LLM week 2

Prompt Engineering

Prompt engineering is a discipline for developing and optimizing prompts to efficiently use LMs for a wide variety of applications and research topics. Since large language models rely on prompts to understand what they need to do, the way the task is described or framed can have a significant impact on the output.

- Clarity
- Relevance
- Progression
- Conciseness
- Adaptability

Text Generation (e.g., story writing, blog generation)

- **Clear context**: Provide a clear setup and tone for the output.
  - Example: “Write a short sci-fi story set in a futuristic city where robots coexist with humans.”
- **Guided structure**: Specify structure (e.g., beginning, middle, end) if needed.
  - Example: “Write a blog post with an introduction, three main points, and a conclusion about the benefits of remote work.”
- **Define style and tone**: Mention the tone (e.g., formal, humorous) explicitly.
  - Example: “Generate a humorous piece on the importance of sleep.”

Question-Answering (e.g., FAQs, chatbot design)

- **Concise and precise question framing**: Ensure the question is direct and unambiguous.
  - Example: Instead of asking, “What is the process for applying to graduate school?” ask “What are the steps to apply for a PhD program in the U.S.?”
- **Contextual clarification**: If multiple interpretations are possible, include extra context.
  - Example: “What is the capital of Turkey? (country, not state).”

Translation (e.g., language conversion, subtitling)

- **Specify translation style**: Clarify if the translation should be formal, colloquial, or literal.
    - Example: “Translate the following text into casual, everyday Spanish.”
- **Cultural awareness**: Address potential cultural differences in expressions or idioms.
    - Example: “Translate this text to French, ensuring idiomatic phrases are adapted to European French.”

Summarization (e.g., executive summaries, news briefs)

- **Length control**: Define the length of the summary (e.g., 100 words, one sentence).
    - Example: “Summarize this article in one sentence.”
- **Focus areas**: Specify what aspects to emphasize (e.g., data, conclusions).
    - Example: “Summarize the key findings of this research article, focusing on results.”

Sentiment Analysis (e.g., social media monitoring)

- **Context-specific wording**: Provide context for the text (e.g., product reviews, customer feedback).
    - Example: “Analyze the sentiment of the following customer feedback and classify it as positive, neutral, or negative.”

Multi-turn Dialogue (e.g., customer service bots)

- **Role-playing**: Set the role and define the context to maintain coherence across turns.
    - Example: “You are a customer support agent helping a user troubleshoot their internet connection. Keep responses short and informative.”

Challenges in Multilingual Prompt Engineering

Data Availability: The availability of high-quality and diverse linguistic data for training and fine-tuning AI models for several languages is a major roadblock in multilingual prompt engineering.

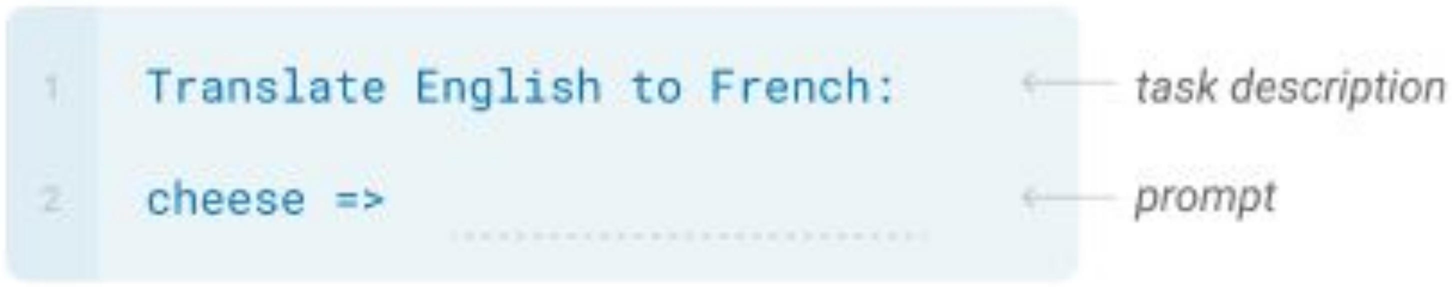
Linguistic Diversity: Linguistic diversity refers to the variation in languages and their usage among cultures. This can pose challenges for multilingual prompt engineering in various ways.

Computational complexity: Computational complexity is the amount of time and resources necessary to solve an issue, and this presents a hurdle in multilingual prompt engineering. Training a multilingual model can be computationally expensive because the model must be trained on data in different languages. This can be incredibly challenging for languages with limited available data.

Ensuring Fairness and Avoiding Bias:: Multilingual models may exhibit biases based on training data disparities.

Zero-shot Prompting

In **zero-shot prompting**, the model is expected to perform a task **without any examples**. The user provides only a prompt or instruction, and the model uses its learned knowledge to predict the answer. No gradient updates are performed.

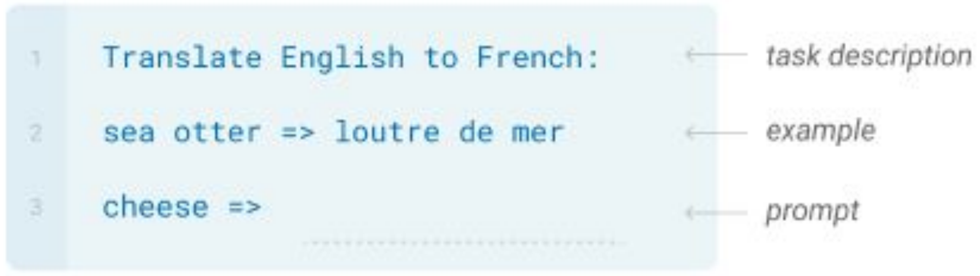


```
Translate English to French:
cheese =>
```

One-shot Prompting

In **one-shot prompting**, the model is provided with **one example** of the task along with the prompt. This helps the model understand the format of the expected answer and can improve its performance. Again, no gradient updates are performed.

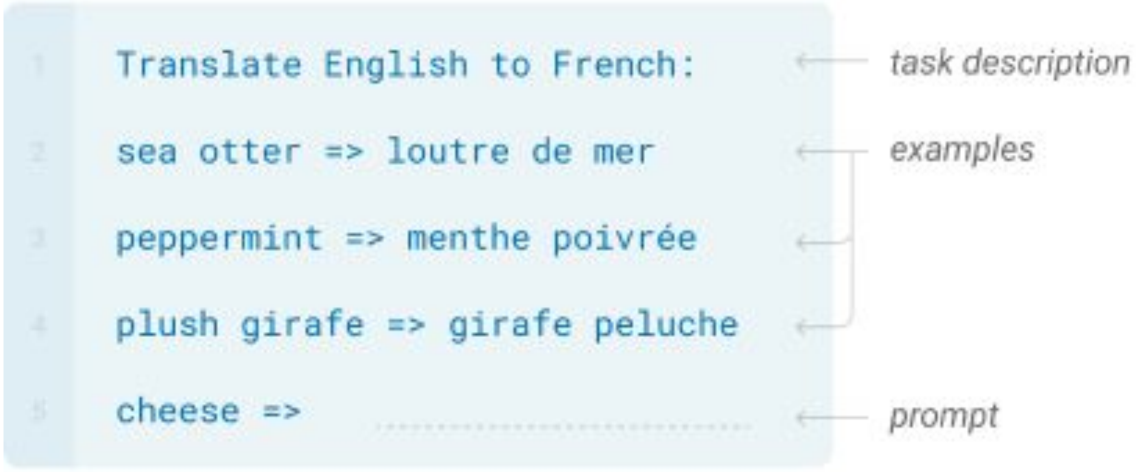
The reason for distinguishing one-shot from few-shot and zero-shot is that it most closely matches the way in which some tasks are communicated to humans.



```
Translate English to French:
sea otter => loutre de mer
cheese =>
```

Few-shot Prompting

In **few-shot prompting**, the model is provided with **a few examples(10 to 100 in the article)** of how the task should be performed. This provides more guidance and allows the model to better generalize how it should approach similar tasks. No gradient updates are performed.

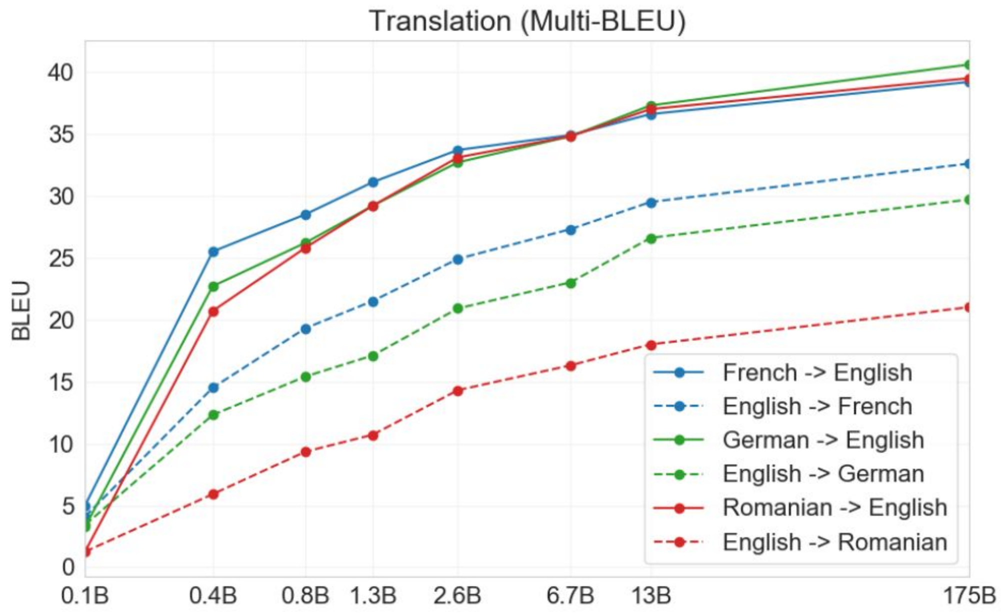


```
Translate English to French:
sea otter => loutre de mer
peppermint => menthe poivrée
plush girafe => girafe peluche
cheese =>
```

Few-shot Prompting

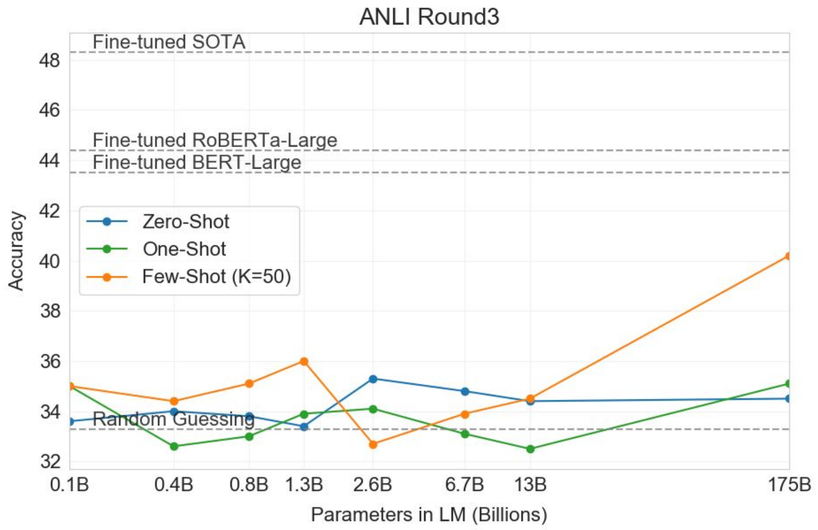
“The main advantages of few-shot are a major reduction in the need for task-specific data and reduced potential to learn an overly narrow distribution from a large but narrow fine-tuning dataset. “

“The main disadvantage is that results from this method have so far been much worse than state-of-the-art fine-tuned models.”

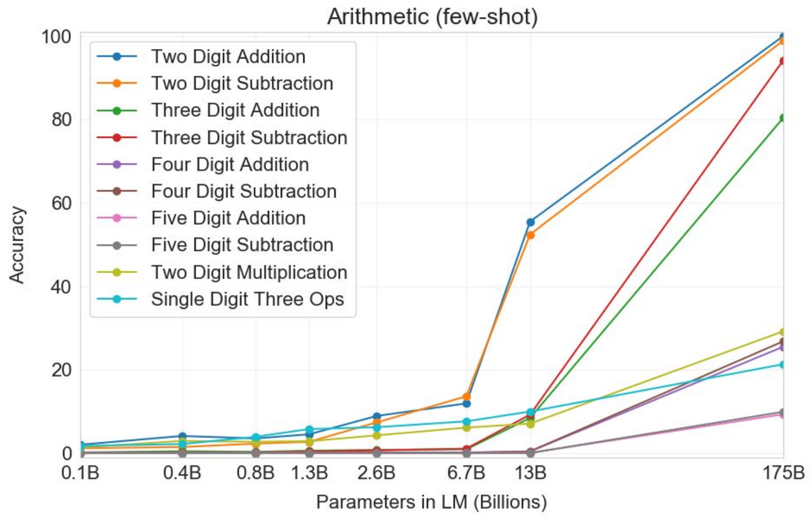


Setting	Winograd	Winogrande (XL)
Fine-tuned SOTA	**90.1**[a]	**84.6**[b]
GPT-3 Zero-Shot	88.3*	70.2
GPT-3 One-Shot	89.7*	73.2
GPT-3 Few-Shot	88.6*	77.7

NLI



ANLI Round3
Fine-tuned SOTA
Fine-tuned RoBERTa-Large
Fine-tuned BERT-Large
Random Guessing
Zero-Shot
One-Shot
Few-Shot (K=50)
Accuracy
48
46
44
42
40
38
36
34
32
0.1B
0.4B
0.8B
1.3B
2.6B
6.7B
13B
175B
Parameters in LM (Billions)



Arithmetic (few-shot)
Two Digit Addition
Two Digit Subtraction
Three Digit Addition
Three Digit Subtraction
Four Digit Addition
Four Digit Subtraction
Five Digit Addition
Five Digit Subtraction
Two Digit Multiplication
Single Digit Three Ops
Accuracy
100
80
60
40
20
0
0.1B
0.4B
0.8B
1.3B
2.6B
6.7B
13B
175B
Parameters in LM (Billions)

Instruction-Based Prompting

Instruction-based prompting is a method where you provide **explicit instructions** to guide a language model in performing a specific task. Instead of relying solely on examples (like in few-shot or one-shot prompting), instruction-based prompts focus on **clear, detailed directives** that inform the model about what to do.

‘is this sentiment positive or negative?’.

Metrics for evaluating the effectiveness of different prompting techniques

1. Relevance

This measures how well the output aligns with the intended goal of the prompt.

How to measure: Use human evaluators to rate outputs on a scale (e.g., 1–5) based on how relevant they are to the prompt’s objective.

2. Accuracy

For tasks with definitive correct answers, accuracy measures how often the prompt produces the correct output.

How to measure: Calculate the percentage of correct outputs over a test set.

3. Consistency

This assesses how consistently the prompt produces similar quality outputs for similar inputs.

How to measure: Use statistical measures like standard deviation of quality scores across multiple runs.

4. Coherence

This evaluates how well-structured and logical the outputs are.

How to measure: Human evaluators can rate outputs for coherence, or you can use automated metrics like perplexity for language tasks.

Standardizing Metrics for Consistent Comparison:

1. **Benchmark Datasets**: Using a set of standardized and widely-accepted datasets for each type of task ensures comparability across models and prompting techniques. For example, datasets like **GLUE** for text classification, **SQuAD** for question answering, **WMT** for translation, and **CNN/DailyMail** for summarization are commonly used.
2. **Unified Metrics**: Establishing common evaluation frameworks across tasks can provide a more consistent measure. For instance:
   - **Cross-task Accuracy or F1-Score**: Evaluating models on multiple tasks and reporting a weighted or averaged score can help standardize comparisons.
   - **Harmonic Mean of Perplexity and Diversity**: For generative models, balancing fluency (low perplexity) with output diversity can provide a more holistic view of performance.
   - **Latency and Computational Efficiency**: For real-time applications, measuring the time or computational resources (e.g., FLOPS, inference time) needed to generate a response can offer insights into efficiency across models and prompt techniques.

3. Evaluation Protocols: Standardizing evaluation protocols (e.g., how many prompt variants are tested, under what conditions) can help normalize results. For example:

- Always evaluate on the same task formulations (e.g., answering fact-based questions or generating creative content).
- Use a consistent number of examples for one-shot and few-shot prompts to avoid inconsistencies in comparison.

4. Cross-lingual and Cross-domain Comparisons: Standardizing metrics that account for performance across different languages (e.g., BLEU for multilingual translation) and domains (e.g., medical vs. financial text generation) can help in comparing model robustness in diverse applications.

Human evaluation

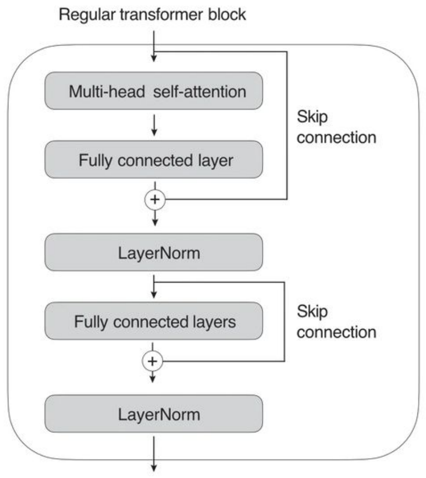
Human evaluation is essential for:

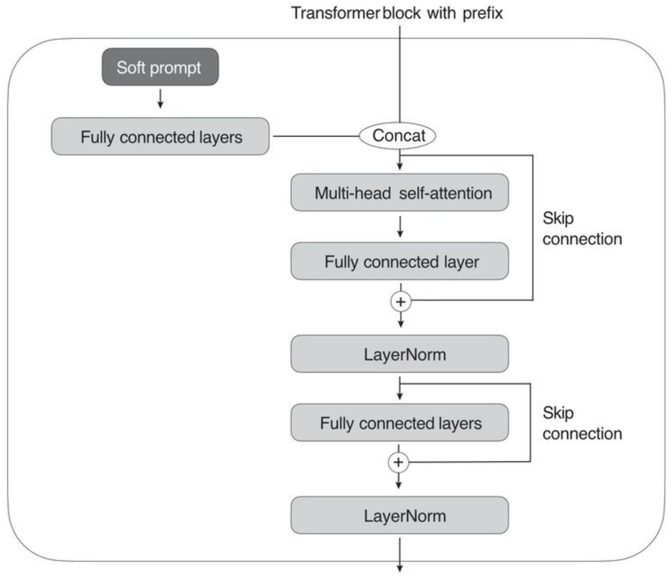
- **Subjective Metrics:** Assessing subjective qualities like relevance, coherence, and fluency.
- **Contextual Understanding:** Evaluating the model's ability to understand and respond to context-dependent prompts.
- **Bias Detection:** Identifying biases in the generated responses.
- **Refinement:** Providing feedback for improving prompts and models

Prefix-tuning

Prefix-tuning is a lightweight fine-tuning method designed to adapt pre-trained language models (LLMs) for specific tasks by learning continuous task-specific vectors that are prepended to the input, without modifying the underlying model's weights.

“an alternative to fine-tuning for conditional generation tasks.”





Advantages of Prefix Tuning

- It allows for efficient adaptation of pre-trained models to new tasks without modifying the original model's weights.
- It requires fewer trainable parameters compared to fine-tuning the entire model, making it more computationally efficient.
- It can be applied to any pre-trained transformer-based model without the need for task-specific architectures.
- Prefix-tuning allows for the independent training of tasks, enabling scalable personalisation without data cross-contamination.
- Each user's data can be isolated, and a ***personalised prefix can be trained for each user***, ensuring privacy and modularity. The independence of tasks also enables efficient batching across users and the creation of ensembles of multiple prefixes trained on the same task.

Limitations of Prefix Tuning

Difficulty in Handling Multi-Task Learning

Reduced Performance on Architectures with Few Parameters

Applicability to Multi-Modal Models

Chain-Of-Thought Prompting

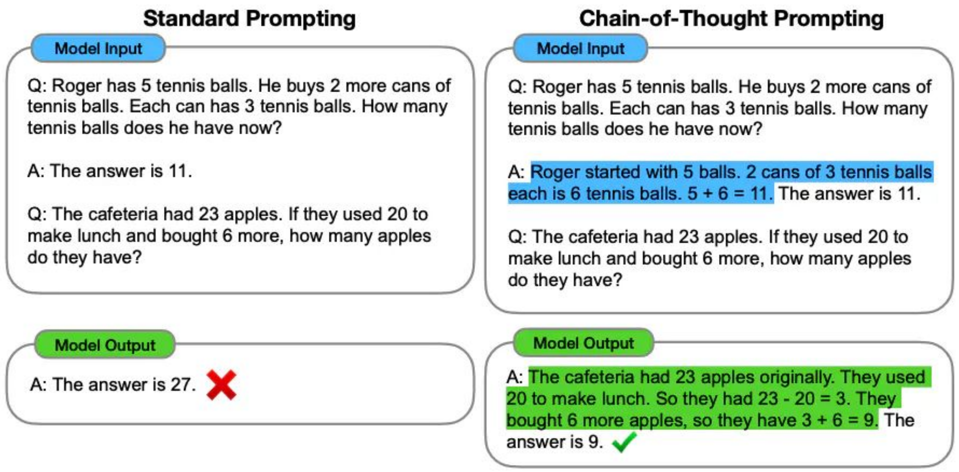
Focuses on reasoning before getting the final answer. Teaches the model with so little examples (even one) to think step by step.

Improves larger models <100B. No effect on smaller models.

Shows the reasoning, good for debugging.

Applicable to every task that can be solved by human language.

Most efficient in problems that require complex reasoning or multi step calculations. It outperforms traditional prompting significantly in such problems.



Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27. ✗

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

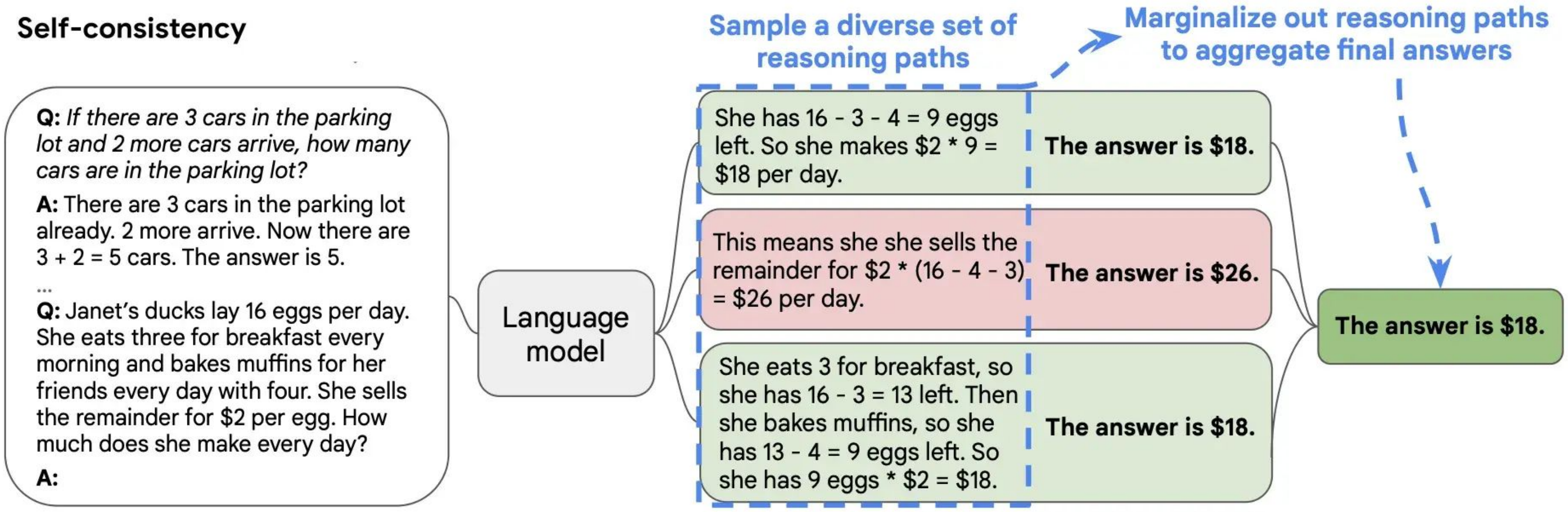
Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Self Consistency Prompting

in SCP, the model generates multiple outputs and checks how many of them overlaps in some contexts to identify most common responses.

Uses those most common responses to generate an answer.


Self-consistency
Q: If there are 3 cars in the parking lot and 2 more cars arrive, how many cars are in the parking lot?
A: There are 3 cars in the parking lot already. 2 more arrive. Now there are 3 + 2 = 5 cars. The answer is 5.
...
Q: Janet's ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the remainder for $2 per egg. How much does she make every day?
A:
Language model
Sample a diverse set of reasoning paths
She has 16 - 3 - 4 = 9 eggs left. So she makes $2 * 9 = $18 per day.
The answer is $18.
This means she she sells the remainder for $2 * (16 - 4 - 3) = $26 per day.
The answer is $26.
She eats 3 for breakfast, so she has 16 - 3 = 13 left. Then she bakes muffins, so she has 13 - 4 = 9 eggs left. So she has 9 eggs * $2 = $18.
The answer is $18.
Marginalize out reasoning paths to aggregate final answers
The answer is $18.

Automatic Prompt Generation

In this method, responses to automatically generated prompts are evaluated and the prompt that has the highest likelihood score is considered as the best prompt.

The evaluation of the responses has different metrics.

This automates the process of finding the best prompt.

Generates prompts that are optimized using model feedback, instead of human intuition.

It increases stability and generalizability of LLMs by creating tailored, contextual prompts

Contextual (Dynamic) Prompting

In contextual prompting, the model modifies incoming prompts in a way that it includes the context. So, model generates answers that fit better to the context.

It can be achieved by letting the model know the past prompts, updating the context with every prompt, so on.

What are the optimal strategies for implementing self-consistency in various types of tasks?

I found some of the strategies very interesting, so I wanted to add.

1. Classification Tasks **Approach**: Generate multiple predictions for each input and select the most frequent one.

Example **Input**: “Classify the sentiment of the following review: ‘The product is great!’”

1. “What is the sentiment of this review: ‘The product is great!’?”
2. “Determine the sentiment: ‘The product is great!’”
3. “How do you feel about this statement: ‘The product is great!’?”

- **Selection**: Count the occurrences of each sentiment (positive, negative, neutral) and choose the one with the highest count.

2. Open-Ended Question Answering Use synonyms and rephrasing to generate diverse outputs.

Question: “What are the causes of climate change?”

1. “List the factors contributing to climate change.”
2. “Explain the reasons behind climate change.”
3. “What leads to climate change?”

- **Evaluation**: Rank responses based on criteria such as completeness, accuracy, and clarity, and select the top answer.

3. Mathematical Problem Solving **Approach**: Ask the model to solve the problem using various techniques or representations (e.g., equations, diagrams).

Problem: “What is the solution to 2x + 5 = 15?”

1. “Solve for x in the equation 2x + 5 = 15.”
2. “Find x when 2x + 5 equals 15.”
3. “What value of x satisfies 2x + 5 = 15?”

- **Consistency Check**: Compare the solutions derived from different methods and confirm if they converge on the same value.

Retrieval-Augmented Prompting

Steps:

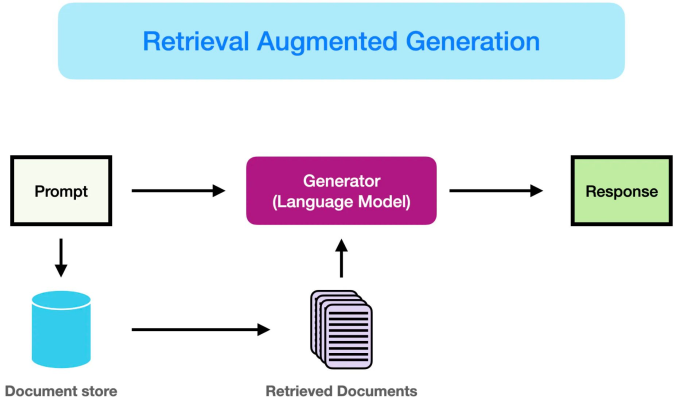
1. **Query**: A query that is formed based on the user's input or the specific task is used to search a knowledge base, such as Wikipedia, scientific literature, or a custom database.

2. **Retrieval**: A retrieval engine (such as a search engine or information retrieval system) searches through the knowledge base and retrieves relevant documents, facts, or information snippets.

3. **Augmented Prompt Creation**: The retrieved information is combined with the original query to create an augmented prompt. This augmented prompt is then passed to the LLM for processing.

4. **Model Generation**: The model uses the augmented prompt, which includes both the query and the retrieved context, to generate a more accurate and informative output.

The image is almost correct. Its RAG but we need RAP

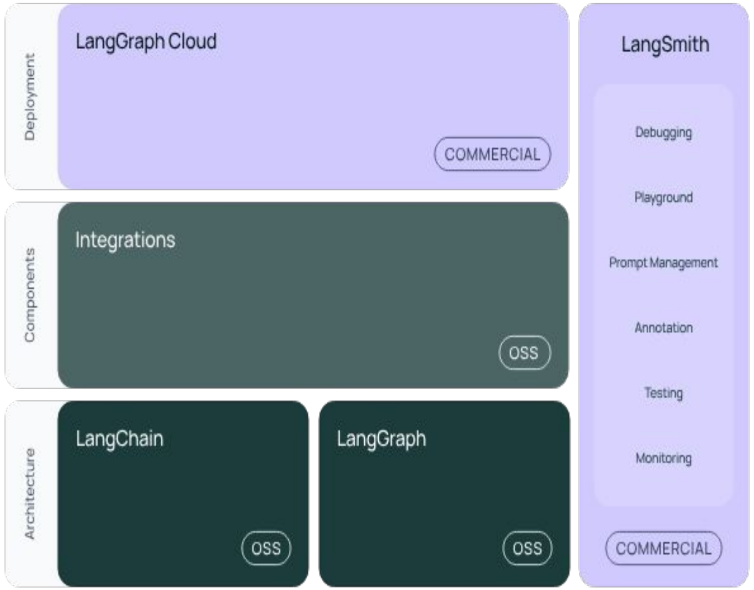


LangChain

LangChain is a framework for developing applications powered by large language models (LLMs). It provides tools to easily chain various LLM components. So we can create complex workflows, automation and integration. This makes it easier to build apps like chatbots, auto agents etc..

LangChain simplifies every stage of the LLM application lifecycle:

- Development: Build your applications using LangChain's open-source building blocks, components, and third-party integrations. Use LangGraph to build stateful agents with first-class streaming and human-in-the-loop support.
- Productionization: Use LangSmith to inspect, monitor and evaluate your chains, so that you can continuously optimize and deploy with confidence.
- Deployment: Turn your LangGraph applications into production-ready APIs and Assistants with LangGraph Cloud.



the framework consists of the following open-source libraries:

- `langchain-core`: Base abstractions and LangChain Expression Language. Provides basic functionality.
- `langchain-community`: Third party integrations.
  - Partner packages (e.g. `langchain-openai`, `langchain-anthropic`, etc.): Some integrations have been further split into their own lightweight packages that only depend on `langchain-core`.
- `langchain`: Chains, agents, and retrieval strategies that make up an application's cognitive architecture. Full framework that builds lc-core and extra tools beyond core capabilities such as APIs, databases or services
- LangGraph: Build robust and stateful multi-actor applications with LLMs by modeling steps as edges and nodes in a graph. Integrates smoothly with LangChain, but can be used without it. It orchestrates LC workflows where different components (lang models, tools, memory etc) chained together. It visualizes how data moves between these components.
- LangServe: Deploy LangChain chains as REST APIs.
- LangSmith: A developer platform that lets you debug, test, evaluate, and monitor LLM applications.

LangChain Expression Language (LCEL)

LangChain Expression Language, or LCEL, is a declarative way to chain LangChain components. LCEL was designed to support putting prototypes in production, with no code changes.

LCEL aims to provide consistency around behavior and customization over legacy subclassed chains such as LLMChain and ConversationalRetrievalChain.

Runnable interface

Many LangChain components implement the `Runnable` protocol, including chat models, LLMs, output parsers, retrievers, prompt templates, and more.

The standard interface includes:

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

Component	Input Type	Output Type
Prompt	Dictionary	PromptValue
ChatModel	Single string, list of chat messages or a PromptValue	ChatMessage
LLM	Single string, list of chat messages or a PromptValue	String
OutputParser	The output of an LLM or ChatModel	Depends on the parser
Retriever	Single string	List of Documents
Tool	Single string or dictionary, depending on the tool	Depends on the tool

Components

- Chat models
- LLMs
- Messages
- Prompt templates
- Example selectors
- Output parsers
- Chat history
- Documents
- Callbacks
- Document loaders
- Text splitters
- Embedding models
- Vector stores
- Retrievers
- Key-value stores
- Tools
- Toolkits Agents

Chat models

Language models that use a sequence of messages as inputs and return chat messages as outputs.

Chat models support the assignment of distinct roles to conversation messages, helping to distinguish messages from the AI, users, and instructions such as system messages.

Although the underlying models are messages in, message out, the LangChain wrappers also allow these models to take a string as input. This means you can easily use chat models in place of LLMs.

When a string is passed in as input, it is converted to a HumanMessage and then passed to the underlying model.

LangChain does not host any Chat Models, rather it rely on third party integrations.

LLMs

Language models that takes a string as input and returns a string. These are traditionally older models

the LangChain wrappers also allow these models to take messages as input. This gives them the same interface as Chat Models. When messages are passed in as input, they will be formatted into a string under the hood before being passed to the underlying model.

LangChain does not host any LLMs, rather it rely on third party integrations.

Messages

Some language models take a list of messages as input and return a message. There are a few different types of messages. All messages have a role, content, and response_metadata property.

The role describes WHO is saying the message. The standard roles are "user", "assistant", "system", and "tool". LangChain has different message classes for different roles.

The content property describes the content of the message. This can be a few different things:

- A string (most models deal with this type of content)
- A List of dictionaries (this is used for multimodal input, where the dictionary contains information about that input type and that input location)

messages can have a name property which allows for differentiating between multiple speakers with the same role.

Prompt templates

Prompt templates help to translate user input and parameters into instructions for a language model. This can be used to guide a model's response, helping it understand the context and generate relevant and coherent language-based output.

Prompt Templates take as input a dictionary, where each key represents a variable in the prompt template to fill in.

Prompt Templates output a PromptValue. This PromptValue can be passed to an LLM or a ChatModel, and can also be cast to a string or a list of messages. The reason this PromptValue exists is to make it easy to switch between strings and messages.

Example selectors

One common prompting technique for achieving better performance is to include examples as part of the prompt. This is known as few-shot prompting. This gives the language model concrete examples of how it should behave. Example Selectors are classes responsible for selecting and then formatting examples into prompts.

Output parsers

Output parser is responsible for taking the output of a model and transforming it to a more suitable format for downstream tasks. Useful when you are using LLMs to generate structured data, or to normalize output from chat models and LLMs.

Chat history

Most LLM applications have a conversational interface. An essential component of a conversation is being able to refer to information introduced earlier in the conversation. At bare minimum, a conversational system should be able to access some window of past messages directly.

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.

Documents

A Document object in LangChain contains information about some data. It has two attributes:

- `page_content: str`: The content of this document. Currently is only a string.
- `metadata: dict`: Arbitrary metadata associated with this document. Can track the document id, file name, etc.

Document Loaders

These classes load Document objects. LangChain has hundreds of integrations with various data sources to load data from: Slack, Notion, Google Drive, etc.

Each DocumentLoader has its own specific parameters, but they can all be invoked in the same way with the .load method.

Text splitters

LangChain has a number of built-in document transformers that make it easy to split, combine, filter, and otherwise manipulate documents.

At a high level, text splitters work as following:

1. Split the text up into small, semantically meaningful chunks (often sentences).
2. Start combining these small chunks into a larger chunk until you reach a certain size (as measured by some function).
3. Once you reach that size, make that chunk its own piece of text and then start creating a new chunk of text with some overlap (to keep context between chunks).

That means there are two different axes along which you can customize your text splitter:

1. How the text is split
2. How the chunk size is measured

Embedding models

Embedding models create a vector representation of a piece of text.

The Embeddings class is a class designed for interfacing with text embedding models. There are many different embedding model providers (OpenAI, Cohere, Hugging Face, etc) and local models, and this class is designed to provide a standard interface for all of them.

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).

Vector stores

One of the most common ways to store and search over unstructured data is to embed it and store the resulting embedding vectors, and then at query time to embed the unstructured query and retrieve the embedding vectors that are 'most similar' to the embedded query. A vector store takes care of storing embedded data and performing vector search.

Most vector stores can also store metadata about embedded vectors and support filtering on that metadata before similarity search, allowing you more control over returned documents.

Vector stores can be converted to the retriever interface as we did in our RAG applications.

Retrievers

A retriever is an interface that returns documents given an unstructured query. It is more general than a vector store. A retriever does not need to be able to store documents, only to return (or retrieve) them. Retrievers can be created from vector stores, but are also broad enough to include Wikipedia search and Amazon Kendra.

Retrievers accept a string query as input and return a list of Document's as output.

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 (KV) storage is helpful.

LangChain includes a BaseStore interface, which allows for storage of arbitrary data. However, LangChain components that require KV-storage accept a more specific BaseStore[str, bytes] instance that stores binary data (referred to as a ByteStore), and internally take care of encoding and decoding data for their specific needs.

Tools

Tools are utilities designed to be called by a model: their inputs are designed to be generated by models, and their outputs are designed to be passed back to models. Tools are needed whenever you want a model to control parts of your code or call out to external APIs.

A tool consists of:

- The name of the tool.
- A description of what the tool does.
- A JSON schema defining the inputs to the tool.
- A function (and, optionally, an async variant of the function).

Toolkits

Toolkits are collections of tools that are designed to be used together for specific tasks. They have convenient loading methods.

All Toolkits expose a get_tools method which returns a list of tools.

Agents

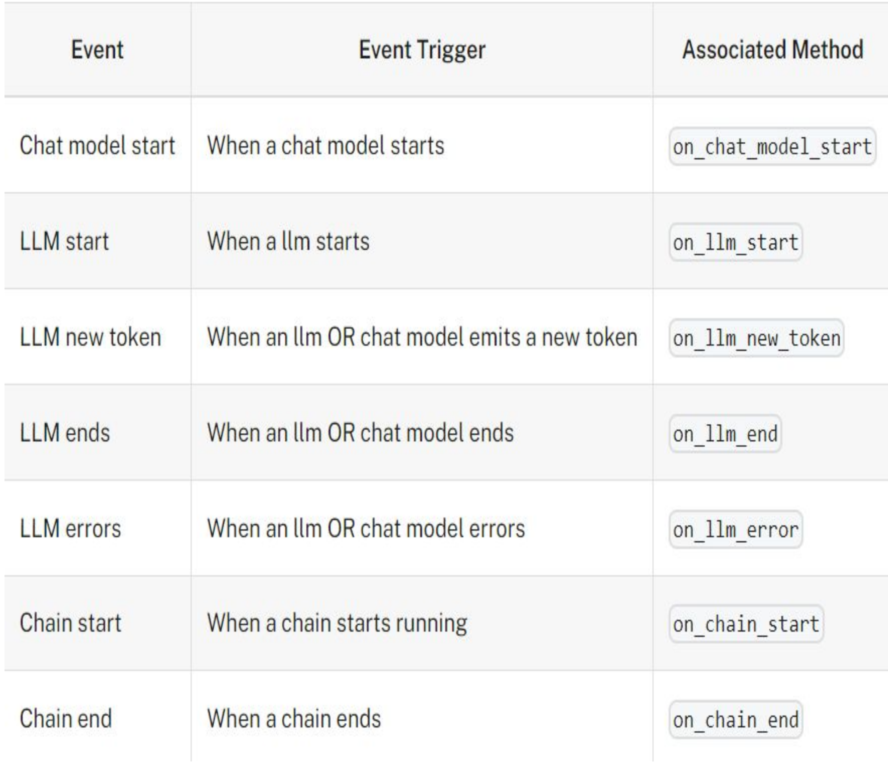
By themselves, language models can't take actions - they just output text. A big use case for LangChain is creating agents. Agents are systems that use an LLM as a reasoning engine to determine which actions to take and what the inputs to those actions should be. The results of those actions can then be fed back into the agent and it determine whether more actions are needed, or whether it is okay to finish.

LangGraph is an extension of LangChain specifically aimed at creating highly controllable and customizable agents

Callbacks

LangChain provides a callbacks system that allows you to hook into the various stages of your LLM application. This is useful for logging, monitoring, streaming, and other tasks.

You can subscribe to these events by using the callbacks argument available throughout the API.



Event	Event Trigger	Associated Method
Chat model start	When a chat model starts	`on_chat_model_start`
LLM start	When a llm starts	`on_llm_start`
LLM new token	When an llm OR chat model emits a new token	`on_llm_new_token`
LLM ends	When an llm OR chat model ends	`on_llm_end`
LLM errors	When an llm OR chat model errors	`on_llm_error`
Chain start	When a chain starts running	`on_chain_start`
Chain end	When a chain ends	`on_chain_end`

Techniques

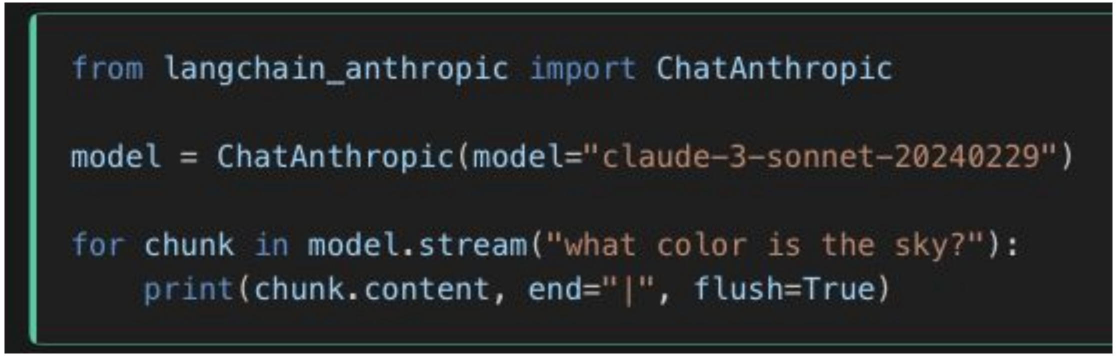
- Streaming
- Function/tool calling
- Structured output
- Few-shot prompting
- Retrieval
- Text splitting
- Evaluation
- Tracing

Streaming

Individual LLM calls run for much longer than traditional requests.

Fortunately, LLMs generate output iteratively, which means it's possible to show intermediate results before the final response is ready.

Consuming output as soon as it becomes available has therefore become a vital part.

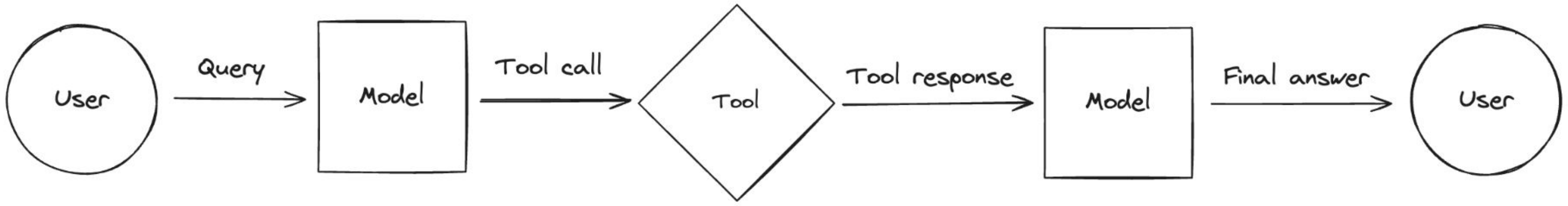


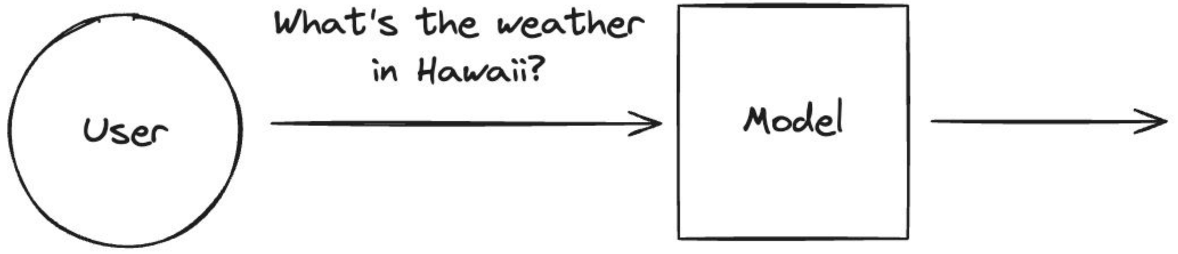
```python
from langchain_anthropic import ChatAnthropic

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

for chunk in model.stream("what color is the sky?"):
    print(chunk.content, end="|", flush=True)
```

Function / tool Calling

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

While the name implies that the model is performing some action, this is actually not the case! The model only generates the arguments to a tool, and actually running the tool (or not) is up to the user.



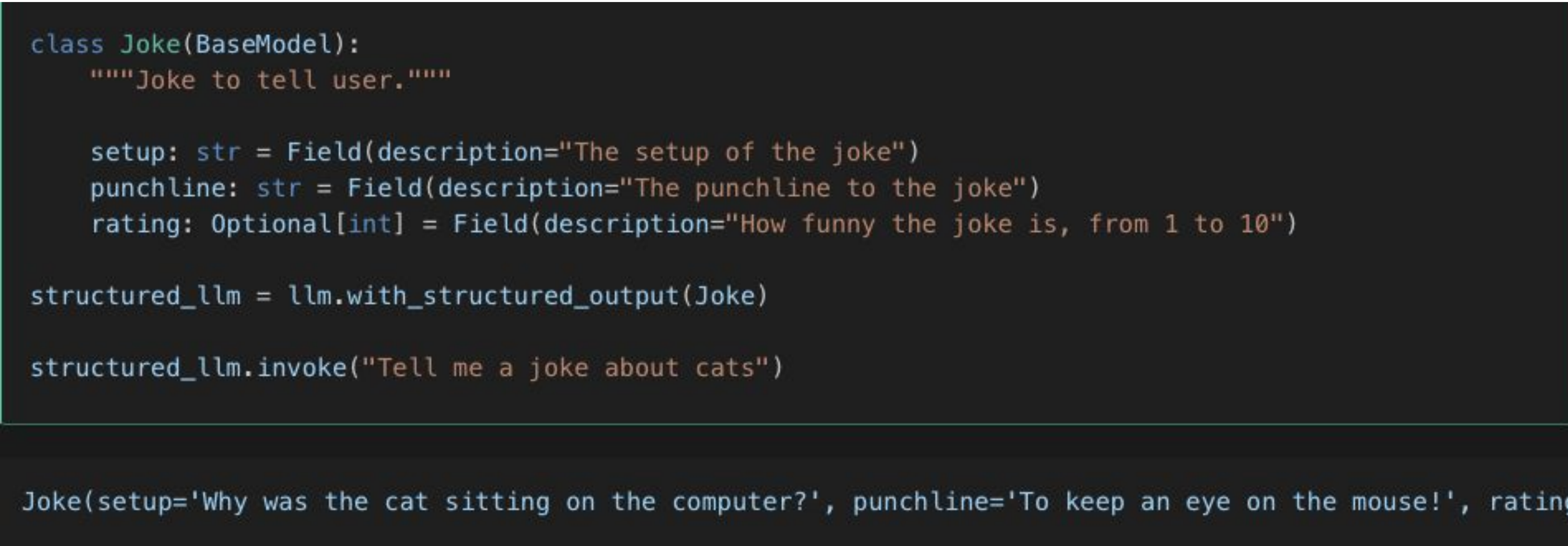


```
AIMessage(
  tool_calls=[{
    name: "get_weather"
    args: {
      location: "Hawaii",
    },
    id: "call_abc123",
    type: "tool_call"
  }]
)
```

Sometimes we might want to have only this output if we want that structured output, not a complete answer

Structured Output

LLMs are capable of generating arbitrary text. This enables the model to respond appropriately to a wide range of inputs, but for some use-cases, it can be useful to constrain the LLM's output to a specific format or structure. This is referred to as structured output.

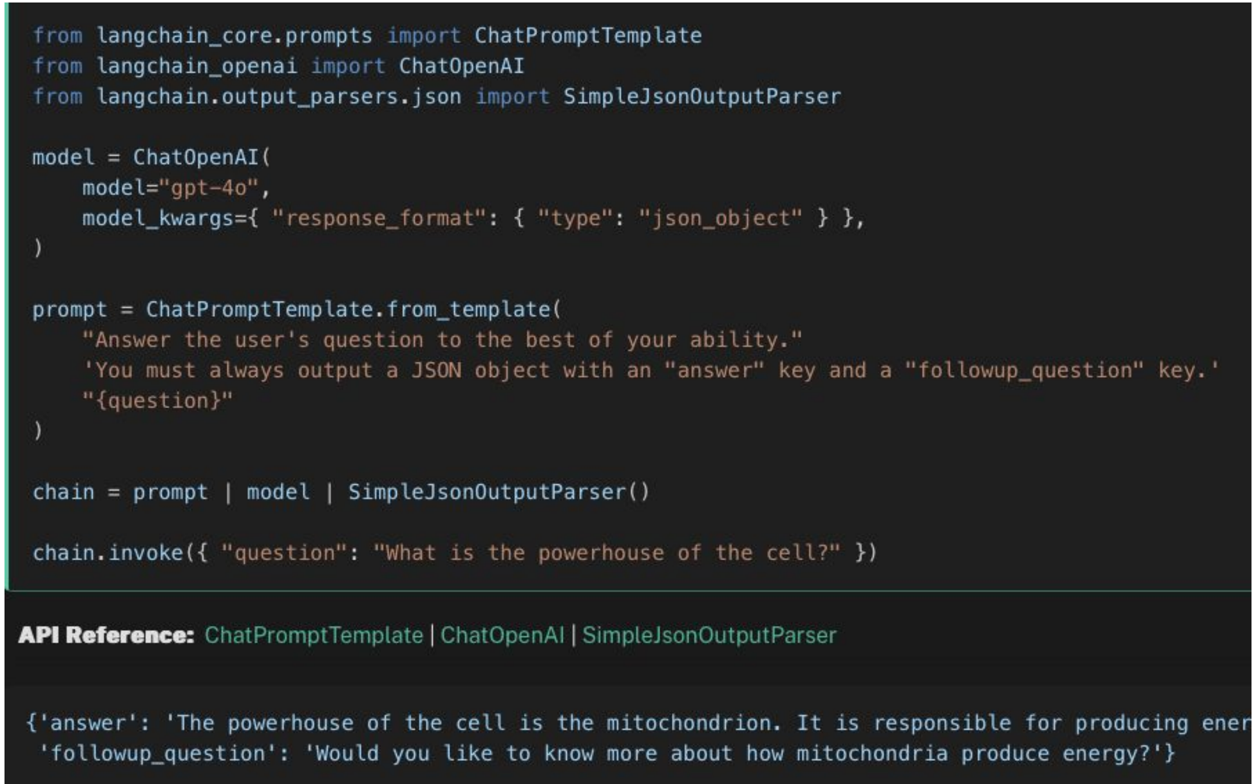


```python
class Joke(BaseModel):
    """Joke to tell user."""

    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")
```

```
Joke(setup='Why was the cat sitting on the computer?', punchline='To keep an eye on the mouse!', ratin
```

JSON Mode

In scope of structured output, we might ask models to give JSON answers only.
Most models support this by enabling in configuration.



```python
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.output_parsers.json import SimpleJsonOutputParser

model = ChatOpenAI(
    model="gpt-4o",
    model_kwargs={ "response_format": { "type": "json_object" } },
)

prompt = ChatPromptTemplate.from_template(
    "Answer the user's question to the best of your ability."
    'You must always output a JSON object with an "answer" key and a "followup_question" key.'
    "{question}"
)

chain = prompt | model | SimpleJsonOutputParser()

chain.invoke({ "question": "What is the powerhouse of the cell?" })
```

API Reference: ChatPromptTemplate | ChatOpenAI | SimpleJsonOutputParser

```
{'answer': 'The powerhouse of the cell is the mitochondrion. It is responsible for producing ener
 'followup_question': 'Would you like to know more about how mitochondria produce energy?'}
```

Evaluation

Evaluation is the process of assessing the performance and effectiveness of your LLM-powered applications. It involves testing the model's responses against a set of predefined criteria or benchmarks to ensure it meets the desired quality standards and fulfills the intended purpose. This process is vital for building reliable applications.

LangSmith helps with this process in a few ways:

- It makes it easier to create and curate datasets via its tracing and annotation features
- It provides an evaluation framework that helps you define metrics and run your app against your dataset
- It allows you to track results over time and automatically run your evaluators on a schedule or as part of CI/Code

Tracing

A trace is essentially a series of steps that your application 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. Tracing gives you observability inside your chains and agents, and is vital in diagnosing issues.