In [1]:
```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Getting Started with LangChain 🦜🔗 + Vertex AI Gemini API

 Run in Colab  Run in Colab Enterprise  View on GitHub  Open in Vertex AI Workbench

Author(s) Rajesh Thallam, Holt Skinner

What is LangChain?

> LangChain is a framework for developing applications powered by large language models (LLMs).

TL;DR LangChain makes the complicated parts of working & building with language models easier. It helps do this in two ways:

1. **Integration** - Bring external data, such as your files, other applications, and API data, to LLMs
2. **Agents** - Allows LLMs to interact with its environment via decision making and use LLMs to help decide which action to take next

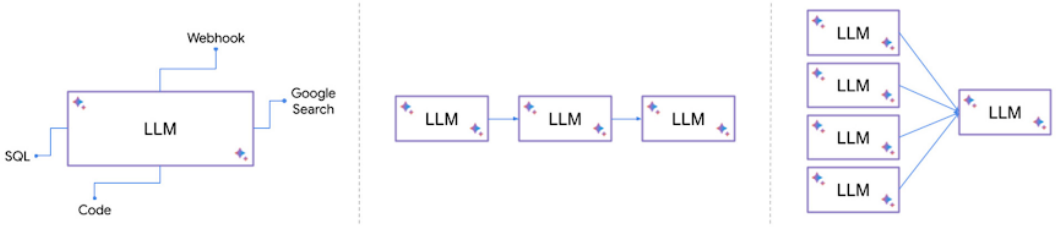
To build effective Generative AI applications, it is key to enable LLMs to interact with external systems. This makes models data-aware and agentic, meaning they can understand, reason, and use data to take action in a meaningful way. The external systems could be public data corpus, private knowledge repositories, databases, applications, APIs, or access to the public internet via Google Search.

Here are a few patterns where LLMs can be augmented with other systems:

- Convert natural language to SQL, executing the SQL on database, analyze and present the results
- Calling an external webhook or API based on the user query

- Synthesize outputs from multiple models, or chain the models in a specific order

It may look trivial to plumb these calls together and orchestrate them but it becomes a mundane task to write glue code again and again e.g. for every different data connector or a new model. That's where LangChain comes in!



Why LangChain?

LangChain's modular implementation of components and common patterns combining these components makes it easier to build complex applications based on LLMs. LangChain enables these models to connect to data sources and systems as agents to take action.

1. **Components** are abstractions that works to bring external data, such as your documents, databases, applications,APIs to language models. LangChain makes it easy to swap out abstractions and components necessary to work with LLMs.

2. **Agents** enable language models to communicate with its environment, where the model then decides the next action to take. LangChain provides out of the box support for using and customizing 'chains' - a series of actions strung together.

Though LLMs can be straightforward (text-in, text-out) you'll quickly run into friction points that LangChain helps with once you develop more complicated applications.

LangChain & Vertex AI

Vertex AI Generative AI models — Gemini and Embeddings — are officially integrated with the LangChain Python SDK, making it convenient to build applications using Gemini models with the ease of use and flexibility of LangChain.

- LangChain Google Integrations

Note: This notebook does not cover all aspects of LangChain. Its contents have been curated to get you to building & impact as quick as possible. For more, please check out LangChain Conceptual Documentation

Objectives

This notebook provides an introductory understanding of LangChain components and use cases of LangChain with the Vertex AI Gemini API.

- Introduce LangChain components
- Showcase LangChain + Vertex AI Gemini API - Text, Chat and Embedding
- Summarizing a large text
- Question/Answering from PDF (retrieval based)
- Chain LLMs with Google Search

References:

- Adapted from LangChain Cookbook from Greg Kamradt
- LangChain Conceptual Documentation
- LangChain Python Documentation

Costs

This tutorial uses billable components of Google Cloud:

- Vertex AI

Learn about Vertex AI pricing, and use the Pricing Calculator to generate a cost estimate based on your projected usage.

In [3]:
```
# Install Vertex AI SDK, LangChain and dependencies
%pip install --upgrade --quiet google-cloud-aiplatform langchain langchain-core
```

```
Note: you may need to restart the kernel to use updated packages.
```

Colab only: Run the following cell to restart the kernel or use the button to restart the kernel. For Vertex AI Workbench you can restart the terminal using the button on top.

In [1]:
```
# Automatically restart kernel after installs so that your environment can acces
import IPython

app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[1]:
```
{'status': 'ok', 'restart': True}
```

Authenticating your notebook environment

- If you are using **Colab** to run this notebook, run the cell below and continue.
- If you are using **Vertex AI Workbench**, check out the setup instructions here.

In [1]:
```
import sys

if "google.colab" in sys.modules:
    from google.colab import auth

    auth.authenticate_user()
```

- If you are running this notebook in a local development environment:
  - Install the Google Cloud SDK.

- Obtain authentication credentials. Create local credentials by running the following command and following the oauth2 flow (read more about the command here):

  ```
  gcloud auth application-default login
  ```

Import libraries

Colab only: Run the following cell to initialize the Vertex AI SDK. For Vertex AI Workbench, you don't need to run this.

In [2]:
```
import vertexai

PROJECT_ID = "qwiklabs-gcp-02-abf532d75b70"  # @param {type:"string"}
REGION = "us-central1"  # @param {type:"string"}

# Initialize Vertex AI SDK
vertexai.init(project=PROJECT_ID, location=REGION)
```

In [3]:
```
from langchain.chains import (
    ConversationChain,
    LLMChain,
    RetrievalQA,
    SimpleSequentialChain,
)
from langchain.chains.summarize import load_summarize_chain
from langchain.memory import ConversationBufferMemory
from langchain.output_parsers import ResponseSchema, StructuredOutputParser
from langchain_chroma import Chroma
from langchain_community.document_loaders import PyPDFLoader, WebBaseLoader
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document
from langchain_core.example_selectors import SemanticSimilarityExampleSelector
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.prompts import PromptTemplate
from langchain_core.prompts.few_shot import FewShotPromptTemplate
from langchain_google_vertexai import ChatVertexAI, VertexAI, VertexAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

```
USER_AGENT environment variable not set, consider setting it to identify your req
uests.
```

Define LangChain Models using the Vertex AI Gemini API for Text, Chat and Vertex AI Embeddings for Text

In [4]:
```
# LLM model
llm = VertexAI(
    model_name="gemini-1.5-flash",
    verbose=True,
)

# Chat
chat = ChatVertexAI(model="gemini-1.5-pro")
```

```
# Embedding
embeddings = VertexAIEmbeddings("text-embedding-004")
```

LangChain Components

Let's take a quick tour of LangChain framework and concepts to be aware of. LangChain offers a variety of modules that can be used to create language model applications. These modules can be combined to create more complex applications, or can be used individually for simpler applications.

- **Models** are the building block of LangChain providing an interface to different types of AI models. Large Language Models (LLMs), Chat and Text Embeddings models are supported model types.
- **Prompts** refers to the input to the model, which is typically constructed from multiple components. LangChain provides interfaces to construct and work with prompts easily - Prompt Templates, Example Selectors and Output Parsers.
- **Memory** provides a construct for storing and retrieving messages during a conversation which can be either short term or long term.
- **Indexes** help LLMs interact with documents by providing a way to structure them. LangChain provides Document Loaders to load documents, Text Splitters to split documents into smaller chunks, Vector Stores to store documents as embeddings, and Retrievers to fetch relevant documents.
- **Chains** let you combine modular components (or other chains) in a specific order to complete a task.
- **Agents** are a powerful construct in LangChain allowing LLMs to communicate with external systems via Tools and observe and decide on the best course of action to complete a given task.

Schema - Nuts and Bolts of working with LLMs

Text

Text is the natural language way to interact with LLMs.

In [5]:
```
# You'll be working with simple strings (that'll soon grow in complexity!)
my_text = "What day comes after Friday?"

llm.invoke(my_text)
```

Out[5]: 'Saturday \n'

Chat Messages

Chat is like text, but specified with a message type (System, Human, AI)

- **System** - Helpful context that tells the AI what to do
- **Human** - Messages intended to represent the user
- **AI** - Messages showing what the AI responded with

For more information, see LangChain Documentation for Chat Models.

In [6]:
```
chat.invoke([HumanMessage(content="Hello")])
```

Out[6]:
```
AIMessage(content='Hello! 👋  \n\nHow can I help you today? 😊 \n', additional
_kwargs={}, response_metadata={'is_blocked': False, 'safety_ratings': [{'catego
ry': 'HARM_CATEGORY_HATE_SPEECH', 'probability_label': 'NEGLIGIBLE', 'blocked':
False, 'severity': 'HARM_SEVERITY_NEGLIGIBLE'}, {'category': 'HARM_CATEGORY_DAN
GEROUS_CONTENT', 'probability_label': 'NEGLIGIBLE', 'blocked': False, 'severit
y': 'HARM_SEVERITY_NEGLIGIBLE'}, {'category': 'HARM_CATEGORY_HARASSMENT', 'prob
ability_label': 'NEGLIGIBLE', 'blocked': False, 'severity': 'HARM_SEVERITY_NEGL
IGIBLE'}, {'category': 'HARM_CATEGORY_SEXUALLY_EXPLICIT', 'probability_label':
'NEGLIGIBLE', 'blocked': False, 'severity': 'HARM_SEVERITY_NEGLIGIBLE'}], 'usag
e_metadata': {'prompt_token_count': 1, 'candidates_token_count': 15, 'total_tok
en_count': 16, 'cached_content_token_count': 0}, 'finish_reason': 'STOP', 'avg_
logprobs': -0.26178555488586425, 'logprobs_result': {'top_candidates': [], 'cho
sen_candidates': []}}, id='run-be4e2d4c-b83c-411a-8194-a300f2586393-0', usage_m
etadata={'input_tokens': 1, 'output_tokens': 15, 'total_tokens': 16})
```

In [7]:
```
res = chat.invoke(
    [
        SystemMessage(
            content="You are a nice AI bot that helps a user figure out what to
        ),
        HumanMessage(content="I like tomatoes, what should I eat?"),
    ]
)

print(res.content)
```

```
A delicious Caprese salad with fresh mozzarella and basil would be perfect!
```

You can also pass more chat history w/ responses from the AI

In [8]:
```
res = chat.invoke(
    [
        HumanMessage(
            content="What are the ingredients required for making a tomato sandw
        )
    ]
)
print(res.content)
```

```
The beauty of a tomato sandwich is in its simplicity! Here's what you need:

**Essentials:**

* **Bread:** White bread is classic, but feel free to use sourdough, wheat, or yo
ur favorite kind.
* **Tomatoes:** Ripe, juicy tomatoes are key.  Heirloom varieties offer fantastic
flavor.
* **Mayonnaise:** Duke's and Hellmann's are popular choices, but use what you lik
e best.
* **Salt & Pepper:** To taste

**Optional but Delicious Additions:**

* **Bacon:** Crispy bacon adds a smoky, salty crunch.
* **Avocado:** For creaminess and healthy fats.
* **Lettuce:** A leaf or two for freshness.
* **Onion:** Thinly sliced red onion for a sharp bite.
* **Cheese:** A mild cheddar or provolone can be nice.

**Pro Tip:**  Salt your tomato slices and let them sit on a paper towel for a few
minutes before assembling the sandwich. This draws out excess moisture and intens
ifies their flavor!
```

Documents

Document in LangChain refers to an unstructured text consisting of `page_content` referring to the content of the data and `metadata` (data describing attributes of page content).

In [9]:
```
Document(
    page_content="This is my document. It is full of text that I've gathered fro
    metadata={
        "my_document_id": 234234,
        "my_document_source": "The LangChain Papers",
        "my_document_create_time": 1680013019,
    },
)
```

Out[9]:
```
Document(metadata={'my_document_id': 234234, 'my_document_source': 'The LangCha
in Papers', 'my_document_create_time': 1680013019}, page_content="This is my do
cument. It is full of text that I've gathered from other places")
```

Text Embedding Model

Embeddings are a way of representing data–almost any kind of data, like text, images, videos, users, music, whatever–as points in space where the locations of those points in space are semantically meaningful. Embeddings transform your text into a vector (a series of numbers that hold the semantic 'meaning' of your text). Vectors are often used when comparing two pieces of text together. An embedding is a relatively low-dimensional space into which you can translate high-dimensional vectors.

LangChain Text Embedding Model is integrated with Vertex AI Embedding API for Text.

BTW: Semantic means 'relating to meaning in language or logic.'

In [10]:
```
text = "Hi! It's time for the beach"
```

In [11]:
```
text_embedding = embeddings.embed_query(text)
print(f"Your embedding is length {len(text_embedding)}")
print(f"Here's a sample: {text_embedding[:5]}...")
```

```
Your embedding is length 768
Here's a sample: [-0.02856108359992504, -0.007161829620599747, 0.0015754484338685
87, -0.010381614789366722, 0.0047200522385537624]...
```

Prompts

Prompts are text used as instructions to your model. For more details have a look at the notebook Intro to Prompt Design.

In [12]:
```
prompt = """
Today is Monday, tomorrow is Wednesday.

What is wrong with that statement?
"""

llm.invoke(prompt)
```

Out[12]:
```
'The statement is wrong because **there is no day called "Wednesday" that comes
directly after "Monday."** \n\nThe correct sequence is:\n\n* Monday\n* Tuesday
\n* Wednesday \n'
```

Prompt Template

Prompt Template is an object that helps to create prompts based on a combination of user input, other non-static information and a fixed template string.

Think of it as an `f-string` in Python but for prompts

In [13]:
```
# Notice "location" below, that is a placeholder for another value later
template = """
I really want to travel to {location}. What should I do there?

Respond in one short sentence
"""

prompt = PromptTemplate(
    input_variables=["location"],
    template=template,
)

final_prompt = prompt.format(location="Rome")

output = llm.invoke(final_prompt)

print(f"Final Prompt: {final_prompt}")
print("-----------")
print(f"LLM Output: {output}")
```

```
Final Prompt:
I really want to travel to Rome. What should I do there?

Respond in one short sentence

-----------
LLM Output: Explore ancient history, savor delicious food, and experience vibrant
culture.
```

Example Selectors

Example selectors are an easy way to select from a series of examples to dynamically place in-context information into your prompt. Often used when the task is nuanced or has a large list of examples.

Check out different types of example selectors here

In [14]:
```python
example_prompt = PromptTemplate(
    input_variables=["input", "output"],
    template="Example Input: {input}\nExample Output: {output}",
)

# Examples of locations that nouns are found
examples = [
    {"input": "pirate", "output": "ship"},
    {"input": "pilot", "output": "plane"},
    {"input": "driver", "output": "car"},
    {"input": "tree", "output": "ground"},
    {"input": "bird", "output": "nest"},
]
```

In [15]:
```python
# SemanticSimilarityExampleSelector will select examples that are similar to you

example_selector = SemanticSimilarityExampleSelector.from_examples(
    # This is the list of examples available to select from.
    examples,
    # This is the embedding class used to produce embeddings which are used to m
    embeddings,
    # This is the VectorStore class that is used to store the embeddings and do
    FAISS,
    # This is the number of examples to produce.
    k=2,
)
```

In [16]:
```python
similar_prompt = FewShotPromptTemplate(
    # The object that will help select examples
    example_selector=example_selector,
    # Your prompt
    example_prompt=example_prompt,
    # Customizations that will be added to the top and bottom of your prompt
    prefix="Give the location an item is usually found in",
    suffix="Input: {noun}\nOutput:",
    # What inputs your prompt will receive
    input_variables=["noun"],
)
```

In [17]:
```
# Select a noun!
my_noun = "student"

print(similar_prompt.format(noun=my_noun))
```

```
Give the location an item is usually found in

Example Input: driver
Example Output: car

Example Input: pilot
Example Output: plane

Input: student
Output:
```

In [18]:
```
llm.invoke(similar_prompt.format(noun=my_noun))
```

Out[18]:
```
'school \n'
```

Output Parsers

Output Parsers help to format the output of a model. Usually used for structured output.

Two main ideas:

1. Format Instructions: An autogenerated prompt that tells the LLM how to format it's response based off desired result

2. Parser: A method to extract model's text output into a desired structure (usually json)

In [19]:
```
# How you would like your response structured. This is basically a fancy prompt
response_schemas = [
    ResponseSchema(
        name="bad_string", description="This a poorly formatted user input strin
    ),
    ResponseSchema(
        name="good_string", description="This is your response, a reformatted re
    ),
]

# How you would like to parse your output
output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
```

In [20]:
```
# See the prompt template you created for formatting
format_instructions = output_parser.get_format_instructions()
print(format_instructions)
```

```
The output should be a markdown code snippet formatted in the following schema, i
ncluding the leading and trailing "```json" and "```":

```json
{
        "bad_string": string  // This a poorly formatted user input string
        "good_string": string  // This is your response, a reformatted response
}
```
```

In [21]:
```
template = """
You will be given a poorly formatted string from a user.
Reformat it and make sure all the words are spelled correctly including country,

{format_instructions}

% USER INPUT:
{user_input}

YOUR RESPONSE:
"""

prompt = PromptTemplate(
    input_variables=["user_input"],
    partial_variables={"format_instructions": format_instructions},
    template=template,
)

prompt_value = prompt.format(user_input="welcom to dbln!")

print(prompt_value)
```

```
You will be given a poorly formatted string from a user.
Reformat it and make sure all the words are spelled correctly including country,
city and state names

The output should be a markdown code snippet formatted in the following schema, i
ncluding the leading and trailing "```json" and "```":

```json
{
        "bad_string": string  // This a poorly formatted user input string
        "good_string": string  // This is your response, a reformatted response
}
```

% USER INPUT:
welcom to dbln!

YOUR RESPONSE:
```

In [22]:
```
llm_output = llm.invoke(prompt_value)
llm_output
```

Out[22]:
```
'```json\n{\n\t"bad_string": "welcom to dbln!",\n\t"good_string": "Welcome to D
ublin!"\n}\n```'
```

In [23]:
```
output_parser.parse(llm_output)
```

Out[23]:
```
{'bad_string': 'welcom to dbln!', 'good_string': 'Welcome to Dublin!'}
```

Indexes

Indexes refer to ways to structure documents for LLMs to work with them.

Document Loaders

Document loaders are ways to import data from other sources. See the growing list of document loaders here. There are more on LlamaIndex as well that work with LangChain Document Loaders.

In [24]:
```
loader = WebBaseLoader("http://www.paulgraham.com/worked.html")
```

In [25]:
```
data = loader.load()
```

In [26]:
```
print(f"Found {len(data)} comments")
print(f"Here's a sample:\n\n{''.join([x.page_content[:150] for x in data[:2]])}"
```

```
Found 1 comments
Here's a sample:

What I Worked On

February 2021Before college the two main things I worked on, outside of school,
were writing and programming. I didn't write essays.
```

Text Splitters

Text Splitters are a way to deal with input token limits of LLMs by splitting text into chunks.

There are many ways you could split your text into chunks, experiment with different ones to see which is best for your use case.

In [27]:
```
loader = WebBaseLoader("http://www.paulgraham.com/worked.html")
pg_work = loader.load()
```

In [28]:
```
text_splitter = RecursiveCharacterTextSplitter(
    # Set a really small chunk size, just to show.
    chunk_size=1000,
    chunk_overlap=20,
)

texts = text_splitter.split_documents(pg_work)
```

In [29]:
```
print(f"You have {len(texts)} documents")
```

```
You have 79 documents
```

In [30]:
```
print("Preview:")
print(texts[0].page_content, "\n")
print(texts[1].page_content)
```

```
Preview:
What I Worked On

February 2021Before college the two main things I worked on, outside of school,
were writing and programming. I didn't write essays. I wrote what
beginning writers were supposed to write then, and probably still
are: short stories. My stories were awful. They had hardly any plot,
just characters with strong feelings, which I imagined made them
deep.The first programs I tried writing were on the IBM 1401 that our
school district used for what was then called "data processing."
This was in 9th grade, so I was 13 or 14. The school district's
1401 happened to be in the basement of our junior high school, and
my friend Rich Draves and I got permission to use it. It was like
a mini Bond villain's lair down there, with all these alien-looking
machines — CPU, disk drives, printer, card reader — sitting up
on a raised floor under bright fluorescent lights.The language we used was an ear
ly version of Fortran. You had to
type programs on punch cards, then stack them in the card reader
```

Retrievers

Retrievers are a way of storing data such that it can be queried by a language model. Easy way to combine documents with language models.

There are many different types of retrievers, the most widely supported is the `VectorStoreRetriever`.

In [31]:
```
loader = WebBaseLoader("http://www.paulgraham.com/worked.html")
documents = loader.load()
```

Here we use Facebook AI Similarity Search (FAISS), a library and a vector database for similarity search and clustering of dense vectors. To generate dense vectors, a.k.a. embeddings, we use LangChain text embeddings model with Vertex AI Embeddings for Text .

In [32]:
```
# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=50

# Split your docs into texts
texts = text_splitter.split_documents(documents)

# Embed your texts
db = FAISS.from_documents(texts, embeddings)
```

In [33]:
```
# Init your retriever. Asking for just 1 document back
retriever = db.as_retriever()
retriever
```

Out[33]:
```
VectorStoreRetriever(tags=['FAISS', 'VertexAIEmbeddings'], vectorstore=<langcha
in_community.vectorstores.faiss.FAISS object at 0x7f72d8b8ecb0>, search_kwargs=
{})
```

In [34]:
```
docs = retriever.get_relevant_documents(
    "what types of things did the author want to develop or build?"
)
```

```
print("\n\n".join([x.page_content[:200] for x in docs[:2]]))
```

```
was interesting for its own sake and not just for its association
with AI, even though that was the main reason people cared about
it at the time. So I decided to focus on Lisp. In fact, I decided
to

tide pools. It felt like I was doing life right. I remember that
because I was slightly dismayed at how novel it felt. The good news
is that I had more moments like this over the next few years.In the
```

```
/var/tmp/ipykernel_5220/990906280.py:1: LangChainDeprecationWarning: The method `
BaseRetriever.get_relevant_documents` was deprecated in langchain-core 0.1.46 and
will be removed in 1.0. Use :meth:`~invoke` instead.
  docs = retriever.get_relevant_documents(
```

Vector Stores

Vector Store is a common type of index or a database to store vectors (numerical embeddings). Conceptually, think of them as tables with a column for embeddings (vectors) and a column for metadata.

Example

Embedding	Metadata
`[-0.00015641732898075134, -0.003165106289088726, ...]`	`{'date' : '1/2/23}`
`[-0.00035465431654651654, 1.4654131651654516546, ...]`	`{'date' : '1/3/23}`

- Chroma & FAISS are easy to work with locally.
- Vertex AI Vector Search (Matching Engine) is fully managed vector store on Google Cloud, developers can just add the embeddings to its index and issue a search query with a key embedding for the blazingly fast vector search.

LangChain VectorStore is integrated with Vertex AI Vector Search.

In [35]:
```
loader = WebBaseLoader("http://www.paulgraham.com/worked.html")
documents = loader.load()

# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=50

# Split your docs into texts
texts = text_splitter.split_documents(documents)
```

In [36]:
```
print(f"You have {len(texts)} documents")
```

```
You have 52 documents
```

In [37]:
```
embedding_list = embeddings.embed_documents([text.page_content for text in texts
```

In [38]:
```
print(f"You have {len(embedding_list)} embeddings")
print(f"Here's a sample of one: {embedding_list[0][:3]}...")
```

```
You have 52 embeddings
Here's a sample of one: [-0.014726029708981514, -0.02730403281748295, -0.04264757
4096918106]...
```

VectorStore stores your embeddings (☝️) and makes them easily searchable.

Memory

Memory is the concept of storing and retrieving data in the process of a conversation. Memory helps LLMs remember information you've chatted about in the past or more complicated information retrieval.

There are many types of memory, explore the documentation to see which one fits your use case.

ConversationBufferMemory

Memory keeps conversation state throughout a user's interactions with a language model. `ConversationBufferMemory` memory allows for storing of messages and then extracts the messages in a variable.

We'll use `ConversationChain` to have a conversation and load context from memory. We will look into Chains in the next section.

In [39]:
```
conversation = ConversationChain(
    llm=llm, verbose=True, memory=ConversationBufferMemory()
)

conversation.predict(input="Hi there!")
```

```
/var/tmp/ipykernel_5220/1992978922.py:2: LangChainDeprecationWarning: Please see
the migration guide at: https://python.langchain.com/docs/versions/migrating_memo
ry/
  llm=llm, verbose=True, memory=ConversationBufferMemory()
/var/tmp/ipykernel_5220/1992978922.py:1: LangChainDeprecationWarning: The class `
ConversationChain` was deprecated in LangChain 0.2.7 and will be removed in 1.0.
Use :meth:`~RunnableWithMessageHistory: https://python.langchain.com/v0.2/api_ref
erence/core/runnables/langchain_core.runnables.history.RunnableWithMessageHistor
y.html` instead.
  conversation = ConversationChain(
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is tal
kative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: Hi there!
AI:

> Finished chain.
```

Out[39]:
```
"Hi there! 👋 It's nice to meet you.  What can I do for you today? I'm ready to
chat about anything, from current events to your favorite hobbies. 😊 \n"
```

In [40]:
```
conversation.predict(input="What is the capital of France?")
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is tal
kative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.

Current conversation:
Human: Hi there!
AI: Hi there! 👋 It's nice to meet you.  What can I do for you today? I'm ready
to chat about anything, from current events to your favorite hobbies. 😊

Human: What is the capital of France?
AI:

> Finished chain.
```

Out[40]:
```
"The capital of France is **Paris**. It's a beautiful city known for its iconic
landmarks like the Eiffel Tower, the Louvre Museum, and the Notre Dame Cathedra
l. Did you know that Paris is also a major center for fashion, art, and cultur
e? 😊 \n"
```

In [41]:
```
conversation.predict(input="What are some popular places I can see in France?")
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:
Human: Hi there!
AI: Hi there! 👋 It's nice to meet you.  What can I do for you today? I'm ready to chat about anything, from current events to your favorite hobbies. 😊

Human: What is the capital of France?
AI: The capital of France is **Paris**. It's a beautiful city known for its iconic landmarks like the Eiffel Tower, the Louvre Museum, and the Notre Dame Cathedral. Did you know that Paris is also a major center for fashion, art, and culture? 😊

Human: What are some popular places I can see in France?
AI:

> Finished chain.
```

Out[41]:
```
"France has so many amazing places to see! Here are a few popular ones:\n\n* **Paris:**  As the capital, it's a must-see! You can visit the Eiffel Tower, Louvre Museum, Notre Dame Cathedral (which is currently being restored after a fire in 2019), walk along the Seine River, explore the charming Latin Quarter, and enjoy delicious French pastries.\n* **The French Riviera:**  This area on the Mediterranean coast is known for its beautiful beaches, luxury resorts, and glamorous cities like Nice, Cannes, and Monaco. You can soak up the sun, go for a swim, visit art galleries, or try your luck at the casino in Monte Carlo. \n* **The Loire Valley:**  This region is famous for its magnificent castles, like Chambord, Chenonceau, and Villandry. You can wander through their grand halls, explore their manicured gardens, and imagine the lives of the French royalty who once lived there. \n* **Mont Saint-Michel:** This stunning island monastery is a UNESCO World Heritage Site and offers breathtaking views of the surrounding bay. You can walk across the causeway to the island, explore the abbey, and learn about its fascinating history. \n* **Provence:** This sunny region in southeastern France is known for its lavender fields, vineyards, and charming villages. You can visit the ancient Roman city of Arles, explore the hilltop villages, and sample the delicious regional cuisine. \n\nDo any of these places sound interesting to you?  I can tell you more about any of them if you'd like! 😊 \n"
```

In [42]:
```
conversation.predict(input="What question did I ask first?")
```

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is tal
kative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.

Current conversation:
Human: Hi there!
AI: Hi there! 👋 It's nice to meet you.  What can I do for you today? I'm ready
to chat about anything, from current events to your favorite hobbies. 😊

Human: What is the capital of France?
AI: The capital of France is **Paris**. It's a beautiful city known for its iconi
c landmarks like the Eiffel Tower, the Louvre Museum, and the Notre Dame Cathedra
l. Did you know that Paris is also a major center for fashion, art, and culture?
😊

Human: What are some popular places I can see in France?
AI: France has so many amazing places to see! Here are a few popular ones:

* **Paris:**  As the capital, it's a must-see! You can visit the Eiffel Tower, Lo
uvre Museum, Notre Dame Cathedral (which is currently being restored after a fire
in 2019), walk along the Seine River, explore the charming Latin Quarter, and enj
oy delicious French pastries.
* **The French Riviera:**  This area on the Mediterranean coast is known for its
beautiful beaches, luxury resorts, and glamorous cities like Nice, Cannes, and Mo
naco. You can soak up the sun, go for a swim, visit art galleries, or try your lu
ck at the casino in Monte Carlo.
* **The Loire Valley:**  This region is famous for its magnificent castles, like
Chambord, Chenonceau, and Villandry. You can wander through their grand halls, ex
plore their manicured gardens, and imagine the lives of the French royalty who on
ce lived there.
* **Mont Saint-Michel:** This stunning island monastery is a UNESCO World Heritag
e Site and offers breathtaking views of the surrounding bay. You can walk across
the causeway to the island, explore the abbey, and learn about its fascinating hi
story.
* **Provence:** This sunny region in southeastern France is known for its lavende
r fields, vineyards, and charming villages. You can visit the ancient Roman city
of Arles, explore the hilltop villages, and sample the delicious regional cuisin
e.

Do any of these places sound interesting to you?  I can tell you more about any o
f them if you'd like! 😊

Human: What question did I ask first?
AI:

> Finished chain.
```

Out[42]: 'You asked me first, "What is the capital of France?" 😊 \n'

Chains ⛓️ ⛓️ ⛓️

Chains are a generic concept in LangChain allowing to combine different LLM calls and action automatically.

Ex:

```
Summary #1, Summary #2, Summary #3 --> Final Summary
```

There are many applications of chains search to see which are best for your use case.

We'll cover a few of them:

1. Simple Sequential Chains

Sequential chains are a series of chains, called in deterministic order.
`SimpleSequentialChain` are easy chains where each step uses the output of an LLM as an input into another. Good for breaking up tasks (and keeping the LLM focused).

In [43]:
```
template = """Your job is to come up with a classic dish from the area that the
% USER LOCATION
{user_location}

YOUR RESPONSE:
"""
prompt_template = PromptTemplate(input_variables=["user_location"], template=tem

# Holds my 'location' chain
location_chain = LLMChain(llm=llm, prompt=prompt_template)
```

```
/var/tmp/ipykernel_5220/1937169603.py:10: LangChainDeprecationWarning: The class
`LLMChain` was deprecated in LangChain 0.1.17 and will be removed in 1.0. Use :me
th:`~RunnableSequence, e.g., `prompt | llm`` instead.
  location_chain = LLMChain(llm=llm, prompt=prompt_template)
```

In [44]:
```
template = """Given a meal, give a short and simple recipe on how to make that d
% MEAL
{user_meal}

YOUR RESPONSE:
"""
prompt_template = PromptTemplate(input_variables=["user_meal"], template=templat

# Holds my 'meal' chain
meal_chain = LLMChain(llm=llm, prompt=prompt_template)
```

In [45]:
```
overall_chain = SimpleSequentialChain(chains=[location_chain, meal_chain], verbo
```

In [46]:
```
review = overall_chain.run("Rome")
```

```
/var/tmp/ipykernel_5220/3989572581.py:1: LangChainDeprecationWarning: The method
`Chain.run` was deprecated in langchain 0.1.0 and will be removed in 1.0. Use :me
th:`~invoke` instead.
  review = overall_chain.run("Rome")
```

```
> Entering new SimpleSequentialChain chain...
For Rome, a classic dish that comes to mind is **Cacio e Pepe**.

This simple yet incredibly delicious pasta dish features just three main ingredie
nts:

* **Cacio:** Pecorino Romano cheese, a sharp and salty sheep's milk cheese.
* **Pepe:** Black pepper, freshly ground for the best flavor.
* **Pasta:** Typically, spaghetti or tonnarelli, a thick, flat pasta common in Ro
me.

The key to a perfect Cacio e Pepe is the perfect balance of creamy cheese, spicy
pepper, and perfectly cooked pasta. The cheese is grated directly into the pasta
while it's still hot, allowing it to melt and create a sauce that clings to the p
asta. The pepper is added generously, providing a robust flavor that complements
the cheese.

Cacio e Pepe is a true Roman classic, enjoyed by locals and tourists alike. It's
a simple dish that showcases the quality of the ingredients and the skill of the
chef.

## Cacio e Pepe: A Roman Classic

**Ingredients:**

* 1 pound spaghetti or tonnarelli
* 1 cup grated Pecorino Romano cheese
* Freshly ground black pepper, to taste
* Salt, to taste

**Instructions:**

1. Bring a large pot of salted water to a boil. Add the pasta and cook according
to package directions until al dente.
2. While the pasta is cooking, grate the Pecorino Romano cheese and set aside.
3. Drain the pasta, reserving about 1 cup of pasta water.
4. Return the pasta to the pot.  Add the reserved pasta water and a generous amou
nt of freshly ground black pepper. Toss to combine.
5. Add the grated Pecorino Romano cheese, a little at a time, while continuously
tossing the pasta. The cheese will melt and create a creamy sauce.
6. Continue adding cheese until the sauce reaches your desired consistency. Add m
ore pasta water if needed to loosen the sauce.
7. Taste and adjust seasoning with salt and pepper as needed.
8. Serve immediately and enjoy!


> Finished chain.
```

2. Summarization Chain

Summarization Chain easily runs through a long numerous documents and get a summary.

There are multiple chain types such as Stuffing, Map-Reduce, Refine, Map-Rerank. Check out documentation for other chain types besides `map-reduce`.

In [47]:

```
loader = WebBaseLoader(
    "https://cloud.google.com/blog/products/ai-machine-learning/how-to-use-groun
)
documents = loader.load()

print(f"# of words in the document = {len(documents[0].page_content)}")

# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=50

# Split your docs into texts
texts = text_splitter.split_documents(documents)

# There is a lot of complexity hidden in this one line. I encourage you to check
chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=True)
chain.run(texts)
```

# of words in the document = 13530

> Entering new MapReduceDocumentsChain chain...

> Entering new LLMChain chain...
Prompt after formatting:
Write a concise summary of the following:

"How to use Grounding for your LLMs with text embeddings | Google Cloud BlogJump to ContentCloudBlogContact sales Get started for free CloudBlogSolutions & technologyAI & Machine LearningAPI ManagementApplication DevelopmentApplication Modernization Chrome EnterpriseComputeContainers & KubernetesData AnalyticsDatabasesDevOps & SREMaps & GeospatialSecuritySecurity & IdentityThreat IntelligenceInfrastructureInfrastructure ModernizationNetworkingProductivity & CollaborationSAP on Google CloudStorage & Data TransferSustainabilityEcosystemIT LeadersIndustriesFinancial ServicesHealthcare & Life SciencesManufacturingMedia & EntertainmentPublic SectorRetailSupply ChainTelecommunicationsPartnersStartups & SMBTraining & CertificationsInside Google CloudGoogle Cloud Next & EventsGoogle Maps PlatformGoogle Workspace Developers & PractitionersTransform with Google CloudContact sales Get started for free AI & Machine LearningVertex AI Embeddings for Text: Grounding LLMs made easyMay 25, 2023Kaz SatoDeveloper Advocate, Cloud AIIvan CheungDeveloper Programs Engineer, Google CloudMany people are now starting to think about how to bring Gen AI and large language models (LLMs) to production services. You may be wondering "How to integrate LLMs or AI chatbots with existing IT systems, databases and business data?", "We have thousands of products. How can I let LLM memorize them all precisely?", or "How to handle the hallucination issues in AI chatbots to build a reliable service?". Here is a quick"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"to build a reliable service?". Here is a quick solution: grounding with embeddings and vector search.What is grounding? What are embedding and vector search? In this post, we will learn these crucial concepts to build reliable Gen AI services for enterprise use. But before we dive deeper, here is an example:Semantic search on 8 million Stack Overflow questions in milliseconds. (Try the demo here)This demo is available as a public live demo here. Select "STACKOVERFLOW" and enter any coding question as a query, so it runs a text search on 8 million questions posted on Stack Overflow.The following points make this demo unique:LLM-enabled semantic search: The 8 million Stack Overflow questions and query text are both interpreted by Vertex AI Generative AI models. The model understands the meaning and intent (semantics) of the text and code snippets in the question body at librarian-level precision. The demo leverages this ability for finding highly relevant questions and goes far beyond simple keyword search in terms of user experience. For example, if you enter "How do I write a class that instantiates only once", then the demo shows "How to create a singleton class" at the top, as the model knows their meanings are the same in the context of computer programming.Grounded to business facts: In this demo, we didn't try having the LLM to memorize the 8 million items with complex and lengthy prompt engineering. Instead, we attached the Stack Overflow dataset to the model as an"

CONCISE SUMMARY:
Prompt after formatting:

Write a concise summary of the following:

"the Stack Overflow dataset to the model as an external memory using vector searc h, and used no prompt engineering. This means, the outputs are all directly "grou nded" (connected) to the business facts, not the artificial output from the LLM. So the demo is ready to be served today as a production service with mission crit ical business responsibility. It does not suffer from the limitation of LLM memor y or unexpected behaviors of LLMs such as the hallucinations.Scalable and fast: T he demo gives you the search results in tens of milliseconds while retaining the deep semantic understanding capability. Also, the demo is capable of scaling out to handle thousands of search queries every second. This is enabled with the comb ination of LLM embeddings and Google AI's vector search technology.The key enable rs of this solution are 1) the embeddings generated with Vertex AI Embeddings for Text and 2) fast and scalable vector search by Vertex AI Vector Search. Let's sta rt by taking a look at these technologies.First key enabler: Vertex AI Embeddings for TextOn May 10, 2023, Google Cloud announced the following Embedding APIs for Text and Image. They are available on Vertex AI Model Garden. Embeddings for Text : The API takes text input up to 3,072 input tokens and outputs 768 dimensional t ext embeddings, and is available as a public preview. As of May 10, 2023, the pri cing is $0.0001 per 1000 characters (the latest pricing is available on the Prici ng for Generative AI models"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"available on the Pricing for Generative AI models page).Embeddings for Image: Ba sed on Google AI's Contrastive Captioners (CoCa) model, the API takes either imag e or text input and outputs 1024 dimensional image/text multimodal embeddings, av ailable to trusted testers. This API outputs so-called "multimodal" embeddings, e nabling multimodal queries where you can execute semantic search on images by tex t queries, or vise-versa. We will feature this API in another blog post soon.In t his blog, we will explain more about why embeddings are useful and show you how t o build and an application leveraging Embeddings API for Text. In a future blog p ost, we will provide a deep dive on Embeddings API for Image.Embeddings API for T ext on Vertex AI Model GardenWhat is embeddings?So, what are semantic search and embeddings? With the rise of LLMs, why is it becoming important for IT engineers and ITDMs to understand how they work? To learn it, please take a look at this vi deo from a Google I/O 2023 session for 5 minutes:Also, Foundational courses: Embe ddings on Google Machine Learning Crush Course and Meet AI’s multitool: Vector em beddings by Dale Markowitz are great materials to learn more about embeddings.LLM text embedding business use casesWith the embedding API, you can apply the innova tion of embeddings, combined with the LLM capability, to various text processing tasks, such as:LLM-enabled Semantic Search: text embeddings can be used to repres ent both the meaning and intent of a"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"to represent both the meaning and intent of a user's query and documents in the embedding space. Documents that have similar meaning to the user's query intent w ill be found fast with vector search technology. The model is capable of generati ng text embeddings that capture the subtle nuances of each sentence and paragraph s in the document.LLM-enabled Text Classification: LLM text embeddings can be use

d for text classification with a deep understanding of different contexts without any training or fine-tuning (so-called zero-shot learning). This wasn't possible with the past language models without task-specific training.LLM-enabled Recommendation: The text embedding can be used for recommendation systems as a strong feature for training recommendation models such as Two-Tower model. The model learns the relationship between the query and candidate embeddings, resulting in next-gen user experience with semantic product recommendation.LLM-enabled Clustering, Anomaly Detection, Sentiment Analysis, and more, can be also handled with the LLM-level deep semantics understanding.Sorting 8 million texts at "librarian-level" precision Vertex AI Embeddings for Text has an embedding space with 768 dimensions. As explained in the video above, the space represents a huge map of a wide variety of texts in the world, organized by their meanings. With each input text, the model can find a location (embedding) in the map.The API can take 3,072 input tokens, so it can digest the overall"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"3,072 input tokens, so it can digest the overall meaning of a long text and even programming code, and represent it as single embedding. It is like having a librarian knowledgeable about a wide variety of industries, reading through millions of texts carefully, and sorting them with millions of nano-categories that can classify even slight differences of subtle nuances.By visualizing the embedding space, you can actually observe how the model sorts the texts at the "librarian-level" precision. Nomic AI provides a platform called Atlas for storing, visualizing and interacting with embedding spaces with high scalability and in a smooth UI, and they worked with Google for visualizing the embedding space of the 8 million Stack Overflow questions. You can try exploring around the space, zooming in and out to each data point on your browser on this page, courtesy of Nomic AI.8 million Stack Overflow questions embedding spaceVisualized by Nomic AI Atlas (Try exploring it here)Examples of the "librarian-level" semantic understanding by Embeddings API with Stack Overflow questionsNote that this demo didn't require any training or fine-tuning with computer programming specific datasets. This is the innovative part of the zero-shot learning capability of the LLM; it can be applied to a wide variety of industries, including finance, healthcare, retail, manufacturing, construction, media, and more, for deep semantic search on the industry-focused business documents without spending time"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"business documents without spending time and cost for collecting industry specific datasets and training models.The second key enabler: fast and scalable Vector SearchThe second key enabler of the Stack Overflow demo shown earlier is the vector search technology. This is another innovation we are having in the data science field.The problem is "how to find similar embeddings in the embedding space". Since embeddings are vectors, this can be done by calculating the distance or similarity between vectors, as shown below.But this isn't easy when you have millions or billions of embeddings. For example, if you have 8 million embeddings with 768 dimensions, you would need to repeat the calculation in the order of 8 million x 768. This would take a very long time to finish. Actually, when we tried this on BigQuery with one million embeddings five years ago, it took 20 seconds.So the researchers have been studying a technique called Approximate Nearest Neighbor (ANN) for faster search. ANN uses "vector quantization" for separating the space int

o multiple spaces with a tree structure. This is similar to the index in relation al databases for improving the query performance, enabling very fast and scalable search with billions of embeddings.With the rise of LLMs, the ANN is getting popu lar quite rapidly, known as the Vector Search technology.In 2020, Google Research published a new ANN algorithm called ScaNN. It is considered one of the best ANN algorithms in the industry, also the"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"the best ANN algorithms in the industry, also the most important foundation for search and recommendation in major Google services such as Google Search, YouTube and many others.Google Cloud developers can take the full advantage of Google's v ector search technology with Vertex AI Vector Search. With this fully managed ser vice, developers can just add the embeddings to its index and issue a search quer y with a key embedding for the blazingly fast vector search. In the case of the S tack Overflow demo, Vector Search can find relevant questions from 8 million embe ddings in tens of milliseconds.With Vector Search, you don't need to spend much t ime and money building your own vector search service from scratch or using open source tools if your goal is high scalability, availability and maintainability f or production systems.Grounding LLM outputs with Vector SearchBy combining the Em beddings API and Vector Search, you can use the embeddings to "ground" LLM output s to real business data with low latency:In the case of the Stack Overflow demo s hown earlier, we've built a system with the following architecture.Stack Overflow semantic search demo architectureThe demo architecture has two parts: 1) building a Vector Search index with Vertex AI Workbench and the Stack Overflow dataset on BigQuery (on the right) and 2) processing vector search requests with Cloud Run (on the left) and Vector Search. For the details, please see the sample Notebook on GitHub.Grounding LLMs with LangChain"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"Notebook on GitHub.Grounding LLMs with LangChain and Vertex AIIn addition to the architecture used for the Stack Overflow demo, another popular way for grounding is to enter the vector search result into the LLM and let the LLM generate the fi nal answer text for the user. LangChain is a popular tool for implementing this p ipeline, and Vertex AI Gen AI embedding APIs and Vector Search are definitely bes t suited for LangChain integration. In a future blog post, we will explore this t opic further. So stay tuned!How to get startedIn this post, we have seen how the combination of Embeddings for Text API and Vector Search allows enterprises to u se Gen AI and LLMs in a grounded and reliable way. The fine-grained semantic unde rstanding capability of the API can bring the intelligence to information search and recommendation in a wide variety of businesses, setting a new standard of use r experience in enterprise IT systems.To get started, please check out the follow ing resources:Stack Overflow semantic search demo: sample Notebook on GitHubVerte x AI Embeddings for Text API documentationVector Search documentationAI & Machine LearningGoogle Cloud advances generative AI at I/O: new foundation models, embedd ings, and tuning tools in Vertex AIBy June Yang • 5-minute readPosted inAI & Mach ine LearningDevelopers & PractitionersRelated articlesPartnersArize, Vertex AI AP I: Evaluation workflows to accelerate generative app development and AI ROIBy Gab e Barcelos • 9-minute readAI & Machine"

CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:

"ROIBy Gabe Barcelos • 9-minute readAI & Machine LearningPyTorch/XLA 2.5: vLLM support and an improved developer experienceBy Manfei Bai • 4-minute readComputePowerful infrastructure innovations for your AI-first futureBy Mark Lohmeyer • 12-minute readAI & Machine LearningGemini models are coming to GitHub CopilotBy Keith Ballinger • 2-minute readFooter LinksFollow usGoogle CloudGoogle Cloud ProductsPrivacyTermsCookies management controlsHelpLanguageEnglishDeutschFrançais한국어日本語"

CONCISE SUMMARY:

> Finished chain.

> Entering new LLMChain chain...
Prompt after formatting:
Write a concise summary of the following:

"This blog post from Google Cloud discusses how to use text embeddings to "ground" large language models (LLMs) in real-world data. Grounding LLMs helps address the challenges of integrating them into production services, such as:

* **Connecting LLMs to existing systems and data:** Provides a way to connect LLMs to databases and business data.
* **Handling large amounts of information:** Enables LLMs to accurately process and learn from extensive datasets like product catalogs.
* **Mitigating hallucination issues:** Improves the reliability of AI chatbots by grounding responses in factual information.

This text explains how to build reliable Gen AI services using "grounding" with embeddings and vector search. It uses a demo searching 8 million Stack Overflow questions as an example. The demo leverages LLM-powered semantic search, meaning it understands the meaning and intent of questions and code snippets, providing better results than simple keyword search. Instead of making the LLM memorize all the data, "grounding" connects the dataset to the model, allowing for faster and more efficient search.

This demo leverages Google AI's Vertex AI Embeddings for Text and Vertex AI Vector Search to connect LLMs to real-world data from the Stack Overflow dataset. This eliminates LLM hallucinations and allows for fast, scalable, and reliable search results with deep semantic understanding. The demo is production-ready and can handle thousands of queries per second.

This blog post introduces Google's Embeddings API for Text, a tool that uses text embeddings to enable semantic search, allowing users to search for text based on meaning rather than exact keywords. The API leverages the power of Large Language Models (LLMs) to create embeddings that represent the meaning and intent of text. The blog also highlights the importance of understanding embeddings in the context of LLMs and provides resources for learning more about this technology. It further discusses how embeddings can be used for various text processing tasks, including semantic search and text similarity detection, showcasing the practical app

lications of this powerful tool.

Large Language Models (LLMs) are revolutionizing text understanding by generating embeddings that capture the meaning and intent of text. These embeddings enable:

* **Fast and accurate search:** Documents with similar meaning to a query are qui ckly found using vector search.
* **Zero-shot text classification:** LLM embeddings understand context without tr aining, classifying text based on meaning.
* **Enhanced recommendations:** LLMs improve recommendation systems by understan ding the semantic relationship between queries and products.
* **Advanced applications:** Clustering, anomaly detection, sentiment analysis, a nd more benefit from LLMs' deep semantic understanding.

Vertex AI Embeddings for Text provides a 768-dimensional embedding space, effecti vely mapping and organizing diverse texts based on their meaning. This allows for precise text sorting at "librarian-level" accuracy.

This text describes a powerful language model capable of understanding the meanin g of long texts and even programming code, representing it as a single embedding. This allows for advanced semantic search and categorization, similar to a librari an meticulously sorting millions of texts into nuanced categories. Nomic AI's Atl as platform visualizes these embeddings, allowing users to explore relationships between texts. The example of 8 million Stack Overflow questions demonstrates the model's ability to understand computer programming concepts without specific trai ning, showcasing its "zero-shot learning" capability. This technology has the pot ential to revolutionize semantic search across various industries.

This text discusses the challenges of searching for similar embeddings in large d atasets and introduces **vector search technology** as a solution. Vector search uses **Approximate Nearest Neighbor (ANN)** algorithms, like Google's **ScaNN**, to quickly find similar embeddings in high-dimensional spaces. This is crucial fo r tasks like analyzing business documents without the need for extensive data col lection and model training. ANN algorithms improve search speed and scalability b y partitioning the embedding space using techniques like **vector quantization**, similar to indexing in relational databases.

Google Cloud's Vertex AI Vector Search leverages Google's advanced neural network algorithms to provide a high-performance, fully managed vector search service. It enables developers to quickly and efficiently search through large datasets of em beddings, achieving blazingly fast results. This eliminates the need for building and maintaining a separate vector search infrastructure, allowing developers to f ocus on their core applications. By integrating with Google's Embeddings API, Vec tor Search can ground Large Language Model outputs to real-world data, enhancing accuracy and relevance. This is demonstrated through a Stack Overflow demo, where the system efficiently searches through 8 million embeddings to find relevant que stions, highlighting the potential of Vector Search in various applications.

This article discusses grounding LLMs with LangChain and Vertex AI, specifically using Embeddings for Text API and Vector Search. This combination allows for reli able and grounded information retrieval, enhancing search and recommendation syst ems in various businesses. The article highlights a Stack Overflow semantic searc h demo and provides resources for getting started with Vertex AI. It suggests fut ure exploration of using LangChain to generate final answer text from vector sear ch results.

```
This text is a list of recent articles published on the Google Cloud blog. The ar
ticles discuss advancements in AI and machine learning, focusing on PyTorch/XLA
2.5, powerful infrastructure innovations, and the integration of Gemini models wi
th GitHub Copilot.
"


CONCISE SUMMARY:

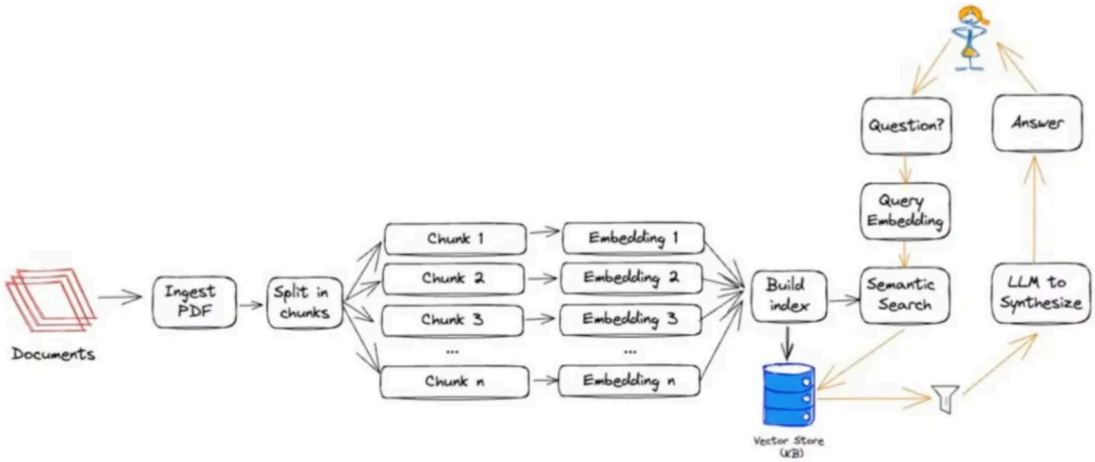
> Finished chain.

> Finished chain.
```

Out[47]:
```
"This article from Google Cloud explores the concept of grounding large languag
e models (LLMs) in real-world data using text embeddings. It explains how embed
ding technology, particularly Google's Embeddings API for Text, enables semanti
c search by capturing the meaning and intent of text.  This allows for more acc
urate and relevant search results, particularly when dealing with large dataset
s. The article highlights the use of vector search, specifically Google's Verte
x AI Vector Search, as a solution for efficiently searching through vast amount
s of embeddings. By combining Embeddings for Text and Vector Search, Google Clo
ud demonstrates how to build reliable and grounded AI services that connect LLM
s to real-world data, enhancing search and recommendation systems. The article
uses a Stack Overflow semantic search demo as a practical example, showcasing t
he potential of this technology for various industries. \n"
```

3. Question/Answering Chain

Question Answering Chains easily do QA over a set of documents using QA chain. There are multiple ways to do this with LangChain. We use **RetrievalQA** chain which uses `load_qa_chain` under the hood.



In [48]:
```
# Load GOOG's 10K annual report (92 pages).
url = "https://abc.xyz/assets/investor/static/pdf/20230203_alphabet_10K.pdf"
loader = PyPDFLoader(url)
documents = loader.load()
```

In [49]:
```
# split the documents into chunks

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=0)
```

```
docs = text_splitter.split_documents(documents)
print(f"# of documents = {len(docs)}")
```

```
# of documents = 263
```

In [50]:
```
# select embedding engine - we use Vertex AI Embeddings API
embeddings
```

Out[50]:
```
VertexAIEmbeddings(client=<vertexai.language_models.TextEmbeddingModel object a
t 0x7f72ee9aba90>, async_client=None, project='qwiklabs-gcp-02-abf532d75b70', l
ocation='us-central1', request_parallelism=5, max_retries=6, stop=None, model_n
ame='text-embedding-004', full_model_name=None, client_options=ClientOptions:
{'api_endpoint': 'us-central1-aiplatform.googleapis.com', 'client_cert_source':
None, 'client_encrypted_cert_source': None, 'quota_project_id': None, 'credenti
als_file': None, 'scopes': None, 'api_key': None, 'api_audience': None, 'univer
se_domain': None}, api_endpoint=None, api_transport=None, default_metadata=(),
additional_headers=None, client_cert_source=None, credentials=None, client_prev
iew=None, temperature=None, max_output_tokens=None, top_p=None, top_k=None, n=
1, seed=None, streaming=False, model_family=None, safety_settings=None, tuned_m
odel_name=None, instance={'max_batch_size': 250, 'batch_size': 250, 'min_batch_
size': 5, 'min_good_batch_size': 18, 'lock': <unlocked _thread.lock object at 0
x7f72eaf350c0>, 'batch_size_validated': False, 'task_executor': <concurrent.fut
ures.thread.ThreadPoolExecutor object at 0x7f72ee9a87f0>, 'get_embeddings_with_
retry': <function _TextEmbeddingModel.get_embeddings at 0x7f72eaed9870>})
```

In [51]:
```
# Store docs in local VectorStore as index
# it may take a while since API is rate limited
db = Chroma.from_documents(docs, embeddings)
```

In [52]:
```
# Expose index to the retriever
retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": 2})
```

In [53]:
```
# Create chain to answer questions

# Uses LLM to synthesize results from the search index.
qa = RetrievalQA.from_chain_type(
    llm=llm, chain_type="stuff", retriever=retriever, return_source_documents=Tr
)
```

In [54]:
```
query = "What was Alphabet's net income in 2022?"
result = qa({"query": query})
print(result)
```

```
/var/tmp/ipykernel_5220/809087267.py:2: LangChainDeprecationWarning: The method `
Chain.__call__` was deprecated in langchain 0.1.0 and will be removed in 1.0. Use
:meth:`~invoke` instead.
  result = qa({"query": query})
```

{'query': "What was Alphabet's net income in 2022?", 'result': "Alphabet's net income in 2022 was $59,972 million. \n", 'source_documents': [Document(metadata={'page': 48, 'source': 'https://abc.xyz/assets/investor/static/pdf/20230203_alphabet_10K.pdf'}, page_content='Alphabet Inc.\nCONSOLIDATED STATEMENTS OF INCOME\n(in millions, except per share amounts)\n Year Ended December 31,\n 2020 2021 2022\nRevenues $ 182,527 $ 257,637 $ 282,836 \nCosts and expenses:\nCost of revenues 84,732 110,939 126,203 \nResearch and development 27,573 31,562 39,500 \nSales and marketing 17,946 22,912 26,567 \nGeneral and administrative 11,052 13,510 15,724 \nTotal costs and expenses 141,303 178,923 207,994 \nIncome from operations 41,224 78,714 74,842 \nOther income (expense), net 6,858 12,020 (3,514) \nIncome before income taxes 48,082 90,734 71,328 \nProvision for income taxes 7,813 14,701 11,356 \nNet income $ 40,269 $ 76,033 $ 59,972 \nBasic net income per share of Class A, Class B, and Class C stock $ 2.96 $ 5.69 $ 4.59 \nDiluted net income per share of Class A, Class B, and Class C stock $ 2.93 $ 5.61 $ 4.56 \nSee accompanying notes.\nTable of Contents Alphabet Inc.\n48'), Document(metadata={'page': 35, 'source': 'https://abc.xyz/assets/investor/static/pdf/20230203_alphabet_10K.pdf'}, page_content='by an increase in compensation expenses of $1.1 billion, largely resulting from a 21% increase in average headcount, \nand an increase in third-party services fees of $815 million . In addition, there was a $551 million increase to the \nallowance for credit losses for accounts receivable, as the prior year comparable period reflected a decline in the \nallowance.\nSegment Profitability\nThe following table presents segment operating income (loss) (in millions).\nYear Ended December 31,\n2021 2022\nOperating income (loss):\nGoogle Services $ 91,855 $ 86,572 \nGoogle Cloud (3,099) (2,968) \nOther Bets (5,281) (6,083) \nCorporate costs, unallocated(1) (4,761) (2,679) \nTotal income from operations $ 78,714 $ 74,842 \n(1) Unallocated corporate costs primarily include corporate initiatives, corporate shared costs, such as finance and legal, including \ncertain fines and settlements, as well as costs associated with certain shared R&D activities. Additionally, hedging gains \n(losses) related to revenue are included in corporate costs and totaled $149 million and $2.0 billion in 2021 and 2022, \nrespectively.\nGoogle Services\nGoogle Services operating income decreased $5.3 billion from 2021 to 2022. The decrease in operating income \nwas primarily driven by increases in compensation expenses and TAC, partially offset by growth in revenues.\nTable of Contents Alphabet Inc.\n35')]}

Executive Overview

The following table summarizes our consolidated financial results (in millions, except for per share information and percentages):

	Year Ended December 31,			
	2021	2022	$ Change	% Change
Consolidated revenues	$ 257,637	$ 282,836	$ 25,199	10 %
Change in consolidated constant currency revenues[1]				14 %
Cost of revenues	$ 110,939	$ 126,203	$ 15,264	14 %
Operating expenses	$ 67,984	$ 81,791	$ 13,807	20 %
Operating income	$ 78,714	$ 74,842	$ (3,872)	(5)%
Operating margin	31 %	26 %		(5)%
Other income (expense), net	$ 12,020	$ (3,514)	$ (15,534)	(129)%
Net income	$ 76,033	$ 59,972	$ (16,061)	(21)%
Diluted EPS	$ 5.61	$ 4.56	$ (1.05)	(19)%

In [55]:
```
query = "How much office space reduction took place in 2023?"
result = qa({"query": query})
print(result)
```

{'query': 'How much office space reduction took place in 2023?', 'result': "The provided information doesn't specify the amount of office space reduction in 2023. It only states that Alphabet Inc. expects to incur exit costs of approximately $0.5 billion relating to office space reductions in the first quarter of 2023. \n", 'source_documents': [Document(metadata={'page': 83, 'source': 'https://abc.xyz/assets/investor/static/pdf/20230203_alphabet_10K.pdf'}, page_content='The following table presents long-lived assets by geographic area, which includes property and equipment, net \nand operating lease assets (in millions):\nAs of December 31,\n 2021 2022\nLong-lived assets:\nUnited States $ 80,207 $ 93,565 \nInternational 30,351 33,484 \nTotal long-lived assets $ 110,558 $ 127,049 \nNote 16. Subsequent Event \nIn January 2023, we announced a reduction of our workforce of approximately 12,000 roles. We expect to \nincur employee severance and related charges of $1.9 billion to $2.3 billion, the majority of which will be recognized in \nthe first quarter of 2023.\nIn addition, we are taking actions to optimize our global office space. As a result we expect to incur exit costs \nrelating to office space reductions of approximately $0.5 billion in the first quarter of 2023. We may incur additional \ncharges in the future as we further evaluate our real estate needs. \nTable of Contents Alphabet Inc.\n83'), Document(metadata={'page': 4, 'source': 'https://abc.xyz/assets/investor/static/pdf/20230203_alphabet_10K.pdf'}, page_content='• the expected timing, amount, and effect of Alphabet Inc.\'s share repurchases;\n• our long-term sustainability and diversity goals;\n• the unpredictability of the ongoing broader economic effects resulting from the war in Ukraine on our future \nfinancial results;\n• the expected financial effect of our announced workforce reduction and office space optimization;\n• our expectation that the change in estimated useful life of servers and certain network equipment will have a \nfavorable effect on our 2023 operating results;\nas well as other statements regarding our future operations, financial condition and prospects, and business strategies. \nForward-looking statements may appear throughout this report and other documents we file with the Securities and \nExchange Commission (SEC), including without limitation, the following sections: Part I, Item 1 "Business;" Part I, Item \n1A "Risk Factors;" and Part II, Item 7 "Management’s Discussion and Analysis of Financial Condition and Results of \nOperations." Forward-looking statements generally can be identified by words such as "anticipates," "believes," \n"estimates," "expects," "intends," "plans," "predicts," "projects," "will be," "will continue," "may," "could," "will likely \nresult," and similar expressions. These forward-looking statements are based on current expectations and \nassumptions that are subject to risks and uncertainties, which could cause our actual results to differ materially from')]}

Additionally, looking ahead to fiscal year 2023:

- In January 2023, we announced a reduction of our workforce of approximately 12,000 roles. We expect to incur employee severance and related charges of $1.9 billion to $2.3 billion, the majority of which will be recognized in the first quarter of 2023.

  In addition, we are taking actions to optimize our global office space. As a result we expect to incur exit costs relating to office space reductions of approximately $0.5 billion in the first quarter of 2023. We may incur additional charges in the future as we further evaluate our real estate needs.

- In January 2023, we completed an assessment of the useful lives of our servers and network equipment, resulting in a change in the estimated useful life of our servers and certain network equipment to six years, which we expect to result in a reduction of depreciation of approximately $3.4 billion for the full fiscal year 2023 for assets in service as of December 31, 2022, recorded primarily in cost of revenues and R&D expenses.

- As AI is critical to delivering our mission of bringing our breakthrough innovations into the real world, beginning in January 2023, we will update our segment reporting relating to certain of Alphabet's AI activities. DeepMind, previously reported within Other Bets, will be reported as part of Alphabet's corporate costs, reflecting its increasing collaboration with Google Services, Google Cloud, and Other Bets. Prior periods will be recast to conform to the revised presentation. See Note 15 of the Notes to Consolidated Financial Statements included in Item 8 of this Annual Report on Form 10-K for information relating to our segments.