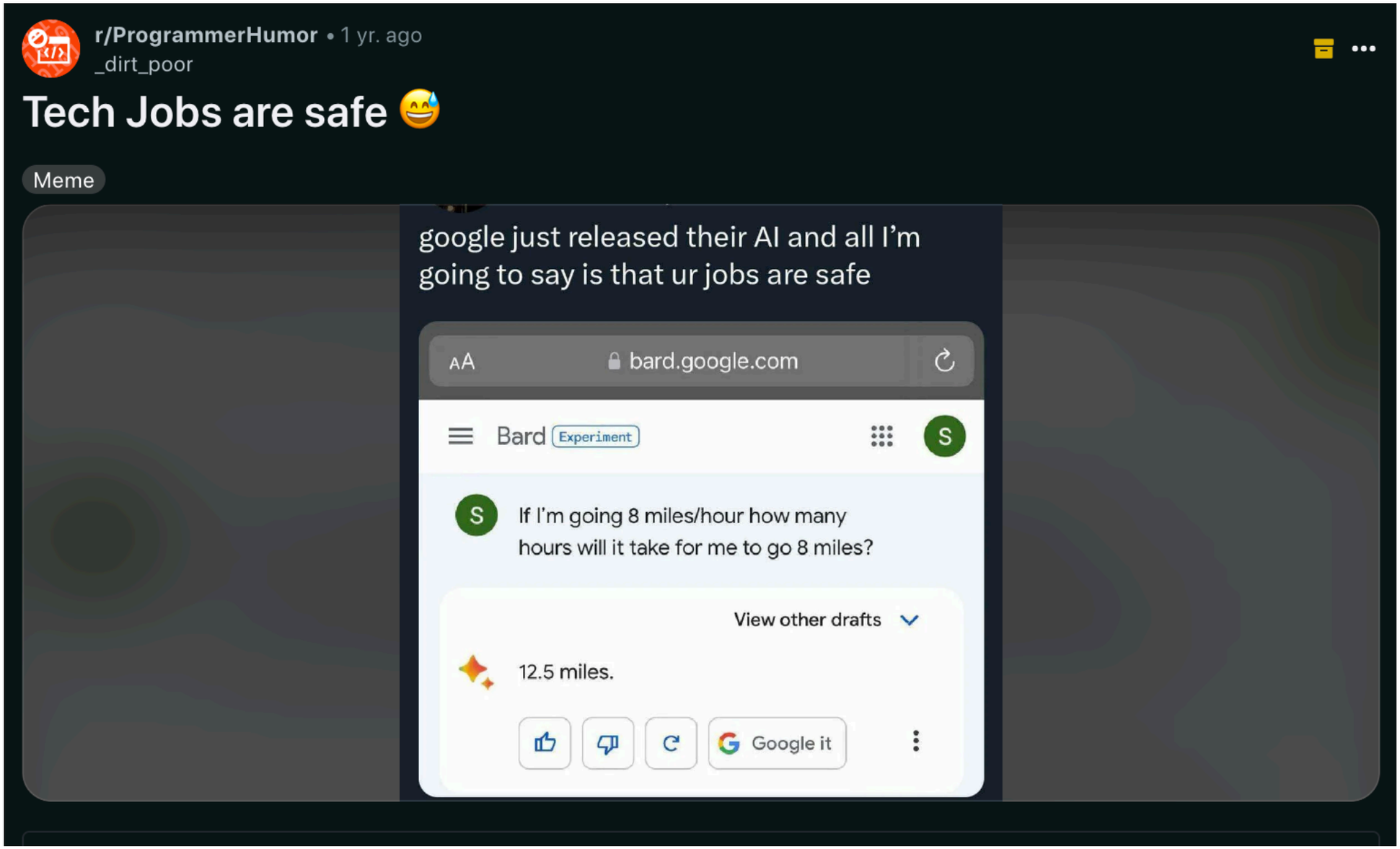
AI Chat Demo

Chatbot with Custom MultiModal Agent using LangChain

Keith Kenemer

Tech Jobs & AI



https://www.reddit.com/r/ProgrammerHumor/comments/11y97dr/tech_jobs_are_safe/

What about agents?

Goal: Chatbot with Custom Multimodal Agent

- **Let's create a gpt-3.5 chatbot (i.e. ChatGPT) for generic queries augmented with a custom multimodal agent to process specific types of queries**

- **The agent should handle the following types of math queries: basic math, symbolic math (e.g. calculus), word problems**

- **The agent should be capable of generating images from a prompt**

- **Proof-of-concept for applications such as : tutor, research assistant, media content assistant, writing assistant, etc**

- **We can leverage the LangChain framework to make development easier**

Generative vs Discriminative Models I

$$p(y \mid x)$$

class label data sample

A discriminative model models the conditional probability of a class label given an input sample x (e.g. classifier).

Analogy: Asking someone which musical genre a particular song belongs to

$$p(x, y)$$

A generative model models the joint distribution of the input samples and associated labels. Able to synthesize data.

Analogy: Asking someone to compose or play a song and tell us the genre

$$p(x \mid y)$$

Analogy: Asking someone to compose or play a song given the genre

Generative vs Discriminative Models II

Discriminative task

Generative models sample from joint distribution

Can be expressed in terms of class conditionals via Baye's Theorem

$$p(y \mid x) = \frac{p(x, y)}{p(x)} = \frac{p(x \mid y)p(y)}{p(x)}$$

Generative model can estimate this

Generative models can be used for discriminative tasks. They are generally considered to be complementary to discriminative models (not a superset) b/c discriminative models have potentially better performance for discriminative tasks

LangChain I

What is LangChain?

LangChain is a framework designed to simplify the creation of applications using large language models (LLMs). As a language model integration framework, LangChain's use-cases largely overlap with those of language models in general, including document analysis and summarization, chatbots, and code analysis.[2]

LangChain relies heavily on the use of prompt templates

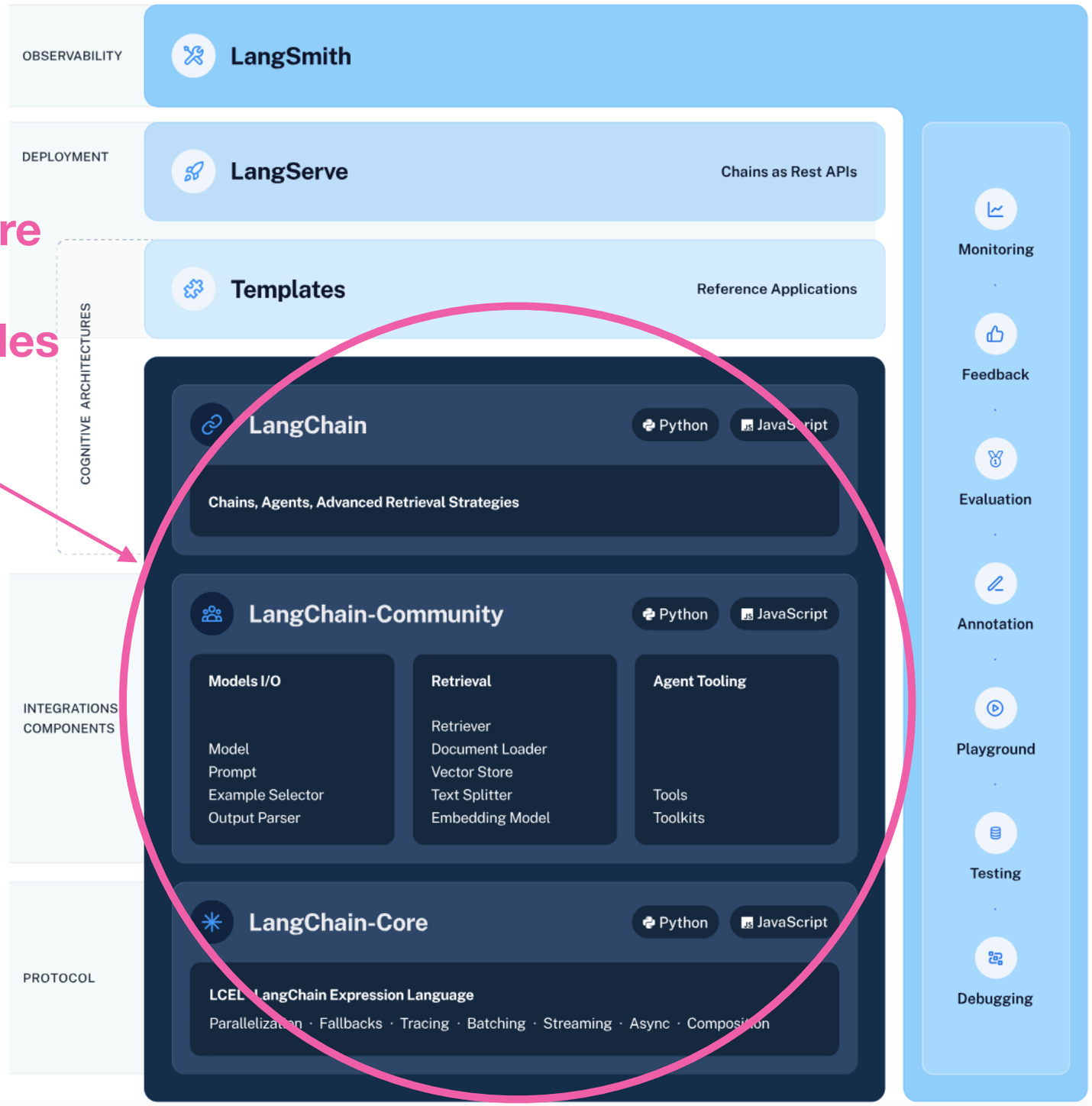
https://en.wikipedia.org/wiki/LangChain

https://www.langchain.com/

https://aws.amazon.com/what-is/langchain/

LangChain II

We'll focus on core LangChain & Community modules



https://python.langchain.com/docs/get_started/introduction

LangChain Expression Language (LCEL)

LCEL makes it easier to build complex processing chains by providing a unified interface (Runnable) and composition primitives

(note: more of convention than a “language”)

Without LCEL

```
from typing import List

import openai


prompt_template = "Tell me a short jok
client = openai.OpenAI()

def call_chat_model(messages: List[dic
    response = client.chat.completions
        model="gpt-3.5-turbo",
        messages=messages,
    )
    return response.choices[0].message

def invoke_chain(topic: str) -> str:
    prompt_value = prompt_template.for
    messages = [{"role": "user", "cont
    return call_chat_model(messages)

invoke_chain("ice cream")
```

LCEL

```
from langchain_core.runnables import R


prompt = ChatPromptTemplate.from_templ
    "Tell me a short joke about {topic
)
output_parser = StrOutputParser()
model = ChatOpenAI(model="gpt-3.5-turb
chain = (
    {"topic": RunnablePassthrough()}
    | prompt
    | model
    | output_parser
)

chain.invoke("ice cream")
```

https://python.langchain.com/docs/get_started/introduction
https://python.langchain.com/docs/expression_language/why

Math & Logic LangChain Modules

```
from langchain.chains import LLMChain
from langchain.chains import LLMMathChain
from langchain.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain_experimental.pal_chain.base import PALChain
from langchain_experimental.llm_symbolic_math.base import LLMSymbolicMathChain
```

These are some of the out-of-the-box math & logic processing chains

- **LLMChain** (generic queries) → **calls LLM with prompt**
- **LLMMathChain**) → **calls numexpr**
- **LLMSymbolicMathChain** → **calls sympy**
- **PALChain** (word problems) → **creates/runs solution function**

numexpr = numerical expression evaluator (python module)

sympy = symbolic math (python module)

PAL = program-aided language model

Classifying the User Query via a Meta Query

The meta-query asks the LLM (gpt-3.5) to categorize the appended user query into one of 5 classes:

- **Generic Query (non-math related)**
- **Basic Math Query**
- **Symbolic Math Query (e.g. calculus)**
- **Word Problem**
- **Image Generation Request**

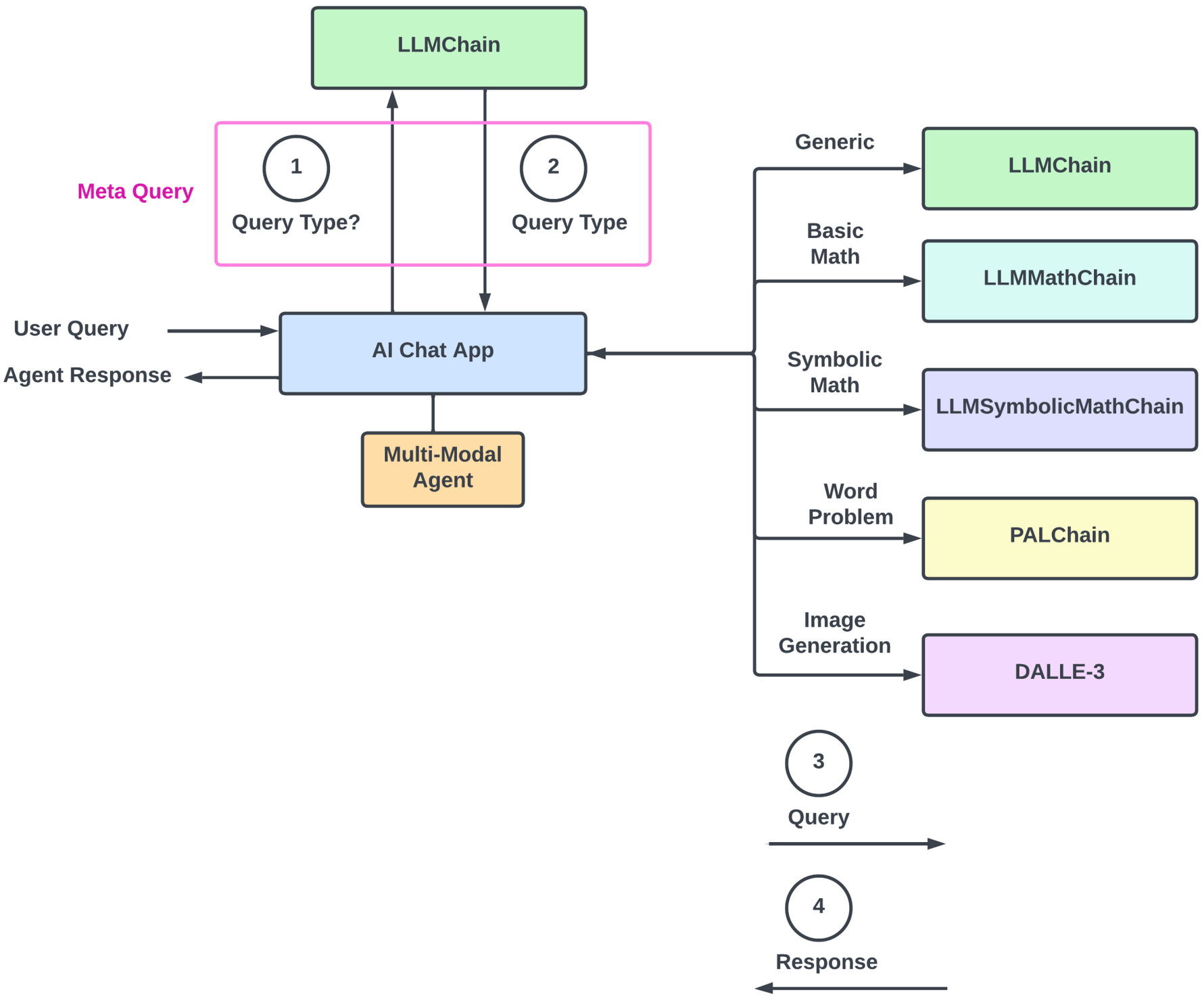
```
# base LLM (determine query type)
self.llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature = TEMPERATURE, request_timeout=REQ_TIMEOUT)
self.prompt_template = PromptTemplate(
    input_variables=["query_type1", "query_type2", "query_type3", "query_type4", "user_input"],
    template="Categorize the following query as either a {query_type1}, {query_type2},"
             "{query_type3}, {query_type4}, or {query_type5}:\n{user_input}")
self.llm_chat_category = LLMChain(llm=self.llm, prompt=self.prompt_template)

# base LLM (generic)
self.prompt_template_generic = PromptTemplate(
    input_variables=["user_input"],
    template="{user_input}")
self.llm_chat_generic = LLMChain(llm=self.llm, prompt=self.prompt_template_generic)


# basic math LLM
self.llm_basic_math = LLMMathChain.from_llm(self.llm, verbose=True)

# symbolic math LLM
self.llm_symbolic_math=LLMSymbolicMathChain.from_llm(self.llm, verbose=True)

# word problem LLM
self.palchain = PALChain.from_math_prompt(llm=self.llm, verbose=True, timeout=REQ_TIMEOUT)

# for image generation requests
self.open_ai_client = OpenAI()
```

Agent Query Processing

The agent switches between processing chains depending on what the category the user query was determined to be.

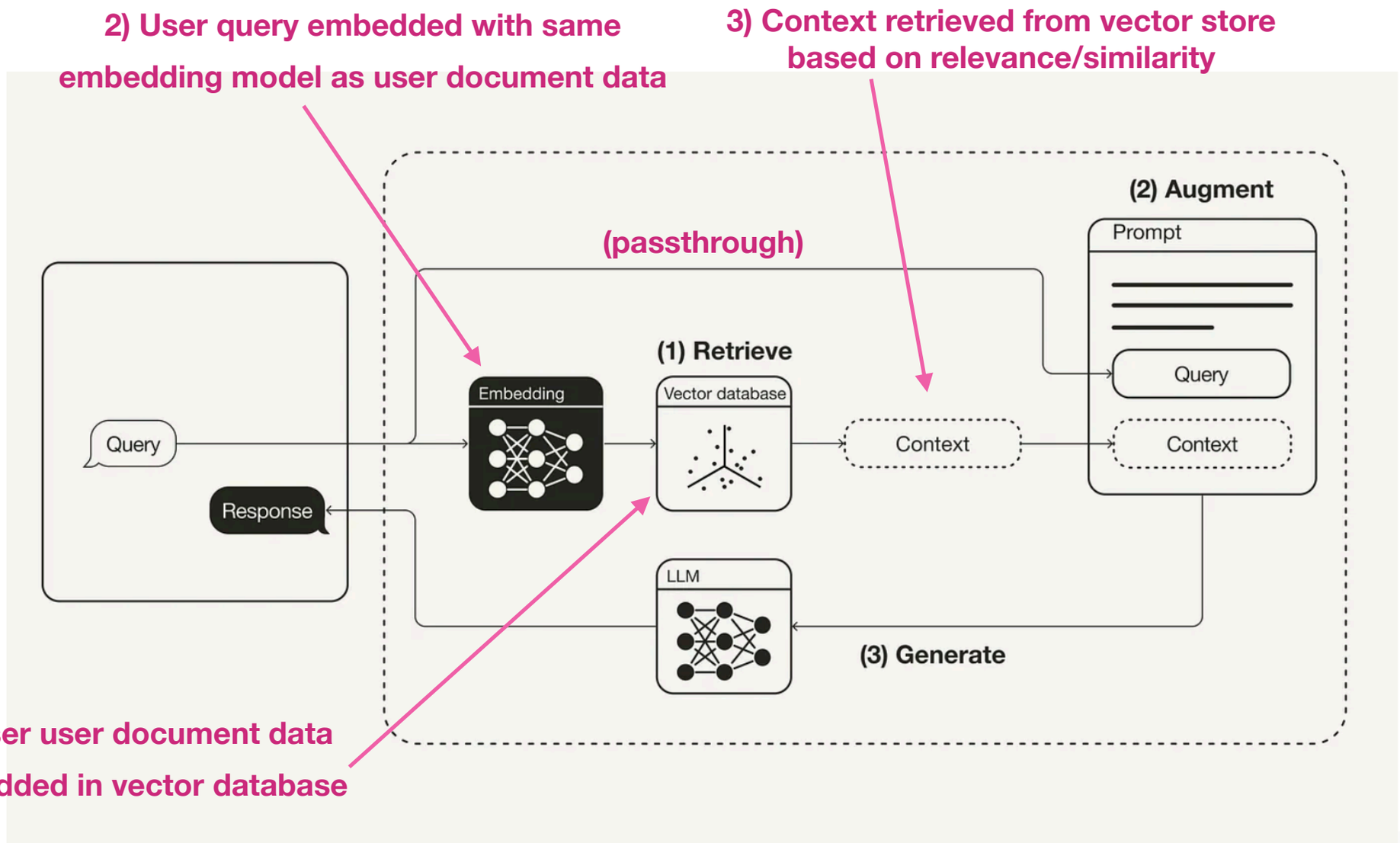
```
# query the base llm to categorize the query type
category_response = self.llm_chat_category.invoke({"query_type1":QUERY_TYPE1,"query_type2":QUERY_TYPE2,
                                                   "query_type3":QUERY_TYPE3, "query_type4":QUERY_TYPE4,
                                                   "query_type5":QUERY_TYPE5,"user_input":user_input})

# call the appropriate chain
category_response_txt = category_response["text"].lower()
print(f"\ncategory response: {category_response_txt}")

if "basic" in category_response_txt:
    print("(interpreting as a basic math query)")
    response = self.llm_basic_math.invoke(user_input)
    response = response["answer"]
elif "symbolic" in category_response_txt:
    print("(interpreting as a symbolic math query)")
    response = self.llm_symbolic_math.invoke(user_input)
    response = response["answer"]
elif "word" in category_response_txt:
    print("(interpreting as a word problem)")
    response = self.palchain.invoke(user_input)
    response = response["result"]
elif "image" in category_response_txt or "generation" in category_response_txt:

    # call DALL-E3 for image request
    print("(interpreting as an image generation request)")
    dalle_response = self.open_ai_client.images.generate(
    model="dall-e-3",
    prompt=user_input,
    size="1024x1024",
    quality="standard",
    n=1,
    )

    # show the result that was pushed in the response url
    url = dalle_response.data[0].url
    self.display_image_from_url(url, 1024,1024)
    response = "image generation complete"
else:
    print("(interpreting as a generic query)")
    response = self.llm_chat_generic.invoke({"user_input":user_input})
    response = response["text"]
```

Demo App Architecture



Retrieval-Augmented Generation (RAG)

- Allows integration of user information as context for queries
- User data is chunked and embeddings stored in vector database
- Relevant context pulled from vector store based on similarity to query
- Retrieved content is used as context within prompt template



https://towardsdatascience.com/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2

Takeaways & Observations

- **Generative Models can also be used for discriminative tasks**
- **Agents expand the capabilities of LLMs by connecting them with other computational tools.**
- **The discriminative capability of LLMs can be useful for building agents since it can inform the agent how to select the appropriate processing chain for different types of queries**
- **LangChain provides a framework which makes it easier to incorporate LLMs into applications by facilitating the construction of complex processing chains and providing many out-of-the-box processing chains**
- **RAG-enabled agents can be implemented using LangChain Community embedding and vector stores**

LangChain Resources

Below are some LangChain resources and tutorials:

https://python.langchain.com/docs/get_started/introduction

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

https://towardsdatascience.com/a-gentle-intro-to-chaining-llms-agents-and-utils-via-langchain-16cd385fca81

https://python.langchain.com/docs/additional_resources/tutorials

https://www.youtube.com/watch?v=DkBc4hfGIe8&t=2s