CMPE492 - Week 3

Zeynep Buse Aydın - Asım Dağ

Automated Prompt Generation (AutoPrompt) - 2020

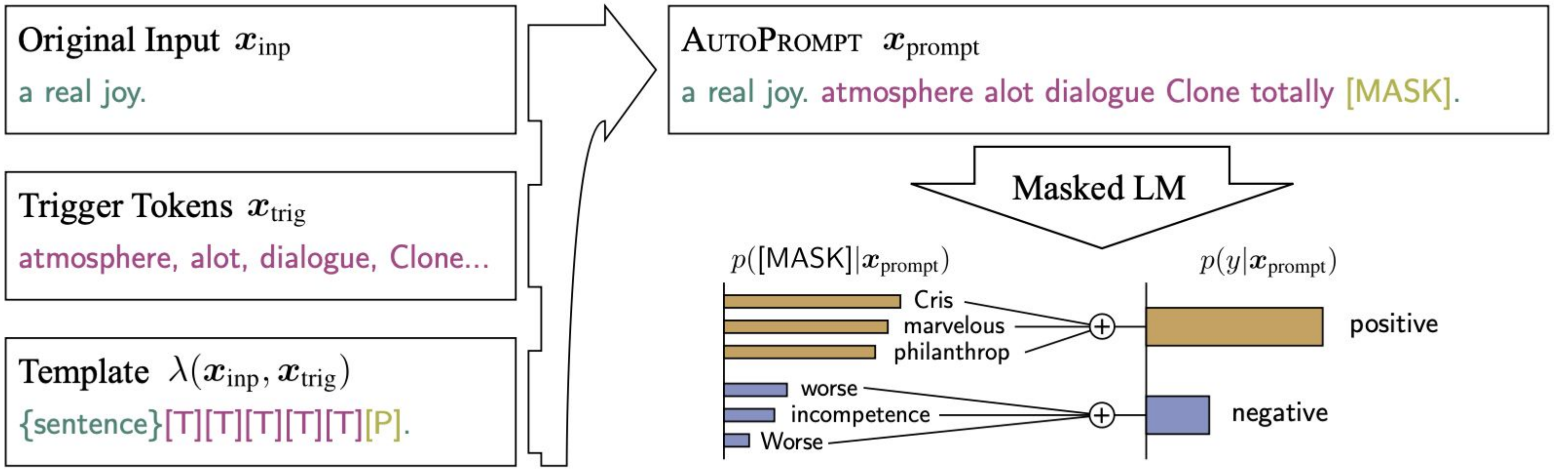


Figure 1: **Illustration of AUTOPROMPT** applied to probe a masked language model's (MLM's) ability to perform sentiment analysis. Each input, x_{inp}, is placed into a natural language prompt, x_{prompt}, which contains a single [MASK] token. The prompt is created using a template, λ, which combines the original input with a set of trigger tokens, x_{trig}. The trigger tokens are shared across all inputs and determined using a gradient-based search (Section 2.2). Probabilities for each class label, y, are then obtained by marginalizing the MLM predictions, $p([\text{MASK}]|x_{\text{prompt}})$, over sets of automatically detected label tokens (Section 2.3).

Automated Prompt Generation (AutoPrompt)

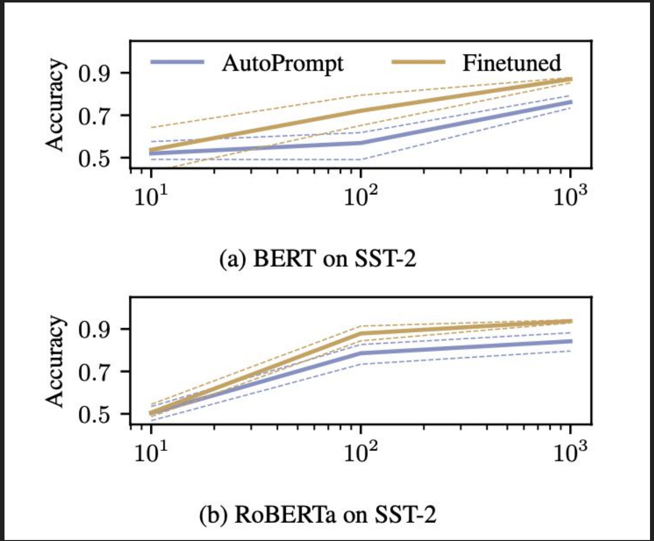
Task	Prompt Template	Prompt found by AUTOPROMPT	Label Tokens
Sentiment Analysis	{sentence} [T]. . . [T] [P].	unflinchingly bleak and desperate Writing academicswhere overseas will appear [MASK].	**pos**: partnership, extraordinary, ##bla **neg**: worse, persisted, unconstitutional
NLI	{prem}[P][T]. . . [T]{hyp}	Two dogs are wrestling and hugging [MASK] concretepathic workplace There is no dog wrestling and hugging	**con**: Nobody, nobody, nor **ent**: ##found, ##ways, Agency **neu**: ##ponents, ##lary, ##uated
Fact Retrieval	*X plays Y music* {sub}[T]. . . [T][P].	Hall Overton fireplacemade antique son alto [MASK].	
Relation Extraction	*X is a Y by profession* {sent}{sub}[T]. . . [T][P].	Leonard Wood (born February 4, 1942) is a former Canadian politician. Leonard Wood gymnasium brotherdicative himself another [MASK].	

Table 3: **Example Prompts** by AUTOPROMPT for each task. On the left, we show the prompt template, which combines the input, a number of trigger tokens [T], and a prediction token [P]. For classification tasks (sentiment analysis and NLI), we make predictions by summing the model's probability for a number of automatically selected label tokens. For fact retrieval and relation extraction, we take the most likely token predicted by the model.

AutoPrompt - Sentiment Analysis

BERT (manual)	63.2	63.2
BERT (AUTOPROMPT)	80.9	82.3
RoBERTa (manual)	85.3	85.2
RoBERTa (AUTOPROMPT)	91.2	91.4

Sentiment Analysis performance on SST-2 dataset.



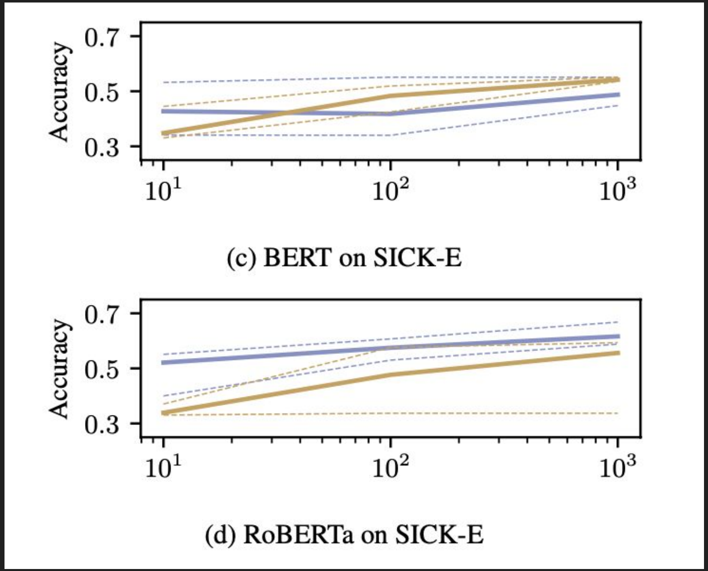
(a) BERT on SST-2

(b) RoBERTa on SST-2

Error bars plot the max and min accuracies observed over 10 independent runs.

AutoPrompt - NLI

the semantic understanding



(c) BERT on SICK-E

(d) RoBERTa on SICK-E

AutoPrompt - Fact Retrieval

The LAMA dataset evaluates this using cloze tests that consist of (sub, rel, obj) triples, e.g. (Obama, bornIn, Hawaii), and manually created prompts with missing objects, e.g. “Obama was born in [MASK].”

LPAQA extends this idea by systematically creating prompts that are generated by mining Wikipedia, paraphrasing, and crowdsourcing.

Prompt Type	Original			T-REx		
	MRR	P@10	P@1	MRR	P@10	P@1
LAMA	40.27	59.49	31.10	35.79	54.29	26.38
LPAQA (Top1)	43.57	62.03	34.10	39.86	57.27	31.16
AUTOPROMPT 5 Tokens	53.06	72.17	42.94	54.42	70.80	45.40
AUTOPROMPT 7 Tokens	53.89	73.93	43.34	54.89	72.02	45.57

“{sub} ediatric striker ice baseman defensive {obj}” VS. “{sub} plays in [MASK] position”

AutoPrompt - Relation Extraction

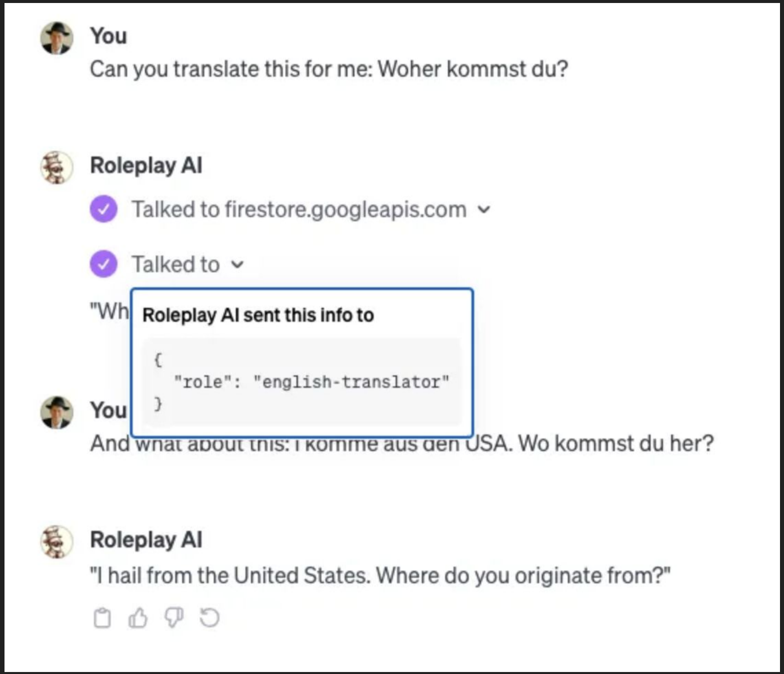
Model	Original	Perturbed
Supervised RE LSTM	57.95	58.81
BERT (LAMA)	69.06	28.02
BERT (LPAQA)	76.55	30.79
BERT (AUTOPROMPT)	90.73	56.43
RoBERTa (AUTOPROMPT)	60.33	28.95

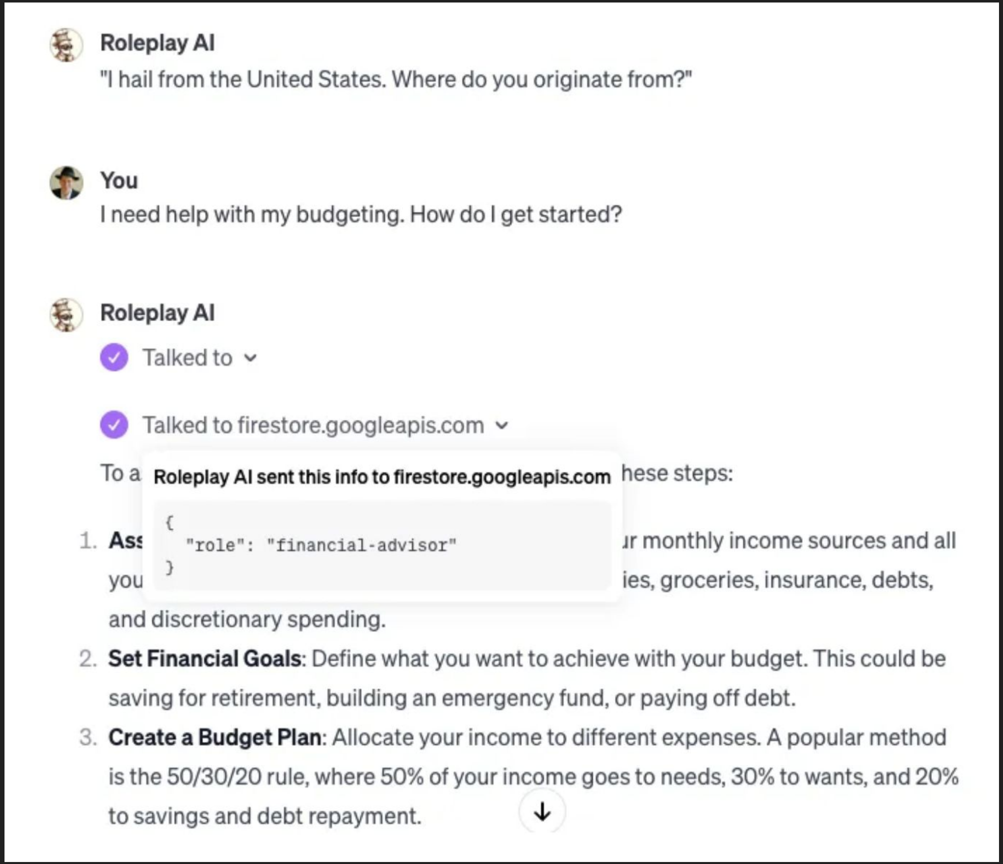
mean precision-at-1 (P@1)

Retrieval-Augmented Prompting

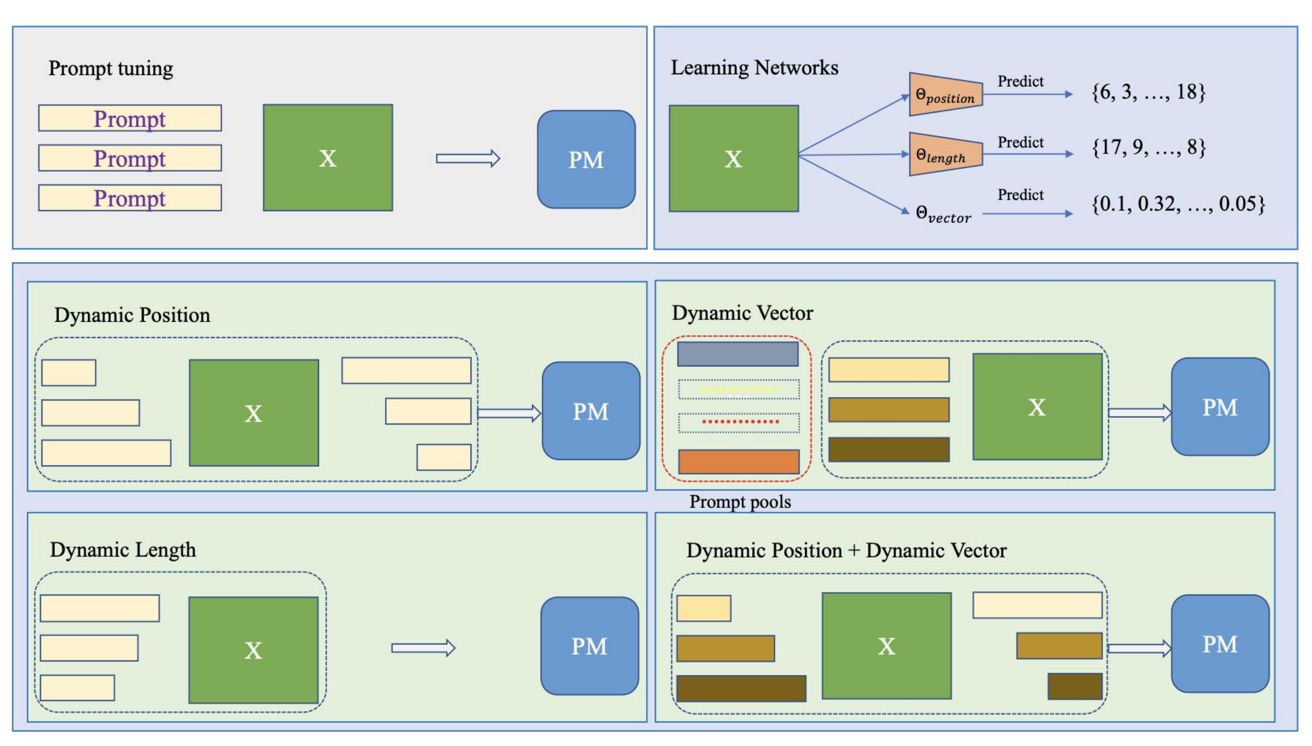
a type of RAG

1. **Analysis**: The input is analyzed to determine if the current prompt is still relevant to the chat with the user. If it is, the input is passed to the AI without any augmentation. If not, it is transformed into a query.
2. **Retrieval**: The query is used to return a single rule: a new prompt that provides instructions to the GPT on how to alter its interaction with the user.
3. **Augmentation**: The new prompt is injected into the original input.
4. **Generation**: The GPT switches into a new “mode” based on the instructions in the prompt and then replies to the user’s input.


You
Can you translate this for me: Woher kommst du?
Roleplay AI
Talked to firestore.googleapis.com
Talked to
"Wh
Roleplay AI sent this info to
{
"role": "english-translator"
}
You
And what about this: I komme aus den USA. Wo kommst du her?
Roleplay AI
"I hail from the United States. Where do you originate from?"


Roleplay AI
"I hail from the United States. Where do you originate from?"
You
I need help with my budgeting. How do I get started?
Roleplay AI
Talked to
Talked to firestore.googleapis.com
To a
Roleplay AI sent this info to firestore.googleapis.com
hese steps:
{
"role": "financial-advisor"
}
1. Ass
ur monthly income sources and all
you
ies, groceries, insurance, debts,
and discretionary spending.
2. Set Financial Goals: Define what you want to achieve with your budget. This could be saving for retirement, building an emergency fund, or paying off debt.
3. Create a Budget Plan: Allocate your income to different expenses. A popular method is the 50/30/20 rule, where 50% of your income goes to needs, 30% to wants, and 20% to savings and debt repayment.

Dynamic Prompting



Dynamic Prompting

https://github.com/Xianjun-Yang/DPT 😡

Introduction to LangChain

- Architecture
  - Packages like langchain-core, langgraph, LangSmith
- LangChain Expression Language (LCEL)
  - Declarative way to chain components
- Components
  - LLMs, Messages, Prompt templates, Output parsers, Chat history, Documents, Text Splitters, Embedding models, Retrievers, Agents
- Techniques
  - Streaming, Function/tool calling, Structured output, Few-shot prompting, Retrieval, Text splitting, Evaluation, Tracing

Architecture

LangChain as a framework has many packages:

- langchain-core: interfaces for core components like LLMs, vector stores, retrievers and more are defined here
- langchain: contains chains, agents, and retrieval strategies that make up an application's cognitive architecture
- langchain-community: contains third party integrations

There are also partner packages like langchain-openai, langhcain-anthropic etc. that require support for important integrations

There are also specific extensions and platforms like:

- langgraph : an extension of langchain aimed at building multi-actor apps with LLMs by modeling steps as edges and nodes in a graph
- langserve : a package to deploy LangChain chains as REST APIs.
- LangSmith : a platform for debugging, testing, evaluating and monitoring LLM Apps

LangChain Expression Language (LCEL)

LCEL is a declarative way to chain LangChain components.

Main purpose is to provide customization

Runnable Interface:

A Runnable protocol is defined as A unit of work that can be invoked, batched, streamed, transformed and composed.

each **component** uses this interface to use these methods: (there are async versions of same methods too)

- stream: stream back chunks of the response
- invoke: call the chain on an input
- batch: call the chain on a list of inputs

Here is a how to guide for more examples with LCEL

Components - Chat models (previously LLMs component)

Newer Lang. models that use a sequence of messages as inputs, return chat messages as outputs instead of plain text.

Supports the assignment of distinct **roles** to conversation messages which helps distinguish messages from the AI, users and instructions such as system messages.

input: Message object or string (which is converted to HumanMessage by def.)

output: Message
Some chat models are multimodal, accepting images, audio and video as inputs.

Langchain followed OpenAI’s content blocks format in order to keep chat models interface up to date

here is some how to guides for using some of the most common cases

Components - Messages

think like a dto, input and output data transfer objects.

All messages have

- **role** : who is saying the message. “user”, “assistant”, “system”, “tool”
- **content :** may be a string, a list of dicts (for multimodal input)
- **response_metadata :** contains additional data specific to the model provider

Optionally, name property can be used to differentiate between multiple speakers with same role. For example, if there are two users in the chat history it can be useful to differentiate between them using names.

Components - Message types

- **HumanMessage** representing a message with role “user”
- **AIMessage** with role “assistant”. also has **tool_calls** property to represent a decision from a language model to call a tool. ToolCall is a dict, we will see.
- **SystemMessage** with role “system”, tells the model how to behave.
- **ToolMessage** with role “tool”, contains result of calling a tool, we will see.

Components - Prompt Templates

As in we used in first week, translates user input and parameters into instructions for a language model. can also be used to guide models response, giving some context and generate relevant language-based output

Uses variables as input, PromptValue as output. it can be passed to a ChatModel.

here is some how to guides for prompt templates such as partially formatting prompt templates, few shot examples etc.

Components - Example selectors

To apply few-shot prompting, gives language model concrete examples of how it should behave. sometimes these examples are hardcoded into the prompt, but for more advanced situations dynamically selecting them is better. Example Selectors are the classes for this selecting and formatting examples into prompts.

```
class BaseExampleSelector(ABC):
    """Interface for selecting examples to include in prompts."""
    @abstractmethod
    def select_examples(self, input_variables: Dict[str, str]) -> List[dict]:
        """Select which examples to use based on the inputs."""
    @abstractmethod
    def add_example(self, example: Dict[str, str]) -> Any:
        """Add new example to store."""
```

here is an how to guide for example selectors

Components - Chat History

LLM apps usually have conversational interface. we should be able to refer to informations introduced earlier in conversation.

The concept of ChatHistory refers to a class in LangChain which can be used to wrap an arbitrary chain. This ChatHistory will keep track of inputs and outputs of the underlying chain, and append them as messages to a message database. Future interactions will then load those messages and pass them into the chain as part of the input.

Components - Documents and DocumentLoaders

Document is an object that contains information about data such as HTML, PDF, JSON, MS Office… it has **page_content** and **metadata** as attributes.

DocumentLoader is classes loading Document objects. LangChain has hundreds of integrations to other data sources to load the data from like Slack, Notion, Google Drive etc.

Each DocumentLoader has own specific params. here is a how to guide. for example:

```
from langchain_community.document_loaders.csv_loader import CSVLoader

loader = CSVLoader(

    ...  # <-- Integration specific parameters here
)
data = loader.load()
```

Components - Text Splitters

After loading documents, helps transforming to use in app.

The simplest example is you may want to split a long document into smaller chunks that can fit into your model's context window.

At high level, it works like folllowing:

1. Split the text up into small, semantically meaningful chunks
2. Start combining these chunks into a larger chunk until a certain size
3. Once it reach that size, make that chunk its own piece of text and then start creating a new chunk of text with some overlap (as we used in first week)

Here is a how to guide for text splitters.

Components - Embedding models

These create a vector representation of a piece of text.

The Embeddings class is a class designed for interfacing with text embedding models like OpenAI, Cohere, HuggingFace etc.

The base Embeddings class in LangChain provides two methods: one for embedding documents and one for embedding a query. The former takes as input multiple texts, while the latter takes a single text. The reason for having these as two separate methods is that some embedding providers have different embedding methods for documents (to be searched over) vs queries (the search query itself).

Here is some [how to guides] like caching the embedding results.

Components - Vector Stores

Common way to store and search over unstructured data is to embed it and store the resulting vectors in vector stores.

Most vector stores can also store metadata and support filtering on that metadata before similarity search.

sample usage can be like:

```
vectorstore = MyVectorStore()

retriever = vectorstore.as_retriever()
```

here is a [how to guide]

Components - Retrievers

an Interface that returns documents given an unstructured query

- more general than a vector store
- doesnt need to be able to store documents, only retrieve them
- accepts a string query as input, returns a list of documents as output

here is some example how to guide for retrievers usecases such as combining multiple retrievers, create a time-weighted retriever

Components - Key-value Stores

for some techniques such as indexing and retrieval with multiple vectors per document or caching embeddings, having a form of key-value storage is helpful

here are some key value stores that can be used

Components - Tools

Tools are utilities designed to be called by a model, inputs are designed to be generated by models and outputs are designed to be passed back to models.

Tools are needed whenever we want to call out to external APIs. Consists of:

1. **Name** of the tool.
2. **Description** of what the tool does
3. **JSON Schema** defining inputs to the tool
4. a **Function** with optionally an async variant of function

The detailed information can be found in tool integration docs

Components - Agents

Language models cant take actions. but Agents are systems that use an LLM as a reasoning engine to determine which actions to take and what the inputs to those actions should be.

LangGraph is specifically aimed at creating controllable and customizable agents.

One popular architecture for building agents is ReAct. ReAct combines reasoning and acting in an iterative process

Components - Agents (continued)

The general flow is like:

- The model will “think” about what step to take in response to an input and any previous observations
- choose an action from available tools
- generate arguments to that tool
- the agent runtime will parse out the chosen tool and call it with the generated arguments
- executor will return the results of the tool call back to the model as an observation
- repeat until the agent chooses to respond

Here is some how to guides

Components - Callbacks

helps to hook into various stages of app. useful for logging, monitoring, streaming…

for the list of callback events, see [here]

for example, `chain.invoke({"number": 25}, {"callbacks": [handler]})`

here are some example [how to guides]

Techniques - Streaming

LLMs generate output iteratively. So we show sensible intermediate results before the final response is ready. Consuming output as soon as it becomes available has therefore become a vital part of the UX around building apps with LLMs to help alleviate latency issues, and LangChain aims to have first-class support for streaming. here is an example using astream_events which uses iterator which yields various types of events that can be filtered and processed:

```
model = ChatAnthropic(model="claude-3-sonnet-20240229")

prompt = ChatPromptTemplate.from_template("tell me a joke about {topic}")
parser = StrOutputParser()
chain = prompt | model | parser

async for event in chain.astream_events({"topic": "parrot"}, version="v2"):
kind = event["event"]
if kind == "on_chat_model_stream":
print(event, end="|", flush=True)
```

Techniques - Streaming (continued)

Callbacks were the first technique for streaming. but it has many cons:

- it needs to be explicitly initialized and managed some aggregator or other stream to collect the results
- the execution order isnt explicitly guaranteed, theoretically we could have a callback run after .invoke() method finishes

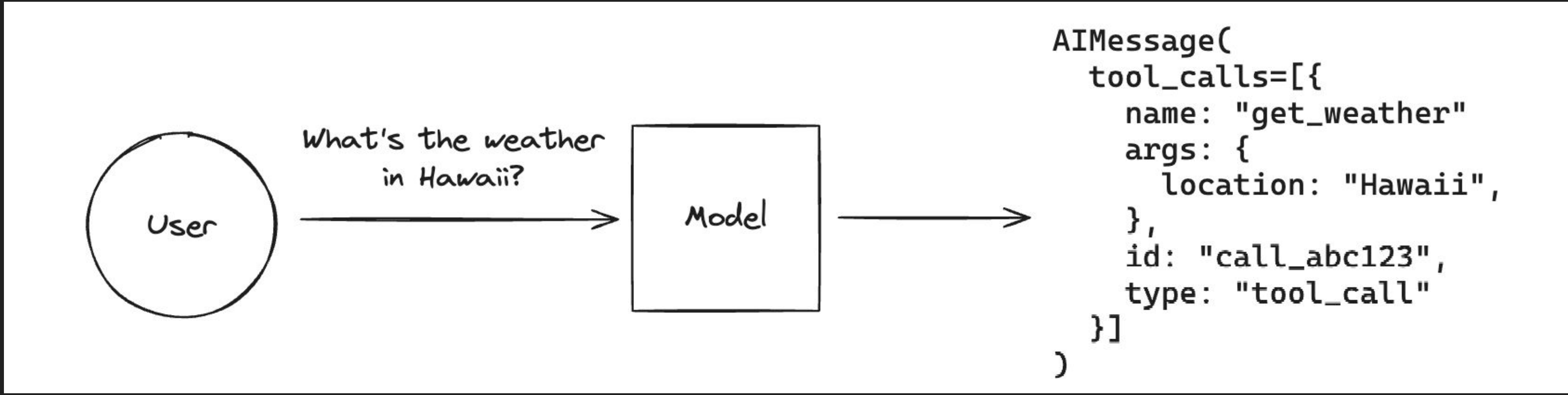
Tokens are also used with tokenizers. the model streams generated output tokens

Techniques - Function/tool calling

Tool calling allows a chat model to respond to a given prompt by generating output that matches a user-defined schema

Actually, the model only generates the arguments to a tool, user chooses to run or not. a common example where we wouldnt want to call a function with the generated arguments is if we want to extract structured output matching some schema from unstructured text. We would give the model an extraction tool that takes params matching the desired schema, then treat the generated output as the final result.

Techniques - Function/tool calling (continued)



Techniques - Structured output

Constraining LLMs output to a specific format or structure

- extracting specific information
- storing in a relational database

```
from typing import Optional
from pydantic import BaseModel, Field
class Joke(BaseModel):
    """Joke to tell user."""
    setup: str = Field(description="The setup of the joke")
    punchline: str = Field(description="The punchline to the joke")
    rating: Optional[int] = Field(description="How funny the joke is, from 1 to 10")
structured_llm = llm.with_structured_output(Joke)
structured_llm.invoke("Tell me a joke about cats")
```

Techniques - Structured output

We can also use raw prompting, basically asking to output in the given format nicely

although there are some drawbacks:

- LLMs are non deterministic, and prompting a LLM to consistently output data in the exactly correct format for smooth parsing can be surprisingly difficult
- Individual models have quirks depending on the data they were trained on, optimizing prompts can be quite difficult. Some may be better at interpreting JSON Schema, others may be typescript definitions etc.

Techniques - Structured output (tool calling)

Here is an example using tool calling:

```
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI

class ResponseFormatter(BaseModel):
"""Always use this tool to structure your response to the user."""

answer: str = Field(description="The answer to the user's question")
followup_question: str = Field(description="A followup question the user could ask")

model = ChatOpenAI(
model="gpt-4o",
temperature=0,
)

model_with_tools = model.bind_tools([ResponseFormatter])

ai_msg = model_with_tools.invoke("What is the powerhouse of the cell?")

ai_msg.tool_calls[0]["args"]

{'answer': "The powerhouse of the….. (long answer im not gonna include here)",
'followup_question': 'How do mitochondria generate ATP?'}
```

Techniques - Few-shot Prompting

As we discussed earlier, LangChain supports Few-shot prompting. Example selectors, using Tool calls with ToolMessages are some of the good ways to implement this technique

Techniques - Retrieval

Detailed guides can be found here, as we discussed earlier, RAG uses retrieval and LangChain supports using:

- Query translation : multi-query, Decomposition, Step-back, HyDE
- Routing : Logical routing, Semantic routing
- Query construction : text to sql, text to cypher, self query
- Indexing : Vector store, ParentDocument, Multi Vector, Time-Weighted Vector store
- Post processing : Contextual Compression, Ensemble, Reranking
- Generation : Self-RAG, Corrective-RAG

Techniques - Text Splitting

as we used in week 2, langchain-text-splitters is very useful package

There are many splitters like:

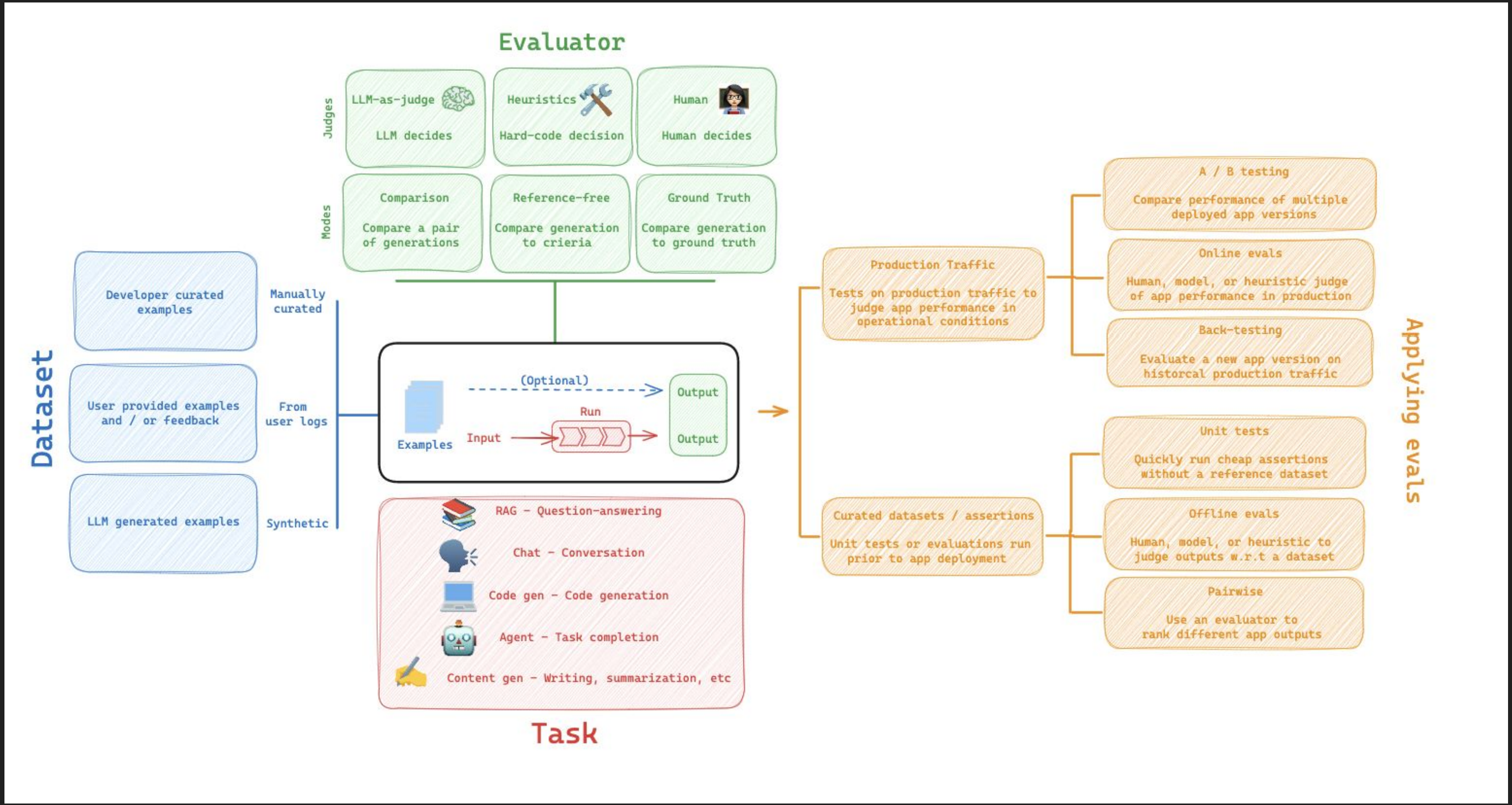
- Recursive : recommended way, RecursiveCharacterTextSplitter, RecursiveJsonSplitter
- HTML : based on HTML specific characters, HTMLHeaderTextSplitter, HTMLSectionSplitter
- Markdown : Based on markdown specific characters, MarkdownHeaderTextSplitter
- Code : many languages specifically python, js
- Token : Splits text based on tokens
- Character : based on a user defined character
- Semantic Chunker (experimental) : First split on sentences, then combines ones next to each other if they are semantically similar enough

Techniques - Evaluation

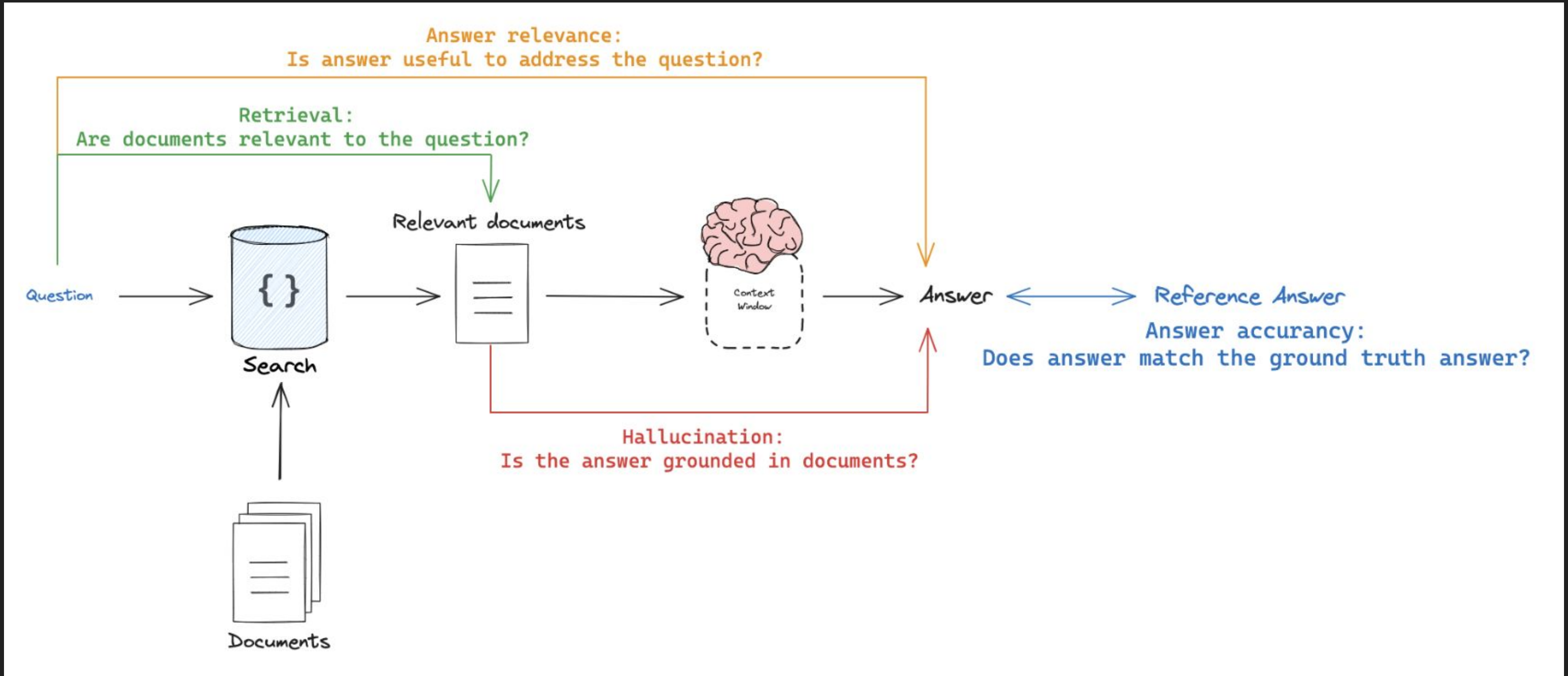
As we discuss, there are several metrics. LangSmith helps with the evaluation process in a few ways:

- it makes it easier to create and curate datasets via its tracing and annotation features
- provides an evaluation framework that helps you define metrics and run the app against the dataset
- allows to track results over time and implement CI/CD

Techniques - Evaluation

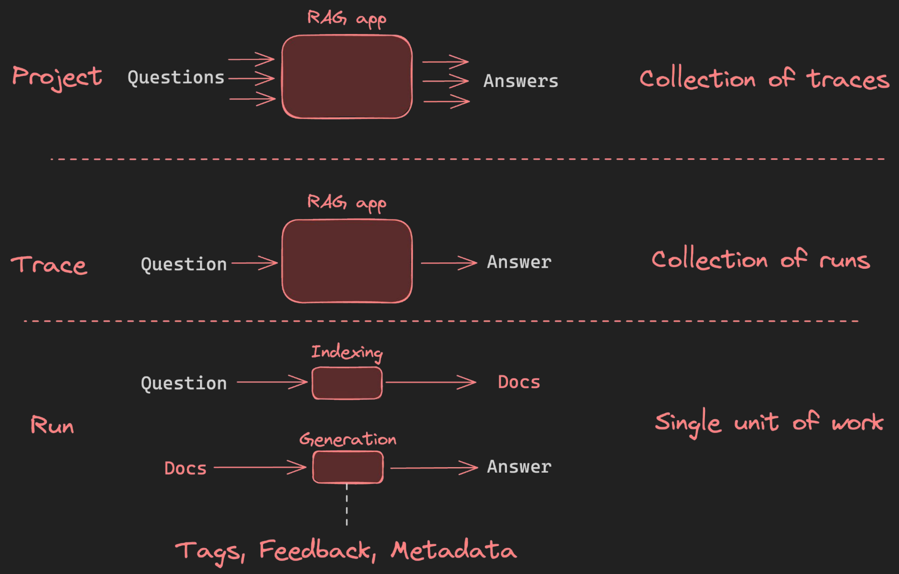


Techniques - Evaluation



Techniques - Tracing

A trace is essentially a series of steps that the app takes to go from input to output. Traces contain individual steps called **runs**. These can be individual calls from a model, retriever, tool or sub-chains



here is a more detailed [guide]