NVIDIA

The Fast Path to Developing with LLMs

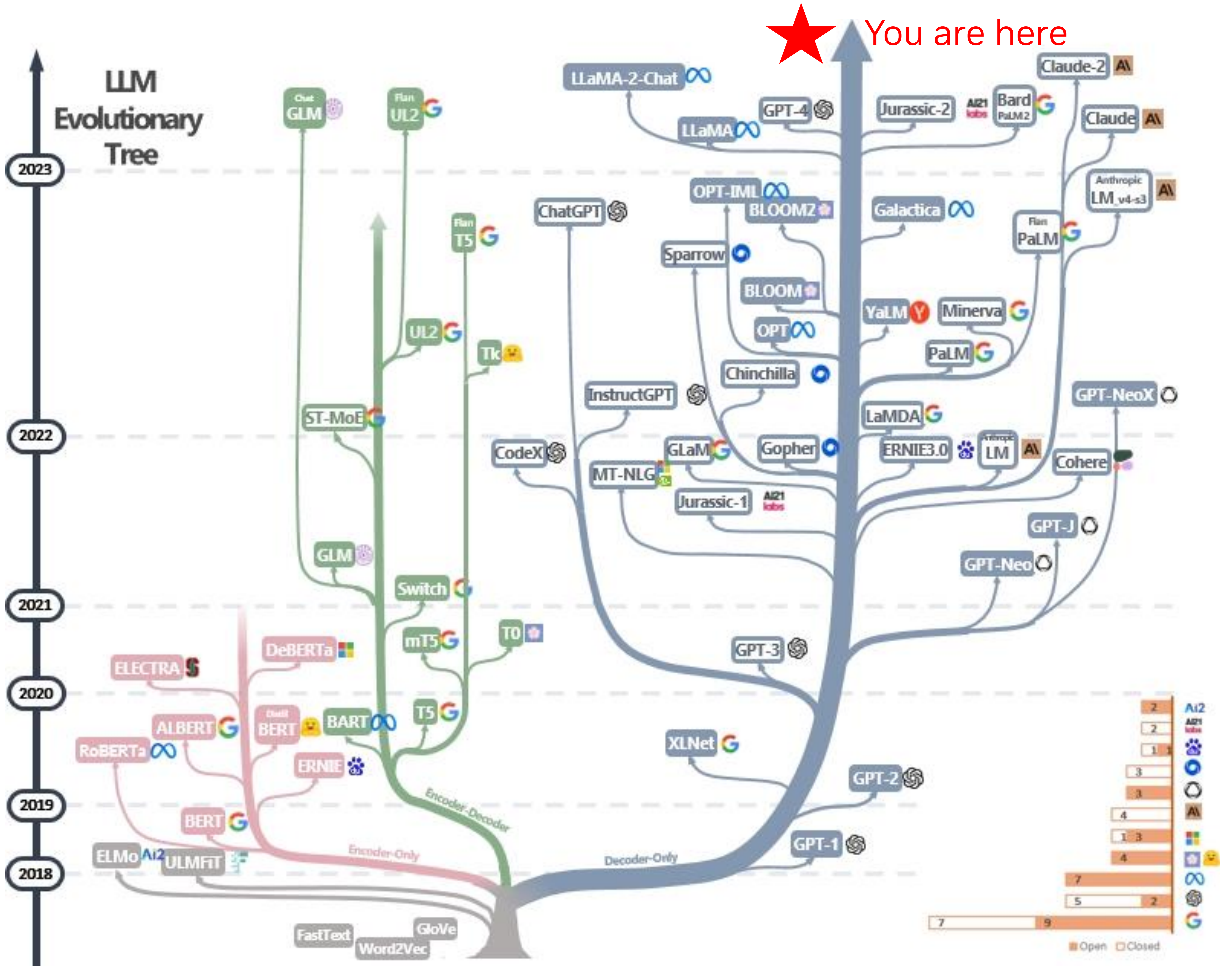
David Taubenheim, Senior Solutions Engineer | 17 November 2023

Agenda

- LLMs in Context
- Using LLM APIs
- Prompt Engineering
- Using LLM Workflow Frameworks
- Combining LLMs with Your Data

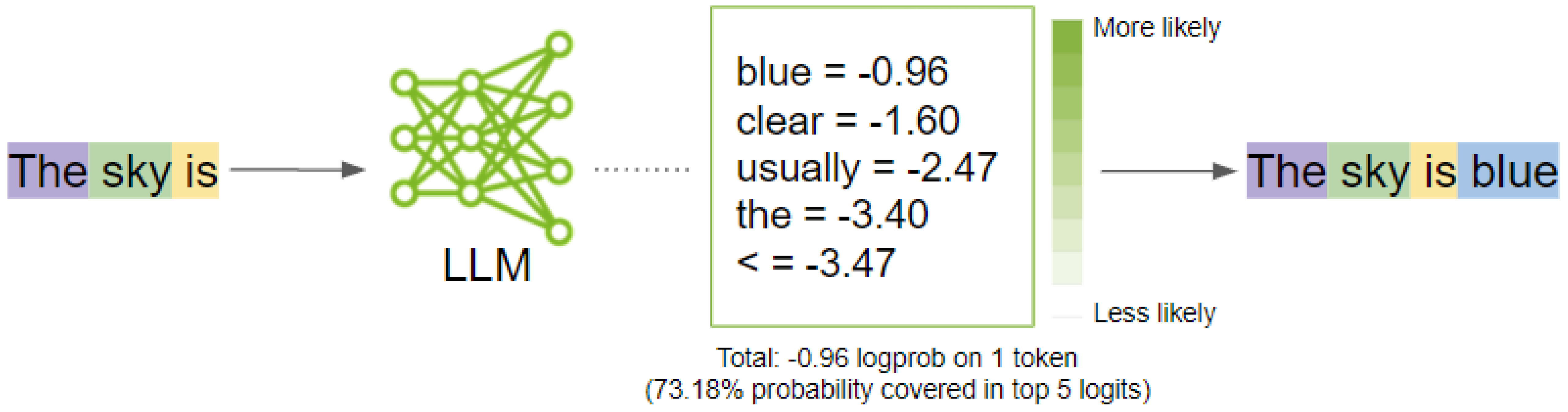
Evolution of Language Models

- Historically, language models were trained for specific tasks, including
  - Text classification
  - Entity extraction
  - Question answering
- 2017: The LLM revolution begins, powered by "transformer" models
  - A deep learning architecture family specializing in processing sequences of datapoints ("tokens")
  - Uses "self-attention" to determine which parts of a sequence help interpret which other parts
  - Introduced by Google/UToronto researchers in "Attention is All You Need" paper
- Now, much larger models trained on extraordinary quantities of data are central to Generative AI



From: ***Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond***
Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, Xia Hu
https://github.com/Mooler0410/LLMsPracticalGuide

Foundation Models and LLMs

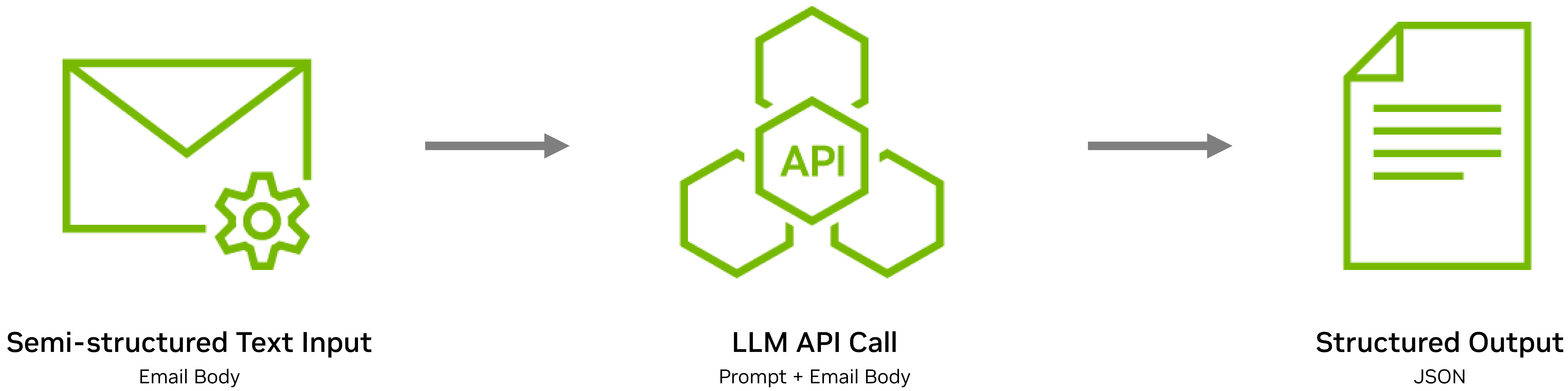


- Transformer models built with unsupervised learning proved to be effective next-token predictors
- Foundation models: Trained on massive unlabeled datasets and can be tuned to specialized applications with comparatively few examples
- Large Language Model: Scaled-up architectures that can accomplish language-related tasks like summarizing, translating, or composing new content

Example App 1: Email Triage

Demo Video:
Email Triage App

How It Works


Semi-structured Text Input
Email Body
API
LLM API Call
Prompt + Email Body
Structured Output
JSON

Using LLM APIs

Common Elements of LLM APIs

Example with OpenAI Chat-GPT

- Import package(s)

```
import os
import openai
from dotenv import load_dotenv, find_dotenv

load_dotenv(find_dotenv())
openai.api_key = os.environ['OPENAI_API_KEY’]

model="gpt-3.5-turbo"
temperature = 0.9

prompt = "Write a haiku about large language models."
messages = [{"role": "user", "content": prompt}]

response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature,
    )

print(response.choices[0].message["content"])
```

```
Endless words unfold,
Giant minds, vast text arrays,
Wisdom from the void.
```

Common Elements of LLM APIs

Example with OpenAI Chat-GPT

- Import package(s)
- Load API key

```
import os
import openai
from dotenv import load_dotenv, find_dotenv

load_dotenv(find_dotenv())
openai.api_key = os.environ['OPENAI_API_KEY’]
```

```
model="gpt-3.5-turbo"
temperature = 0.9

prompt = "Write a haiku about large language models."
messages = [{"role": "user", "content": prompt}]

response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature,
    )

print(response.choices[0].message["content"])
```

```
Endless words unfold,
Giant minds, vast text arrays,
Wisdom from the void.
```

Common Elements of LLM APIs

Example with OpenAI Chat-GPT

- Import package(s)
- Load API key
- Select model and parameters

```
import os
import openai
from dotenv import load_dotenv, find_dotenv

load_dotenv(find_dotenv())
openai.api_key = os.environ['OPENAI_API_KEY’]

model="gpt-3.5-turbo"
temperature = 0.9
```

```
prompt = "Write a haiku about large language models."
messages = [{"role": "user", "content": prompt}]

response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature,
    )

print(response.choices[0].message["content"])
```

```
Endless words unfold,
Giant minds, vast text arrays,
Wisdom from the void.
```

Common Elements of LLM APIs

Example with OpenAI Chat-GPT

- Import package(s)
- Load API key
- Select model and parameters
- Set prompt

```
import os
import openai
from dotenv import load_dotenv, find_dotenv

load_dotenv(find_dotenv())
openai.api_key = os.environ['OPENAI_API_KEY’]

model="gpt-3.5-turbo"
temperature = 0.9

prompt = "Write a haiku about large language models."
messages = [{"role": "user", "content": prompt}]

response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature,
    )

print(response.choices[0].message["content"])
```

```
Endless words unfold,
Giant minds, vast text arrays,
Wisdom from the void.
```

Common Elements of LLM APIs

Example with OpenAI Chat-GPT

- Import package(s)
- Load API key
- Select model and parameters
- Set prompt
- Call API

```
import os
import openai
from dotenv import load_dotenv, find_dotenv

load_dotenv(find_dotenv())
openai.api_key = os.environ['OPENAI_API_KEY’]

model="gpt-3.5-turbo"
temperature = 0.9

prompt = "Write a haiku about large language models."
messages = [{"role": "user", "content": prompt}]

response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature,
    )

print(response.choices[0].message["content"])
```

```
Endless words unfold,
Giant minds, vast text arrays,
Wisdom from the void.
```

Selecting a Large Language Model

Example tasks and corresponding benchmarks

Task Type	Benchmarks
Reasoning	HellaSwag, WinoGrande, PIQA
Reading comprehension/ question answering	BoolQ, TriviaQA, NaturalQuestions
Math word problems	Math, GSM8K, svamp, mathqa, algebra222
Coding	HumanEval, MBPP
Multi-task	MMLU, BBH, GLUE
Separating fact from fiction in training data	TruthfulQA
Multi-turn	MTBench, QuAC
Multilingual	XCOPA, TyDiQA-GoldP
Long context	SCROLLS

- Selection factors
  - Benchmark scores *on relevant benchmark*
  - Quality and quantity of training data
  - Human evaluation/validation
  - Inference latency
  - Cost of deployment, use, or price per tokens
  - Context size
  - License terms
- Domain specificity
  - Models tuned to specific domains can sometimes perform as well as models that are orders of magnitude larger but have only been pretrained

Benchmark Example

Hugging Face LLM Leaderboard

T	Model	ARC	HellaSwag	MMLU	TruthfulQA
●	tiiuae/falcon-180B	69.71	88.98	70.44	45.66
●	tiiuae/falcon-180B	69.8	88.95	70.54	45.67
●	meta-llama/Llama-2-70b-hf	67.32	87.33	69.83	44.92
●	huggyllama/llama-65b	63.48	86.09	63.93	43.43
●	llama-65b	63.48	86.09	63.93	43.43
●	tiiuae/falcon-40b	61.95	85.28	56.98	41.72
●	llama-30b	61.26	84.73	58.47	42.27
●	TigerResearch/tigerbot-70b-base	62.46	83.61	65.49	52.76
●	kittn/mistral-7B-v0.1-hf	60.24	83.34	64.01	42.12
●	kittn/mistral-7B-v0.1-hf	59.98	83.32	64.13	42.15
●	mistralai/Mistral-7B-v0.1	59.98	83.31	64.16	42.15
●	mosaicml/mpt-30b-chat	58.36	82.41	50.98	52

From https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

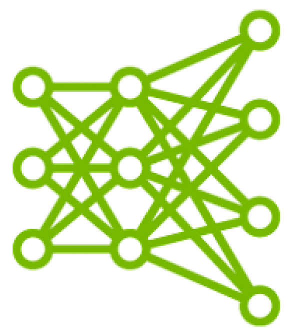
Prompt Engineering

Prompting Methodologies

Prompt design is crucial to obtaining good results from an LLM

Zero-Shot

Asking the foundation model to perform a task with no in-prompt example

Q: What is the capital of France? →  → A: 'Paris'

LLM

Lower token count
More space for context

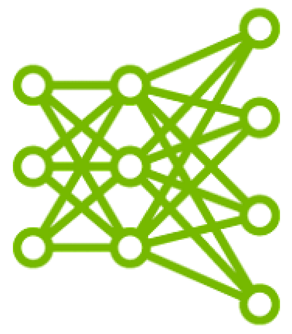
Few-Shot

Providing examples as context to the foundation model related to a task

Q: What is the capital of Spain?
A: {'answer': 'Madrid'}

Q: What is the capital of Italy?
A: {'answer': 'Rome'}

Q: What is the capital of France?

→  → A: {'answer': 'Paris'}

LLM

Better aligned responses
Higher accuracy on complex questions

Email Generation Prompt

Making synthetic test data

- Synthetic data generation (SDG) to protect privacy
  - Iterates over customer name, feedback subject, and product
  - Avoids sending actual confidential email data outside the company
- Important: Check model license
  - Commercial vs. noncommercial
  - Prohibitions on using one LLM's output to train another LLM

```
template_string = "You are a musician named {customer} who purchased a {product} \
from a musical instrument and audio equipment manufacturer named Melodious. \
Write an email to the company's customer support team to {feedback}. \
When you write emails, you get right to the point and avoid pleasantries\
like I hope this email finds you well or I hope you're having a great day. \
Start with a Subject line. Do not be overly formal or polite. Be concise."
```

```
Hello Melodious Customer Support,

I am Zhiyong, a musician who recently purchased a CG Series Grand Piano from your esteemed
company. I wanted to share some exciting news with you.

I have been invited to perform at a prestigious musical event next month, and I am delighted
to inform you that I will be showcasing the exceptional quality and sound of the CG Series
Grand Piano during my performance.

As a professional musician, it is crucial for me to have a reliable instrument that delivers
impeccable sound and performance. The CG Series Grand Piano has exceeded my expectations in
every aspect, and I am confident it will captivate the audience with its rich tonal range
and exceptional touch sensitivity.

I will make sure to mention Melodious as the manufacturer of this remarkable instrument
during my performance, as I believe it deserves recognition for its outstanding
craftsmanship.

Thank you for providing musicians like me with such high-quality instruments. I will keep
you updated on the event and share any media coverage that might arise from it.

Best regards,

Zhiyong
```

Chain of Thought Prompts

How to Engage LLMs with Reasoning

- Ask the model to take a series of intermediate steps before producing the desired result
- Performance gain can be significant compared to zero-shot prompting
- Adding “Let’s think step by step” or "Let's think about this logically" to the prompt can improve the result from some models
- Can supply specific steps if there's a consistent process

```
triage_prompt = """You are an efficient administrative assistant, \
sorting messages for customer service representatives at a musical instrument \
and audio equipment manufacturer named Melodious. You receive an email, shown \
below in tick marks, from a customer, regarding a product. \
Think logically step by step to assist the customer service representative. \

Step 1: If the customer is writing about a specific product,
determine which type from this list.

Products:
"Acoustic Pianos", \
"Digital Pianos and Keyboards", \
"Piano Accessories", \
"String Instruments", \
"Woodwind and Brass Instruments", \
"Woodwind and Brass Accessories", \
"Professional Audio Equipment".

Step 2: If message mentions a product in the list above, \
write a specific one-sentence summary of the exact issue. \

Step 3: Determine the tone of the email and provide it.

Step 4: Classify how urgently a response is warranted, using the \
following categories: "Urgent Response", "Not Urgent Response", \
or "No Response Required"

Step 5: Output your answers with the following headers: \
Customer Name, Product, Product Category, Summary, Tone, Response \
Urgency.

Use the following format:

Step 1: <step 1 reasoning>
Step 2: <step 2 reasoning>
Step 3: <step 3 reasoning>
Step 4: <step 4 reasoning>
Step 5: <step 5 reasoning>
``Email: {body}``
"""
```

Inspired by DeepLearning.AI "ChatGPT Prompt Engineering for Developers"

```python
triage_prompt = """You are an efficient administrative assistant, \
sorting messages for customer service representatives at a musical instrument \
and audio equipment manufacturer named Melodious. You receive an email, shown \
below in tick marks, from a customer, regarding a product. \
Think logically step by step to assist the customer service representative. \

Step 1: If the customer is writing about a specific product,
determine which type from this list.

Products:
"Acoustic Pianos", \
"Digital Pianos and Keyboards", \
"Piano Accessories", \
"String Instruments", \
"Woodwind and Brass Instruments", \
"Woodwind and Brass Accessories", \
"Professional Audio Equipment".

Step 2: If message mentions a product in the list above, \
write a specific one-sentence summary of the exact issue. \

Step 3: Determine the tone of the email and provide it.

Step 4: Classify how urgently a response is warranted, using the \
following categories: "Urgent Response", "Not Urgent Response", \
or "No Response Required"

Step 5: Output your answers with the following headers: \
Customer Name, Product, Product Category, Summary, Tone, Response \
Urgency.

Use the following format:

Step 1: <step 1 reasoning>
Step 2: <step 2 reasoning>
Step 3: <step 3 reasoning>
Step 4: <step 4 reasoning>
Step 5: <step 5 reasoning>
``Email: {body}``
"""
```

```
Step 1: The customer is writing about a specific product, which is the CG Series
Grand Piano.

Step 2: The exact issue mentioned in the email is the customer's satisfaction and
praise for the exceptional quality and performance of the CG Series Grand Piano.

Step 3: The tone of the email is positive and appreciative.

Step 4: A response is not urgently required as the customer is expressing
satisfaction and praise.

Step 5:
Customer Name: Zhiyong
Product: CG Series Grand Piano
Product Category: Acoustic Pianos
Summary: Customer expressing satisfaction and praise for the exceptional quality
and performance of the CG Series Grand Piano.
Tone: Positive and appreciative
Response Urgency: No Response Required
```

Designing A Prompt For Analysis

Incoming Email Analysis

- The more sophisticated ("aligned") a model is, the fewer explicit cues it typically needs
- Common prompt elements
  - **Role**: Dictate a job title along with a descriptive adjective or two
  - **Instructions**: Describe step-by-step what you want done with action verbs
  - **Context**: Bring relevant background info into the prompt
  - **Output format**: Many options
  - **Specificity**: Be exacting in what you want; no need to be too brief
- Elements to avoid
  - Vagueness
  - Unfounded assumptions
  - Overly-broad topics
  - Unnecessary brevity: Recent context window sizes are 32k or even 128k tokens (approx. a 300 page book)

```
triage_prompt = """You are an efficient administrative assistant, sorting \
    messages for customer service representatives at a musical instrument \
    and audio equipment manufacturer named Melodious. You receive an \
    email, shown below in tick marks, from a customer, regarding a product.\
    Read the email and then perform the following actions:
    (1) Determine the customer's name.
    (2) Determine which product the customer is talking about.
    (3) Classify the product into one of the following categories:
        "Acoustic Pianos", \
        "Digital Pianos and Keyboards", \
        "Piano Accessories", \
        "String Instruments", \
        "Woodwind and Brass Instruments", \
        "Woodwind and Brass Accessories", \
        "Professional Audio Equipment".
    (4) Write a specific one-sentence summary of the exact issue, without \
         using the name of the product.
    (5) Determine the tone of the email and provide it.
    (6) Classify how urgently a response is warranted, using the following \
        categories: "Urgent Response", "Not Urgent Response", and "No \
        Response Required"
    (7) Organize your answers into a JSON object with the following keys:
    Customer Name, Product, Product Category, Summary, Tone, Response Urgency.
    ``Email: {body}``"""
```

Output Formatting

Pulling answers out of a response

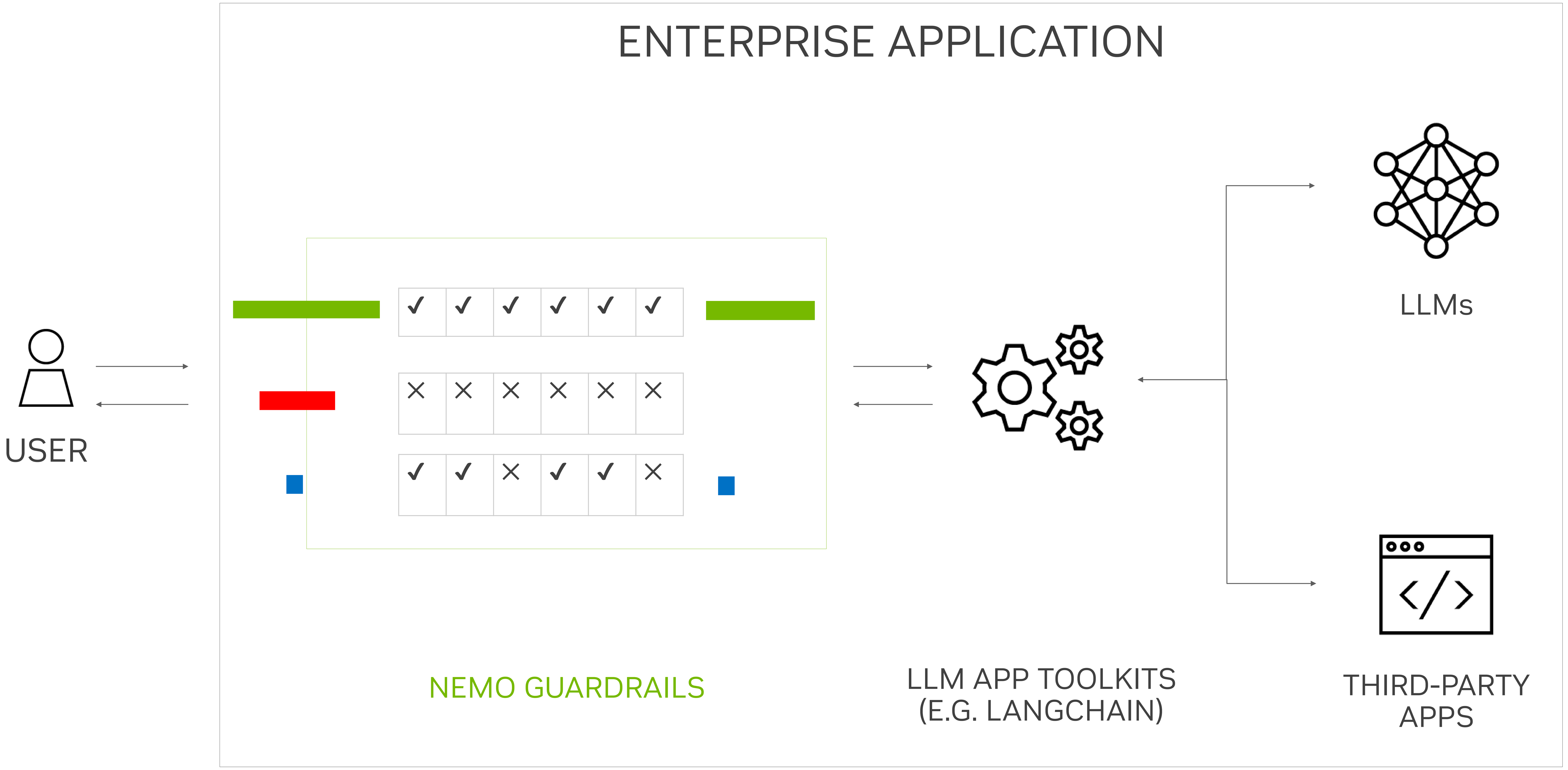
- Your prompt can specify the output format
  - JSON
  - CSV
  - HTML
  - Markdown
  - Lists
  - Tables
  - YAML
  - Code
  - … list is always growing
- Output received via API will typically be a string and require a conversion step for structured formats
  - But some APIs now ensure JSON object output
- Even high-end LLMs can produce imperfect formats — tuning can help, but also need error-checking

```
(7) Organize your answers into a JSON object with the following keys:
    Customer Name, Product, Product Category, Summary, Tone, Response Urgency.

{
  "Customer Name": "Zhiyong",
  "Product": "CG Series Grand Piano",
  "Product Category": "Acoustic Pianos",
  "Summary": "Positive feedback and praise for the CG Series Grand Piano",
  "Tone": "Positive",
  "Response Urgency": "No Response Required"
}
```

Preventing Undesirable LLM Behavior: Toxicity Checks & Guardrails

Add Boundaries To Ensure LLM Systems Operate According to Use Cases




```
#COLANG pattern
   define user XXX1
     "Entry prompt about XXX1-
prompt"
   define bot XXX1
     "Answer for topic XXX1"
   define flow XXX1
     Step 1
     Step 2
     Step 3
     ...
```

TOPICAL

Focus interactions within a specific domain

SAFETY

Prevent hallucinations, toxic or misinformative content

SECURITY

Prevent executing malicious calls and handing power to a 3rd party app

Evaluation Type Depends on Data

Structured Data Generation

Examples: QA, metadata generation, entity extraction
Known right answers

Generate test dataset: inputs and outputs → Run LLM on inputs → Compare LLM outputs to test outputs → Score

Unstructured Data Generation

Examples: Text generation, autocompletion, summarization
Many possible “good” answers

Create test dataset: inputs → Run LLM on inputs → Human or AI applies rubric or A/B testing → Score

Example App 2: Research Summarization

Demo Video:
Issue Research

How It Works

Data Preparation

Document input → Document processing → Conversion to vectors → Vector storage

Live Interaction

Prompt → Document retrieval → LLM API Call → Text output

Workflow Frameworks

Simplifying Development

Modularity and Flexibility

Simple Standalone Use

LLM

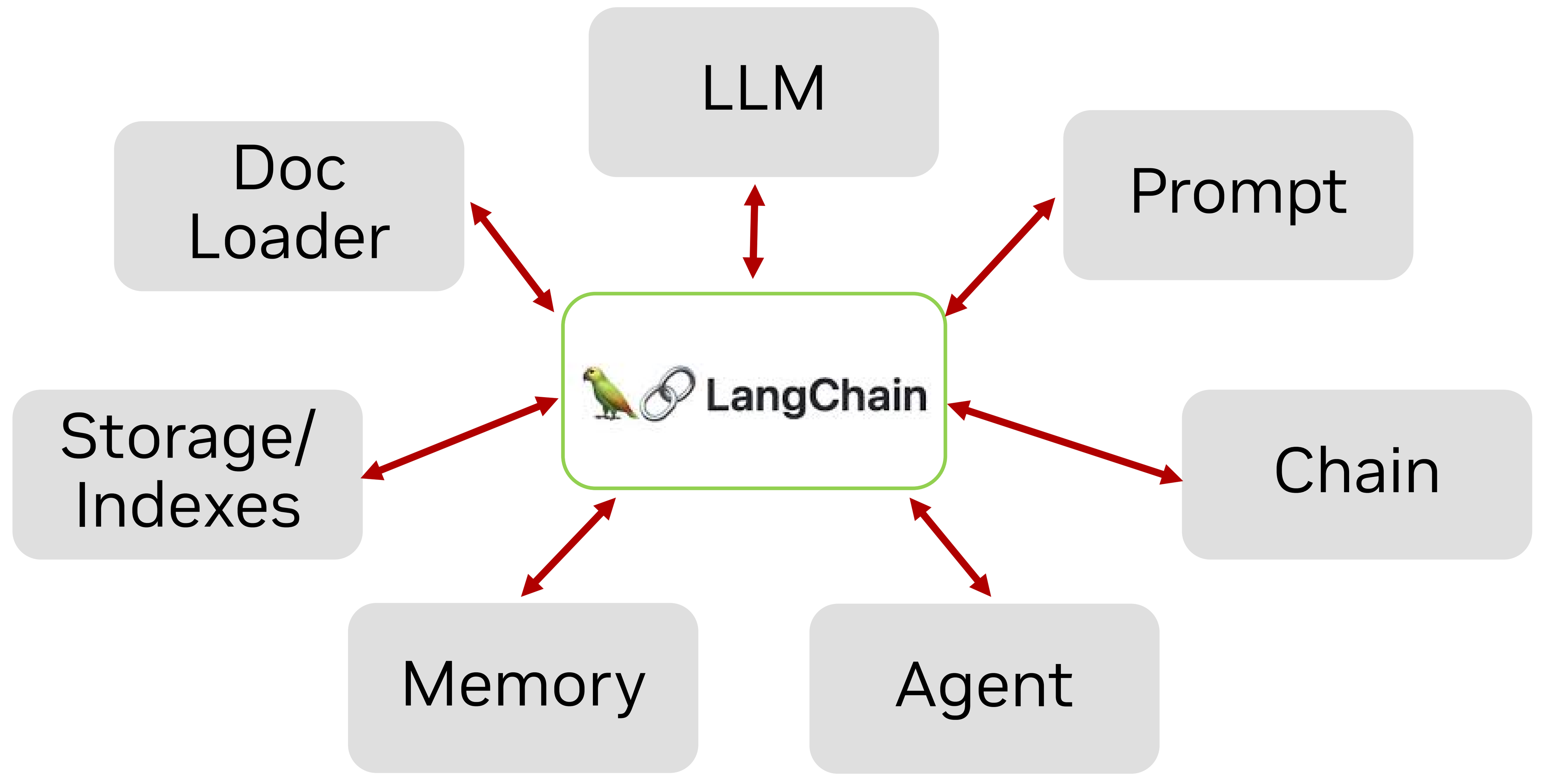
```
from langchain.schema import SystemMessage, HumanMessage
from langchain.chat_models import ChatOpenAI

# swappable LLM
LLM = ChatOpenAI(openai_api_key=openai_api_key, model_name='gpt-3.5-turbo')
```

Prompt

```
# injecting the parameters into standardized chat messages
def write_poem(topic, language, llm=LLM):
    chat_messages = [
        SystemMessage(content=f'You are a poet who composes beautiful poems in 
{language}.'),
        HumanMessage(content=f'Please write a four-line rhyming poem about {topic}.')
    ]
    return(llm(chat_messages).content)
```

Building Components for Complex Graph-Like Chains



Simplifying Development

Modularity and Flexibility

Simple Standalone Use

LLM

```
from langchain.schema import SystemMessage, HumanMessage
from langchain.chat_models import ChatOpenAI

# swappable LLM
LLM = ChatOpenAI(openai_api_key=openai_api_key, model_name='gpt-3.5-turbo')
```

Prompt

```
# injecting the parameters into standardized chat messages
def write_poem(topic, language, llm=LLM):
    chat_messages = [
        SystemMessage(content=f'You are a poet who composes beautiful poems in
{language}.'),
        HumanMessage(content=f'Please write a four-line rhyming poem about {topic}.')
    ]
    return(llm(chat_messages).content)
```

Building Components for Complex Graph-Like Chains

LLM

```
from langchain.prompts.chat import ChatPromptTemplate,SystemMessagePromptTemplate,
HumanMessagePromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain

# swappable LLM
LLM = ChatOpenAI(openai_api_key=openai_api_key, model_name='gpt-3.5-turbo')
```

Prompt

```
# standardized chat messages
system_template = 'You are a poet who composes beautiful poems in {language}.'
system_prompt = SystemMessagePromptTemplate.from_template(system_template)


human_template = 'Please write a five-line rhyming poem about {topic}.'
human_prompt = HumanMessagePromptTemplate.from_template(human_template)


full_prompt = ChatPromptTemplate.from_messages([system_prompt, human_prompt])
```

Chain

```
# connectable chain
chain = LLMChain(llm=LLM, prompt=full_prompt)

# flowing the parameters into the chain so they can be used by potentially
multiple prompts
def run_chain(topic, language):
    return(chain.run(topic=topic, language=language))
```

Examples of Frameworks

Unitary Call

```python
# LLM Object init

llm = ChatOpenAI(openai_api_key=os.environ["OPENAI_API_KEY"],
    model_name='gpt-3.5-turbo')


# Simple request example
def write_poem(topic, language, llm=llm):
    chat_messages = [
        SystemMessage(content='You are a poet who composes beautiful poems in '\
            f'{language}.'),
        HumanMessage(content=f'Please write a four line rhyming poem about {topic}.')
    ]

    return llm(chat_messages).content
```

```python
# LLM Object init + Node creation

llm = PromptModel(model_name_or_path="gpt-3.5-turbo", api_key=OPENAI_API_KEY)
prompt_node_llm = PromptNode(llm)

# Simple request example
def write_poem(topic, language, llm=prompt_node_llm):
    agent_behavior_desc = 'You are a poet who composes beautiful poems in '\
        f'{language}.'
    agent_prompt = agent_behavior_desc + """ Please write a four line rhyming poem
about {query}."""

    conversational_agent = ConversationalAgent(
        prompt_node=llm,
        prompt_template=agent_prompt,)

    return conversational_agent.run(query=topic))
```

```python
# LLM Object init - default is "text-davinci-003"

llm = Agent(prompt_driver=OpenAiChatPromptDriver(
        api_key=os.environ["OPENAI_API_KEY"], model="gpt-3.5-turbo")

# Simple request example
def write_poem(topic, language, llm=llm):
    rule_description = f'You are a poet who composes beautiful poems in {language}'
    llm.add_task(
        PromptTask("Please write a four lines rhyming poem about : '{{ args[0] }}'"\
            " and start with 'Voici mon poeme:'",
            rules=[Rule(value=rule_description)]))

    return llm.run(topic)
```

Chain or Pipeline Call

```python
# CHAINING

# standardized chat messages
system_template = 'You are a poet who composes beautiful poems in {language}.'
system_prompt = SystemMessagePromptTemplate.from_template(system_template)


human_template = 'Please write a four line rhyming poem about {topic}.'
human_prompt = HumanMessagePromptTemplate.from_template(human_template)


full_prompt = ChatPromptTemplate.from_messages([system_prompt, human_prompt])

# connectable chain
chain = LLMChain(llm=llm, prompt=full_prompt)


# flowing the parameters into the chain so they can be
# used by potentially multiple prompts
def run_chain(topic, language):
    return chain.run(topic=topic, language=language)
```

```python
# PIPELINE with multiple models
def translate_fr_poem(topic, other_lang):
    prompt_node_poem_prompt = PromptTemplate(
        prompt = "Write a four line poem on the topic of {query} in French")
    prompt_node_poem = PromptNode(LLM,
        default_prompt_template=prompt_node_poem_prompt)

    pipeline_poem = Pipeline()
    pipeline_poem.add_node(component=prompt_node_poem, name="poem", inputs=["query"])
    poem_fr = pipeline_poem.run(query=topic)

    if other_lang == "en":
        translator = TransformersTranslator(
            model_name_or_path="Helsinki-NLP/opus-mt-fr-en")
        document_poem = poem_fr["results"]
        res = translator.translate(documents=document_poem, query=None)
        return res
    else:
        return poem_fr["results"]

translate_fr_poem("growing mushrooms under the shining moon", "en")
```

```python
# PIPELINE with multiple tasks, predefined, with rules, loader
def create_fr_poem_on_topic(info_to_know, demand):
    artifacts_onx = PdfLoader().load(info_to_know)

    task1 = TextSummaryTask(artifacts_onx[0].value)
    task2 = PromptTask("Follow the query '{{ demand }}' using it as context and "\
        "support the summary {{parent_output}}",
        rules=[
                  Rule("You are a poet who composes beautiful poems."),
                  Rule("Born in France, you speak French."),
                  Rule("Sometimes you like to add a pun to your poem."),
            ])

    pipeline = Pipeline()
    pipeline.add_task(task1)
    pipeline.add_task(task2)
    pipeline.run()


create_fr_poem_on_topic("pdf/Mushrooms.pdf", "Write me a poem on a topic you know")
```

LangChain
Open Source
Large user community
Extensive out-of-the-box integrations
Enterprise: LangSmith, LangChain Hub

Haystack
Open source by DeepSet
Designed for scaled search/retrieval
Evaluation pipelines for system eval
Deployable as REST API

Griptape
Open source or managed
Commercial support
Optimized for scalability and cloud
Encryption, access control, security

Simple Local Vector DB

LangChain components

Document input

```
# Data loading
text_loader = WebBaseLoader("https://en.wikipedia.org/wiki/Poetry")
pages = text_loader.load()
```

Document Processing

```
# Chunking
text_splitter = RecursiveCharacterTextSplitter(chunk_size = 300, chunk_overlap = 50)
chunks = text_splitter.split_text(pages[0].page_content)
```

Conversion to Vectors

Storage

```
# Embedding (with an LLM-based embedding model, in this case)
embedding_model = OpenAIEmbeddings(openai_api_key=api_key)
vector_db = FAISS.from_texts(chunks, embedding=OpenAIEmbeddings())
```

Retrieval

```
# Converting the vectorstore to a retriever
retriever = vector_db.as_retriever()
context_docs = retriever.get_relevant_documents("can you help on defining the big picture
on the tetrameter metric")
```

Combining LLMs with Your Data

Retrieval Augmented Generation (RAG)

Motivation

- Decouples an LLM from only being able to act on original training data
- Obviates the need to retrain the LLM with the latest data
- LLMs limited by context window sizes

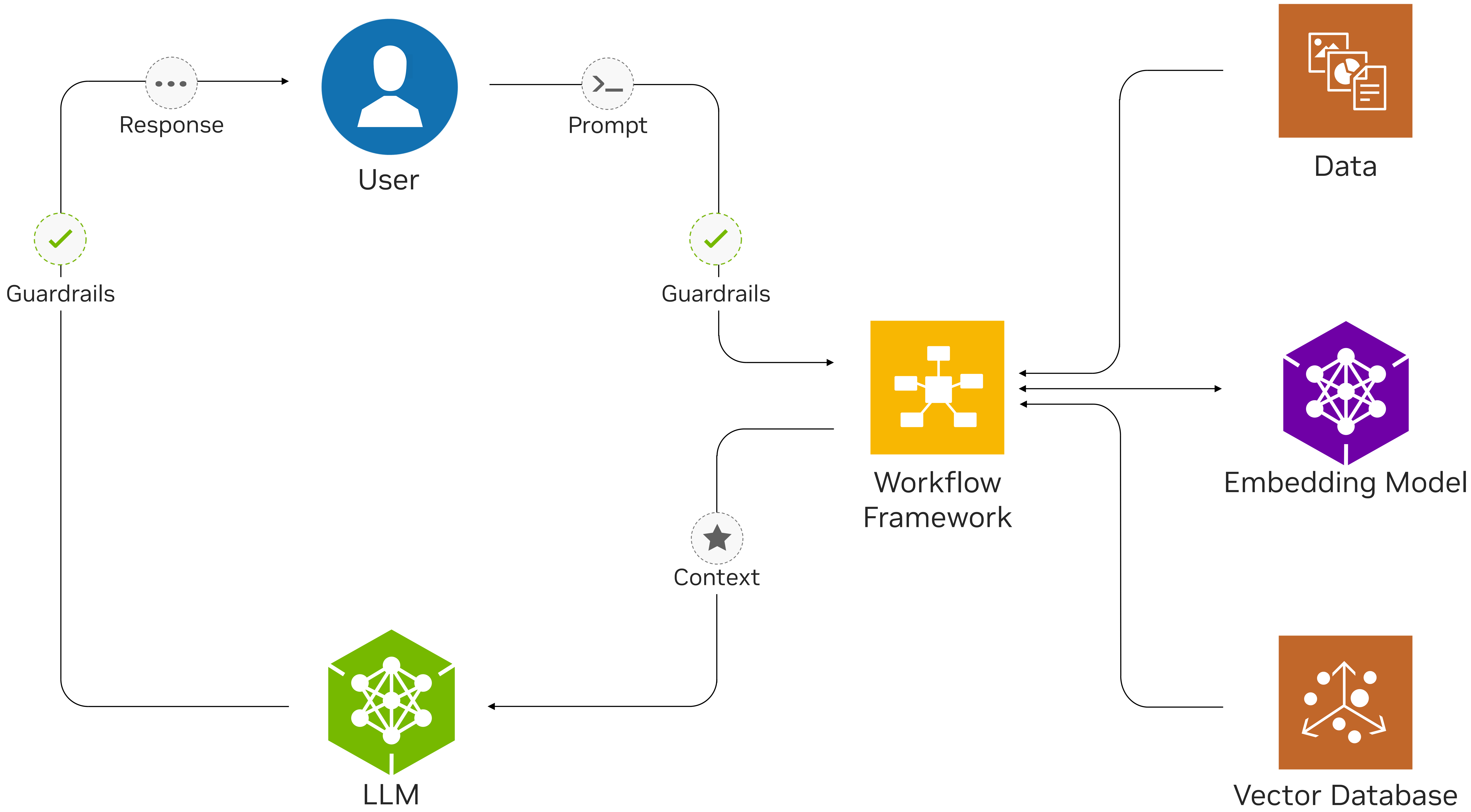
Concept

- Connect LLM to data sources at inference time
  - ex. Databases, Web, Documents, 3rd Party APIs, etc.
- Find relevant data
- Inject relevant data into the prompt

Components of the Email Assistant Application

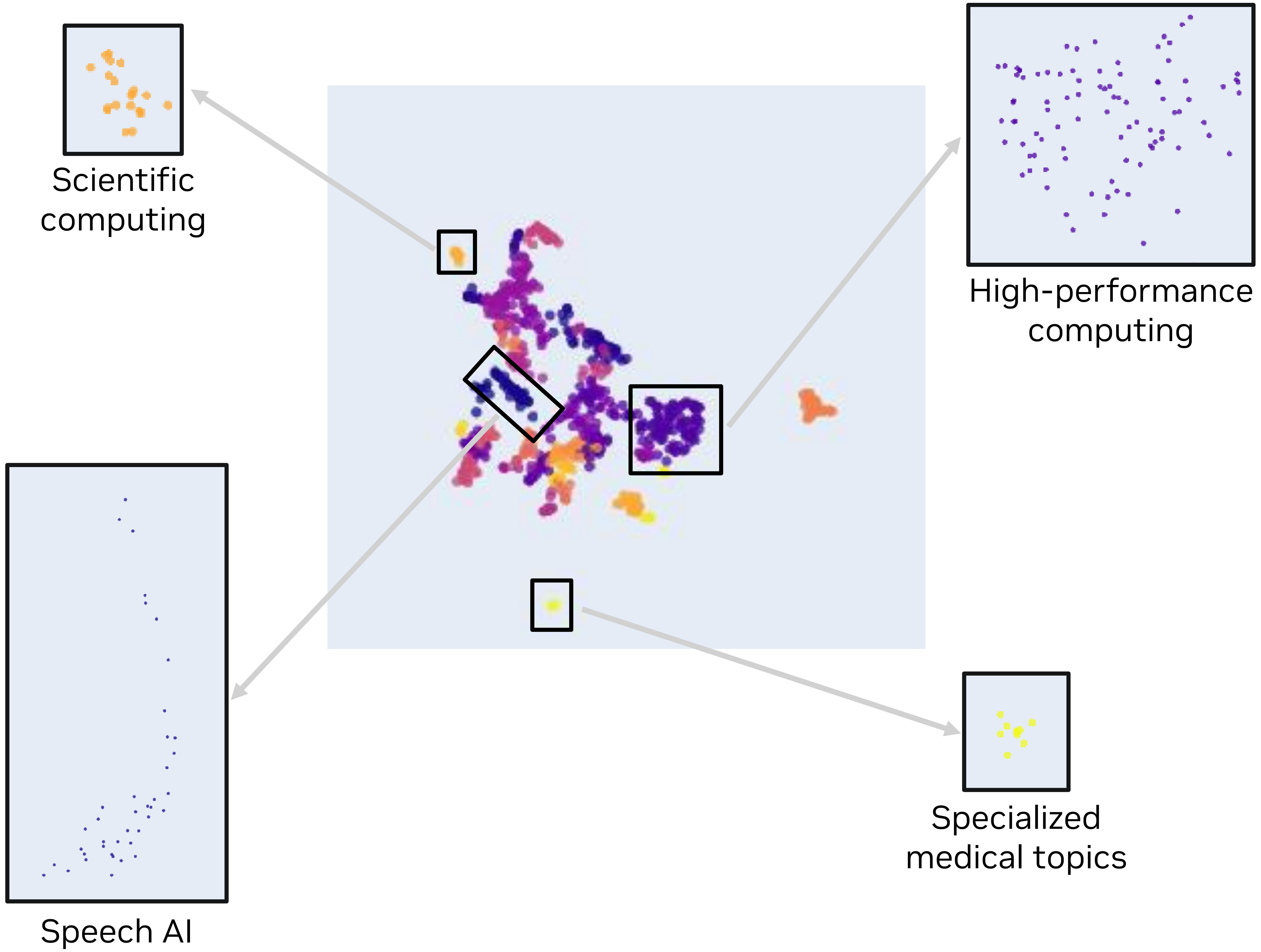
1. Human input (prompt)
2. Vectorization (embedding)
3. Retrieve vectors and calculate distance
4. Extract closest matching docs
5. Inject relevant docs into the prompt
6. Output becomes up-to-date, more accurate, with ability to cite source

Canonical RAG Workflow



Embeddings and the Vector Database

Searching via semantic similarity



2D representation of a 768-dimension embedding space

- Embeddings are data (text, image, or other data) represented as numerical vectors
  - Input text -> embedding model -> output vector
- Part of **semantic search**
  - Model trained to embed similar inputs close together
- Useful for: classification, clustering, topic discovery
- Many pretrained and trainable embedding model sources
  - Modern ones are often deep neural networks

Query: Who will lead the construction team?
Chunk 1: The construction team found lead in the paint.
Chunk 2: Ozzy has been picked to lead the group.

Chunk 1 shares more keywords with the query, but semantic search can differentiate the meanings of "lead" and understand that "team" and "group" are similar, so Chunk 2 may be more helpful for the query.

Stage 1: Data Preparation

Loading and Chunking Data

```
## loading with PyPDFLoader (or UnstructuredMarkdownLoader)

loaded_pdfdoc = PyPDFLoader("pdf/llm-ebook-part1.pdf")
pdf_pages = loaded_pdfdoc.load()


#first page that contains the metadata
#each page is a 'Document', containing both text and metadata
page0 = pdf_pages[0]

#pdf first page content and metadata
page0.page_content
>> 'A Beginner’s Guide to \nLarge Language Models\n
Part 1\nContributors:\nAnna...

Page0.metadata
>> {'source': 'pdf/llm-ebook-part1.pdf', 'page': 0}
```

```
## chunking

from langchain.text_splitter import
RecursiveCharacterTextSplitter

text_r_chunking = RecursiveCharacterTextSplitter(
    # separator list - depending on the type of document
    separators=["\n\n", "\n", "!"],
    chunk_size=500,
    chunk_overlap=80,
    length_function=len
)

chunks = text_r_chunking.split_documents(pdf_pages)

chunks
>>[Document(page_content='A Beginner’s Guide to \nLarge
Language Models\nPart 1\nContributors:\nAnnamalai
Chockalingam\nAnk.....
ur Patel\nShashank Verma\nTiffany Yeung', metadata={'source':
'pdf/llm-ebook-part1.pdf', 'page': 0}),
Document(page_content='A Beginner’s Guide to Large Language
Models 2 Table of Contents Preface .....
```

STORAGE	DOC LOADERS	DOC CHUNKING
Document loading, splitting/chunking, storing	Load from file formats: PDF, JSON… Return list of document objects	Manage context window size limitation Improve relevance of content

Stage 2: Chunking

Challenges and Considerations

```
# 20 page PDF file
loaded_pdfdoc = PyPDFLoader("pdf/llm-ebook-part1.pdf")
pdf_pages = loaded_pdfdoc.load()


# initial splitter
text_r_chunking = RecursiveCharacterTextSplitter(
    # separator list - depending on the type of document
    separators=["\n\n", "\n", "!"],
    chunk_size=500,
    chunk_overlap=80,
    length_function=len
)

docs_pdf = text_r_chunking.split_documents(pdf_pages)
docs_pdf[0]
>>Document(page_content='A Beginner’s Guide to \nLarge
Language Models\nPart 1\nContributors:\nAnnamalai
Chockalingam\nAnkur Patel\nShashank Verma\nTiffany Yeung',
metadata={'source': 'pdf/llm-ebook-part1.pdf', 'page': 0})
```

```
# second version of the splitter, smaller chunk size
text_r_chunking_bis = RecursiveCharacterTextSplitter(
    # separator list - depending on the type of document
    separators=["\n\n", "\n", "!"],
    chunk_size=30,
    chunk_overlap=10,
    length_function=len


docs_pdf_bis =
text_r_chunking_bis.split_documents(pdf_pages)
docs_pdf_bis[0]
>>Document(page_content='A Beginner’s Guide to ',
metadata={'source': 'pdf/llm-ebook-part1.pdf', 'page': 0})
...
```

SPLITTING METHODS

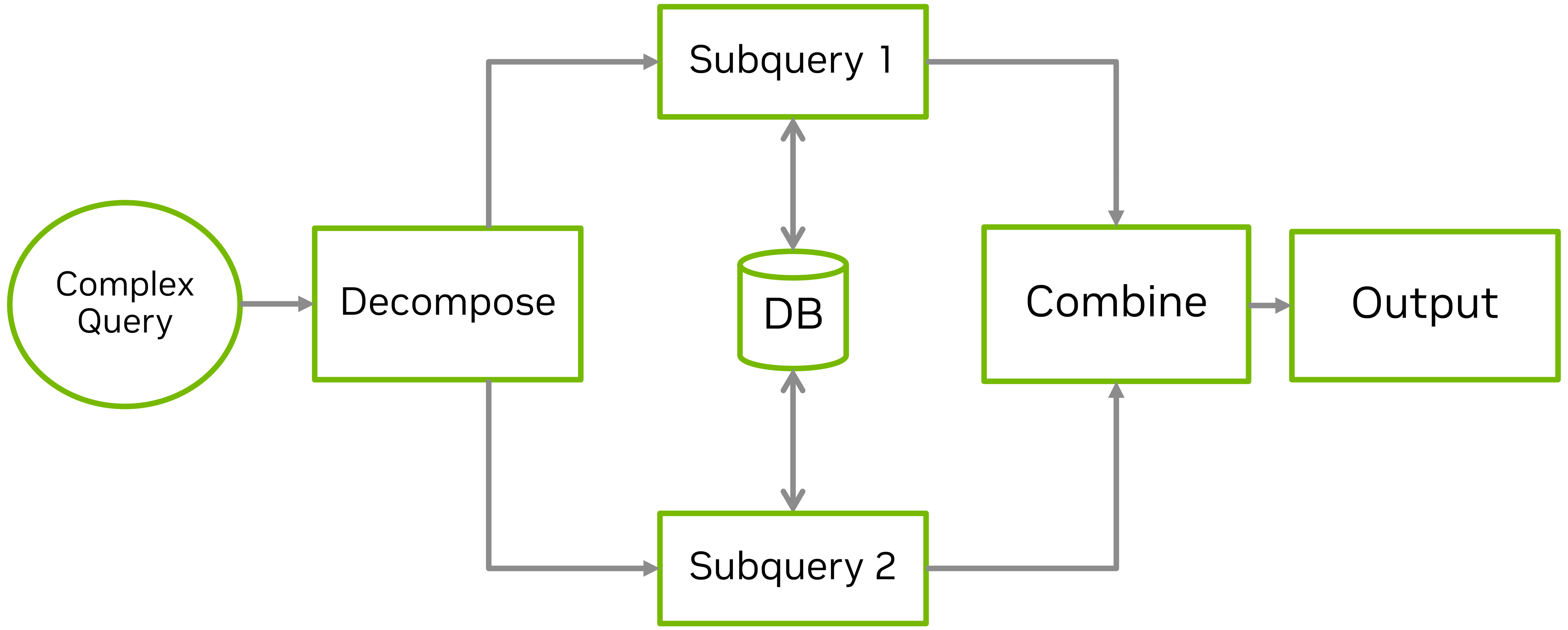
Need to pick right splitting method to ensure no-loss of information, ex. Split on separators

CHUNK SIZE

Smaller chunk size (fine grained) vs. large chunk size (holistic)
Needs experimentation to find right-size chunk based on doc types

Stage 3: Retrieval Optimization

Optimization retrieval to accelerate performance




```
from langchain.text_splitter import MarkdownHeaderTextSplitter

markdown_document =
"# A Beginner's Guide to LLMs\n\n
## Introduction to LLMs\n\n
A large language model is a type of artificial intelligence System\n\n
### What are LLMs \n\n
LLMs are deep learning algorithms that can recognize, extract, summarize \n\n
### Foundation Models vs. Fine-Tuned\n\n
Currently, the most popular method is customizing a model using parameter-efficient
customization techniques, such as p-tuning"

headers_to_split_on = [ ("#", "Header 1"), ("##", "Header 2"), ("###", "Header 3"),]

markdown_splitter
= MarkdownHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
md_header_splits = markdown_splitter.split_text(markdown_document)

>>[Document(page_content='A large language model is a type of artificial
intelligence System etc', metadata={'Header 1': 'A Beginner's Guide to LLMs', 'Header 2':
'Introduction to LLMs'}),
 Document(...,
 Document(page_content='Currently, the most popular method is customizing a model using
parameter-efficient customization techniques, such as p-tuning', metadata={'Header 1': 'A
Beginner's Guide to LLMs', 'Header 2': 'Introduction to LLMs','Header 3': 'Foundation Models
vs. Fine-Tuned'})]
```

SUBQUERY CHAINING

Decompose prompt to multiple retrieval stages

RE-RANKING

Retrieve more results, and rank on multiple attributes to improve query relevance

CONTEXT AWARENESS

Extend context window for chunks, smaller chunks lose context

Building the App

Optimization retrieval to accelerate performance

```
# Used for langchain
document_prompt = PromptTemplate(
    input_variables=["title", "page_content"],
    template="Title: {title}\nContent: {page_content}",
)

prompt = PromptTemplate.from_template(
    'After conducting research on the topic of "{query}", '
    "you found the following resources. While these resources should be relevant to the topic,"
    "some may not be relevant. Use relevant resources as context to write a high-level "
    "overview of the topic in one paragraph. Include the names of SDKs, libraries,"
    "models, or frameworks if they are relevant.\n{context}"

<<<<.................................................................................>>>>

    docs = _results_as_docs(results)

    class MyCustomHandler(BaseCallbackHandler):
        def on_llm_new_token(self, token: str, **kwargs) -> None:
            (...)
```

```
llm = ChatOpenAI(
    model="gpt-3.5-turbo-16k",
    max_tokens=667,
    streaming=True,
    callbacks=[MyCustomHandler()],
)


llm_chain = LLMChain(llm=llm, prompt=prompt)
chain = StuffDocumentsChain(
    llm_chain=llm_chain,
    document_prompt=document_prompt,
    document_variable_name="context",
)

summary = chain.run(query=query,
    input_documents=docs).strip()

return {"summary": summary}
```

WORKFLOW

Retrieve docs, filter top N docs, and feed into LLM to summarize

LLMChain

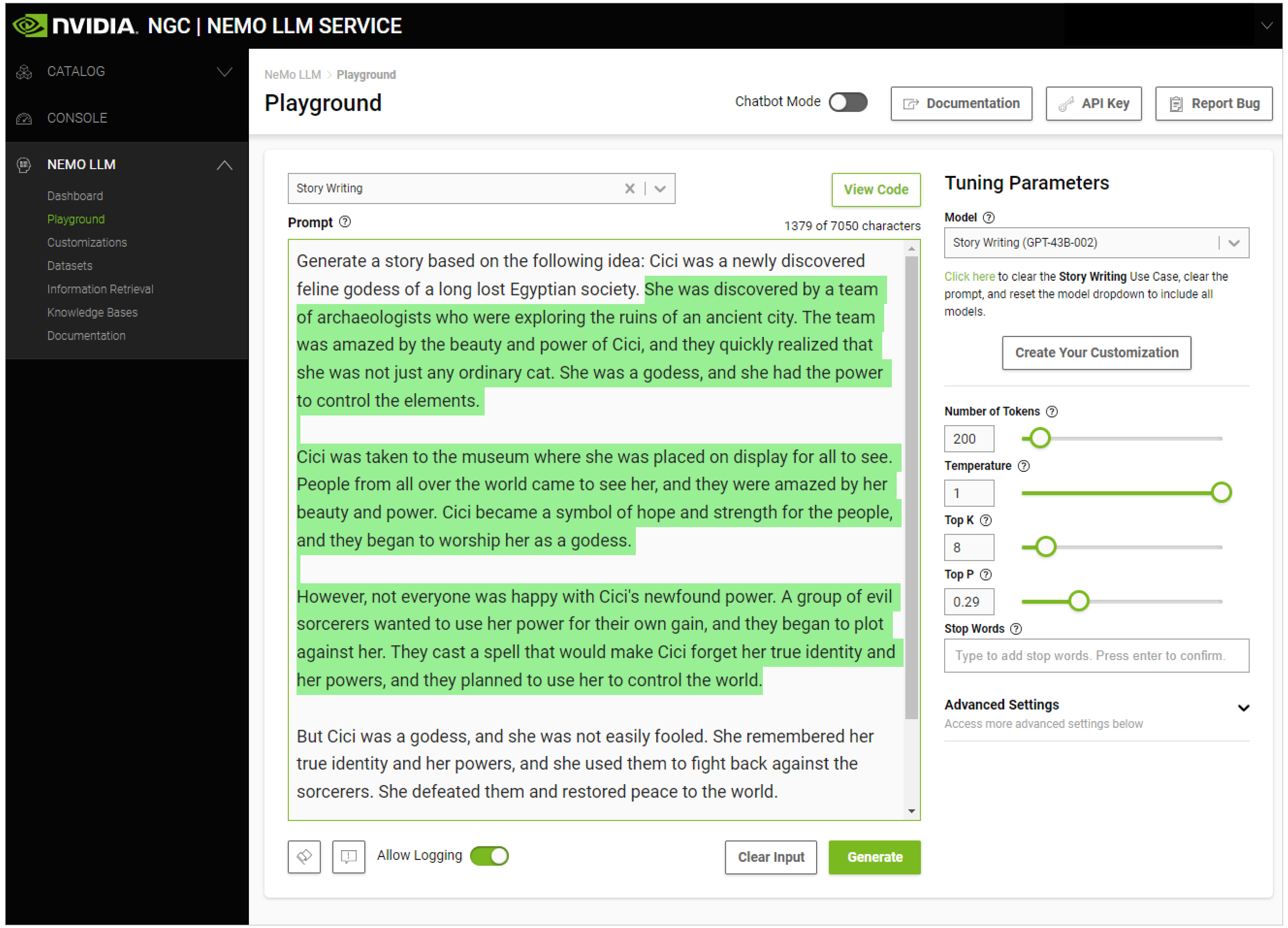
Combine LLM and composite prompt

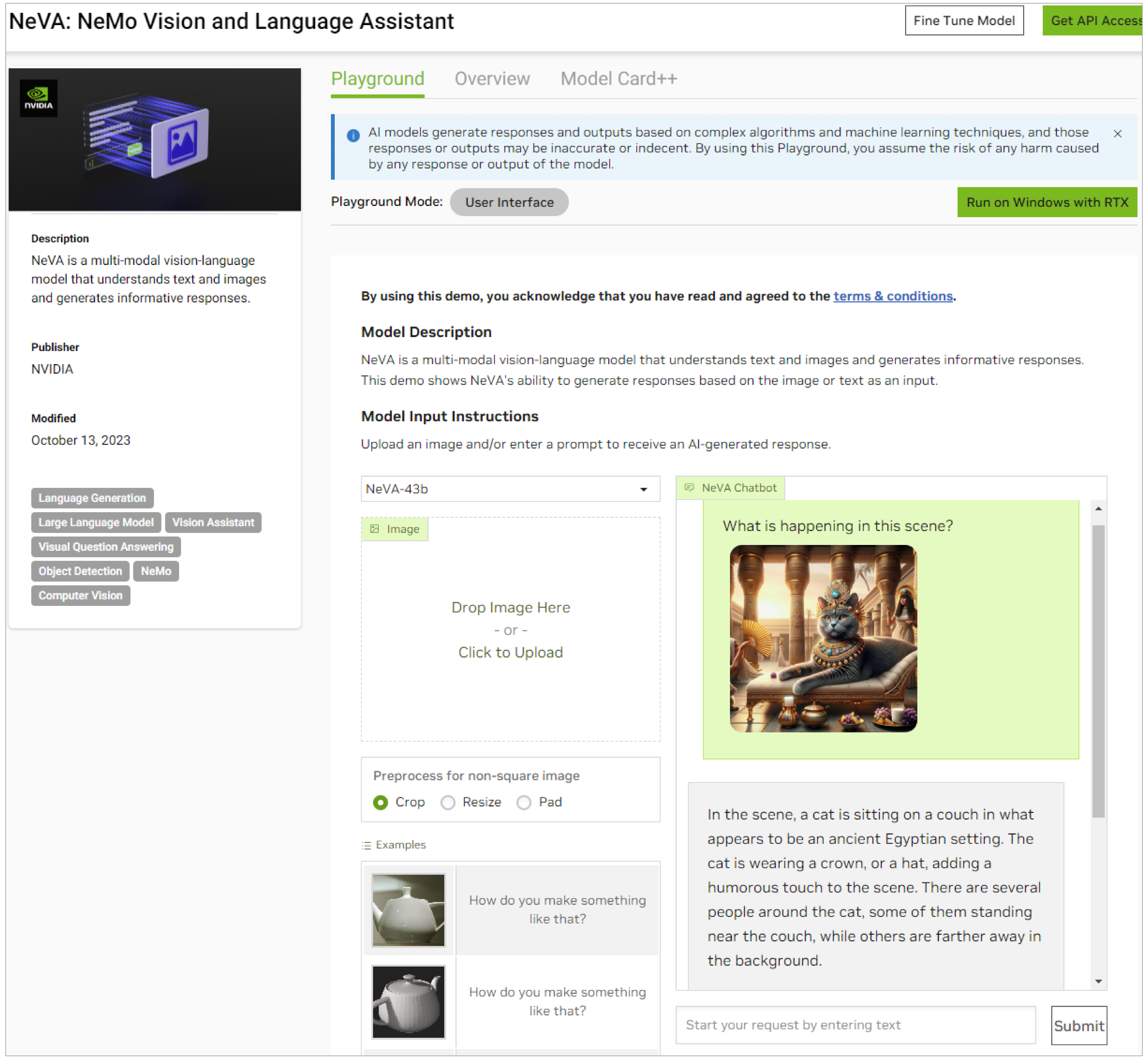
StuffDocumentsChain

Combine documents to feed into LLM as context within prompt

Explore NVIDIA AI Foundation Models

Nemotron-3, Code Llama, NeVA, Stable Diffusion XL, Llama 2, CLIP





https://llm.ngc.nvidia.com/playground

https://catalog.ngc.nvidia.com/ai-foundation-models

What Did You Learn?

- Core concepts of LLM architecture and foundation models
- Factors for selecting between and evaluating LLM APIs
- Prompt engineering basics
- Workflow frameworks for LLMs
- Retrieval Augmented Generation (RAG)
- How these concepts apply to a demo app handling an overflow of email

Many thanks to my colleagues Benjamin Bayart, Chris Pang, and Chris Milroy for major contributions to this session!

Q&A

Apply to NVIDIA Inception for startups:
NVIDIA.com/startups

Join the NVIDIA Developer community to get access to technical training, technology, AI models and 600+ SDKs:

developer.nvidia.com/join

Explore More Gen AI/LLM Training:

25% off Workshops for LLM Day registrants*

USE CODE: **TRAIN-LLM**

Instructor-Led Workshops

> Generative AI With Diffusion Models

> Rapid Application Development Using LLMs

> Efficient Large Language Model (LLM) Customizations

Self-Paced Courses

> Generative AI Explained (Free)

> Generative AI With Diffusion Models

View our comprehensive Gen AI/LLM learning path, covering fundamental to advanced topics
> Gen AI/LLM Learning Path

*****_Offer valid for any of the DLI public workshops scheduled through March 01, 2024._