04

LangChain + Google

The information in this presentation is classified:

Google confidential & proprietary

⚠️ This presentation is shared with you under NDA.

- Do **not** record or take screenshots of this presentation.
- Do **not** share or otherwise distribute the information in this presentation with anyone **inside** or **outside** of your organization.

Thank you!

In this module, you learn to ...

01 Simplify your generative AI code using LangChain

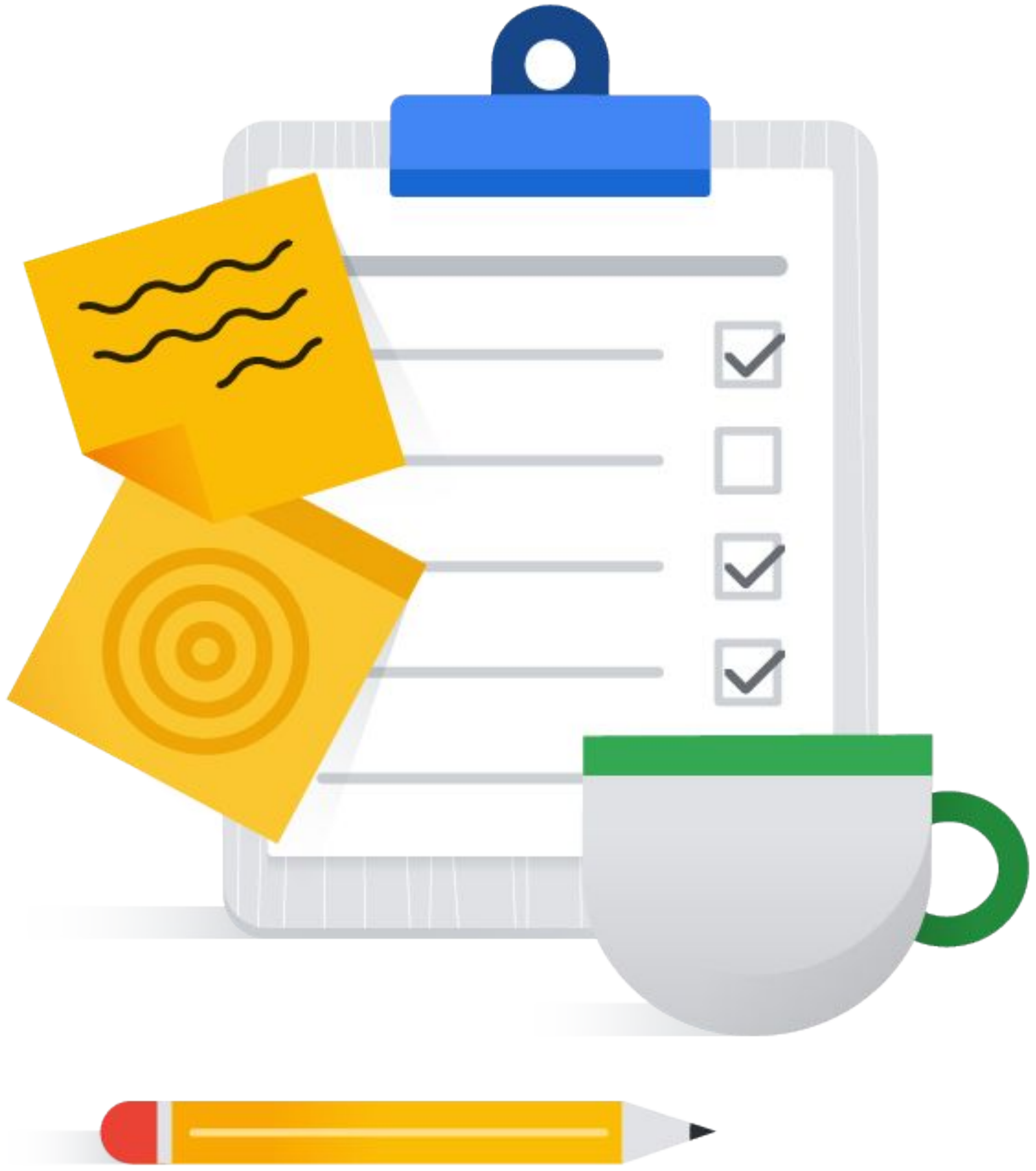
02 Load text from a variety of sources using Loaders

03 Explore LangChain Google Community components

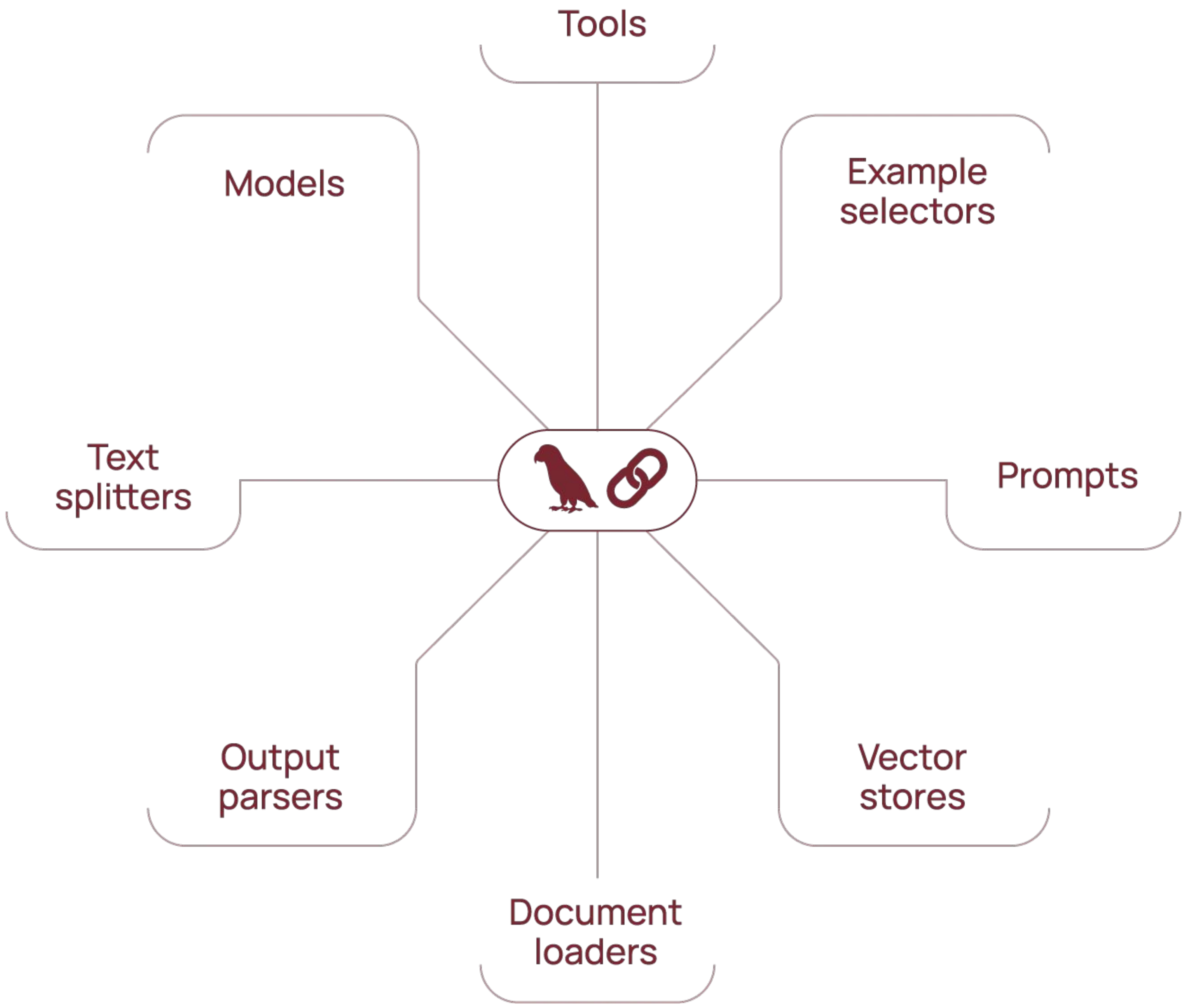
Topics

01 Basics of LangChain

02 LangChain Google Community Components



LangChain is a framework that simplifies connecting models, vector databases, and other LLM application components from various providers.



A chain allows you to use the | character as a Unix-style ‘pipe’ operator connecting components like prompt templates, LLMs, and output parsers:

```
# A chain represented in LangChain Expression Language (LCEL)
chain = prompt | model | output_parser
```

Chains implement the runnable interface which includes a `stream()` function

The `stream()` function will output data in chunks. This would produce output the same way as Gemini streaming

```
chain = prompt | llm | StrOutputParser()
for chunk in chain.stream({"topic": "COBOL", "language": "English"}):
    print(chunk, end="", flush=True)
```

Because LangChain standardizes the interfaces of models, you can easily switch from one model…

```
model_claude = ChatAnthropic(model='claude-3-opus-20240229',
               api_key=anthropic_api_key)
prompt = ChatPromptTemplate.from_template("tell me a short
joke about {topic}")
output_parser = StrOutputParser()

chain = prompt | model_claude | output_parser
chain.invoke({"topic": "pizza delivery"})
```

… to another model, without otherwise modifying your chain.

```
model_gemini = GoogleGenerativeAI(model="gemini-pro",
                           google_api_key=google_api_key)
prompt = ChatPromptTemplate.from_template("tell me a short
joke about {topic}")
output_parser = StrOutputParser()

chain = prompt | model_gemini | output_parser
chain.invoke({"topic": "pizza delivery"})
```

Using LangChain to call Gemini Pro

```
!pip install langchain-google-genai

from langchain_google_genai import GoogleGenerativeAI

# API Key Created in APIs & Services > Credentials
# and restricted to the Generative Language API
google_api_key = “”

model_gemini = GoogleGenerativeAI(model="gemini-pro",
                           google_api_key=google_api_key,
                           temperature=0.7, top_p=0.6)

model_gemini.invoke("Provide instructions for making a good sandwich.")
```

Other features like Safety Settings are still available

```
from google.generativeai.types.safety_types import HarmBlockThreshold, HarmCategory

safety_settings = {
  HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT: HarmBlockThreshold.BLOCK_ONLY_HIGH,
  HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE,
  HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE,
  HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT: HarmBlockThreshold.BLOCK_LOW_AND_ABOVE,
}
model_gemini = GoogleGenerativeAI(model="gemini-pro",
                           google_api_key=google_api_key,
                           temperature=0.7, top_p=0.6,
                           safety_settings=safety_settings)

model_gemini.invoke("Provide instructions for making a good sandwich.")
```

LangChain provides utilities for loading text from different sources

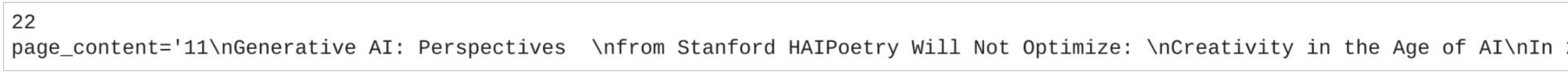
- CSV loader
- Directory loader
- HTML loader
- JSON loader
- Markdown loader
- PDF loader

PyPDFLoader example

```
from langchain.document_loaders import PyPDFLoader

loader = PyPDFLoader("Generative_AI_HAI_Perspectives.pdf")
pages = loader.load_and_split()

print(len(pages))
print(pages[10])
```



```
22
page_content='11\nGenerative AI: Perspectives  \nfrom Stanford HAIPoetry Will Not Optimize: \nCreativity in the Age of AI\nIn
```

WebBaseLoader example

```
from langchain.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://www.example.com/machine-learning-on-google-cloud")
data = loader.load()
```

Splitters aid in chunking content to fit within token windows or for chunking documents for retrieval

- Split by a given character (i.e. a line break of “`/n/n`”)
- Split code by reasonable heuristics for given languages
- Split by HTML headers (<h1>, <h2>, etc.)
- Split by Markdown headers (#, ##, etc.)
- Split recursively by certain characters (with a default order of `["\n\n", "\n", " ", ""]`)
- Split by token count

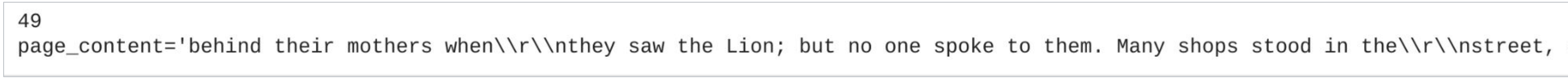
RecursiveCharacterTextSplitter example

```
from langchain.text_splitter import RecursiveCharacterTextSplitter, Language

html_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.HTML, chunk_size=5000, chunk_overlap=5
)
loader = WebBaseLoader("https://www.gutenberg.org/cache/epub/55/pg55.txt")
data = loader.load()

html_docs = html_splitter.create_documents([str(data)])

print(len(html_docs))
print(html_docs[20])
```



```
49
page_content='behind their mothers when\\r\\nthey saw the Lion; but no one spoke to them. Many shops stood in the\\r\\nstreet,
```

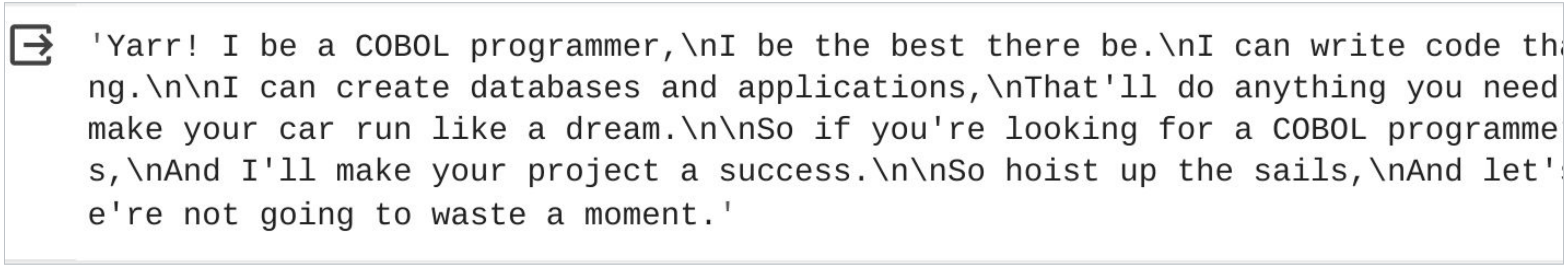
Prompt templates create a reusable interface for formatting prompts

```
from langchain.prompts import PromptTemplate
prompt_template = PromptTemplate.from_template(
    """
    Context: You write in the style of {style}.
    Write me a {output} about {thing}.
    """)

llm(prompt_template.format(style="a pirate", output="poem", thing="COBOL Programming"))
```

Note the parameters in curly braces {}

Set the parameters when invoking the model



```
'Yarr! I be a COBOL programmer,\nI be the best there be.\nI can write code th
ng.\n\nI can create databases and applications,\nThat'll do anything you need
make your car run like a dream.\n\nSo if you're looking for a COBOL programme
s,\nAnd I'll make your project a success.\n\nSo hoist up the sails,\nAnd let'
e're not going to waste a moment.'
```

Output parsers allow you to structure the output returned from the model

- LLMs return text
- Use parsers to return formatted text in whatever format you specify
- Built-in output parsers include:
  - List parser to return a collection of comma separated items
  - Datetime parser to format dates
  - Pydantic (JSON) parser to return JSON defined by a scheme
  - Others...
- You can also create custom output parsers

The List parser outputs a comma separated collection

```
from langchain.output_parsers import CommaSeparatedListOutputParser
from langchain.prompts import PromptTemplate
output_parser = CommaSeparatedListOutputParser()
format_instructions = output_parser.get_format_instructions()

prompt = PromptTemplate(
    template="""List five {subject}, List the items with no formatting.
    {format_instructions}""",
    input_variables=["subject"],
    partial_variables={"format_instructions": format_instructions}
)
_input = prompt.format(subject="ice cream flavors")
output = llm(_input)
output_parser.parse(output)
```

```
['chocolate', 'vanilla', 'strawberry', 'mint chocolate chip', 'cookie dough']
```

Pydantic (JSON) parser

```
from langchain.output_parsers import PydanticOutputParser
from langchain.pydantic_v1 import BaseModel, Field, validator
from langchain.prompts import PromptTemplate

class Joke(BaseModel):
    setup: str = Field(description="question to set up a joke")
    punchline: str = Field(description="answer to resolve the joke")

    @validator("setup")
    def question_ends_with_question_mark(cls, field):
        if field[-1] != "?":
            raise ValueError("Badly formed question!")
        return field
```

Add validation logic to the output. The setup field has to end with a question mark

Using the Pydantic JSON parser

```
prompt = PromptTemplate(
    template="Answer the user query.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)
prompt_and_model = prompt | llm
output = prompt_and_model.invoke({"query": "Tell me a joke about Python programming."})

parser = PydanticOutputParser(pydantic_object=Joke)
parser.invoke(output)
```

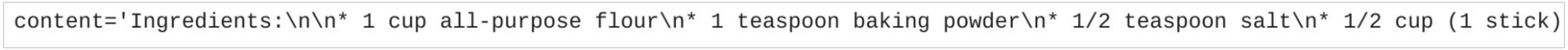
This sets up an input chain

Use the parser to format the results

```
Joke(setup='Why did the Python programmer get a dog?', punchline='Because he wanted a companion object!')
```

A chat is a conversation that includes System, Human, and AI messages

```
response = chat(
    [
        SystemMessage(content="You are a bot who knows about cooking"),
        HumanMessage(content="What's a good dessert for Thanksgiving"),
        AIMessage(content="Pumpkin pie is always a winner."),
        HumanMessage(content="Great, what is the recipe?")
    ]
)
print(response)
```



```
content='Ingredients:\n\n* 1 cup all-purpose flour\n* 1 teaspoon baking powder\n* 1/2 teaspoon salt\n* 1/2 cup (1 stick)
```

Chat prompt templates allow you to inject data into a conversation

```
from langchain.prompts import ChatPromptTemplate
chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful {job}. Your name is {name}."),
        ("human", "Hello, how are you doing?"),
        ("ai", "I'm doing well, thanks!"),
        ("human", "{user_input}"),
    ]
)
messages = chat_template.format_messages(job="Chef", name="Julia",
                      user_input="What is your name and what do you do?")
chat(messages)
```

```
AIMessage(content="My name is Julia, and I'm a helpful Chef. I can help you with your cooking needs.")
```

LangChain memory can be used to manage a conversation over time

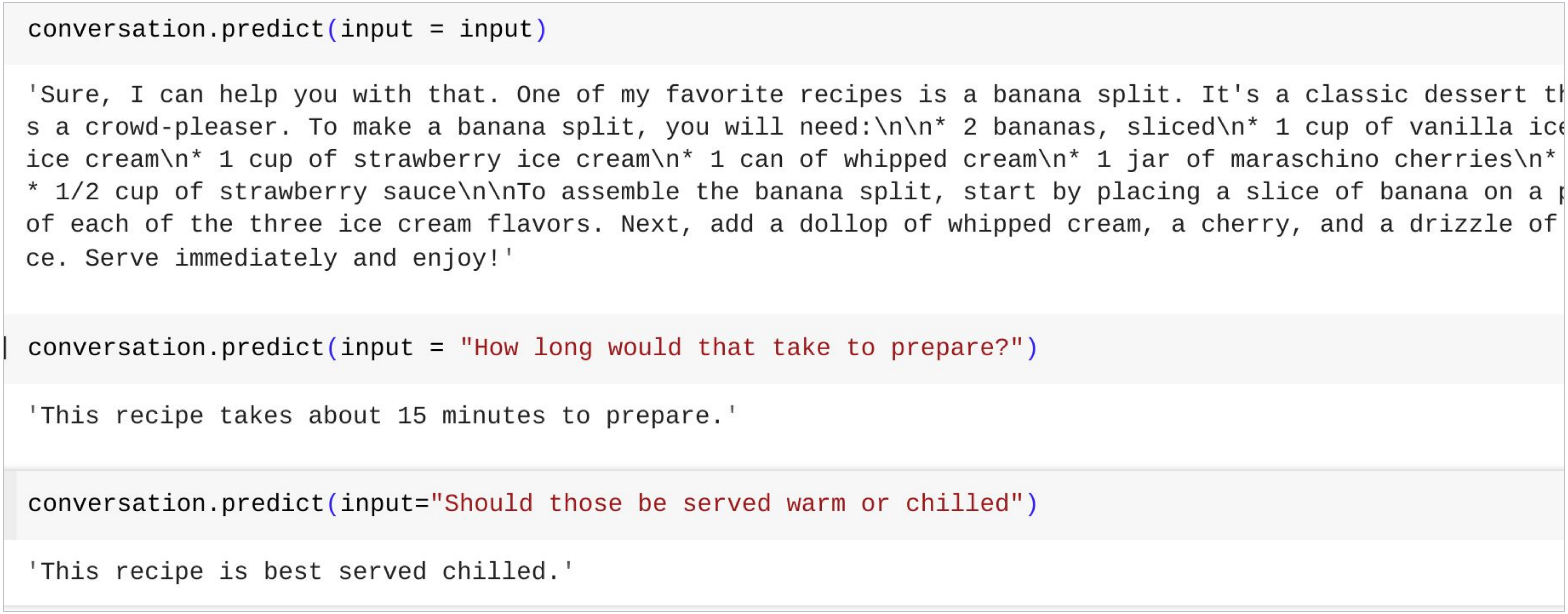
```
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
conversation = ConversationChain(llm=chat, memory=memory, verbose=False)

input = """
    System: You are a Chef named Julia.
    Human: What is a good recipe for dinner that includes bananas?
"""


conversation.predict(input = input)
```

Memory will automatically store your conversation so you can ask follow up questions



```
conversation.predict(input = input)
```

```
'Sure, I can help you with that. One of my favorite recipes is a banana split. It's a classic dessert th
s a crowd-pleaser. To make a banana split, you will need:\n\n* 2 bananas, sliced\n* 1 cup of vanilla ice
ice cream\n* 1 cup of strawberry ice cream\n* 1 can of whipped cream\n* 1 jar of maraschino cherries\n*
* 1/2 cup of strawberry sauce\n\nTo assemble the banana split, start by placing a slice of banana on a p
of each of the three ice cream flavors. Next, add a dollop of whipped cream, a cherry, and a drizzle of
ce. Serve immediately and enjoy!'
```

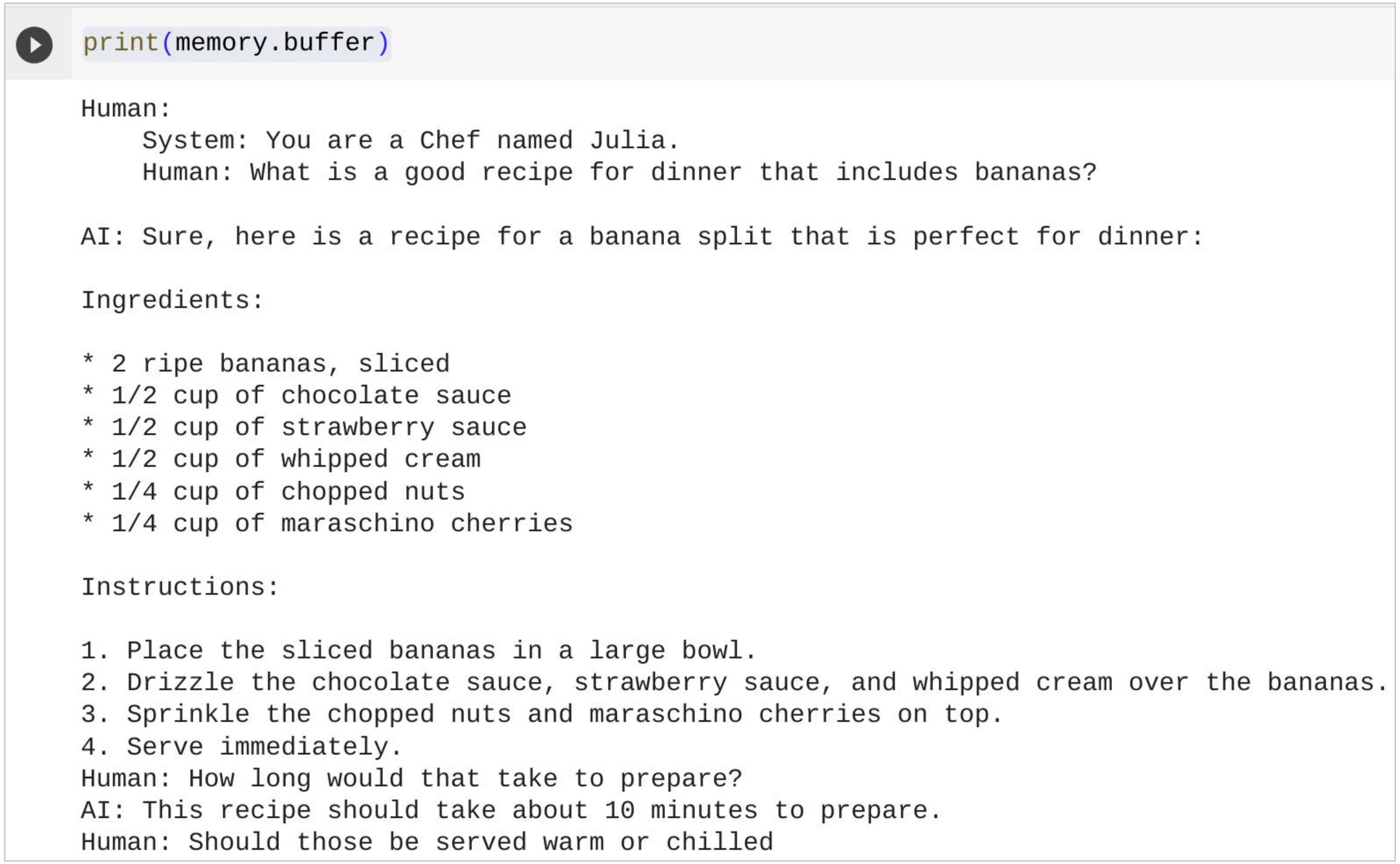
```
conversation.predict(input = "How long would that take to prepare?")
```

```
'This recipe takes about 15 minutes to prepare.'
```

```
conversation.predict(input="Should those be served warm or chilled")
```

```
'This recipe is best served chilled.'
```

Print the memory buffer to view the history of the conversation



```
print(memory.buffer)
```

```
Human:
    System: You are a Chef named Julia.
    Human: What is a good recipe for dinner that includes bananas?

AI: Sure, here is a recipe for a banana split that is perfect for dinner:

Ingredients:

* 2 ripe bananas, sliced
* 1/2 cup of chocolate sauce
* 1/2 cup of strawberry sauce
* 1/2 cup of whipped cream
* 1/4 cup of chopped nuts
* 1/4 cup of maraschino cherries

Instructions:

1. Place the sliced bananas in a large bowl.
2. Drizzle the chocolate sauce, strawberry sauce, and whipped cream over the bananas.
3. Sprinkle the chopped nuts and maraschino cherries on top.
4. Serve immediately.
Human: How long would that take to prepare?
AI: This recipe should take about 10 minutes to prepare.
Human: Should those be served warm or chilled
```

Retrievers return documents given an unstructured query. Vector stores are some examples.

```
from langchain_google_community import VertexAISearchRetriever

retriever = VertexAISearchRetriever(
    project_id=PROJECT_ID, location_id=LOCATION_ID,
    data_store_id=DATA_STORE_ID, max_documents=3,
)

query = "What are Alphabet's Other Bets?"

result = retriever.invoke(query)
for doc in result:
    print(doc)
```

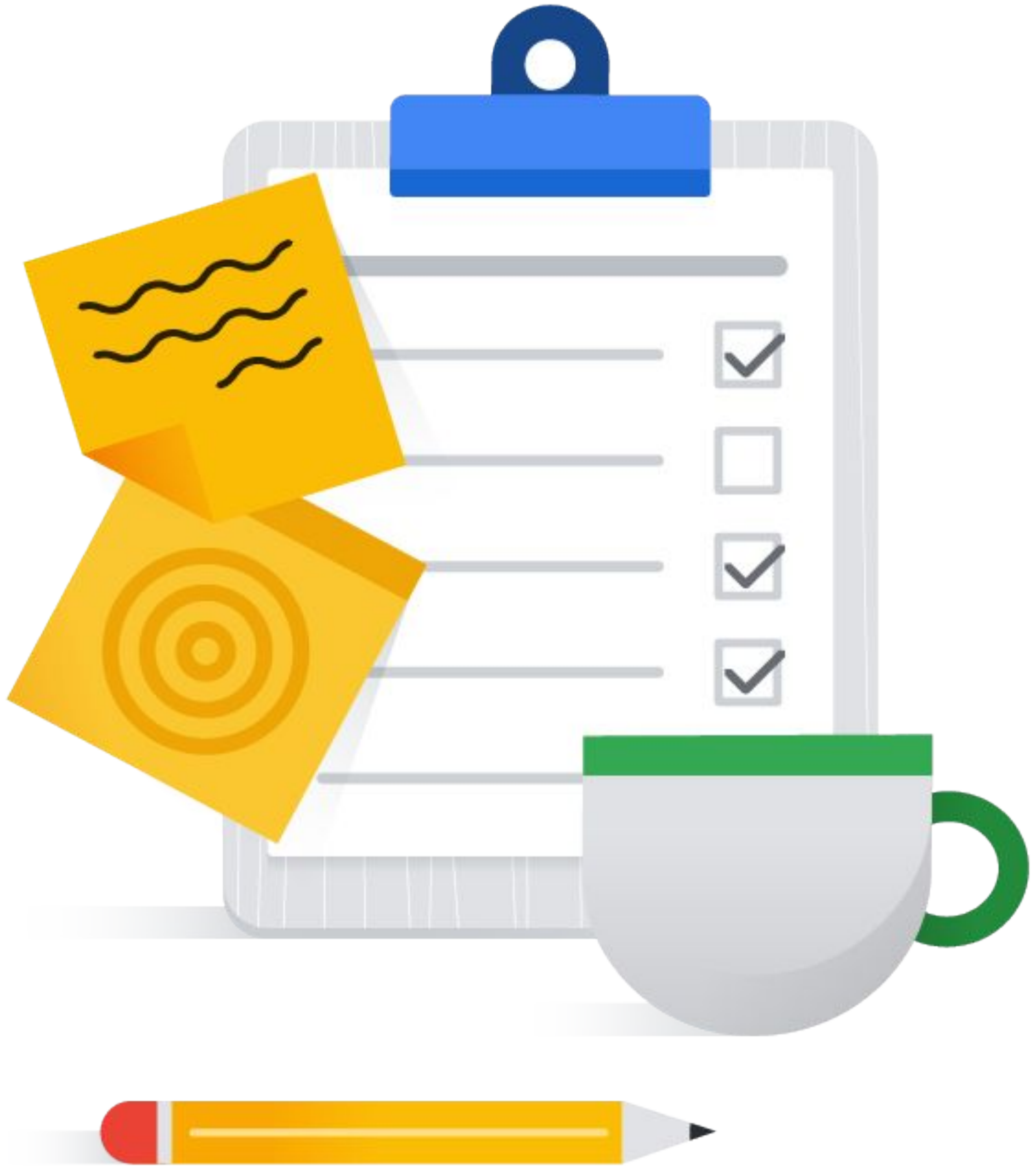
Other components include:

- Dynamic selection of exemplars related to a given query
- Caching of LLM calls and responses
- An interface to models you've deployed from Vertex AI Model Garden
- Google integrations with many document sources, tools, and vector search-enabled databases
- A large library of third-party tools designed for LLM agent function calling, like search engines designed to return text for LLMs

Topics

01 Basics of LangChain

02 LangChain Google Community Components



LangChain Google Community components include integrations with various Google services

Document loaders

- AlloyDB
- BigQuery
- Bigtable
- CloudSQL
- Storage
- and others…

Document Transformers

- Document AI
- Google Translate

Vector Stores

- AlloyDB
- BigQuery Vector Search
- Memory Store
- Spanner
- Firestore
- Vector Search
- and others…

- See: https://python.langchain.com/v0.2/docs/integrations/platforms/google/

BigQuery Loader example

```
from langchain_google_community import BigQueryLoader
query = "SELECT text FROM `bigquery-public-data.hacker_news.full` where title = "Another
AirBnB Host Horror Story" limit 1;”

loader = BigQueryLoader(query)
data = loader.load()

model = VertexAI(model_name="gemini-pro")
prompt = """
Summarize the following article in one sentance,plus a few short bullets
Article: {0}
""".format(data[0].page_content)
response = model.invoke(prompt)
```

Run a query in BigQuery

Document AI uses machine learning to parse documents

```
from langchain_core.document_loaders.blob_loaders import Blob
from langchain_google_community import DocAIParser

parser = DocAIParser(
    location="us", processor_name=PROCESSOR_NAME, gcs_output_path=GCS_OUTPUT_PATH
)

blob = Blob(
path="gs://cloud-samples-data/gen-app-builder/search/alphabet-investor-pdfs/2022Q1_alphab
et_earnings_release.pdf"
)

docs = list(parser.lazy_parse(blob))
```

Need to create a Document AI OCR Processor in your project

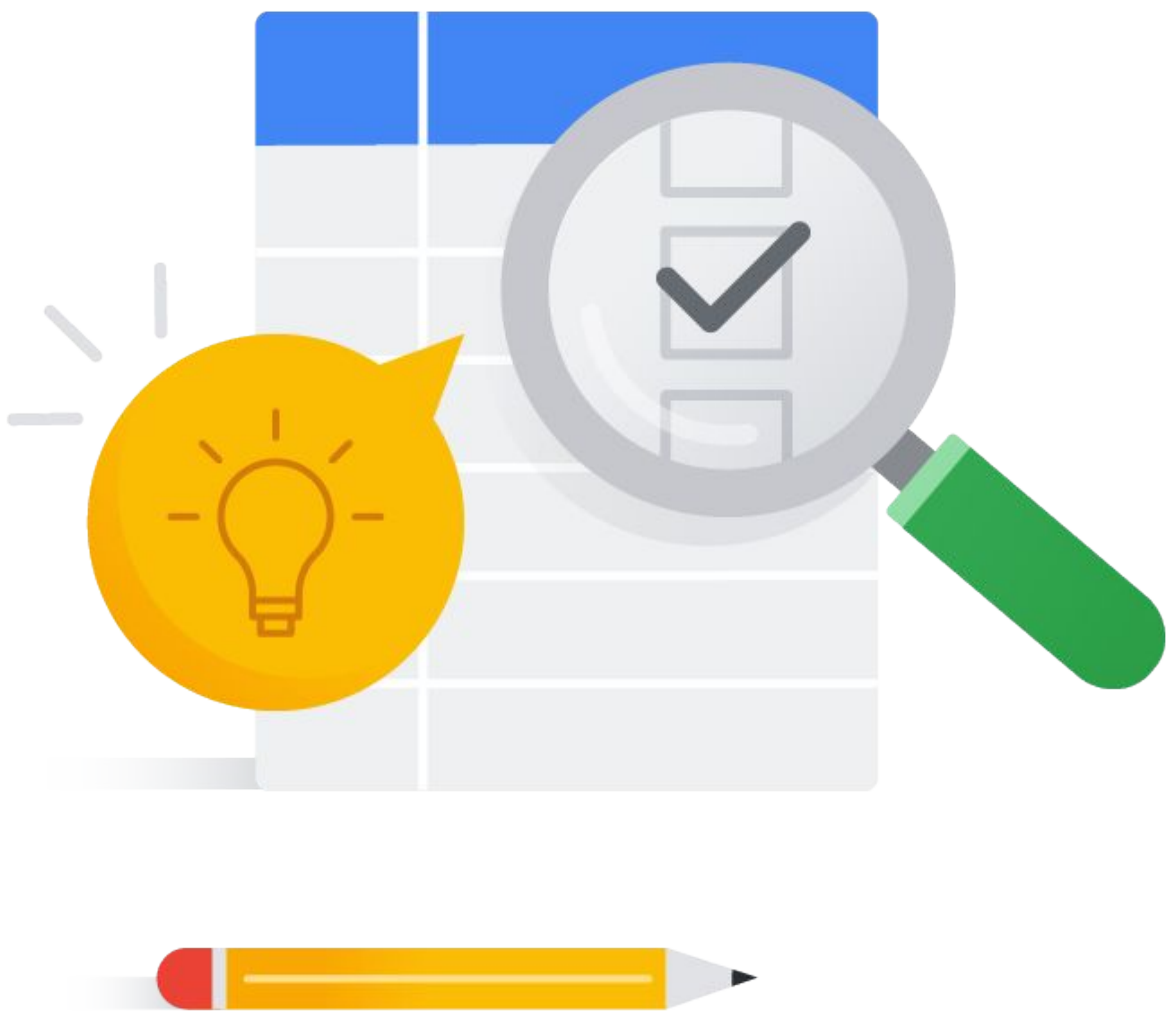
Load a PDF

Parse the PDF

Lab

1.5 hours

Getting Started with LangChain + Gemini



In this module, you learned to ...

01 Simplify your generative AI code using LangChain

02 Load text from a variety of sources using Loaders

03 Explore LangChain Google Community components

Questions and answers

Quiz question

When you retrieve information external to the LLM and use that information as part of an LLM request, it is known as what?

A: `MapReduce`

B: Stuffing

C: Retrieval-Augmented Generation (RAG)

D: Chaining

Quiz question

When you retrieve information external to the LLM and use that information as part of an LLM request, it is known as what?

A: `MapReduce`

B: Stuffing

C: Retrieval-Augmented Generation (RAG)

D: Chaining

Quiz question

You have a document that is too large to summarize in a single request. What pattern might you implement?

A: `MapReduce`

B: Stuffing

C: Retrieval-Augmented Generation (RAG)

D: Chaining

Quiz question

You have a document that is too large to summarize in a single request. What pattern might you implement?

A: `MapReduce`

B: Stuffing

C: Retrieval-Augmented Generation (RAG)

D: Chaining

Quiz question

Which of the following are features of LangChain? (Choose all that apply)

A: Support for multiple models using the same interface

B: Document loaders

C: Prompt templates

D: Output parsers

E: LangChain Expression Language

F: Memory

Quiz question

Which of the following are features of LangChain? (Choose all that apply)

A: Support for multiple models using the same interface

B: Document loaders

C: Prompt templates

D: Output parsers

E: LangChain Expression Language

F: Memory

Google Cloud