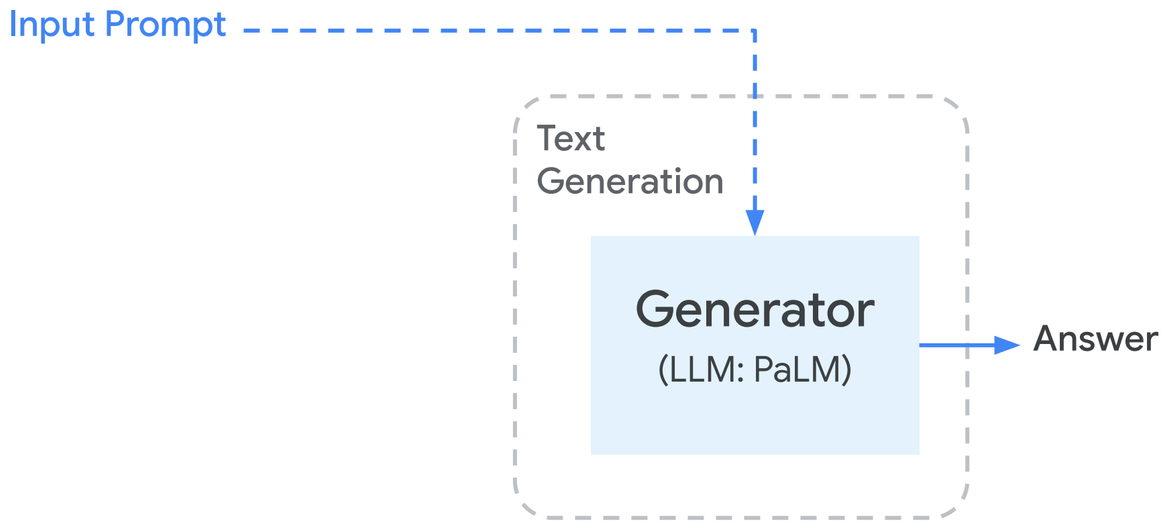
Provide semantic search to find answers from a proprietary dataset and obtain responses with summarizations grounded on the data

- Search answers should not paraphrase the stored data; they should answer the question
- Search answers should have a pointer to the document where the answer was found to be able to verify that answers are grounded on customer data
- Answers should not be made up if the data is not found in the search results
- Dataset is proprietary to the customer and it needs to stay private
- Dataset is constantly growing (it’s not stale)

What is Retrieval Augmented Generation (RAG)?

The problem:

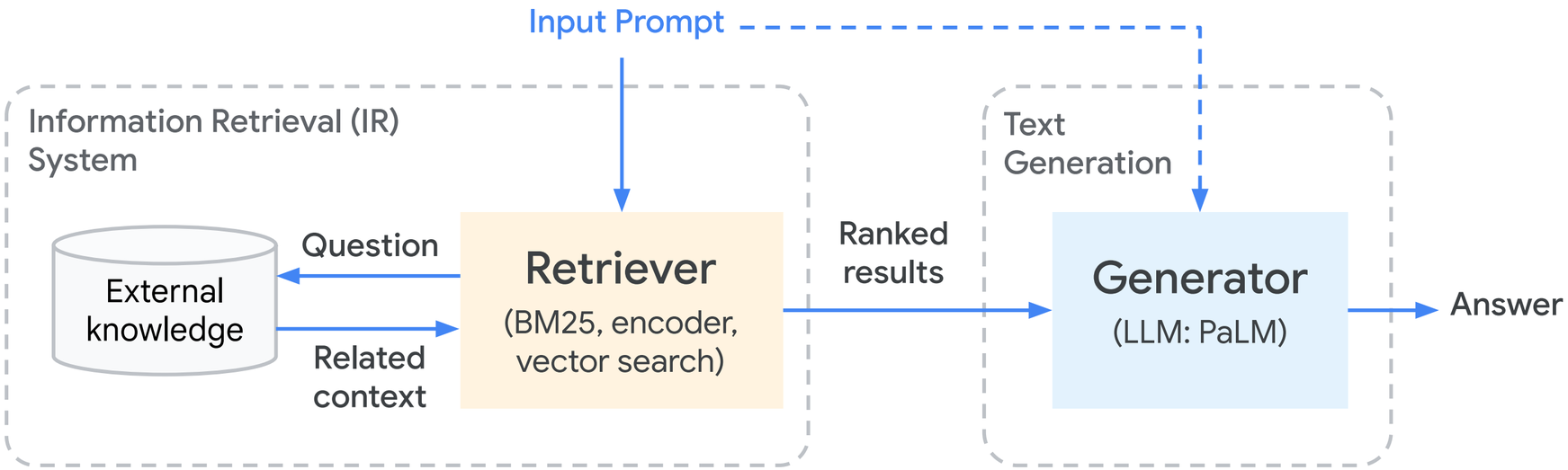
- LLMs don’t know your business proprietary or domain specific data
- LLMs don’t have real-time information
- LLMs find it hard to provide accurate citation



What is Retrieval Augmented Generation (RAG)?

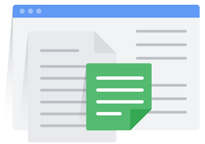
The solution:

- Feed the LLM relevant context in real-time, by using an information retrieval system



RAG architecture example powered by embeddings

1. Index the relevant content



Document corpus
or
Knowledge Base

Vertex AI
Text Embeddings API

Long documents may need to be split into smaller chunks

Embeddings and Content Index

Vertex AI
Vector Search

Your favourite low latency **key-value store**

2. Fetch relevant info and augment prompt

Vertex AI
Text Embeddings API

User prompt

"... lorem ipsum lorem ipsum..."

Fetch top-k semantic matches and append to prompt

Augmented prompt

"... lorem ipsum lorem ipsum...

Context:
...doc 1...
...doc 2...
...
...doc k... "

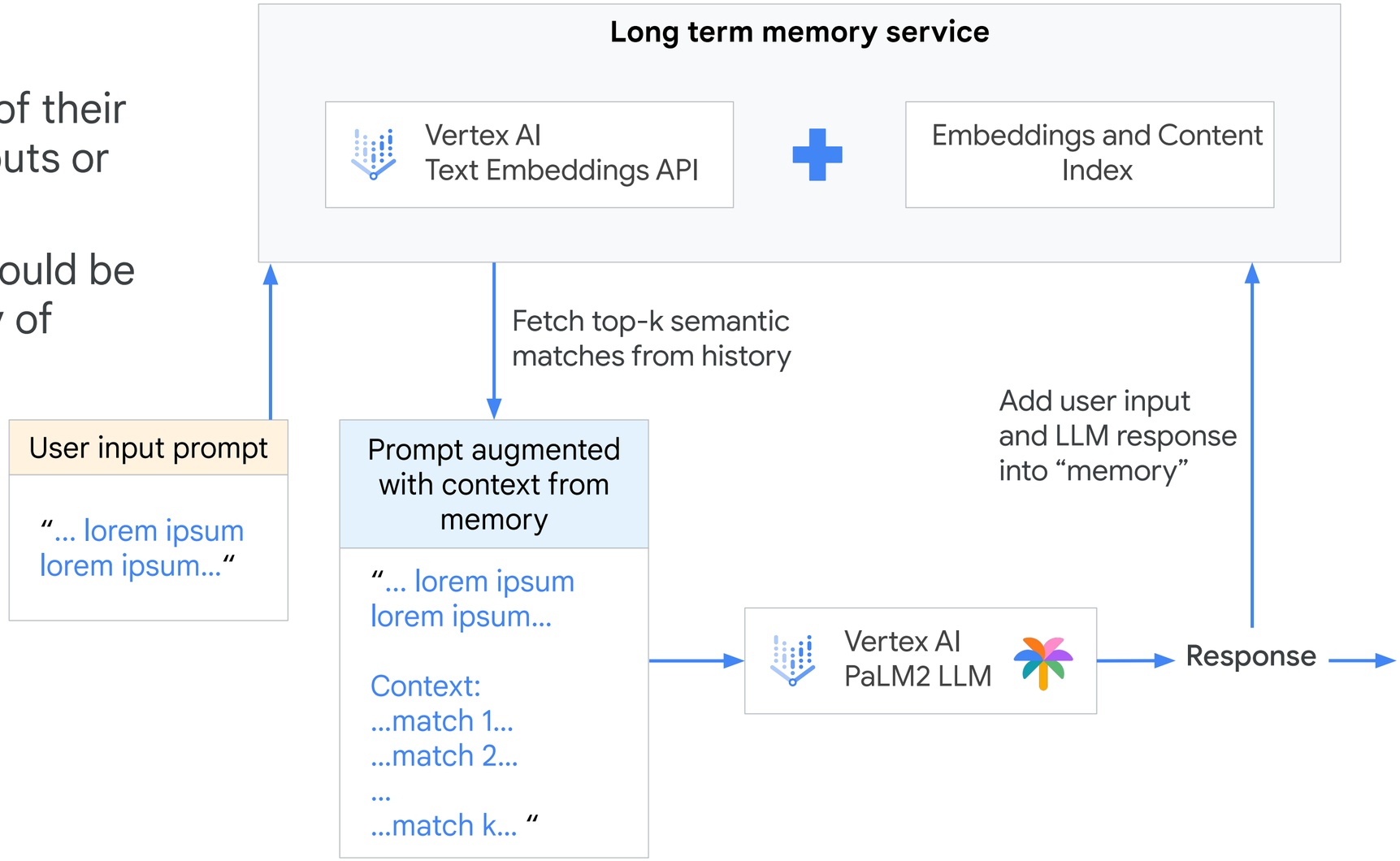
Vertex AI
PaLM2 LLM

Response

RAG architecture to provide memory to LLMs

The problem:

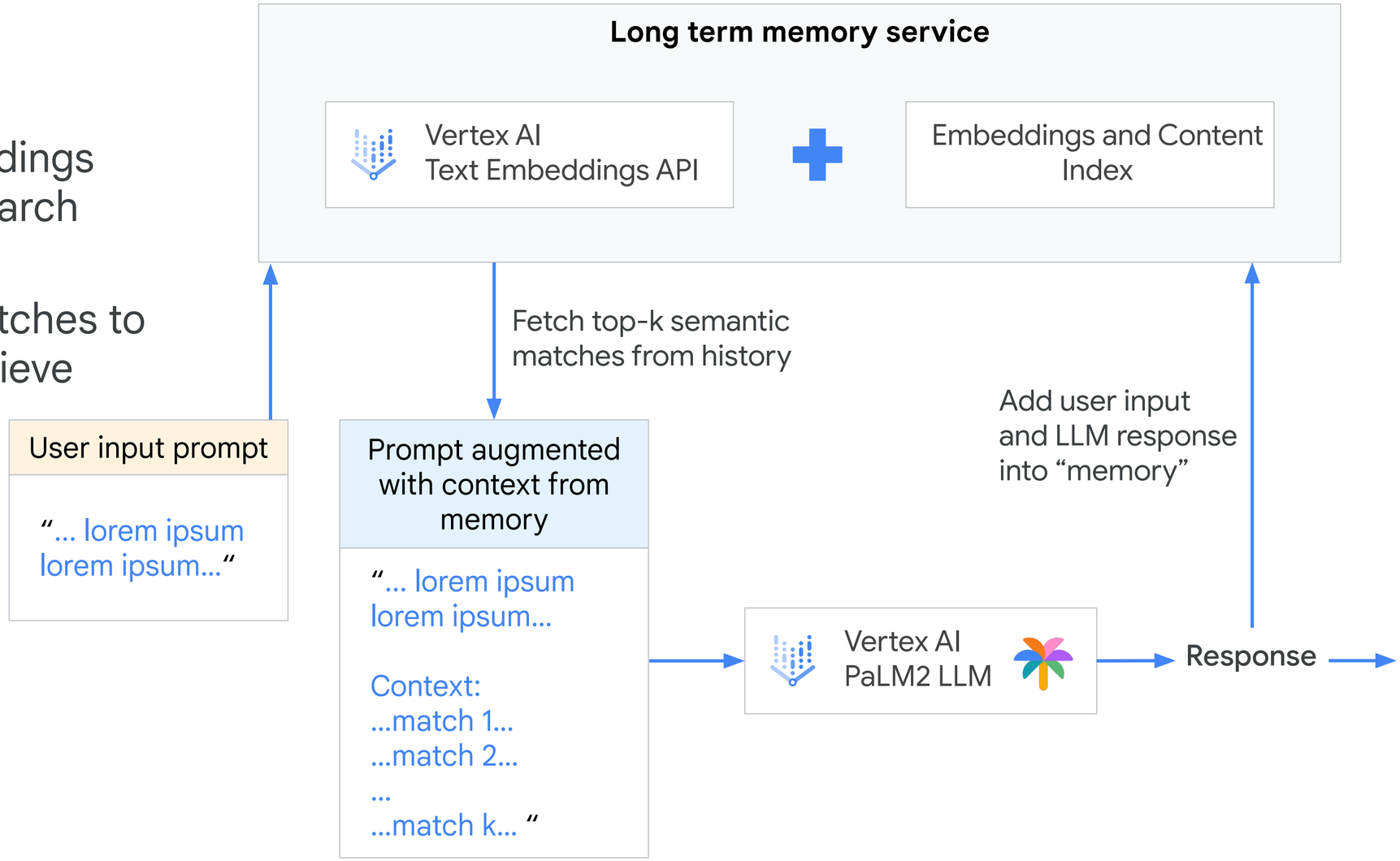
- LLMs have no memory of their own and forget past inputs or chats
- For chat assistants, it would be great to have a memory of previous conversations



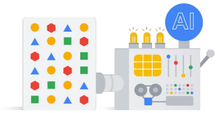
RAG architecture to provide memory to LLMs

The solution:

- Encode turns of the conversation as embeddings and store in a vector search solution
- Fetch most relevant matches to latest user input, to retrieve historical context in real-time



Benefits of RAG vs Fine Tuning



RAG

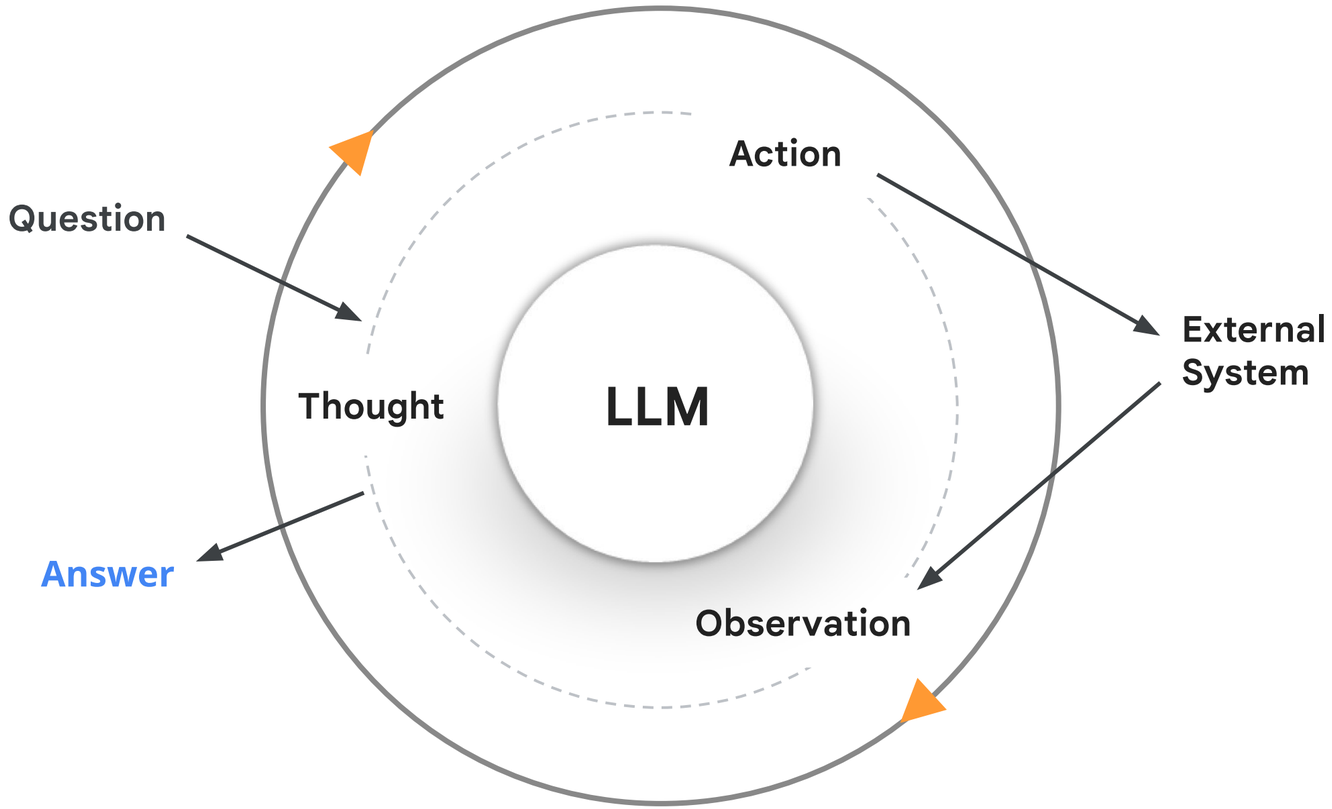
- RAG is usually cheaper
- You can use different versions of the LLM with the same knowledge base, without the need to re-train
- You can keep ingesting documents on-demand
- You can ground the answer in a specific known document source

Fine Tuning

- Fine tuning an LLM and keeping it up-to-date is more expensive than creating embeddings in RAG solutions
- Inferencing in bigger fine-tuned LLMs can be more costly (sometimes) than an LLM with the context provided by the embedding

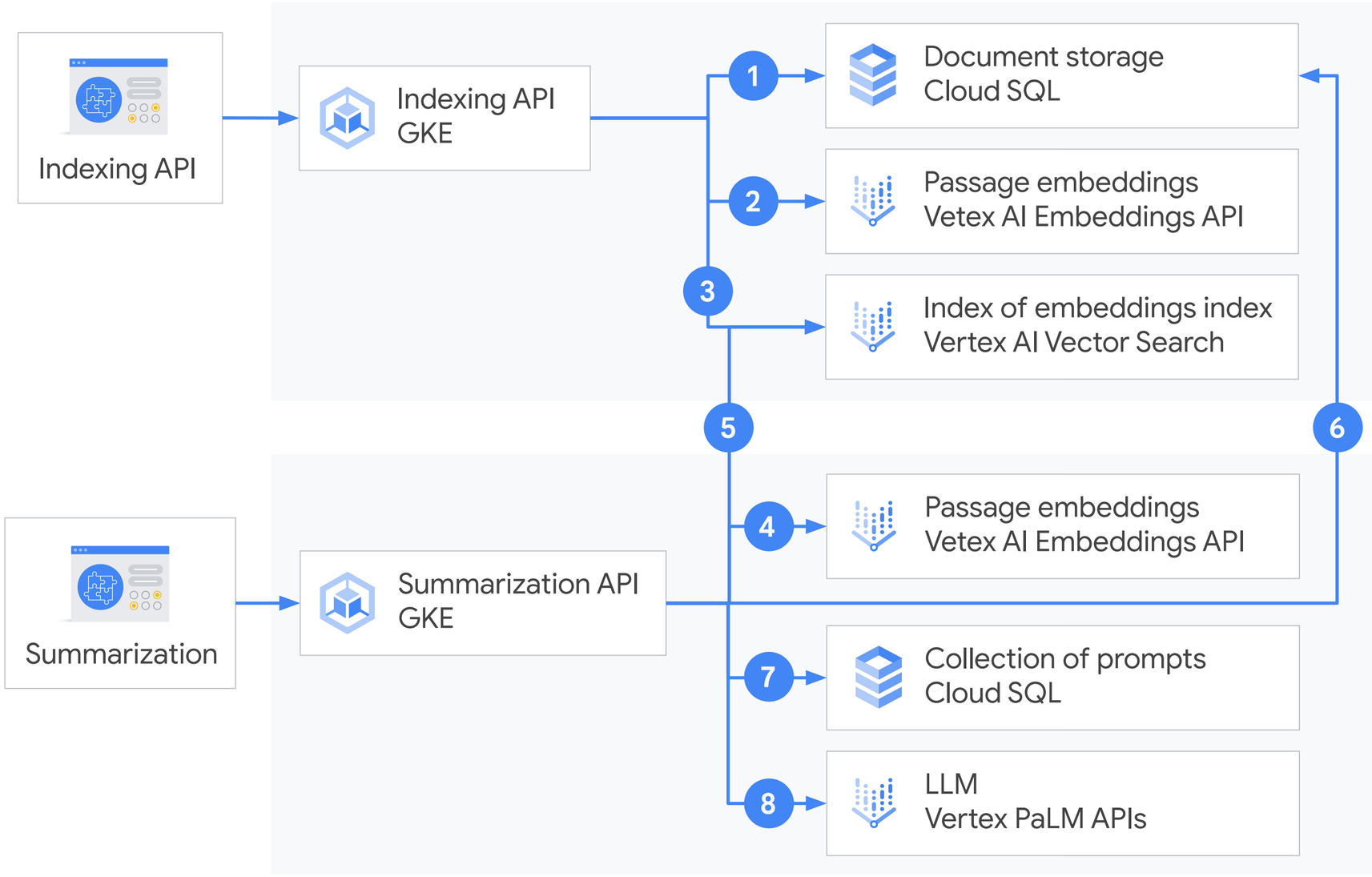
RAG + ReAct

- LLM to query an external system via an API to augment the data
- The LLM acts an an agent, that has thoughts and performs actions

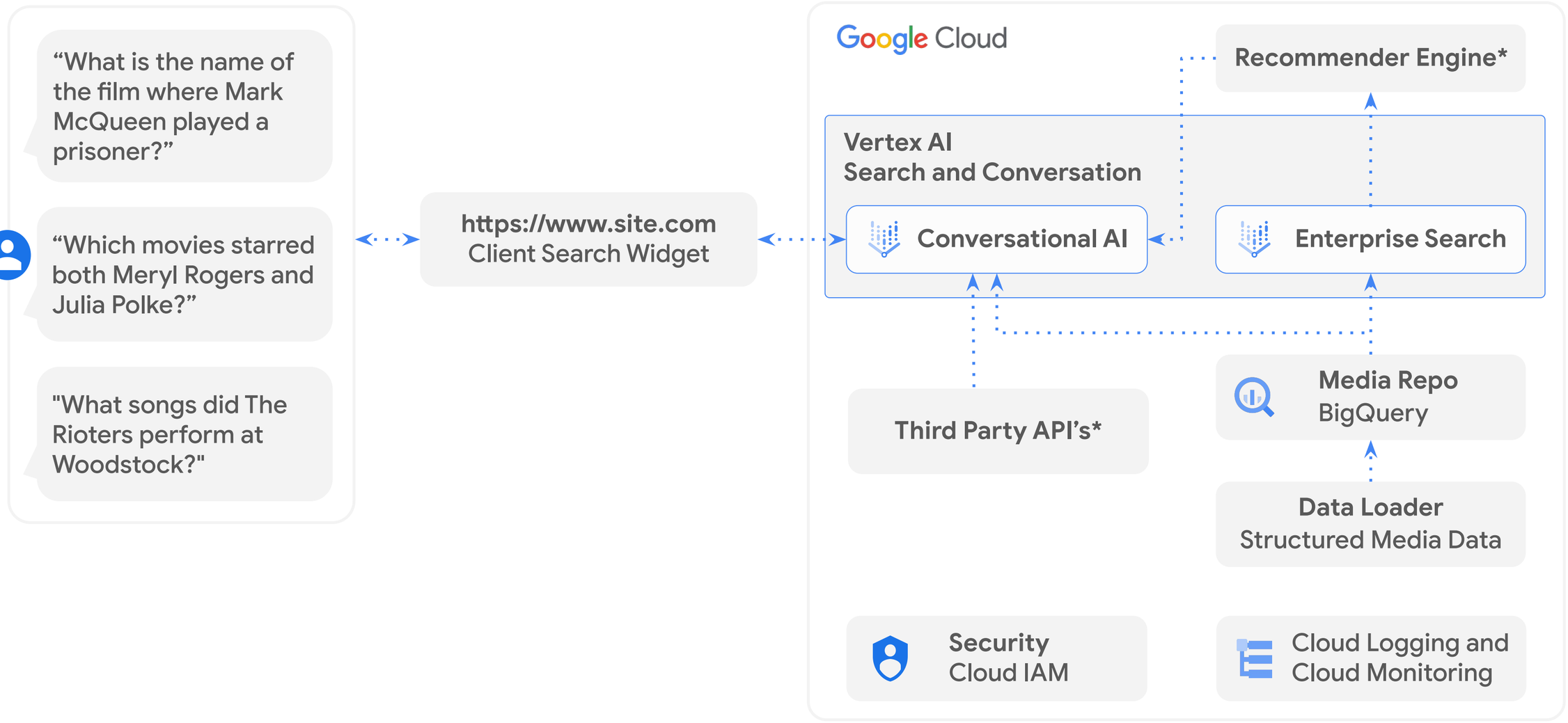


Architectural diagram

1. Split the document into passages, and save both raw document and passages.
2. Encode passages into embeddings.
3. Load them into Vertex AI Vector Search.
4. Encode the question.
5. Get top-k most closest passages from the Vertex AI Vector Search.
6. Retrieve passages from the Document Storage.
7. Get the prompt and enrich it with passing passages.
8. Generate the final answer.



Alternative architecture with Vertex AI Search



Discussion out-of-the-box (OoTB) vs do-it-yourself (DIY)

Out-of-the-box Vertex AI Search and Conversation	**Do-it-yourself** Embeddings + Vector Search + Document AI
Implementation in minutes	Implementation in hours or days
Only batch data refresh available	Batch and streaming data refresh available
Supports: • BigQuery tables, HTML, PDF with embedded text, TXT format • Preview: PPTX and DOCX Does not support: Images, videos, audio	Can be used with any data format Parsing documents with Document AI before ingesting them as embeddings provides better outcomes than OOTB
Prompt templates in Preview	Can create a prompt template and send it to the LLM

Topics

01	Introducing the case study
02	Ingestion
03	Serving
04	Workflow and improvements

RAG considerations for ingesting data

The embeddings APIs have limits in terms of:

- The amount of documents ingested simultaneously
- The size of the documents

To ingest bigger documents chunk with the right approach for you:

- Fixed-size chunking: i.e. 500 tokens
- Context-aware chunking: sections or related paragraphs
- Hierarchical chunking: i.e. Markdown, LaTex, or objects like tables

Other considerations:

- Chunk overlap
- Retrieve additional text to the embedding retrieved
  (ingestion size does not need to be the same as retrieved size)
- Add chunk metadata such as the title, document, page, etc

Option 1: Manually generating batch embeddings

- `textembedding-gecko@001` is limited to 5 input texts of 3,072 tokens each
- Write a function that generates batches of 5 limiting to 10 API calls per second

```
def encode_text_to_embedding_batched(
    sentences: List[str], api_calls_per_second: int = 10, batch_size: int = 5
) -> Tuple[List[bool], np.ndarray]:

    embeddings_list: List[List[float]] = []

    # Prepare the batches using a generator
    batches = generate_batches(sentences, batch_size)

    seconds_per_job = 1 / api_calls_per_second
```

Option 1: Manually generating batch embeddings

```
with ThreadPoolExecutor() as executor:
    futures = []
    for batch in tqdm(
        batches, total=math.ceil(len(sentences) / batch_size), position=0
    ):
        futures.append(
            executor.submit(functools.partial(encode_texts_to_embeddings), batch)
        )
        time.sleep(seconds_per_job)

    for future in futures:
        embeddings_list.extend(future.result())

is_successful = [
    embedding is not None for sentence, embedding in zip(sentences, embeddings_list)
]
embeddings_list_successful = np.squeeze(
    np.stack([embedding for embedding in embeddings_list if embedding is not None])
)
return is_successful, embeddings_list_successful
```

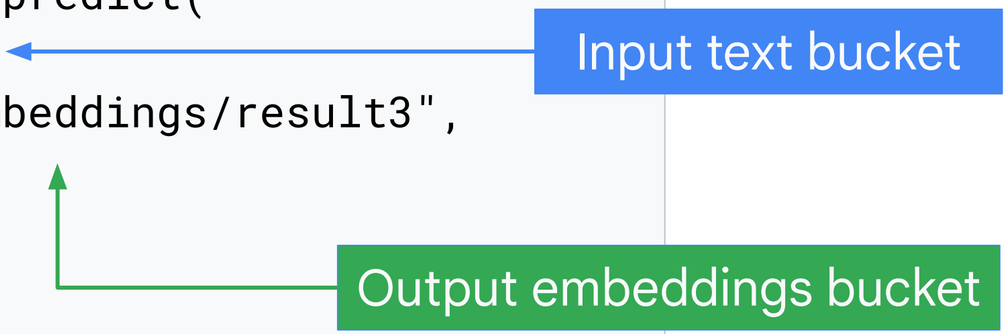
Compute batches of 5 for processing

Make sure to catch errors in prod

Option 2: API for generating batch embeddings [Preview]

- Source in BigQuery table or as a JSON Lines (JSONL) file in Cloud Storage.
- Each request can include up to 30,000 prompts
- Output in a Cloud Storage bucket

```
from vertexai.preview.language_models import TextEmbeddingModel
textembedding_model =
TextEmbeddingModel.from_pretrained("textembedding-gecko")
batch_prediction_job = textembedding_model.batch_predict(
  dataset=["gs://BUCKET_NAME/test_table.jsonl"],
  destination_uri_prefix="gs://BUCKET_NAME/tmp/embeddings/result3",
)
print(batch_prediction_job.display_name)
print(batch_prediction_job.resource_name)
print(batch_prediction_job.state)
```



Option 2: API for generating batch embeddings [Preview]

JSONL input example

```
{“content”:”Give a short description of a machine learning model:”}
{“content”:”Best recipe for banana bread:”}
```

JSONL output example

```
{“instance”:{”content”:”Give...”},”predictions”:[{“embeddings”:{statistics”:{“token_cou
{“instance”:{”content”:”Best...”},”predictions”:[{“embeddings”:{statistics”:{“token_cou
```

Option 2: API for generating batch embeddings [Preview]

BigQuery input example

This example shows a single column BigQuery table.

content
"Give a short description of a machine learning model:"
"Best recipe for banana bread:"

BigQuery output example

content	predictions
"Give a short description of a machine learning model:"	`'[{"embeddings": { "statistics":{"token_count":8,"truncated":false}, "Values":[0.1,....] } } ]'`
"Best recipe for banana bread:"	`'[{"embeddings": { "statistics":{"token_count":3,"truncated":false}, "Values":[0.2,....] } } ]'`

Choosing an embeddings storage solution

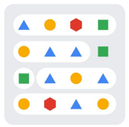
Vertex AI Vector Search

- Use in addition to a database
- Better for bigger datasets
- Result filtering with tags
- Cheaper
- Supported index distance functions:
  - Dot product distance (default)
  - Euclidean L2 distance
  - Cosine distance
  - Manhattan L1 distance

AlloyDB for PostgreSQL

- Collocate data and embeddings
- Smaller datasets already in PostgreSQL
- Generate embeddings with textembedding-gecko from SQL
- Hybrid search with Postgres full-text search
- More expensive
- Supported index distance functions:
  - Dot product (or inner product)
  - Euclidean L2 distance
  - Cosine distance

The best index solution might vary on the dataset



Index solution options

- Compare the recall of indexes to the Dot product distance
- Dot product distance is computed using brute force; it can perform well for small datasets
- Cosine distance usually gives the same results as the Dot product distance. It is recommended to use Dot product with L2 normalization instead for an equivalent.
- Manhattan L1 distance is usually the fastest
- Euclidean L2 distance is usually the most accurate

Embeddings workflow with AlloyDB

- Create a column to store vector embeddings in AlloyDB

```
ALTER TABLE items ADD COLUMN qa_embedding vector(768);
```

- Populate the new column

```
UPDATE items SET qa_embedding = embedding('textembedding-gecko@001', complaints);
```

- Index the new column

```
CREATE INDEX qa_embd_idx ON items USING ivf (qa_embedding vector_l2_ops) WITH
(lists = 20, quantizer = 'SQ8');
```

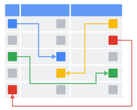
- Query the new column

```
SELECT id, name FROM items ORDER BY complaint_embedding <->
embedding('textembedding-gecko@001',
'How big is the Google Pixel 8?') LIMIT 10;
```

Topics

01	Introducing the case study
02	Ingestion
03	Serving
04	Workflow and improvements

Create and deploy a Vector Search index endpoint



Serving requests and types of endpoints

- Serving requests
  - Enable min and max amount of instances for the endpoint to serve traffic
  - Enable autoscaling on the nodes serving the endpoint
- Types of endpoints:
  - Public: protected by IAM based oauth - use Service Accounts
  - Private Service Access (via VPC Peering) - only within the same or peered VPC (no internet access)
  - Private Service Connect: allow different VPC to access the endpoint.

Filter search results based on attributes: data ingestion

- When ingesting embeddings into Vector Search, add attributes to make filtering possible.
- Specify the namespaces and tokens associated with each input vector.
- Attributes are organized in categories or namespaces

```
"id": "42", "embedding": [0.5, 1.0],
    "restricts": [
        {"namespace": "shape", "allow": ["circle", "circumference"]},
        {"namespace": "color", "allow": ["blue"]}
    ]}

{"id": "43", "embedding": [0.6, 1.0],
     "restricts": [
         {"namespace": "shape", "allow": ["square"]},
         {"namespace": "color", "allow": ["red"]}
]}
```

Attribute or namespace

Token or value

Filter search results based on attributes

- You can filter results by selecting attributes or tokens.
- From the filtered values, Nearest Neighbor Search (NNS) is performed.

- Examples:
  - To filter for “red circles” use: `{color:red},{shape:circle}`
  - To filter for “red and blue square use: `{color:red,blue},{shape:square}`
  - To specify an object with no color, omit the "color" namespace in the restricts field.
  - To filter out “blue” shapes use: `{color:red,!blue}`

- Ideas:
  - Use filters to classify documents: public, private, confidential...
  - Use filters to differentiate domains of knowledge in an organization: marketing, engineering, sales...

Topics

01	Introducing the case study
02	Ingestion
03	Serving
04	Workflow and improvements

Augment responses with LLM integrations

- Get responses from Vector Search

```
NUM_NEIGHBOURS = 10
response = my_index_endpoint.find_neighbors(
    deployed_index_id=DEPLOYED_INDEX_ID,
    queries=test_embeddings,
    num_neighbors=NUM_NEIGHBOURS,
)
```

- Get the text from the database

```
full_answer = queryDB(f"SELECT text FROM table_of_contents WHERE id =
{response[0]}")
```

- Use the LLM to summarize the response for the user:

```
summary = llm(f"Summarize the following text:/n {full_answer}")
```

Optimize search by caching responses with MemoryStore