



Using LangChain with PaLM

In this module, you learn to ...

01

Simplify your GenAI code using LangChain

02

Load text from a variety of sources using Loaders

03

Explore chains to work with long or multiple documents



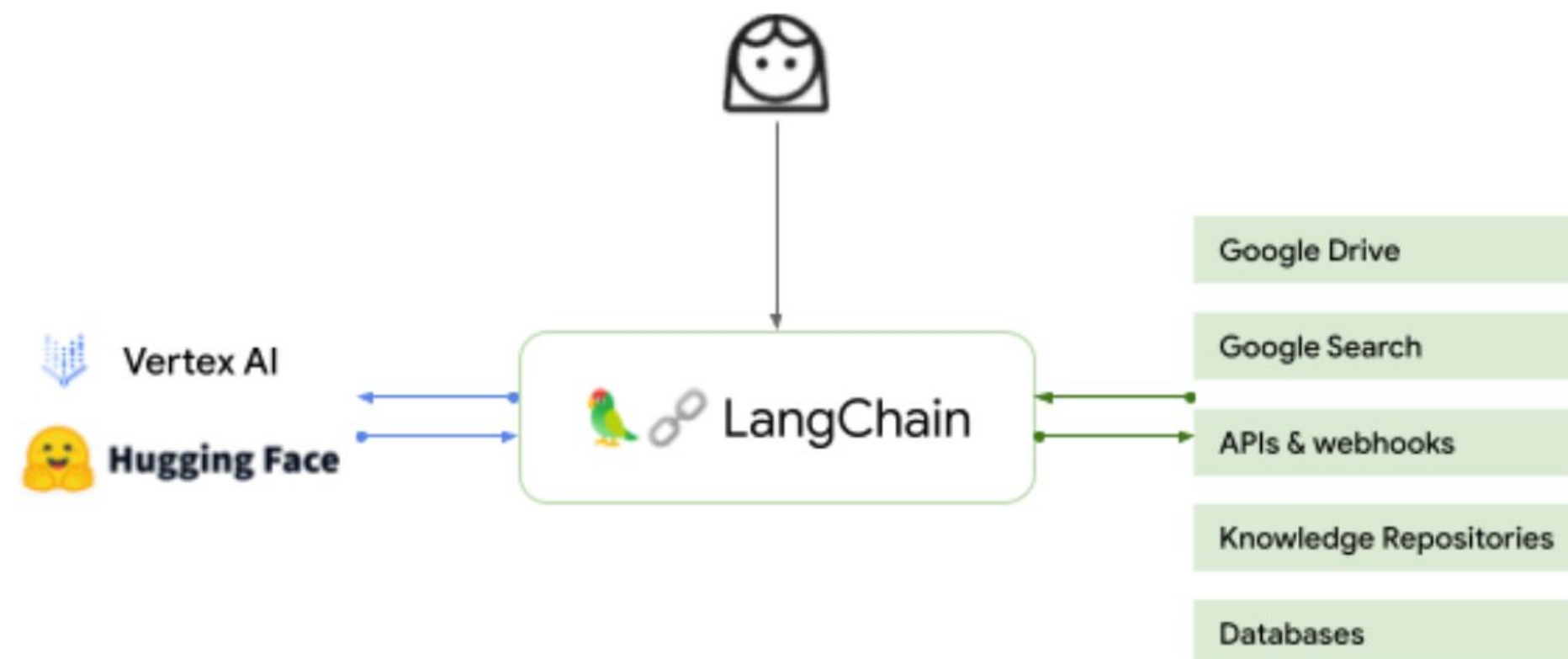
Topics

01	Getting Started with LangChain
02	LangChain Document Loaders
03	Working with Large or Multiple Documents



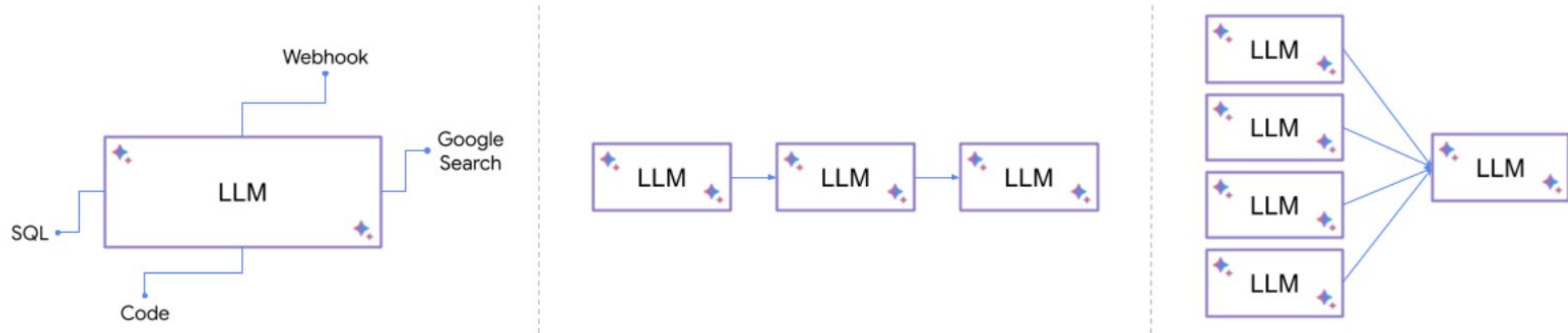
LangChain is a framework that makes developing apps with large language models easier

- Integrations bring external data, such as your files, other applications, and API data, to the LLMs
- Agents allow LLMs to interact with their environment via decision making
 - Use LLMs to help decide which action to take next
- Applications using LLMs need to be able to interact with data external to themselves
 - The web
 - Databases
 - APIs
 - ect.



There are many use cases where LLMs need to communicate with external 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

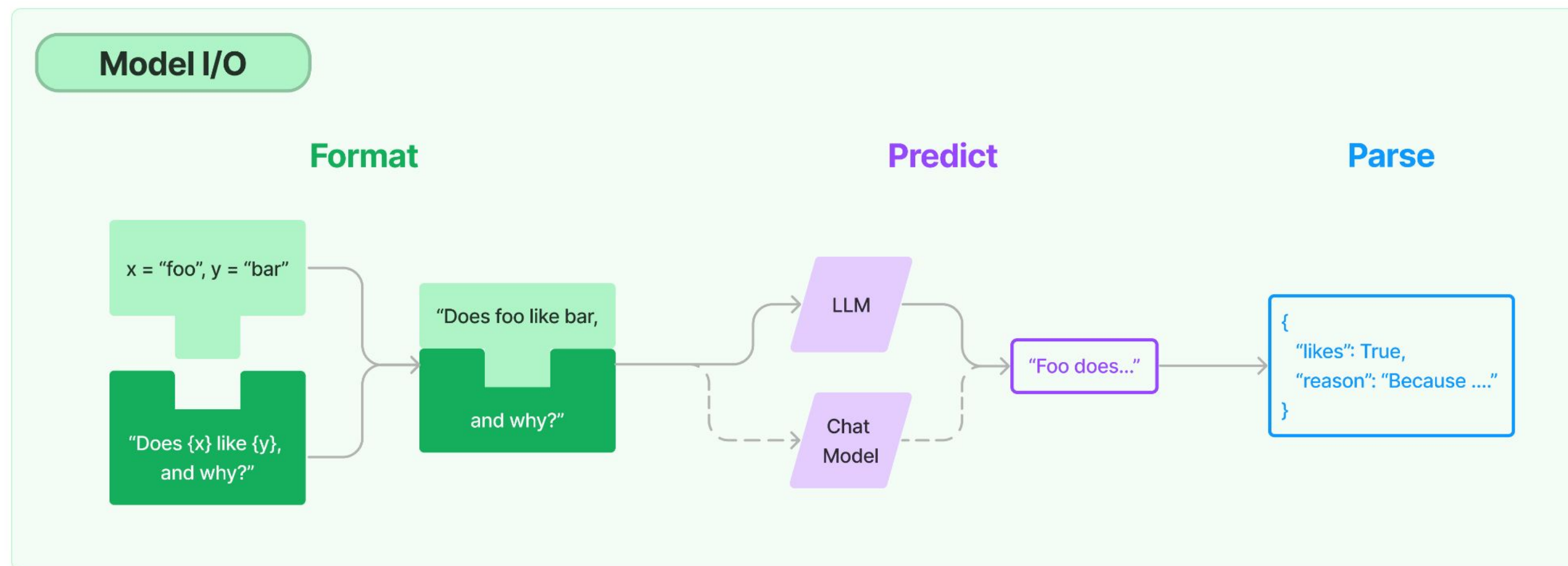


LangChain features

- Support for multiple large language models using the same interface
 - OpenAI, VertexAI, Anthropic, etc.
- Prompt templates make building complex prompts easier and consistent
- Output parser help control the format of model results
- Chains allow for easier management of complex workflows
- LangChain Expression Language (LCEL) makes programming chains more concise
- Built-in components for handling typical LLM tasks
 - Manage memory in a chat
 - Load external data
 - Managing a sequence of LLM requests
 - Many more...

Model I/O

- LangChain contains abstractions for:
 - Interacting with multiple large language models
 - Formatting prompts
 - Formatting output



Using LangChain with Vertex AI

```
! pip install google-cloud-aiplatform langchain
```

```
import vertexai
import langchain
from langchain.llms import VertexAI
from google.cloud import aiplatform

PROJECT_ID = "your-proj-id-here"
vertexai.init(project=PROJECT_ID, location="us-central1")

print(f"LangChain version: {langchain.__version__}")
print(f"Vertex AI SDK version: {aiplatform.__version__}")
```


Using the PaLM text generation model

```
llm = VertexAI(  
    model_name="text-bison@001",  
    max_output_tokens=256,  
    temperature=0.1,  
    top_p=0.8,  
    top_k=40,  
    verbose=True,  
)
```

Instantiate the LLM

```
llm("Tell me a joke")
```

Prompt the LLM

```
'What do you call a fish with no eyes? Fsh!'
```

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 thng.\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's e're not going to waste a moment.'

Custom Prompt Templates allow you to add validation when using prompts

- Example: We want to a large language model to describe what a function does
 - Below is the prompt

```
PROMPT = """\
Given the function name and source code, generate an English language explanation of the
function.
Function Name: {function_name}
Source Code:
{source_code}
Explanation:
"""
```

We will have the model
explain this function



```
def rb(arg):
    r = 0
    b = arg.bit_length()
    for i in range(b):
        r <=<= 1
        r |= (arg & 1)
        r >>= 1
    return r
```

Custom Prompt Template class

```
class FunctionExplainerPromptTemplate(StringPromptTemplate, BaseModel):
    @validator("input_variables")
    def validate_input_variables(cls, v):
        """Validate that the input variables are correct."""
        if len(v) != 1 or "function_name" not in v:
            raise ValueError("function_name must be the only input_variable.")
        return v

    def format(self, **kwargs) -> str:
        source_code = inspect.getsource(kwargs["function_name"])
        prompt = PROMPT.format(
            function_name=kwargs["function_name"].__name__, source_code=source_code
        )
        return prompt

    def _prompt_type(self):
        return "function-explainer"
```

Using the Custom Prompt Template

- Below is the function we want explained

```
fn_explainer = FunctionExplainerPromptTemplate(input_variables=["function_name"])  
prompt = fn_explainer.format(function_name=rb)  
llm(prompt)
```

```
'The function rb takes an integer argument and returns the reverse binary representation of the argument. The function works by first calculating the number of bits in the argument using the bit_length() method. It then iterates over the bits of the argument, starting with the least significant bit, and shifting the result to the left by one bit and ORing it with the current bit of the argument. The result is then shifted to the right by one bit and returned.'
```

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}, Only 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!')
```

Chains allow you to run multiple LangChain components

```
from langchain.prompts import PromptTemplate
from langchain.schema import StrOutputParser

prompt = (
    PromptTemplate.from_template("Write me a poem about {topic}"
    + ", make it rhyme"
    + "\n\nand in {language}")
)

chain = LLMChain(llm=llm, prompt=prompt, output_parser=StrOutputParser())
chain.run(topic="COBOL", language="English")
```

```
'COBOL, the language of business,\nIs used to create systems that are complex.\nIt's been around for decades,\nAnd it's still going strong.\nIt's used by businesses all over the world,\nTo keep their operations running smoothly.\n\nCOBOL is a powerful language,\nWith a rich set of features.\nIt's easy to learn,\nAnd it's very versatile.\nYou can use COBOL to create any type of business applica-
tion,\nFrom simple data entry systems to complex enterprise-wide systems.\n\nIf you're looking for a language to learn,\nCOBOL is a great option.\nIt's a stable language,\nWith a large community of users.\nAnd it's still in high demand,\nSo you'll be able to find a job easily.\n\nSo if you're ready to learn a new language,\nWhy not give COBOL a try?\nYou won't be disappointed.'
```

LangChain Expression Language (LCEL) is a more concise way of creating a chain

```
from langchain.prompts import PromptTemplate
from langchain.schema import StrOutputParser

prompt = (
    PromptTemplate.from_template("Write me a poem about {topic}"
    + ", make it rhyme"
    + "\n\nand in {language}")
)

chain = prompt | llm | StrOutputParser()
chain.invoke({"topic": "COBOL", "language": "English"})
```


This produces the same results as the prior slide

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

```
from langchain.prompts import PromptTemplate
from langchain.schema import StrOutputParser

prompt = (
    PromptTemplate.from_template("Write me a poem about {topic}"
    + ", make it rhyme"
    + "\n\nand in {language}")
)

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



The `stream()` function will output data in chunks. This would produce output the same way as Bard or ChatGPT

Using the PaLM chat model

```
from langchain.chat_models import ChatVertexAI
from langchain.schema import HumanMessage, SystemMessage, AIMessage

chat = ChatVertexAI(model_name="chat-bison@001",
    max_output_tokens=256,
    temperature=0.1,
    top_p=0.8,
    top_k=40,
    verbose=True, )

chat([HumanMessage(content="What's a good recipe for a Halloween Party?")])
```

```
AIMessage(content="Here are some recipes for a Halloween Party:\n\n* **Halloween Party Punch:** This punch is a great way to get the party started. It's made with orange juice, cranberry juice, ginger ale, and vodka.\n* **Halloween Party Cookies:** These cookies are a fun and festive treat for your guests. They're made with chocolate chips, white chocolate chips, and candy corn.\n* **Halloween Party Pizza:** This pizza is a great way to feed a crowd. It's made with a pizza dough, pizza sauce, cheese, and a variety of Halloween-themed toppings.\n* **Halloween Party Dip:** This dip is a great way to get your guests mingling. It's made with cream cheese, sour cream, shredded cheese, and a variety of Halloween-themed toppings.\n* **Halloween Party Trail Mix:** This trail mix is a great way to keep your guests energized. It's made with a variety of nuts, seeds, dried fruit, and candy.")
```

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 that is a crowd-pleaser. To make a banana split, you will need:\n\n* 2 bananas, sliced\n* 1 cup of vanilla 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 plate. Add a dollop of each of the three ice cream flavors. Next, add a dollop of whipped cream, a cherry, and a drizzle of strawberry sauce. 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 see 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

Topics

01	Getting Started with LangChain
02	LangChain Document Loaders
03	Working with Large or Multiple Documents



LangChain Document Loaders

- LLMs are trained on huge amounts of public data, but in real-world applications you will need to retrieve your own data and pass it to the model
- LangChain has a number of built-in document loaders for retrieving external data
 - CSV loader
 - Directory loader
 - HTML loader
 - JSON loader
 - Markdown loader
 - PDF loader



PDF loader 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 2018,
```


Use WebBaseLoader to get data from the internet

```
from langchain.document_loaders import WebBaseLoader

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

Once you have the data, pass it to the model for summarization

```
from langchain.document_loaders import PyPDFLoader

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

llm("Summarize the following: {0}".format(pages[10:15]))
```

```
'The first document is about the potential impact of generative AI on the arts. The author argues that generative AI raises important questions about authenticity, economic valuation, provenance, creator compensation, and copyright. The author also argues that generative AI may simply automate a highly reductive notion of both the creative process and of the learning process itself.\n\nThe second document is about the potential impact of generative AI on the rule of law. The author argues that generative AI could help individuals prepare legal documents, attorneys in legal research and writing, and judges to improve the accuracy and efficiency of painfully slow forms of adjudication. However, the author also warns that generative AI models can lie, hallucinate, and make up facts, cases, and doctrine.\n\nThe third document is about the potential impact of generative AI on the creative industry. The author argues that generative AI is a source of scientific excitement but also anxiety abo...'
```

Topics

- | | |
|----|--|
| 01 | Getting Started with LangChain |
| 02 | LangChain Document Loaders |
| 03 | Working with Large or Multiple Documents |



What if the document is too large for the model to process in one request?

```
from langchain.document_loaders import WebBaseLoader

# This is the book The Wizard of Oz
loader = WebBaseLoader("https://www.gutenberg.org/cache/epub/55/pg55.txt")
data = loader.load()

llm("Summarize the following: {0}".format(data))
```

InvalidArgument: 400 The request cannot be processed. The most likely reason is that the provided input exceeded the model's input token limit.

We need to split long text into multiple documents

```
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, and I
```

LangChain provides a variety of splitters

- [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](#)

LangChain suggests four primary strategies for working with multiple documents

- Stuff
- Map Reduce
- Map Re-rank
- Refine

Stuffing ignores the token limit, but just combines them all to be fed into the model context

- Suppose you have multiple small documents and you want to combine them and summarize them together
- Saves calls to the LLM, reducing costs and increasing speed
- Create a chain that:
 - Combines the documents
 - Sends them to the LLM for summarization
 - Formats the results
- Can more easily hit a model's token limit if you're not checking to make sure your documents are under the limit

<https://python.langchain.com/docs/modules/chains/document/stuff>

Stuffing Chain

```
doc_prompt = PromptTemplate.from_template("{page_content}")

chain = (
    {
        "content": lambda docs: "\n\n".join(
            format_document(doc, doc_prompt) for doc in docs
        )
    }
    | PromptTemplate.from_template(
        "List the key points in the following content:\n\n{content}")
    | llm
    | StrOutputParser()
)
print(chain.invoke(docs))
```

Combine a collection of documents into one. This retrieves the `page_content` property from each doc.

Map-Reduce is a strategy for summarizing documents that are too large for a single request

- Steps
 - Divide the document into pieces
 - Summarize each piece (Map)
 - Combine the summaries
 - Collapse the summaries
 - Summarize the summaries (Reduce)
- LangChain implements this with the MapReduce chain

https://python.langchain.com/docs/modules/chains/document/map_reduce

The Map chain

```
map_chain = (  
    {"context": partial_format_document}  
    | PromptTemplate.from_template(  
        "Summarize this content in fewer than 256 words:\n\n{context}")  
    | llm  
    | StrOutputParser()  
)  
  
# A wrapper chain to keep the original Document metadata  
map_as_doc_chain = (  
    RunnableParallel({"doc": RunnablePassthrough(), "content": map_chain})  
    | (lambda x: Document(page_content=x["content"], metadata=x["doc"].metadata))  
) .with_config(run_name="Summarize (return doc)")
```


The collapse chain consolidates subsets of documents

```
collapse_chain = (  
    {"context": format_docs}  
    | PromptTemplate.from_template(  
        "Summarize this content so it is smaller than 1000 words:\n\n{context}")  
    | llm  
    | StrOutputParser())  
  
def collapse(docs, config, token_max=4000, ):  
    collapse_ct = 1  
    while get_num_tokens(docs) > token_max:  
        config["run_name"] = f"Collapse {collapse_ct}"  
        invoke = partial(collapse_chain.invoke, config=config)  
        split_docs = split_list_of_docs(docs, get_num_tokens, token_max)  
        docs = [collapse_docs(_docs, invoke) for _docs in split_docs]  
        collapse_ct += 1  
    return docs
```

The Reduce chain combines the final subset summaries

```
reduce_chain = (  
    {"context": format_docs}  
    | PromptTemplate.from_template("Combine these summaries:\n\n{context}")  
    | llm  
    | StrOutputParser()  
) .with_config(run_name="Reduce")
```

The chain that combines the other chains

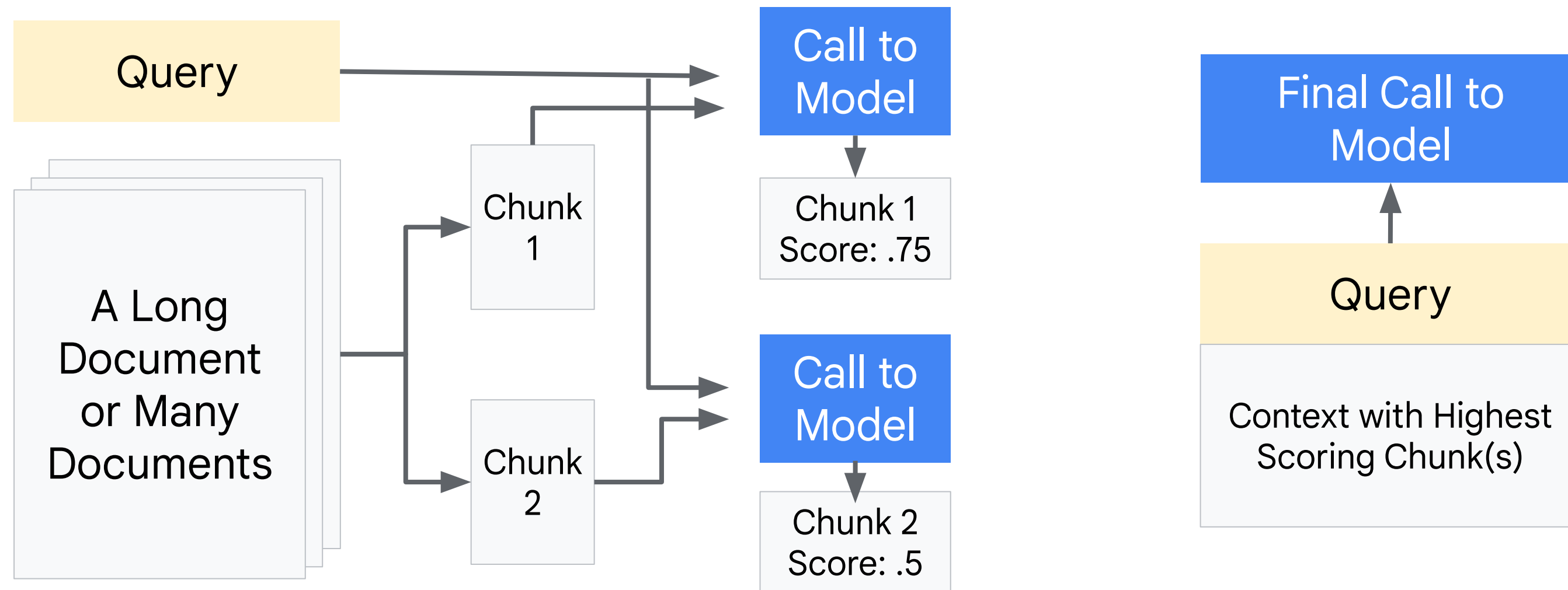
```
map_reduce = (map_as_doc_chain.map() | collapse | reduce_chain).with_config(  
    run_name="Map reduce"  
)
```

To run it, convert the splits into a collection of Document objects and then invoke the chain

```
docs = [  
    Document(  
        page_content=splits.page_content,  
        metadata={"source": "https://www.gutenberg.org/cache/epub/55/pg55.txt"},  
    )  
    for splits in html_docs  
]  
  
print(map_reduce.invoke(docs[10:12], config={"max_concurrency": 5}))
```

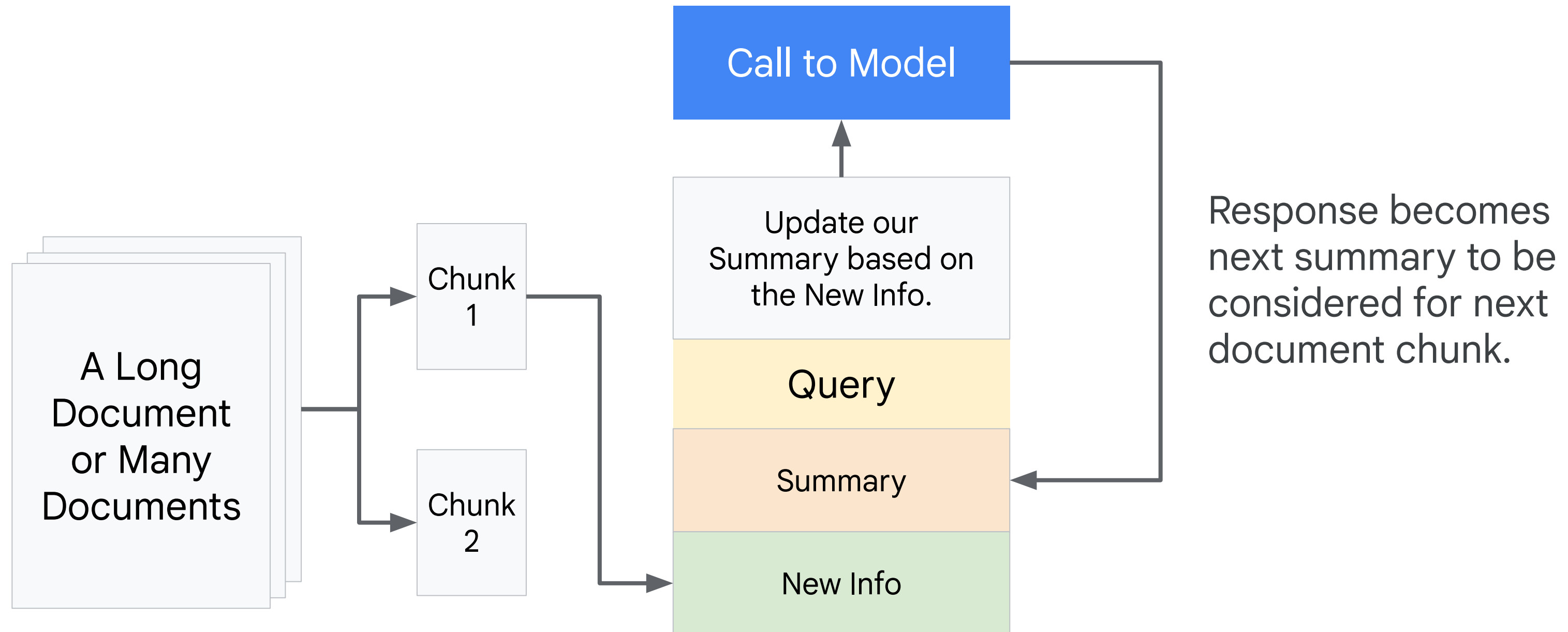
Dorothy and her companions are walking through the forest when they come across a Cowardly Lion. The Cowardly Lion is a character in the book The Wonderful Wizard of Oz. He is a large, fierce lion

Map Re-rank breaks documents into smaller chunks, has the model score each document, returning only the highest scoring document(s)



https://python.langchain.com/docs/modules/chains/document/map_rerank

Refine iterates over documents, with the option for the model to update a response at each iteration.

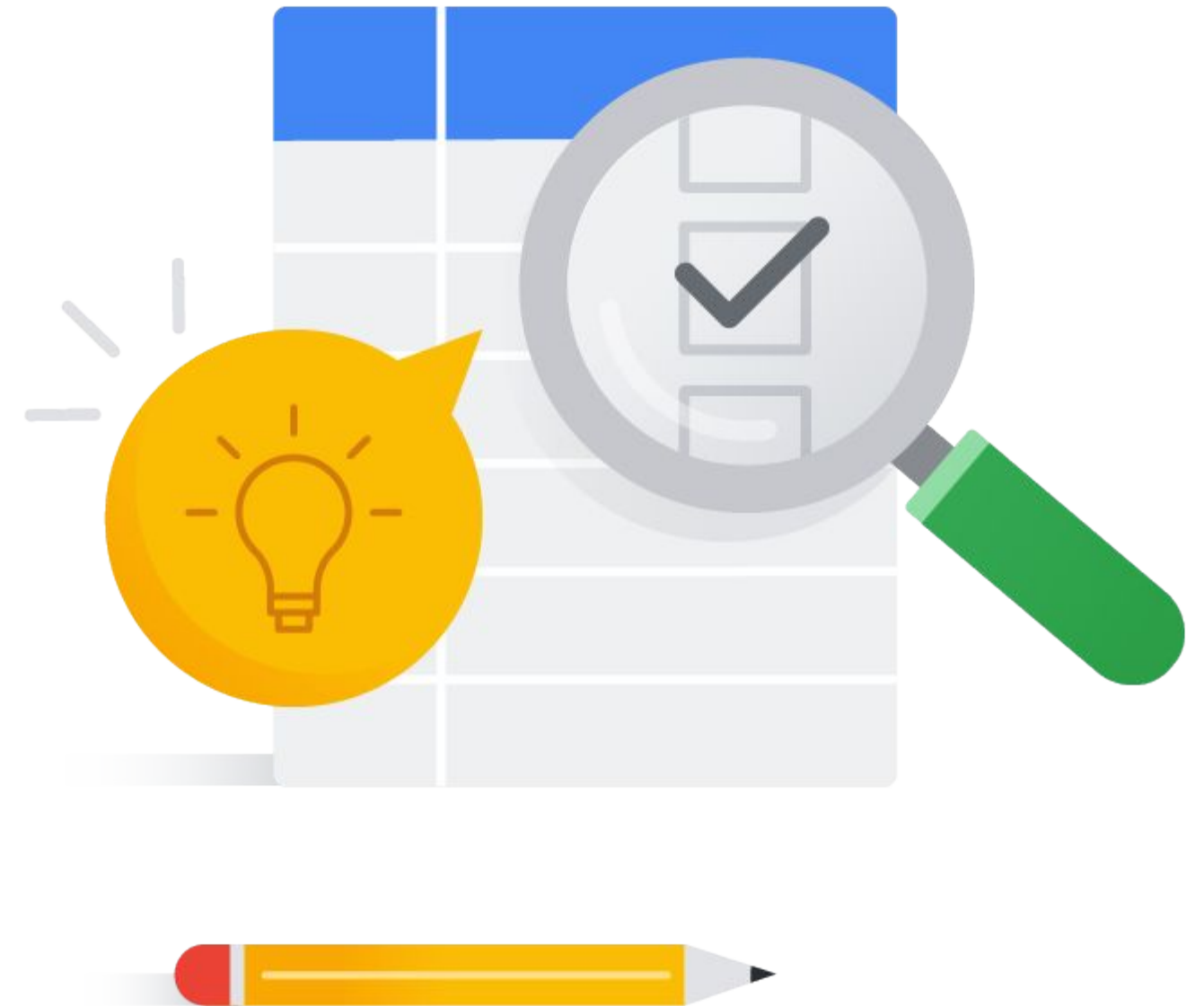


<https://python.langchain.com/docs/modules/chains/document/refine>

Lab

🕒 30 min ⚙️

Lab: Getting Started with LangChain + Vertex AI PaLM API

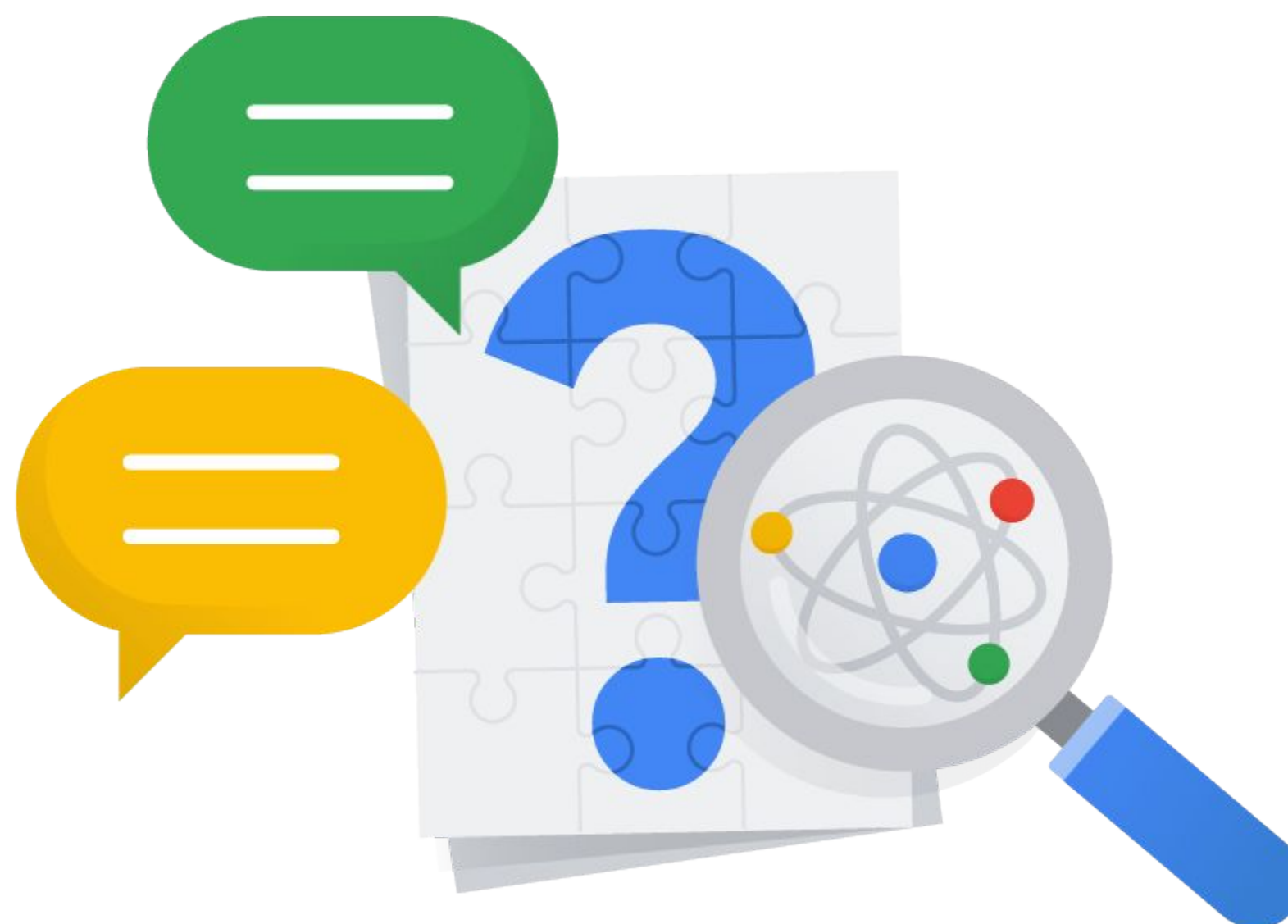


In this module, you learned to ...

- 01 Simplify your GenAI code using LangChain
- 02 Solve complex GenAI problems using LangChain chains
- 03 Explore chains to work with Documents



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