Lecture 8.1- LLM Components

Generative AI Teaching Kit

This lecture

- Semi-supervised and Few Shot Learning
- Prompting and Prompt templating
- The ChatGPT Pipeline
- LLM Components and Chains

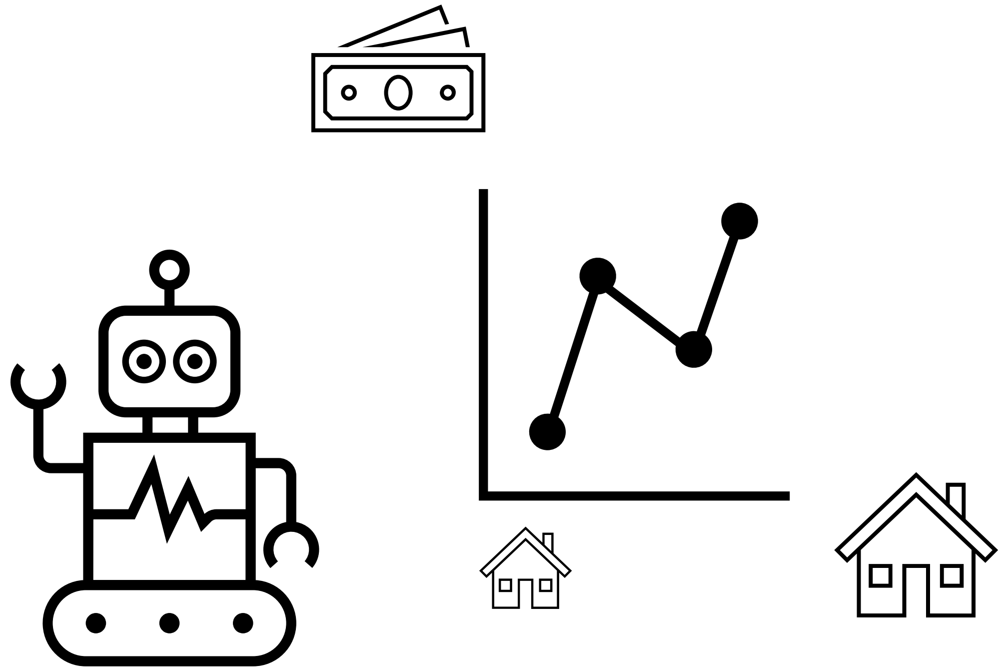
GPTs and Few Shot Learning

The breakthrough that changed the world

How training a model used to work

<u>Traditional Machine Learning</u>

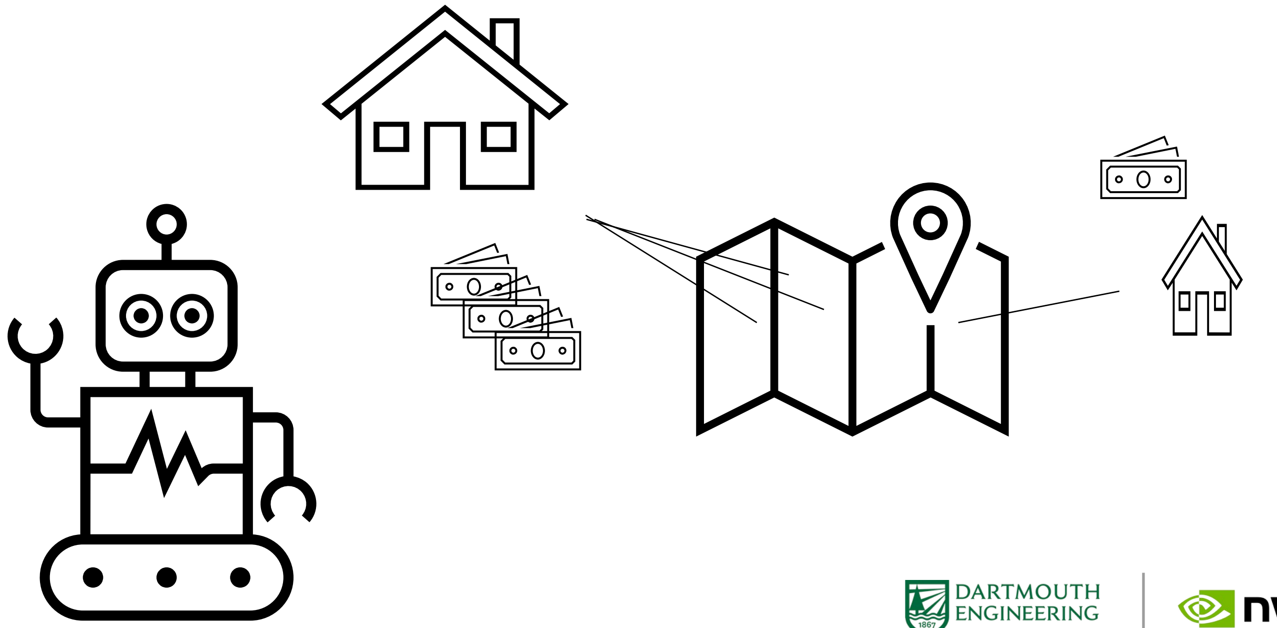
- Gather data on specific problem
  - Predicting house prices
  - Finding similarly priced houses
- Build a **<u>single</u>** model to predict/classify/cluster based on data



Model A

<u>One Model : One Task</u>

- Train on specific task
- Need data for that specific task
- Useful only for that task once trained



Model B

Supervised vs. Unsupervised Learning

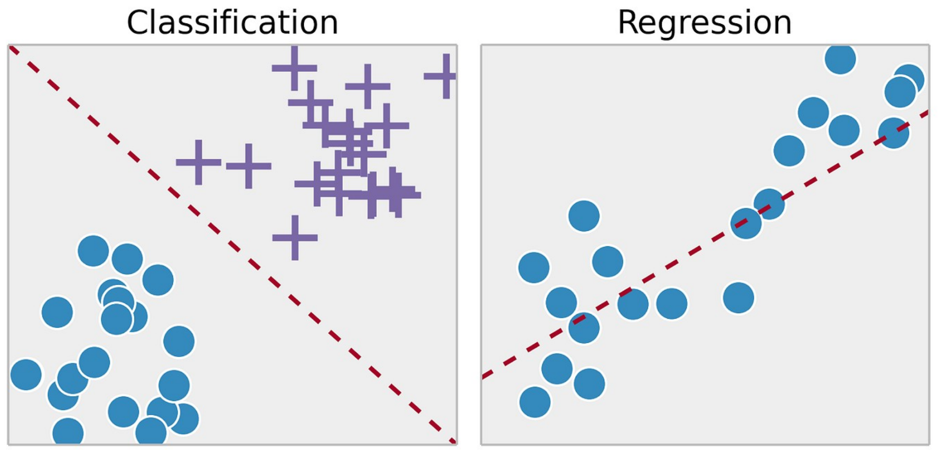
Data and modeling in traditional Machine Learning comes in two flavors:

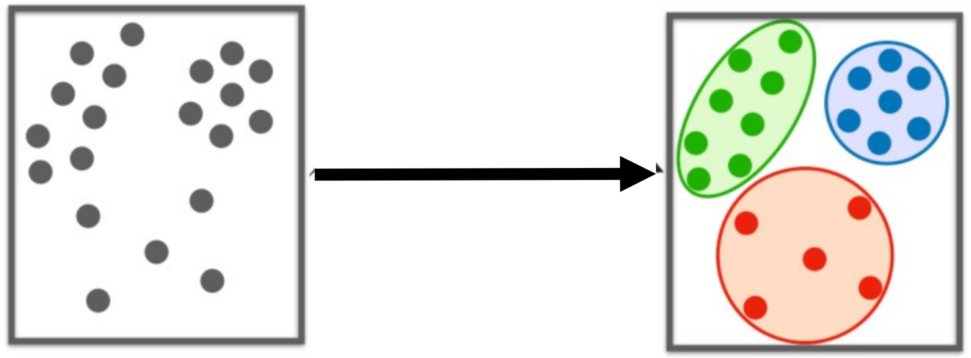
Labeled/Supervised and Unlabeled/Unsupervised

Labeled data allows supervised learning to find a relationship between input data and the label/value

Example methods:

- Linear/Logistic Regression
- Random Forests
- Deep Learning





Unlabeled data cannot be used to find the same kinds of relationships, but can be used for relative similarities and differences to be found in the data itself

The Problem with Unsupervised Learning

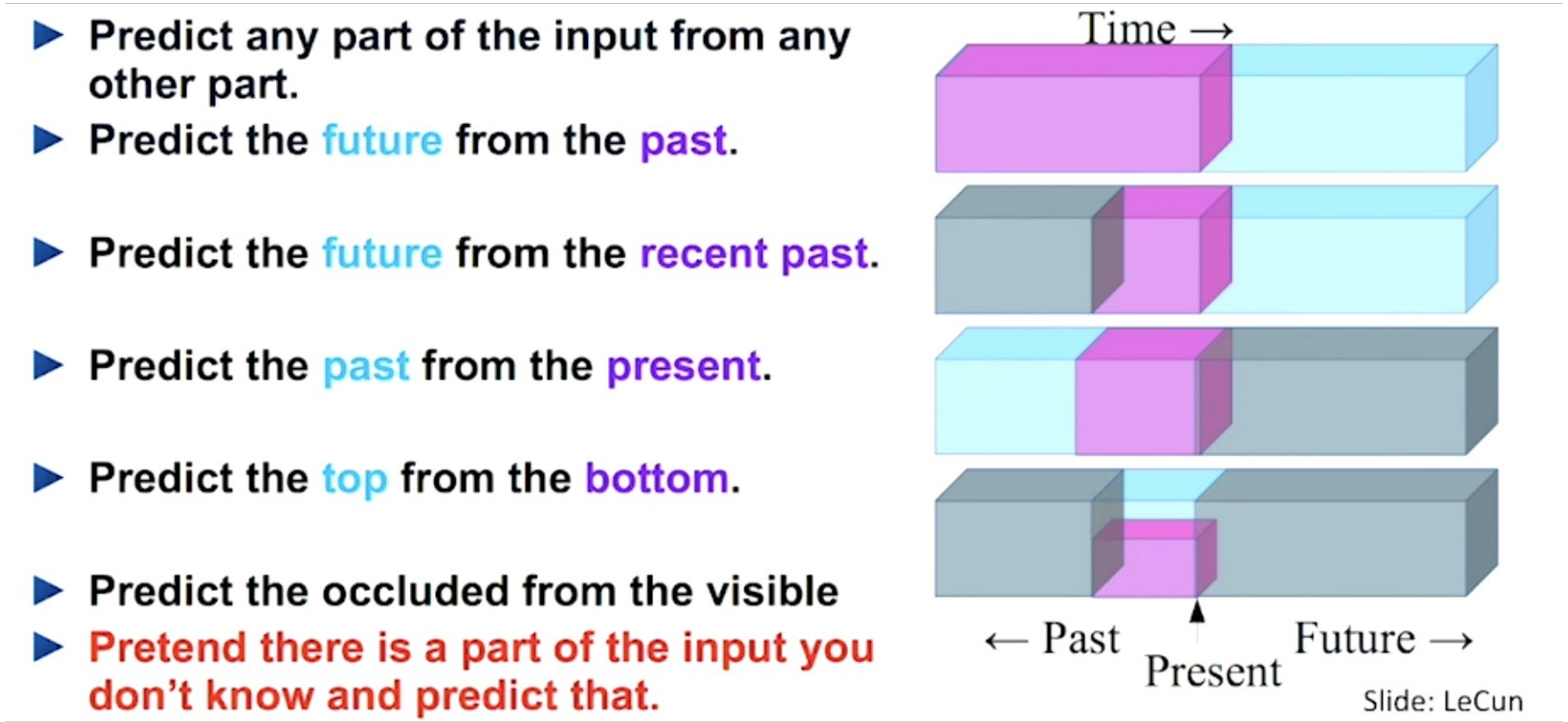
- Labeling data is **hard**, and **costly**
- The vast majority of data is **not** labeled:
  - Text
  - Images
  - Sensor data
  - Video footage
  - Transactions
  - Medical records, etc.
- Unsupervised learning cannot offer as much utility as supervised learning
  - No Prediction
  - No Classification
  - Only similarity/clustering

How can we use all of this data?

Self-supervised Learning – Self taught

What if we used the data as its own label?

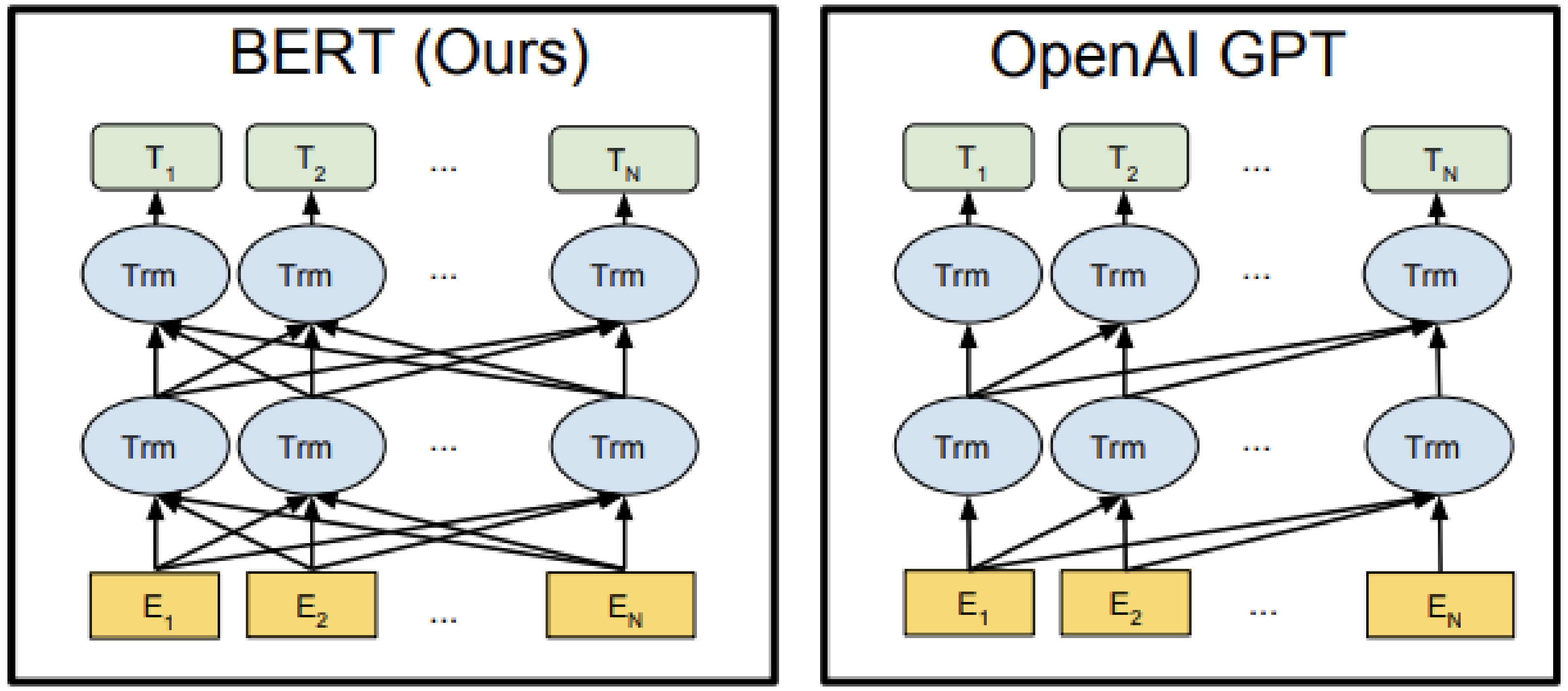
Self-supervised learning masks different parts of the data and treats this as input/output pairs.



Autoregressive Learning – Learning to speak

Training a transformer recap

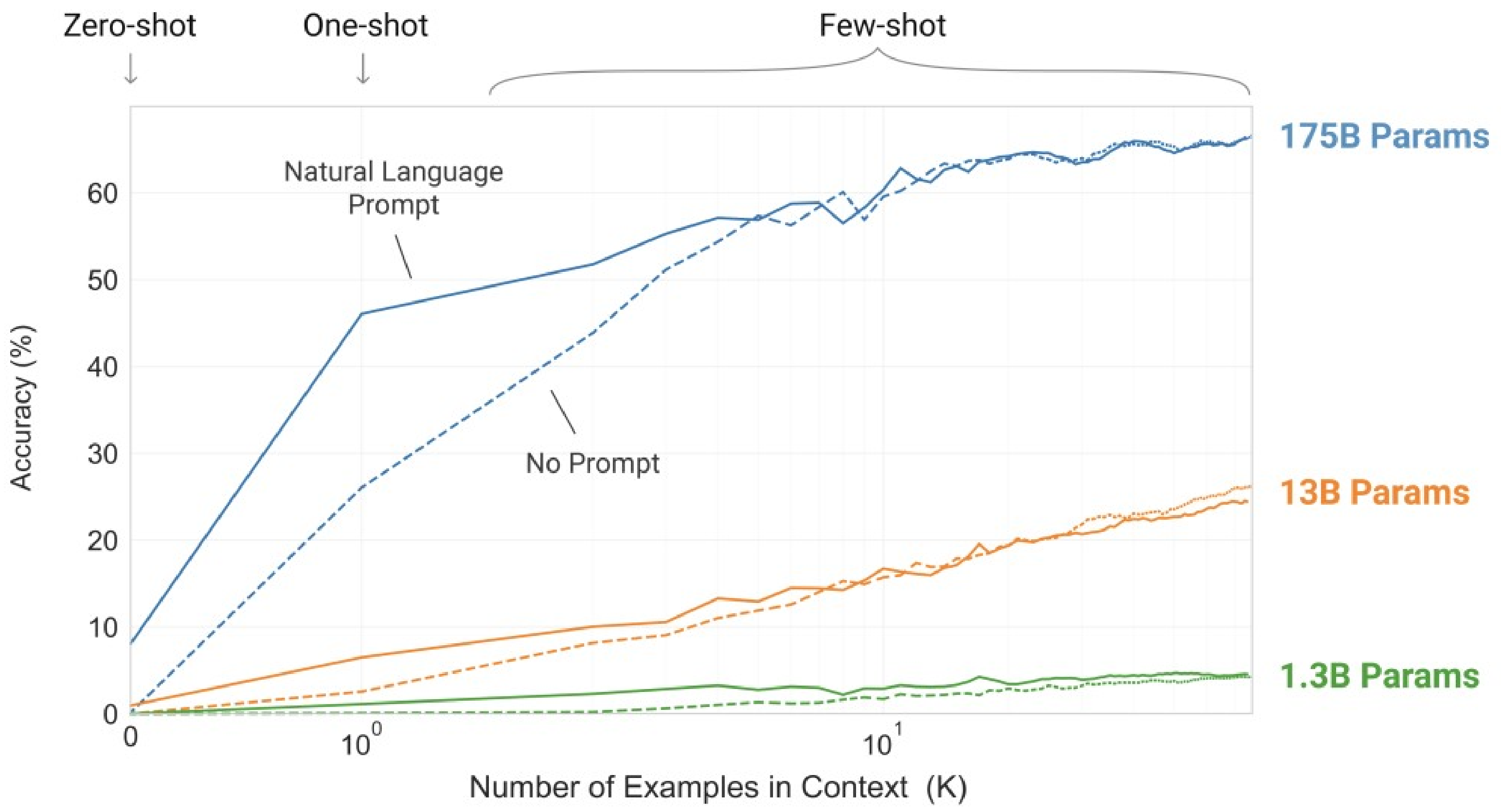
- Transformers like BERT/GPT use the unlabeled text data as the sources of both input and output.
- In Autoregressive training, and in Masked Language Modeling the model aims to fill in the missing (or next) token.
- It uses the attention mechanism to add extra context to determine what the right token should be.
- This means no labels are needed and all kinds of data can be used from text, to images, to speech, to video!



“Language Models are Few Shot Learners”

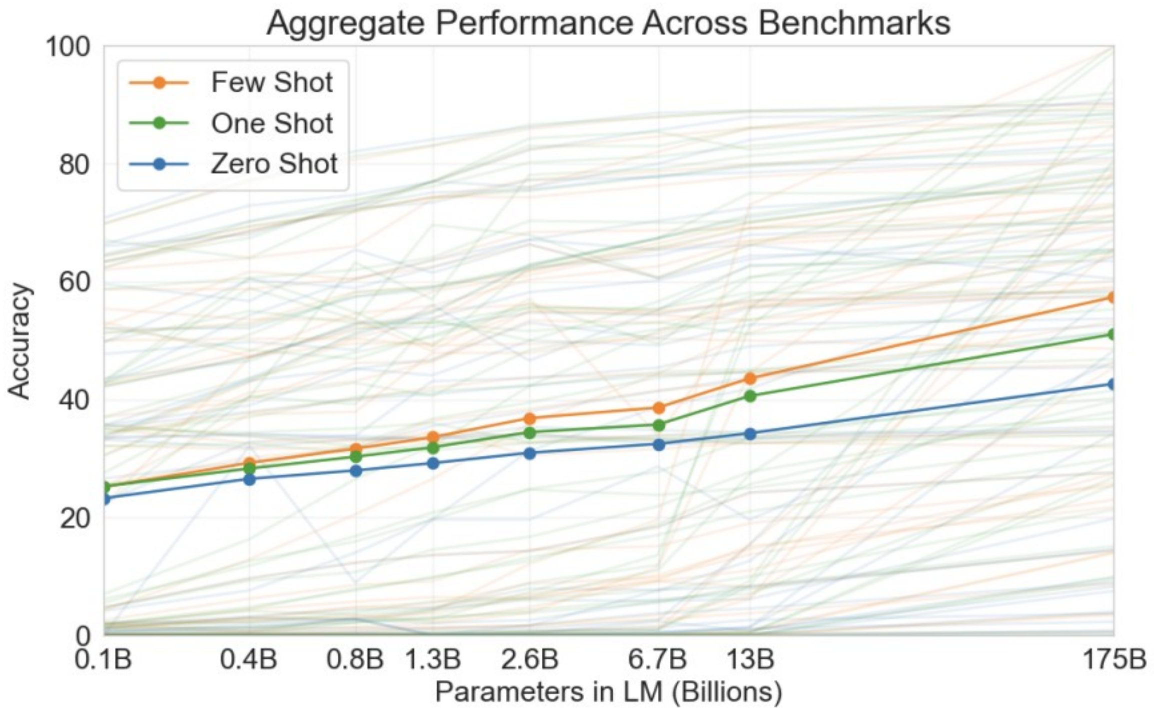
GPT-3 was trained on Billions of tokens of text in a semi-supervised manner. This dataset was a mixture of text sources from the internet.

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4



With such a vast parameter count, a model this size had not been built before on unlabeled data, and a new phenomenon was discovered, **Few Shot Learning.**

Few Shot Learning is when, as the input to the model, a number of samples (or “shots”) of the desired input<=>output behavior is given, before providing the example that the model needs to work on.



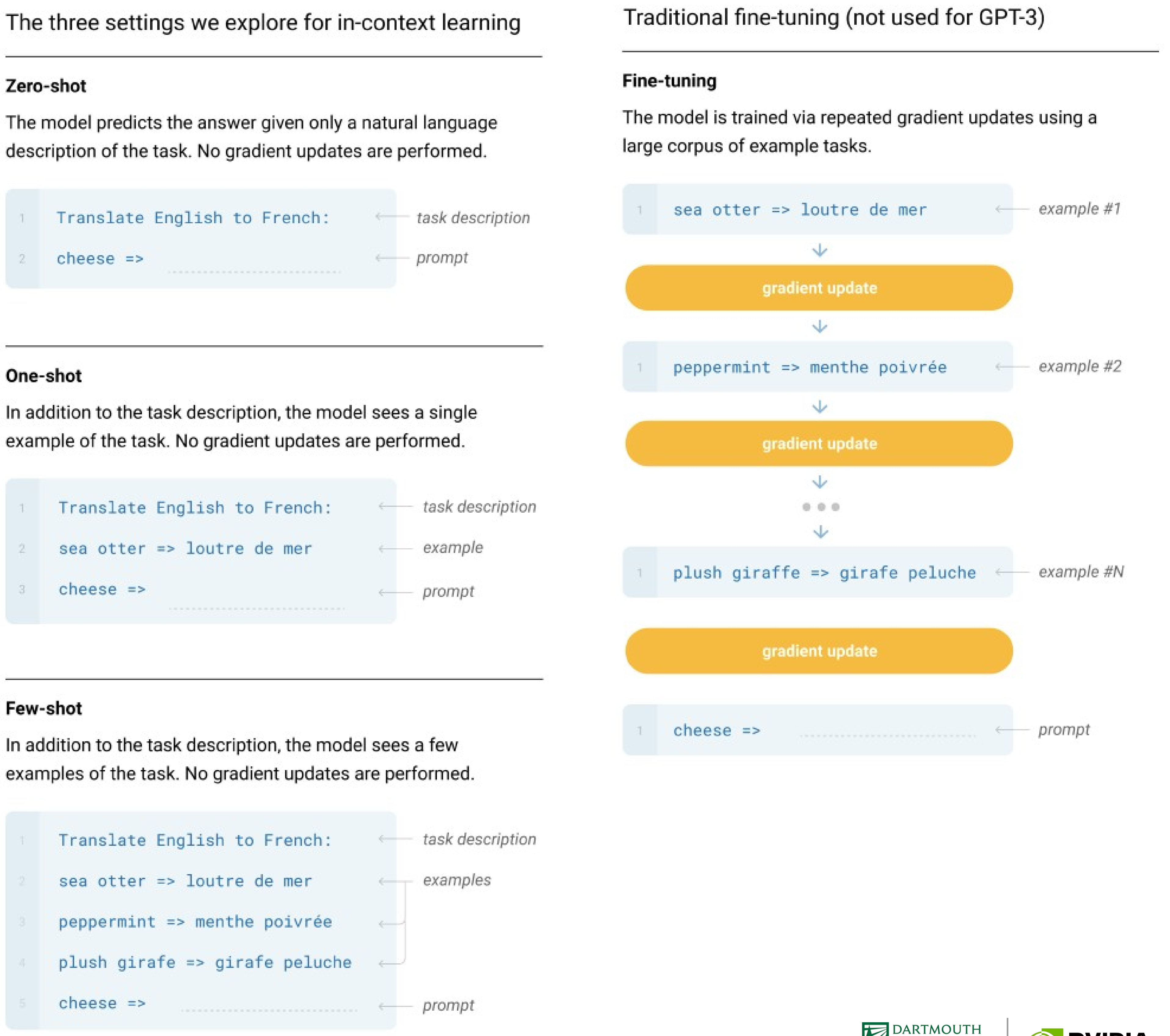
This was found to have significant effect on the performance of GPT-3 in tasks it was not trained on. It was only trained to predict the next token based on the tokens it had seen in the training corpus.

In-Context Learning vs. Fine-tuning

Few Shot Learning is an example of “**in-context learning**” where the transformer uses its vast set of parameters and attention matrices to focus on the input given and attempts to ”understand” what the task is.

This is in stark contrast to something like fine-tuning, or supervised learning where the task is structured in the input-output pairs and the mode simply learns the relationship between them.

This flexibility of LLMs to seemingly learn new tasks after training has allowed for a new industry to be formed, Generative AI. The fundamental concept we will need to understand then, is how to prompt these models.



DARTMOUTH ENGINEERING NVIDIA

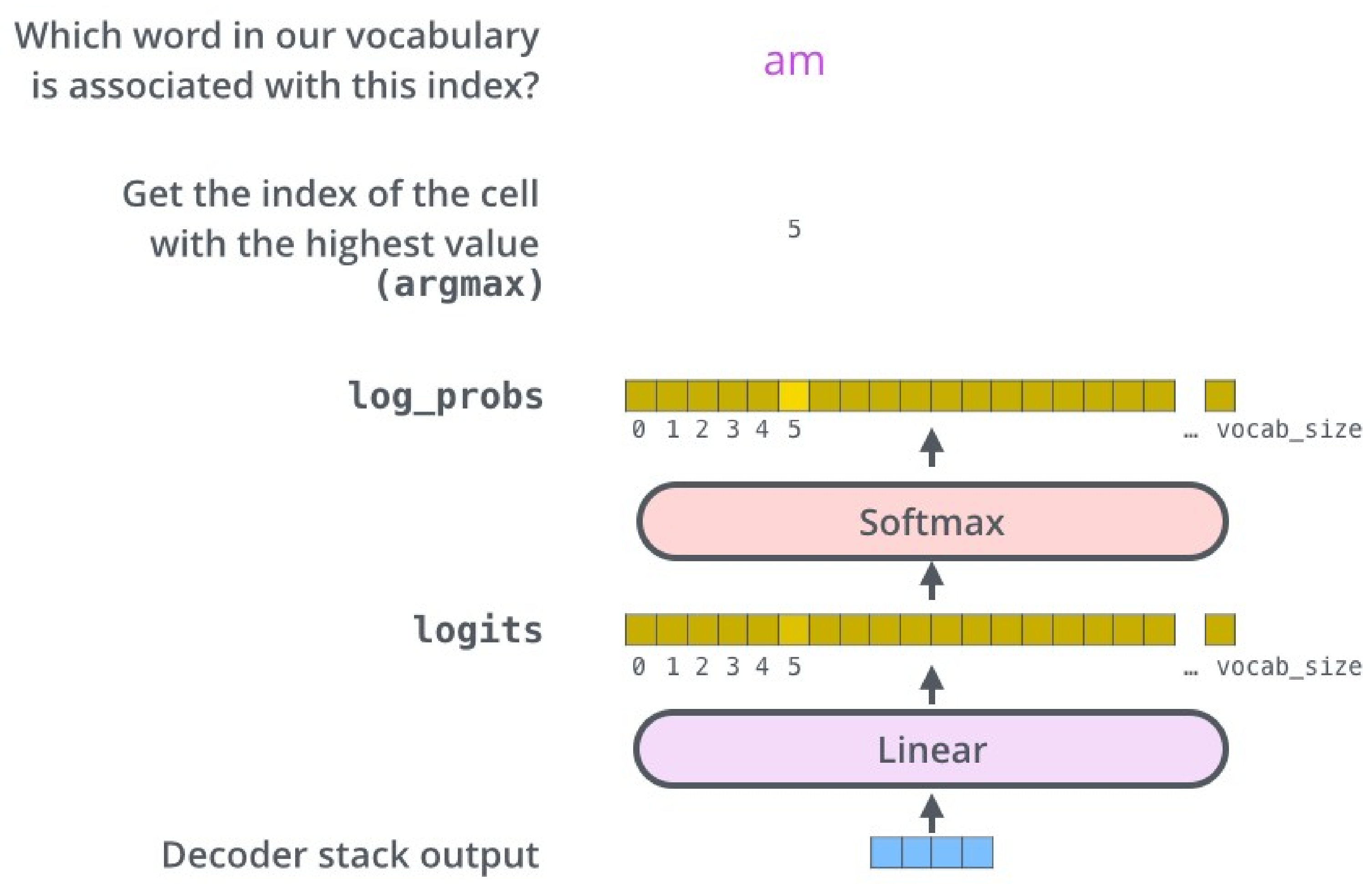
Prompting and Prompt templating

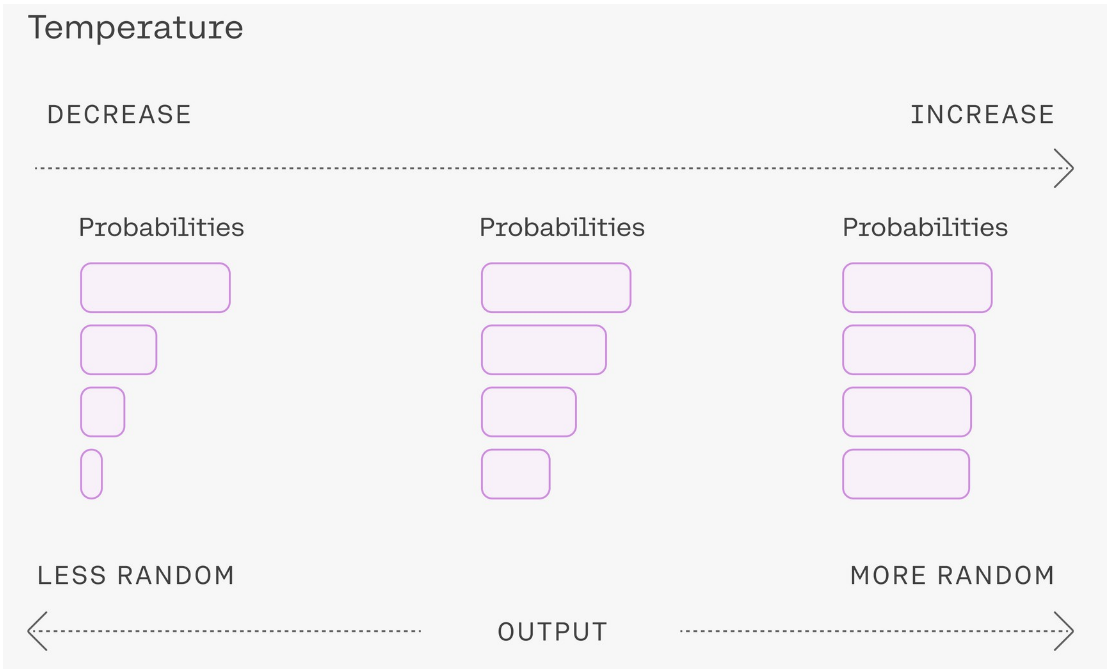
Saying the right thing

How LLMs choose the next word

Before we discuss prompting, let's review why there is non-determinism in LLMs, by design:

- At the final layer, the last output is used to select a token index, which corresponds to a natural language token.
- This choice is made using a probability distribution which can be affect by the ***temperature*** value
- This means that for a given input, we cannot always know exactly what the next token will be…





Consistency Challenges in LLMs – Input/Output

Temperature isn't the only reason LLM outputs aren't predictable.

Hallucinations

- An LLM has access to an astoundingly high dimensional space, constructed from the embedding dimensions, and number of parameters
- Only a tiny subset of this space will be accessed during training, with the number of potential input token configurations to the model effectively infinite
- This can lead LLMs, at inference time, to output data that may have no real connection to the input data, simply because it hasn't been well constrained
- This is known as hallucination. This occurs when the LLM attempts to extrapolate in a region of this high dimensional space where it has not been well trained.

Prompting an LLM – How does it affect the model?

To over come hallucination, we can format our inputs in such a manner that we can generate consistency. Our **prompts and prompt templates** will be key.

By giving a consistent input to our models, we can promote a consistent activation of the right parameters and utilization of the attention mechanism

Good Prompts for an LLM

1. **Prompt**: *"Explain the impact of climate change on global food production."*
   - **Why It's Good**: This prompt is clear, specific, and focused on a particular topic. It asks for an explanation, which guides the LLM to provide detailed information on a well-defined subject.
2. **Prompt**: *"Summarize the key themes in George Orwell's novel '1984'."*
   - **Why It's Good**: The prompt is precise, directing the LLM to focus on the key themes of a specific book. It avoids ambiguity and is likely to result in a coherent, relevant response.

Bad Prompts for an LLM

1. **Prompt**: *"Tell me everything about history."*
   - **Why It's Bad**: This prompt is too vague and broad. "History" is an immense topic, and the lack of specificity can lead to a disjointed or overly general response. The LLM might struggle to determine what exactly the user wants to know.
2. **Prompt**: *"Why is the sky green?"*
   - **Why It's Bad**: This prompt is based on a false premise (the sky is not green). While the LLM might provide an answer, it could result in an incorrect or nonsensical explanation, particularly if the model doesn't recognize the premise as false.

From ChatGPT

What we've learned about prompting LLMs

Prompt templating is now a very common practice with multiple libraries now built to support simple templating

```
prompt_template = PromptTemplate(
    input_variables=["theme"],  # List of input variable names
    template="Write a story about {theme}"  # String template with placeholders
)
```

These templates typically align with various tasks:

- Summarization
- Analysis
- Translation
- Extraction

```
summary_template = PromptTemplate.from_template(
    """Provide a concise summary of the following factual topic:

{topic}

Here are some key points to consider including:

* Background information
* Main ideas
* Supporting details
* Significance or impact

Strive for objectivity and neutrality in your summary."""
)

# Generate a summary prompt for climate change
climate_summary_prompt = summary_template.format(topic="Climate Change")
print(climate_summary_prompt)
```

Each of which will have standardized templates that have been found to work best with various LLMs

Prompt Formats in Chat Models

Not all prompts are the same

In an API request, most Chat LLMs utilize three types of prompts/messages:

1) The System Message
2) The Assistant Response
3) The User Prompt

After each interaction, the messages array is expanded to contain the history of the assistant and user prompt, and the system message/prompt is pre-pended each time the user sends a message (though this isn't stored in the array)

```
client = OpenAI()

completion = client.chat.completions.create(
  model="gpt-4o",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello!"}
  ]
)
```

Example API request format for OpenAI

System Prompts

These special, static, prompts are pre-pended to the user prompts and the models are trained to obey these system prompts.

- **Guidance**: It sets the rules or guidelines for how the AI should respond to user inputs. This could include instructions on maintaining a certain tone (e.g., formal, casual), following specific ethical guidelines, or limiting responses to certain types of information.
- **Context**: It provides the AI with context about the interaction. For example, the system prompt might indicate that the AI should behave as a customer service representative or as a tutor in a specific subject.
- **Consistency**: It ensures that the AI's responses are consistent throughout the interaction, adhering to the predefined rules or context.

Example of a System Prompt
Let's say you're designing an AI assistant for a customer service chatbot. The system prompt might look something like this:

System Prompt:

"You are a helpful and polite customer service assistant. Always address customers respectfully, provide clear and concise answers, and offer additional help when possible. Avoid technical jargon and ensure the customer feels valued. If you cannot assist directly, offer to escalate the issue to a human representative."

```
You are ChatGPT, a large language model trained by OpenAI, based on the GPT-4 architecture.
Knowledge cutoff: 2023-10
Current date: 2024-08-12

Image input capabilities: Enabled
Personality: v2

# Tools

## bio

The `bio` tool allows you to persist information across conversations. Address your message `to=bio` and
write whatever information you want to remember. The information will appear in the model set context below
in future conversations.

## dalle

// Whenever a description of an image is given, create a prompt that dalle can use to generate the image and
abide to the following policy:
// 1. The prompt must be in English. Translate to English if needed.
// 2. DO NOT ask for permission to generate the image, just do it!
// 3. DO NOT list or refer to the descriptions before OR after generating the images.
// 4. Do not create more than 1 image, even if the user requests more.
// 5. Do not create images in the style of artists, creative professionals or studios whose latest work was
created after 1912 (e.g. Picasso, Kahlo).
// - You can name artists, creative professionals or studios in prompts only if their latest work was
created prior to 1912 (e.g. Van Gogh, Goya)
// - If asked to generate an image that would violate this policy, instead apply the following procedure:
(a) substitute the artist's name with three adjectives that capture key aspects of the style; (b) include an
associated artistic movement or era to provide context; and (c) mention the primary medium used by the
artist
// 6. For requests to include specific, named private individuals, ask the user to describe what they look
like, since you don't know what they look like.
// 7. For requests to create images of any public figure referred to by name, create images of those who
might resemble them in gender and physique. But they shouldn't look like them. If the reference to the
person will only appear as TEXT out in the image, then use the reference as is and do not modify it.
// 8. Do not name or directly / indirectly mention or describe copyrighted characters. Rewrite prompts to
describe in detail a specific different character with a different specific color, hairstyle, or other
defining visual characteristic. Do not discuss copyright policies in responses.
// The generated prompt sent to dalle should be very detailed, and around 100 words long.
// Example dalle invocation:
// ```
// {
// "prompt": "<insert prompt here>"
// }
// ```
```

Part of ChatGPT's system prompt

LLM Pipelines

Chat models are more than just a transformer

Using an LLM – More than just a transformer

By far the most common ways to interact with open-source models is using the HuggingFace Transformers library.

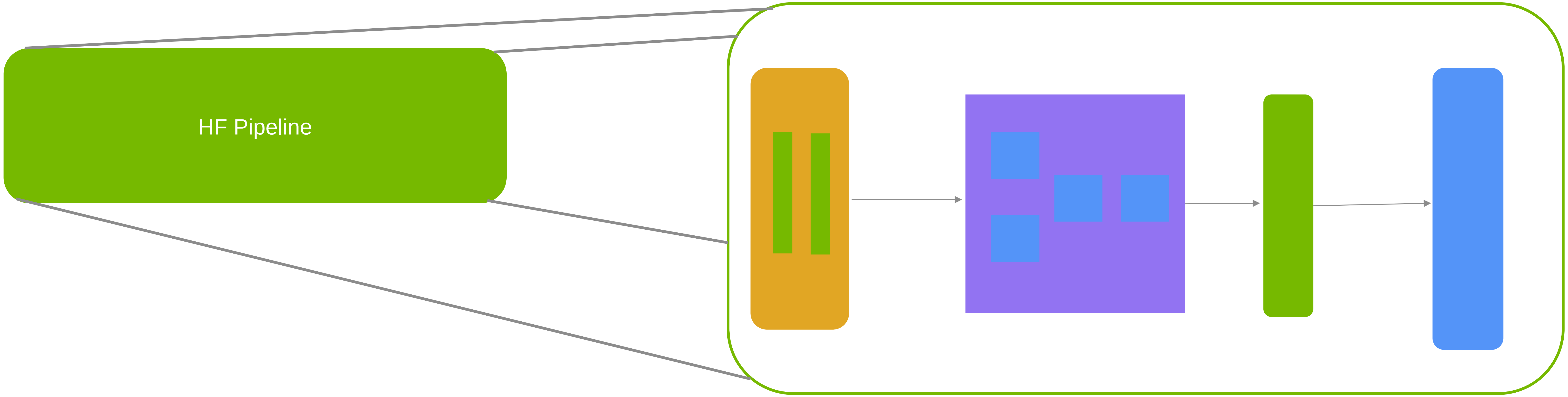
Here you can use high level APIs to interact with LLMs, such as the pipeline module.


```
>>> pipe = pipeline("text-classification")
>>> pipe("This restaurant is awesome")
[{'label': 'POSITIVE', 'score': 0.9998743534088135}]
```

Using an LLM – More than just a transformer

However, there is more going on underneath this API call than you might realize!

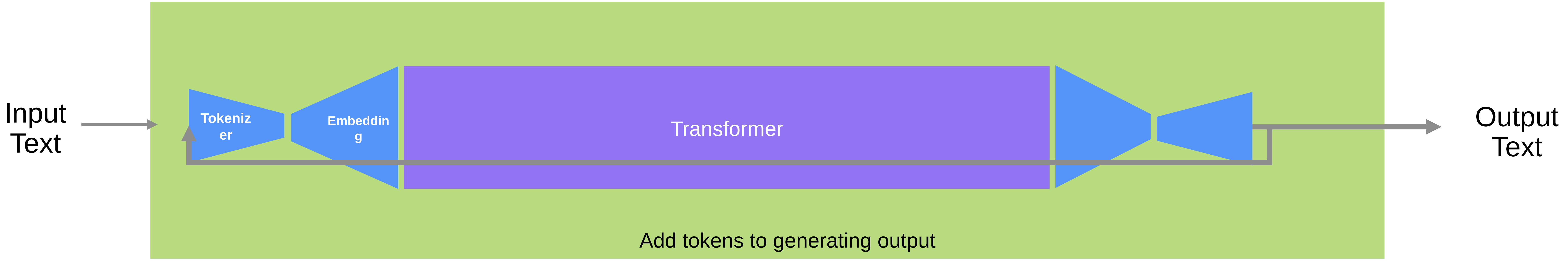
This abstraction is useful to quickly solve a particular task, but if we delve deeper, we can start to build more complex structures



Components of the Pipeline – Overview

Let’s look at the insides of this pipeline:

- Text is converted to token IDs via the tokenizer
- These tokens are then converted to word embeddings in the transformer
- The transformer converts these vectors to enriched vectors
- The last layer of the transformer selects a token
- The tokenizer decodes the token ID to a natural language token



LLM Pipelines – Anatomy of a Conversation

We can see in the messages array from a sample conversation that the format of messages goes back and forth with the user and the assistant.

The system message first aligns the model.

The assistant then works with both the system message and the user's prompt to complete the task.

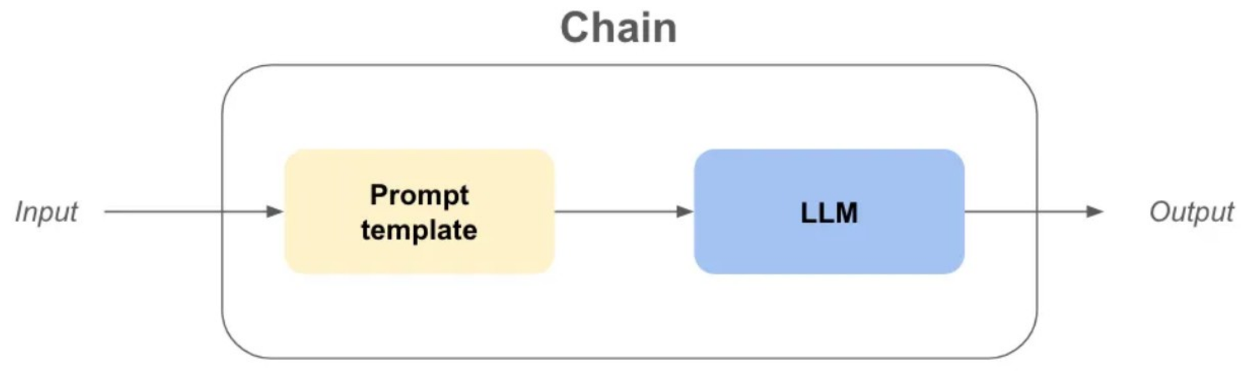
With different prompts, the assistant will focus on how to answer differently.


```json
[
    {
        "role": "system",
        "content": "You are a knowledgeable and friendly assistant that helps users with general knowledge questions. Answer clearly and concisely."
    },
    {
        "role": "user",
        "content": "Can you explain how photosynthesis works?"
    },
    {
        "role": "assistant",
        "content": "Photosynthesis is a process used by plants, algae, and some bacteria to convert light energy, usually from the sun, into chemical energy stored in glucose. The process takes place in the chloroplasts of plant cells, where chlorophyll captures light energy. This energy is used to convert carbon dioxide and water into glucose and oxygen. The overall equation is: 6 CO2 + 6 H2O + light energy -> C6H12O6 + 6 O2."
    },
    {
        "role": "user",
        "content": "Why is photosynthesis important?"
    },
    {
        "role": "assistant",
        "content": "Photosynthesis is crucial because it provides the primary source of energy for nearly all life on Earth. It produces oxygen, which is essential for the respiration of most living organisms, and it is the basis of the food chain, as plants serve as the primary producers."
    }
]
```

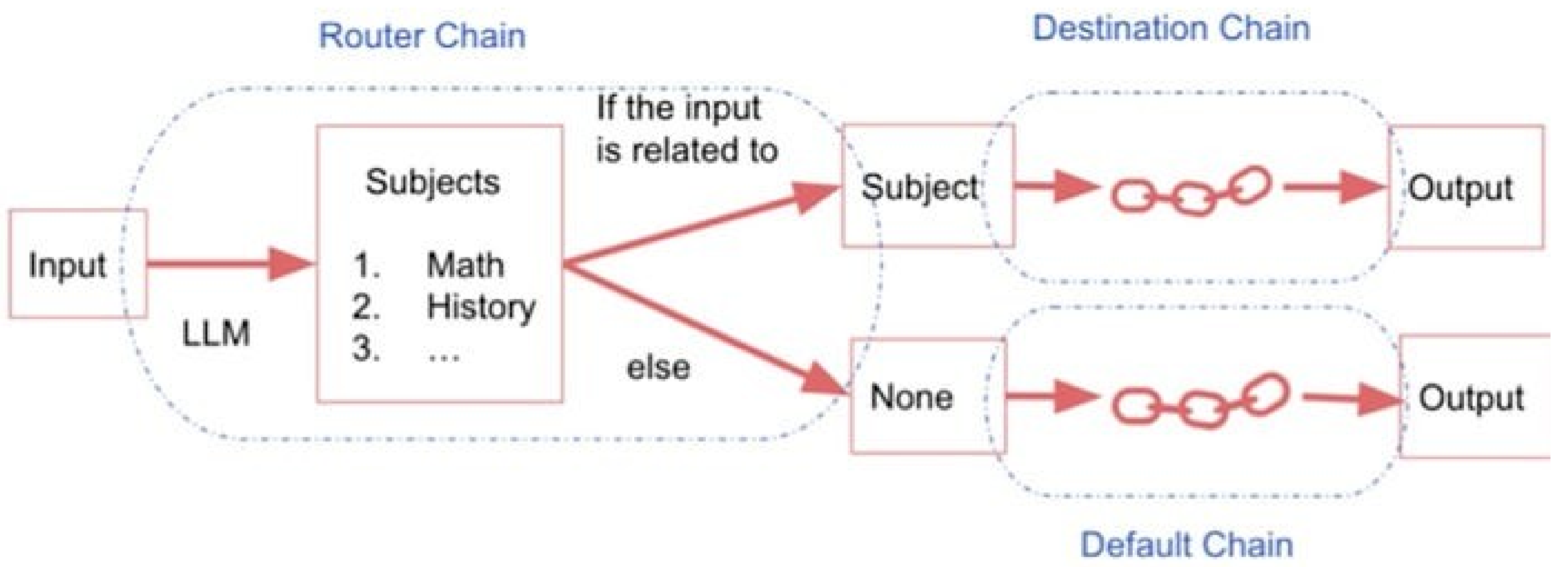
LLM Pipelines – More Flexibility with Chains

We can combine prompts and prompt templates, with these LLM pipeline for more flexibility

By doing so, we can connect more than just a single prompt to an LLM, but create chains of logical flow



Prompt, LLM and Chain



LLM Chains

Logical Flow Patters with LLMs

LLM Chains - Expanding Pipelines Further

LLM chains can be thought of as a way build structure and allow for flexibility.

The simplest chain consists of:

1. An input prompt template

2. An LLM

3. (optional) An output parser

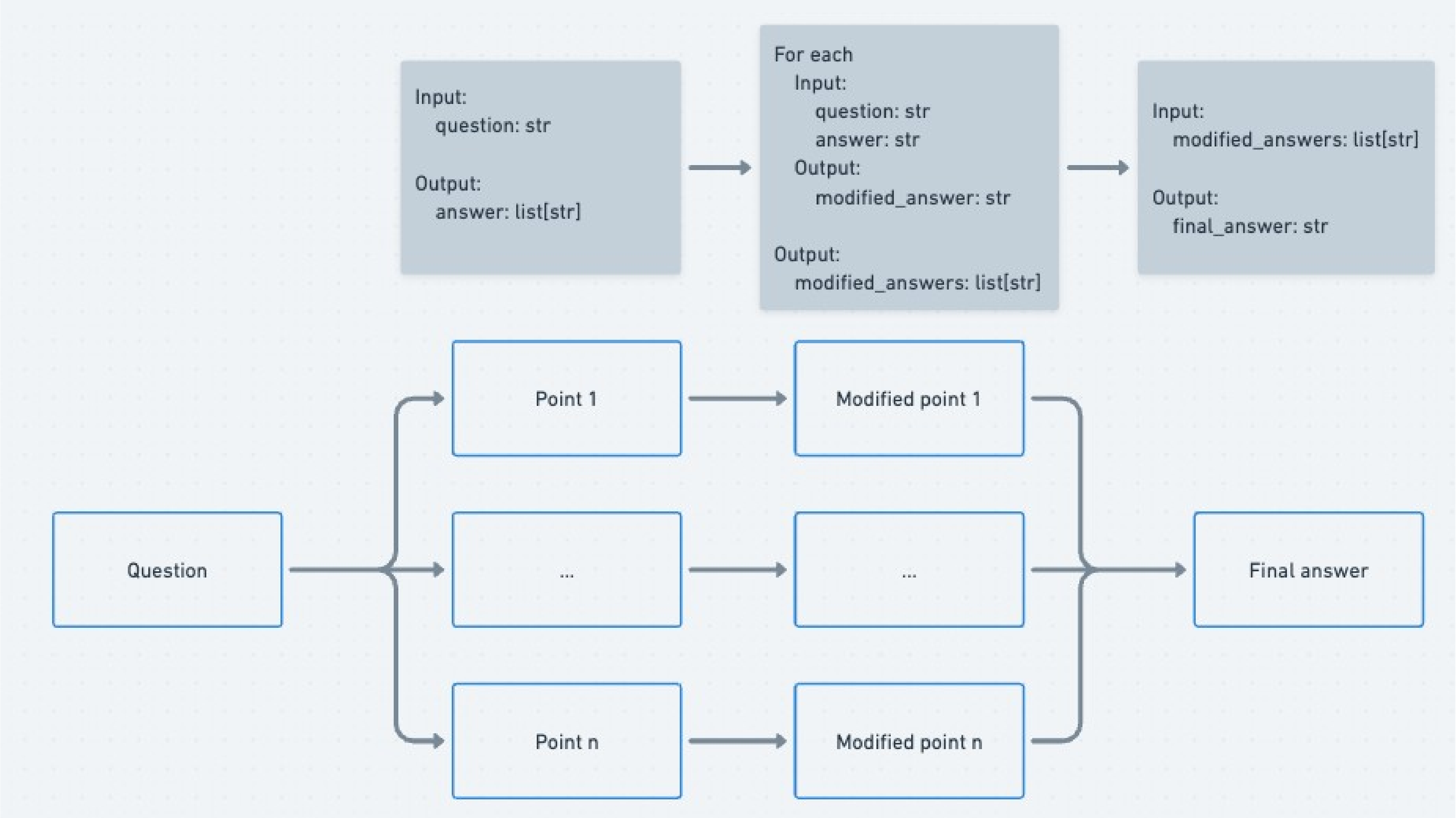
```
# Step 1: Define the Input Template
input_template = PromptTemplate(
    template="Question: {question}\nAnswer:",input_variables=["question"])

# Step 2: Initialize the LLM (e.g., using OpenAI's GPT)
llm = OpenAI(model="text-davinci-003")

# Step 3: Define the Output Parser
# Example: Extracting a simple numerical answer from the LLM's response
using regex
output_parser = RegexParser(
    pattern=r"Answer:\s*(\d+)",output_key="parsed_answer")

# Step 4: Create the LLM Chain
llm_chain = LLMChain(
    llm=llm,prompt=input_template,output_parser=output_parser)

# Step 5: Define the Sequential Chain
sequential_chain = SimpleSequentialChain(chains=[llm_chain],verbose=True)
```

Once created, a user can simply interact with the chain object, passing in the relevant information and receiving the desired output

```
# Step 6: Run the Sequential Chain with an Example Input
input_data = {"question": "What is 2 + 2?"}
output = sequential_chain.run(input_data)
```

Combining Prompts and LLMs as Chains

With this approach, we can connect multiple paths in parallel and serial, potentially trying multiple models, different tasks for each, and combining them for the user at the output

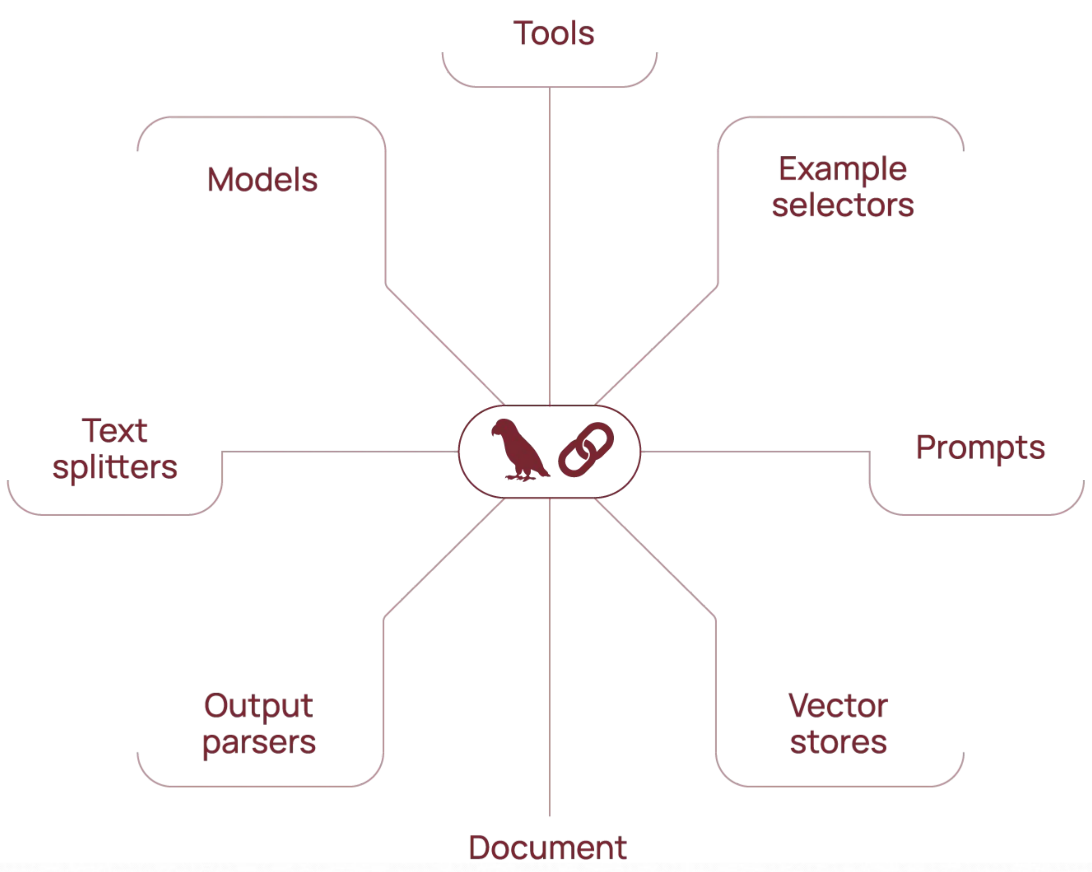


LangChain

Not long after the launch of ChatGPT in Nov 2022, a new library appeared that was focused on the integration of LLMs as a core logical processing piece that connected components together.

The library **LangChain** has since become the de-facto chain library for the GenAI community and now has 100s of integrations into every mainstream and opensource model.

LangChain allows developers to connect not just prompts and LLMs but also to other components, making it possible to build full applications that connect data and AI, all using a single framework and natural language as the user's input



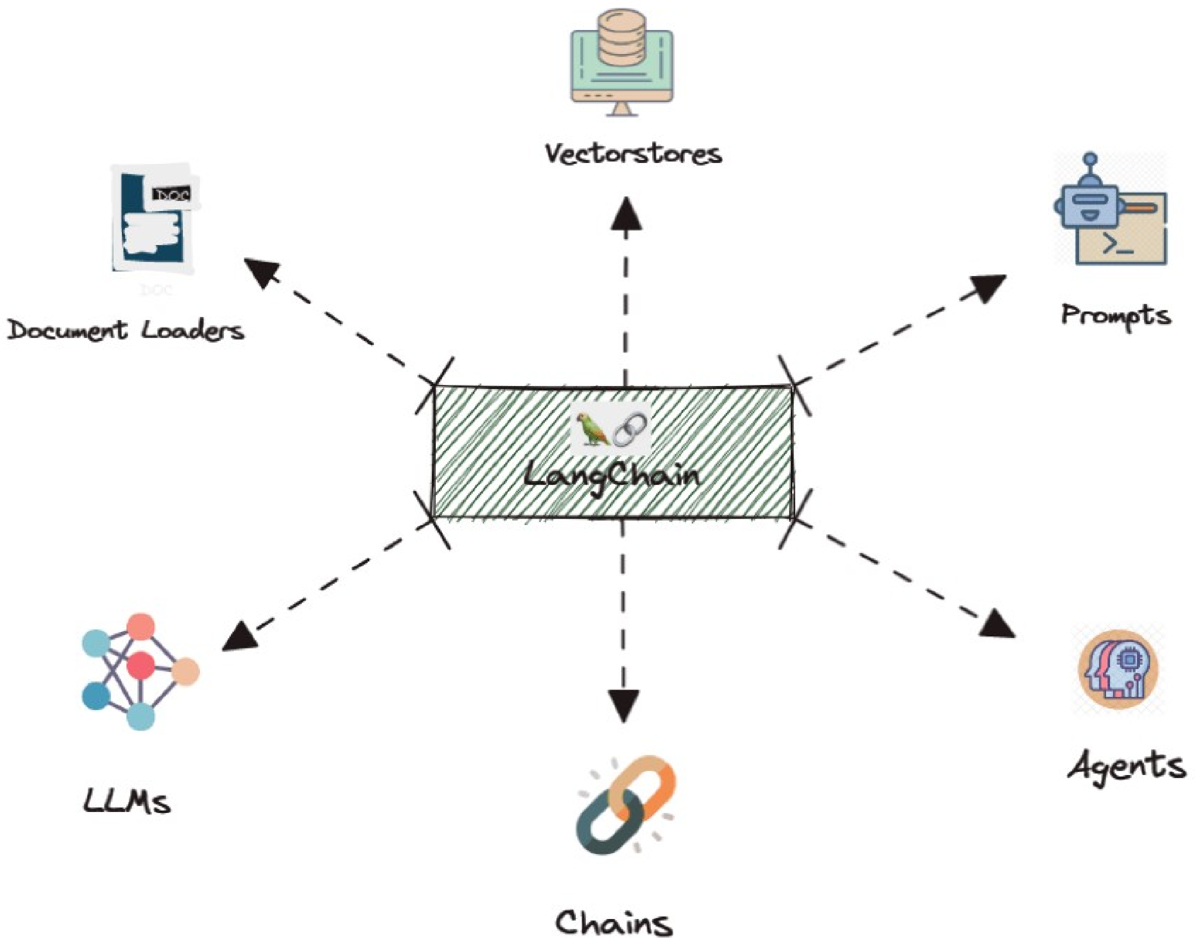
Python

```python
>>> from langchain_community.vectorstores import (
    Chroma
    )
>>> from langchain_core.output_parsers import StrOutputParser
>>> from langchain_core.prompts import ChatPromptTemplate
>>> from langchain_core.runnables import RunnablePassthrough
>>> from langchain_openai import ChatOpenAI

chain = (
    RunnablePassthrough.assign(
         context=Chroma(...).as_retriever()
    ) | ChatPromptTemplate(...)
    | ChatOpenAI(...)
    | StrOutputParser()
)
```

Combining LLMs with other components

With chains, we can connect more than just prompts to LLMs

We can connect data sources, other LLMs, compute environments, basically anything that has a digital interface can be connected using libraires like LangChain



In the next lesson we'll take a deeper dive into one of the most common applications, Retrieval Augmented Generation, or RAG!

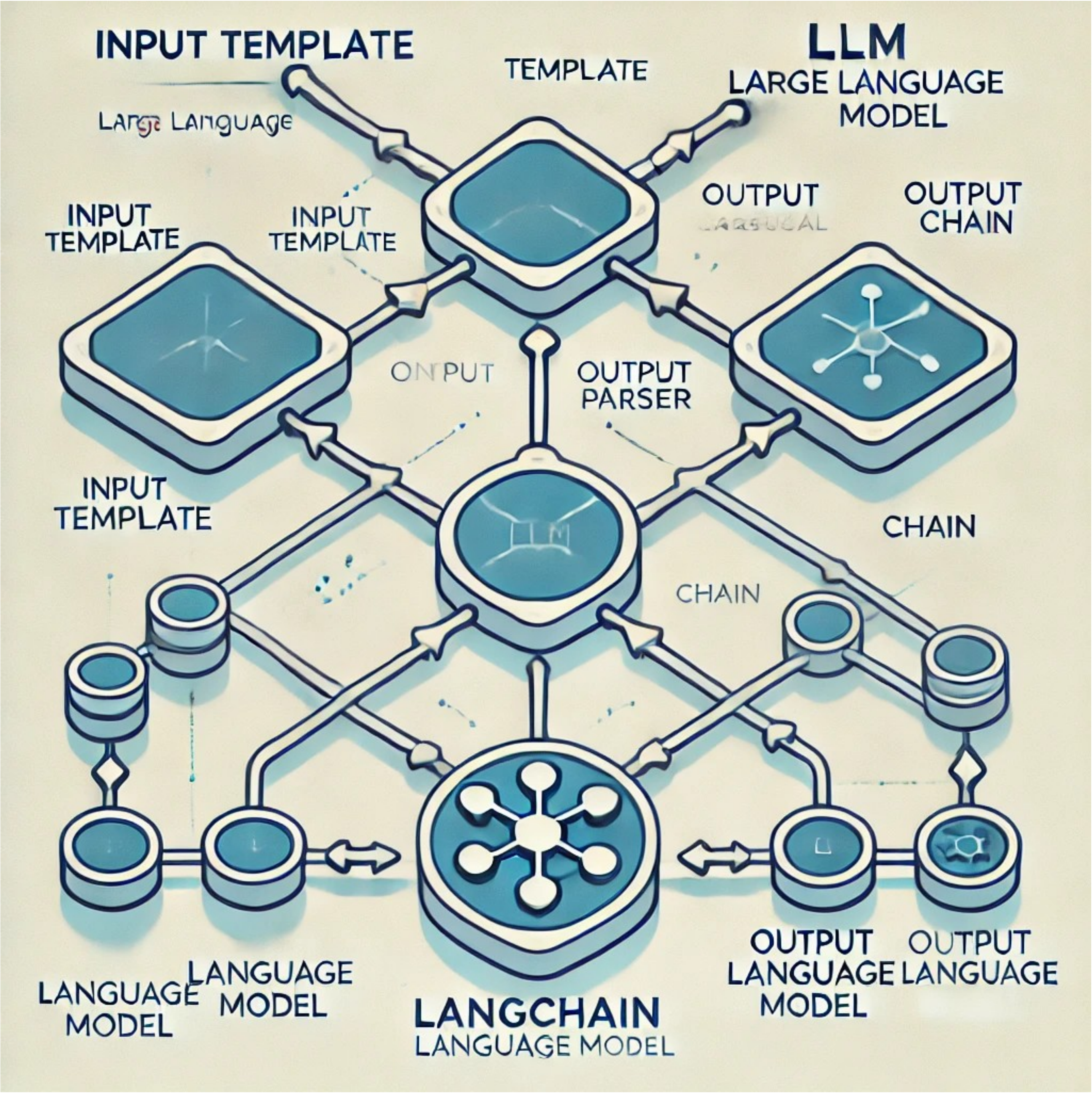
Wrap Up

LLM Components

- Today we reviewed how prompts enable in-context learning
- Pipelines allow us to abstract the complexities of the inner workings of transformers to make interactions easier
- Explored the different types of prompting, including the system prompt
- Introduced the concept of LLM chains and how the LangChain library can be used to make complex logical flows

----

In the next lesson we will start our discussion on how to build more complex chains and LLM applications, starting with **R**etrieval **A**ugment **G**eneration.



(LangChain graph generated by ChatGPT)

Thank you!