



**16 April 2025**

# **Foundational Concepts and Tools**



**LLM**

**Large Language Model**

# Language Models

- Language model is designed to predict the likelihood of a sequence of words occurring in a language.
- These models learn the statistical properties and patterns present in each language to generate text.
- Language models can broadly classify into two categories:
  - Generative language models: designed to generate new text based on the patterns learnt from the training data.
    - The model takes a prompt or starting sequence and then generate the next word or sequence of words one step at a time.
  - Predictive language models: predict the likelihood of the next word in a given context.
    - Widely used in tasks like autocomplete, next-word prediction and machine translation.

# RNN, LSTM, GRU and Transformer

Description	Recurrent Neural Network (RNN)	Long Short Term Memory (LSTM)	Gated Recurrent Unit (GRU)	Transformers
<b>Overview</b>	RNNs are foundational sequence models that process sequences iteratively, using the output from the previous step as an input to the current step.	LSTMs are an enhancement over standard RNNs, designed to better capture long-term dependencies in sequences.	GRUs are a variation of LSTMs with a simplified gating mechanism.	Transformers move away from recurrence and focus on self-attention mechanisms to process data in parallel
<b>Key characteristics</b>	<ul style="list-style-type: none"> <li>- Recurrent connections allow for the retention of "memory" from previous time steps.</li> </ul>	<ul style="list-style-type: none"> <li>- Uses gates (input, forget, and output) to regulate the flow of information.</li> <li>- Has a cell state in addition to the hidden state to carry information across long sequences.</li> </ul> <p>© AIMA.com Research</p>	<ul style="list-style-type: none"> <li>- Contains two gates: reset gate and update gate.</li> <li>- Merges the cell state and hidden state.</li> </ul>	<ul style="list-style-type: none"> <li>- Uses Self-attention mechanisms to weigh the importance of different parts of the input data.</li> <li>- Consists of multiple encoder and decoder blocks.</li> <li>- Processes data in parallel rather than sequentially.</li> </ul>
<b>Advantages</b>	<ul style="list-style-type: none"> <li>- Simple structure.</li> <li>- Suitable for tasks with short sequences.</li> </ul>	<ul style="list-style-type: none"> <li>- Can capture and remember long-term dependencies in data.</li> <li>- Mitigates the vanishing gradient problem of RNNs.</li> </ul>	<ul style="list-style-type: none"> <li>- Fewer parameters than LSTM, often leading to faster training times.</li> <li>- Simplified structure while retaining the ability to capture long-term dependencies.</li> </ul>	<ul style="list-style-type: none"> <li>- Can capture long-range dependencies without relying on recurrence</li> <li>- Highly parallelizable, leading to faster training on suitable hardware.</li> </ul>
<b>Disadvantages</b>	<ul style="list-style-type: none"> <li>- Suffers from the vanishing and exploding gradient problem, making it hard to learn long-term dependencies</li> <li>- Limited memory span</li> </ul>	<ul style="list-style-type: none"> <li>- More computationally intensive than RNNs</li> <li>- Complexity can lead to longer training times.</li> </ul>	<ul style="list-style-type: none"> <li>- Might not capture long-term dependencies as effectively as LSTM in some tasks.</li> </ul>	<ul style="list-style-type: none"> <li>- Requires a large amount of data and computing power for training.</li> <li>- Can be memory-intensive due to the attention mechanism, especially for long sequences.</li> </ul>
<b>Use Cases</b>	<p>Due to its limitations, plain RNNs are less common in modern applications.</p> <p>Used in simple language modeling, time series prediction</p>	Machine translation, speech recognition, sentiment analysis, and other tasks that require understanding of longer context.	Text generation, sentiment analysis, and other sequence tasks where model efficiency is a priority.	<ul style="list-style-type: none"> <li>- State-of-the-art performance in various NLP tasks, including machine translation, text summarization.</li> <li>- Forms the backbone for models like BERT and GPT.</li> </ul>
<b>Model variants</b>	Vanilla RNN, Bidirectional RNN, Deep (Stacked) RNN	Vanilla LSTM, Bidirectional LSTM, Peephole LSTM, Deep (Stacked) LSTM	GRU	Original Transformer (Seq-to-Seq), Encoder only (Eg: BERT), Decoder only (Eg: GPT), Text to Text (Eg: T5)

Source: <https://aiml.com/compare-the-different-sequence-models-rnn-lstm-gru-and-transformers/>

# The Transformer Architecture

- Transformers are a revolutionary architecture which was introduced in 2017 by Vaswani et al. 2017 in the paper “Attention is All You Need”.
- Transformers are superior to LSTMs (Long-Short-Term-Memory networks) and RNNs (Recurrent Neural Networks) for several reasons.
- Transformers outperform RNNs and LSTMs and excel in parallelism, long-range dependency handling and scalability.
- This makes the architecture of choice for many modern NLP tasks.

[Paper: \[1706.03762\] Attention Is All You Need](#)

# Why Transformer?

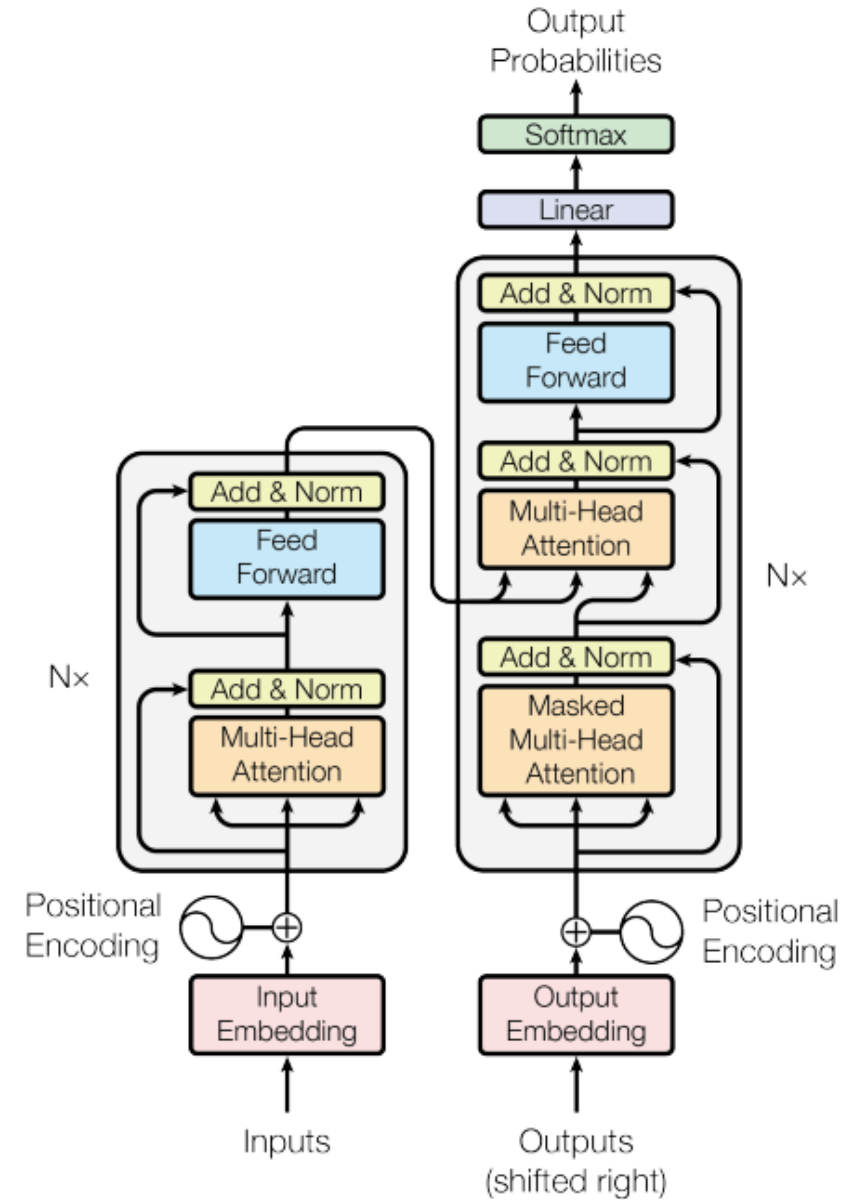
## Advantages:

- Process data in parallel rather than sequentially.
- Able to capture long-range dependencies without relying on recurrence.
- Highly parallelizable leading to faster training on suitable hardware.
- State Of The Art (SOTA) performance for tasks like translation, text summarization etc.

## Disadvantages:

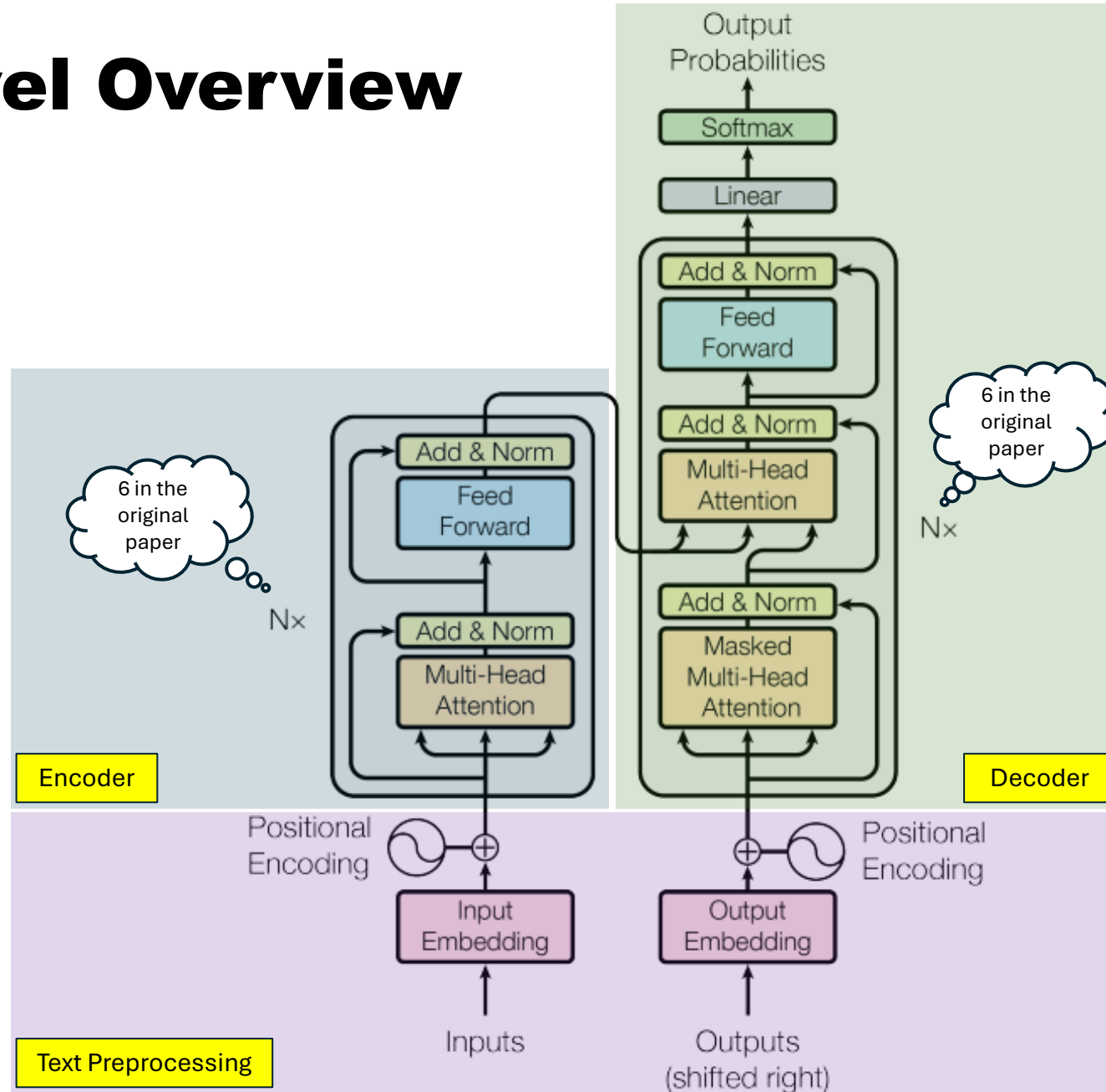
- Memory intensive due to the attention mechanism especially for long sequences.
- Computationally expensive and require a large amount of data for training effectively.

# Transformer Architecture (2017)



Source: <https://arxiv.org/pdf/1706.03762.pdf>

# High Level Overview

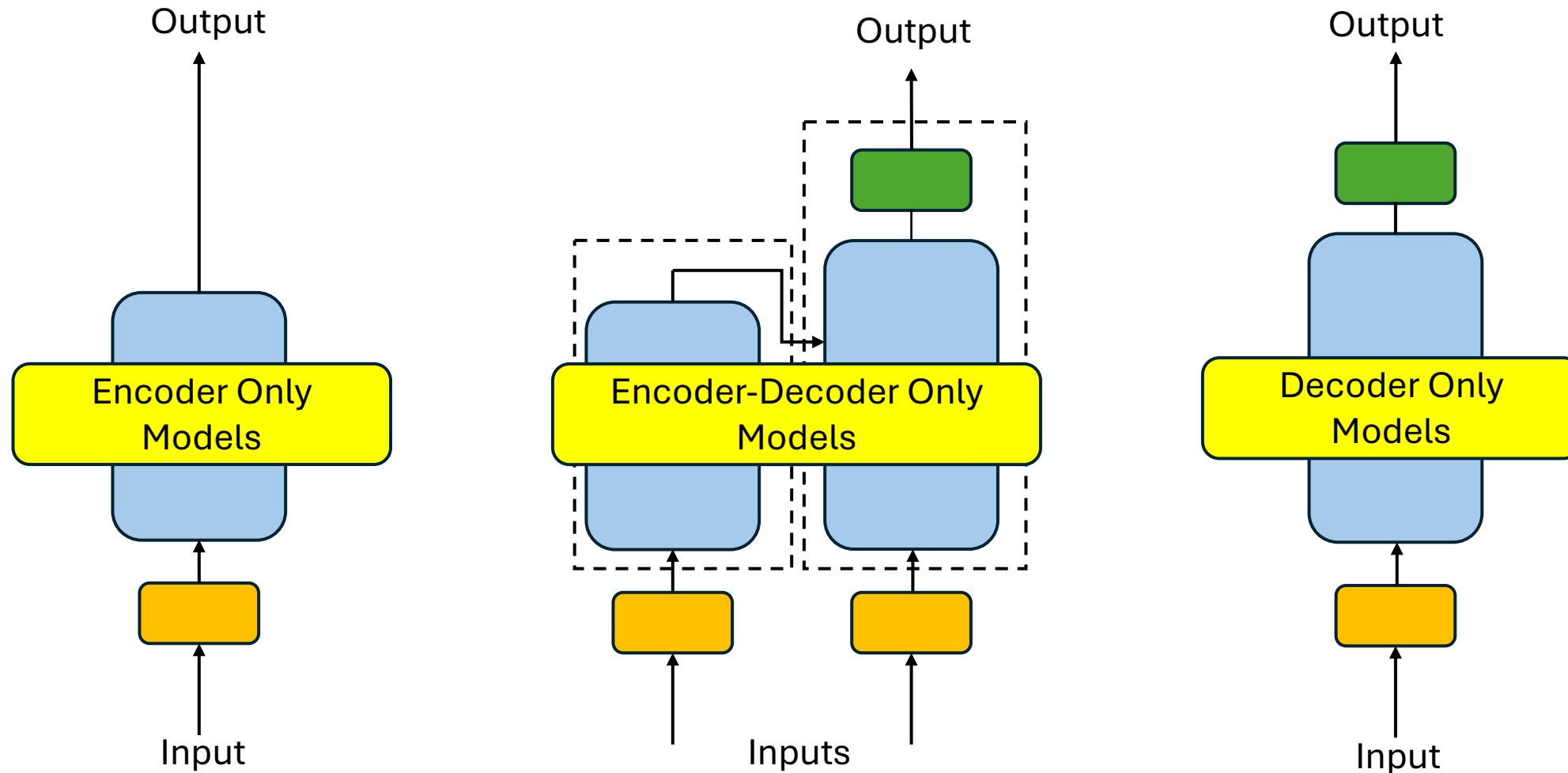




# Transformer Architecture

- All language models use some variants of the transformer architecture.
- The original proposed architecture was for solving sequence-to-sequence tasks i.e., translation.
- It was extended to solve a variety of different problems.
  - Semantic similarity of text
  - Classifying images
- Encoder processes the input sequence and produces an output sequence.
- Decoder generates its own output sequence, given the encoder's output sequence as input.
- Transformer takes a sequence of input and produces a new sequence as output.

# Architecture Variants



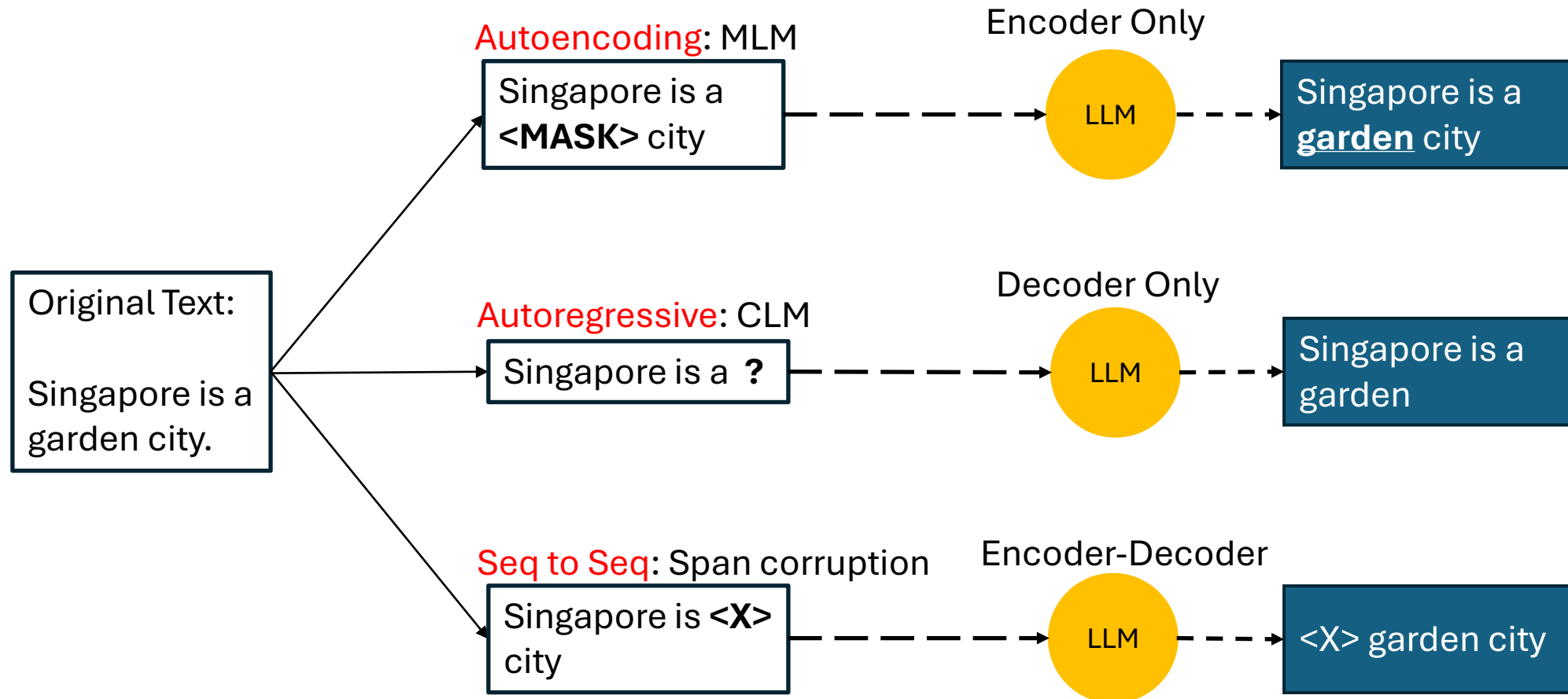
# Model Characteristics

- Encoder (also known as Autoencoding models)
  - Pretrained using Masked Language Modelling (MLM)
  - Objective is to reconstruct text ('denoising')
  - Bidirectional context
  - Good use cases for: Sentiment Analysis, Named Entity Recognition (NER) and Word Classification
  - Models: BERT, ROBERTA, distilBERT
- Decoder (also known as Autoregressive models)
  - Pretrained using Causal Language Modelling (CLM)
  - Objective is to predict the next token
  - Unidirectional context
  - Good use cases for: Text generation
  - Models: GPT, BLOOM

# Model Characteristics

- Encoder-Decoder (aka Sequence-to-Sequence models)
  - Pretrained techniques varies, for example, span corruption
  - Objective is to reconstruct the span of words
  - Good uses cases for Translation, Text Summarization and Question Answering
  - Models: T5, BART, Marian

# Pre-Training



MLM: Masked Language Modelling  
CLM: Causal Language Modelling

# Text Processing

- **Tokenization**
  - Convert raw text (corpus) into a format that the model can understand.
  - It involves splitting the text into tokens (usually words or sub-words).
  - There are many methods to tokenize text:
    - NLTK (Natural Language Toolkit)
    - spaCy
    - Tokenizer from HuggingFace Transformers
    - Stanford NLP
    - Gensim
    - etc.

**GPT-3.5 & GPT-4** **GPT-3 (Legacy)**

The Transformer model, introduced in the "Attention is All You Need" paper, is the neural network architecture at the core of the large language models.

Clear Show example

Tokens	Characters
30	152

An example:  
OpenAI  
Tokenizer

The Transformer model, introduced in the "Attention is All You Need" paper, is the neural network architecture at the core of the large language models.

TEXT TOKEN IDS

Source: <https://platform.openai.com/tokenizer>

# Tokenizer

- The computers aren't very good with words but are great with numbers.
- NLP models have limitation on the maximum number of token it can process.
- Tokenization breaks down a continuous stream of text into smaller, discrete units called tokens.
- Token helps to provides a structured way to break down text into manageable pieces for the model to process.
- Tokens are typically words, sub-words, characters (including punctuations) depending on the specific tokenization approach used.
- Example: **I'm** hungry vs **I am** hungry.
  - "I'm" → 1 word
  - "I am" → 2 words

# Tokenizer

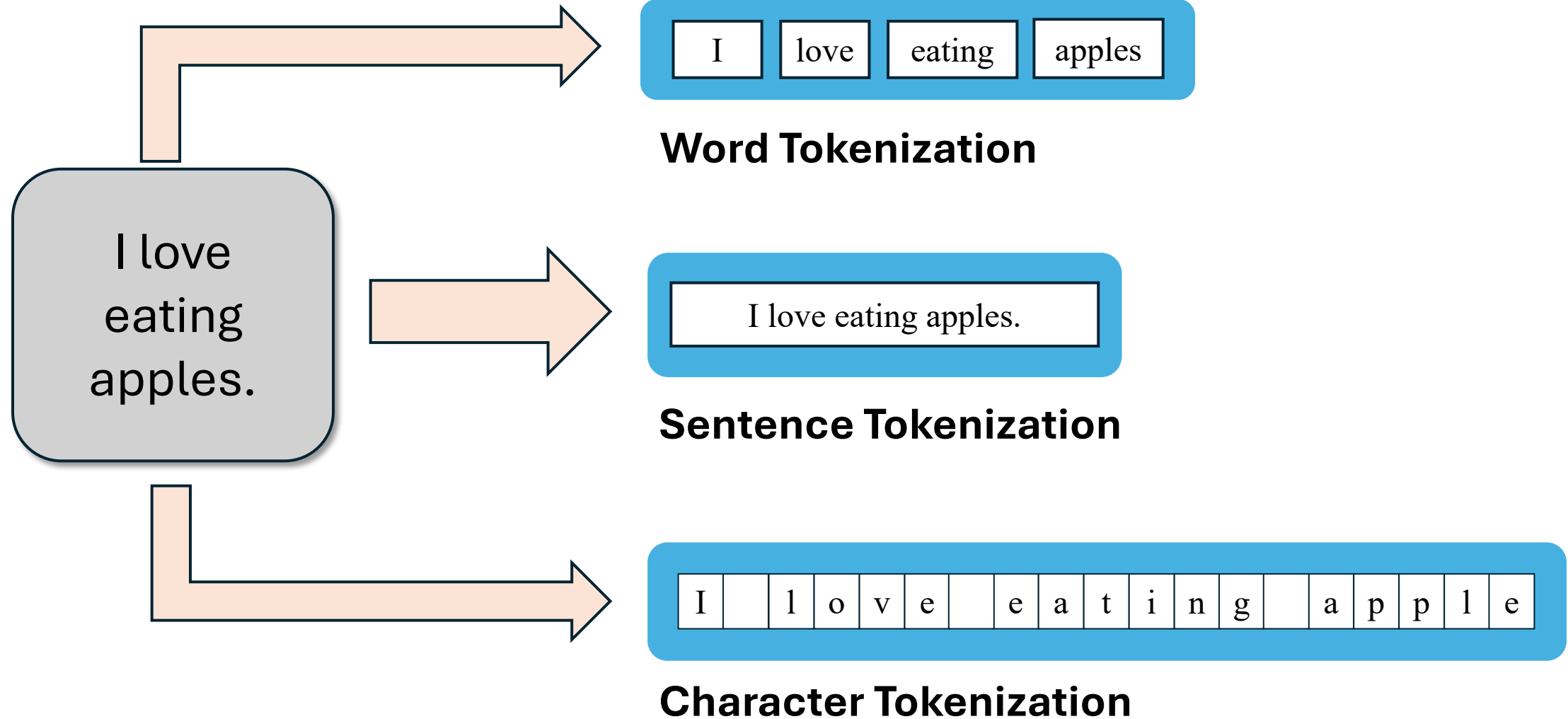
- Tokenizer is an integral part of a language model used during training.
- When a model is trained, it learns not just the relationship between words but also specific tokenization patterns introduced by the tokenizer.
- Tokens are basic units that the model processes and different tokenizers can produce different token sequences for the same text.
- Using a different tokenizer during inference than the one used during training is not recommended.
  - Can lead to token misalignment
  - Out-of-vocabulary problems
  - Degraded performance



# Tokenization Approaches

- Character-level tokenization
  - Represent the entire English language with limited characters i.e. upper/lower case character plus punctuation.
  - Not very useful since the meaning of “Cat” or “Cradle” is different although the “C” in both words means the same.
- Word-level tokenization
  - More meaningful than character-level tokenization but requires a much larger dictionary.
- Sentence-level tokenization
  - Capture meaningful phrases but resulted in an absurdly large dictionary.
- Each method has its pros and cons.
- Best solution provides decent compromises.

# Tokenization



# Embeddings

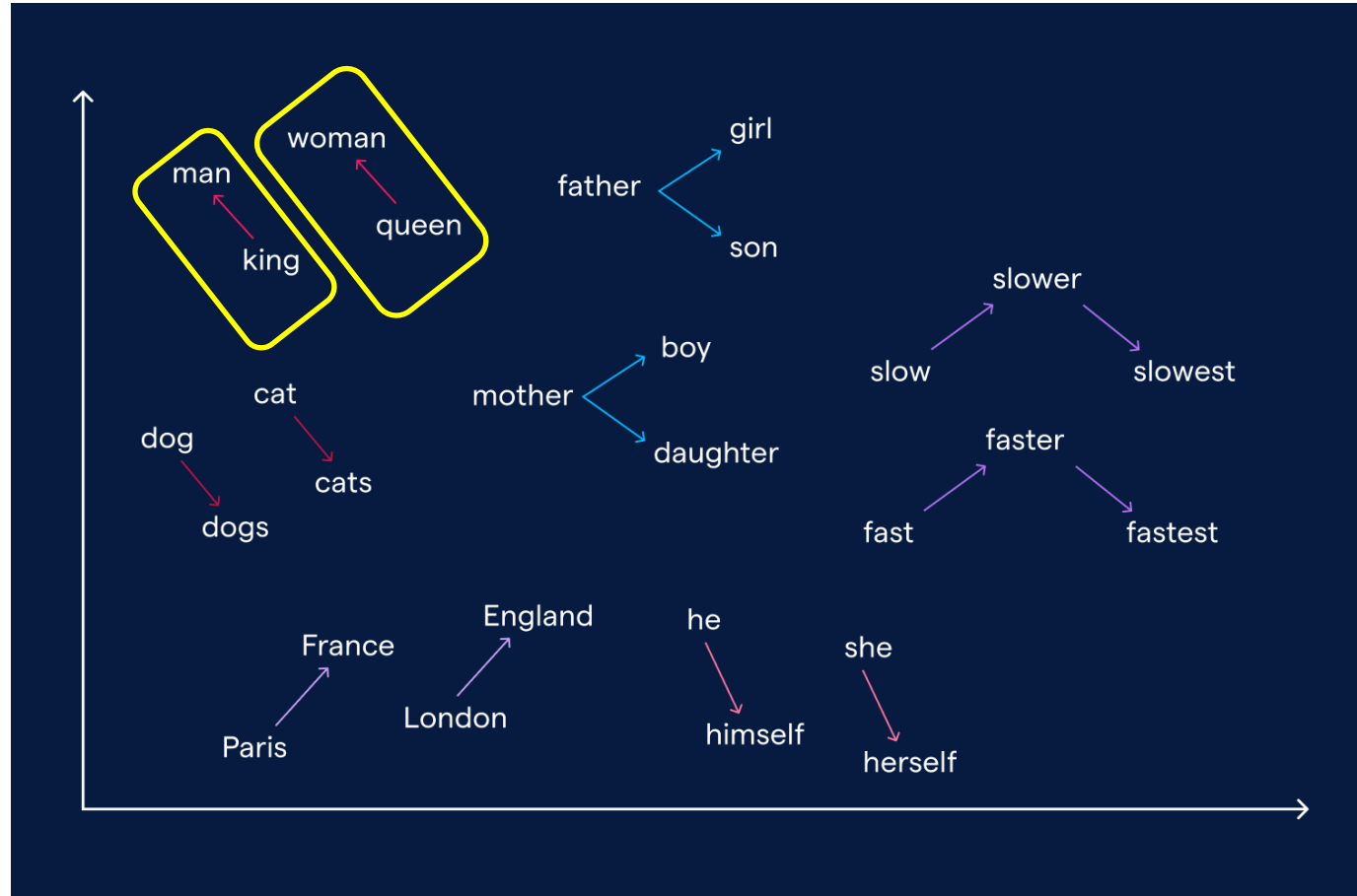
- The next step after tokenizing the text is to convert the words to numbers.
- Transform text into something that machines can interpret is called “word embedding”.
- Word embeddings are numerical illustrations of a text.
- Sentence and texts include organized sequences of information
- Goal is to develop a representation of words that capture their meanings, meaningful connections plus other sorts of situations in which the words are employed.

# Embeddings

- The aim is to generate numeric which forms the semantic representation from those tokens.
- A list of numbers which holds the semantic representation of a word.  
[-2, 4, -3.7, 41 ....., -0.98]
- One of the important characteristics is that if we plot it on a 2D graph, similar terms will be closer than dissimilar terms.
- Words that are found in similar contexts will have similar embeddings.
- An embedding contains a dense vector of floating-point values.
- Embeddings utilizes an efficient representation in which related words are encoded in the same manner.

# Embeddings

- The vector from **king**→**man** is very similar to **queen**→**woman**.



Source: [Feature Form](#)

# Pre-trained Word Embedding

- Modern Natural Language Processing (NLP) uses word embeddings that has been previously trained on a large corpus of text.
- It is hence called “Pre-trained Word Embeddings”.
- Captures both the connotative and syntactic meaning of a word.
- Examples of pre-trained word embeddings are:
  - Word2Vec
  - Continuous Bag-of-Words (CBOW)
  - Skip-Gram Model
  - GloVe
  - fastText
  - ELMo
  - BERT

# Positional Encodings

- Now that we have a way to represent (and learn) words (numbers), can we make it better?

The hunter chased the boar.  
The boar chased the hunter.

- The two sentences above can be represented using the same embeddings.
- But the two sentences have different meaning.
- Problem:
  - The embeddings contain semantic meanings, but not exact order meaning.
  - And ORDER usually matters.
- We can compute the positional encoding with sine and cosine functions of varying frequencies.

# Positional Encodings

- Using embeddings to represent words without order is not good enough.
- Transformer add **positional encodings** to the embeddings.
- Calculate a position vector (a list of numbers again!) for every word and summing the two vectors to form another vector.
- The positional encodings are typically added to the input embeddings before sending them into the transformer model.
- Positional encodings are necessary to introduce the notion of position or order.



# Positional Encodings

- The common approach is to use sine and cosine functions to generate these positional encodings.

In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Source: [arxiv.org/pdf/1706.03762.pdf](https://arxiv.org/pdf/1706.03762.pdf)

- $PE_{(pos, 2i)}$  and  $PE_{(pos, 2i+1)}$  are the components of the positional encoding for position  $pos$  and dimension  $2i$  and  $2i + 1$  respectively.
- $d_{model}$  : dimension of model embedding i.e. 512 in original paper.
- $pos$  : position of the token in the sequence.
- $i$  used for making alternate even and odd sequences.

# References

- Generative AI exists because of transformer  
[Financial Times](#)



- YouTube
  - [Visual Guide to Transformer Neural Networks - \(Episode 1\) Position Embeddings](#)
  - [How large language models work, a visual intro to transformers | Chapter 5, Deep Learning \(27 mins\)](#)
  - [Attention in transformers, visually explained | Chapter 6, Deep Learning \(26 mins\)](#)
  - [The matrix math behind transformer neural networks, one step at a time!!! \(23 mins\)](#)
  - [Transformer Neural Networks, ChatGPT's foundation, Clearly Explained!!! \(36 mins\)](#)
- Transformer Explainer  
<https://poloclub.github.io/transformer-explainer/>



# Prompt Engineering

# Introduction

- Prompt engineering is the process of designing and refining text inputs (prompts) to effectively interact with AI models like OpenAI ChatGPT.
- Its core purpose is to generate desired and relevant outputs.
- It combines creativity, logic and an understanding of AI behavior to bridge human intentions with machine-generated response.
- Key features are:
  - Interactive Communication: Enables meaningful and task-specific conversations with AI models.
  - Thoughtful Prompt Design: Crafting specific instructions to reduce ambiguity.
  - Iterative Design: Often involves testing and improving prompts to optimize results.
  - Versatile Application: Applicable in diverse areas such as writing, coding, decision support, and automation.

# Why Prompt Engineering Is Important?

- AI is becoming integral in industries like education, healthcare, marketing, and more.
- Prompt engineering skills enable individuals to maximize AI's utility.
- Poorly crafted prompts yield irrelevant or inaccurate results.
- Prompt engineering is a key competency in the AI-driven job market.
- Empowers individuals to design solutions without extensive programming knowledge.
- As AI systems evolve, prompt engineering ensures you stay ahead in understanding and utilizing these technologies effectively.

# Brief Overview of How it Works



Training Data



Learning Process

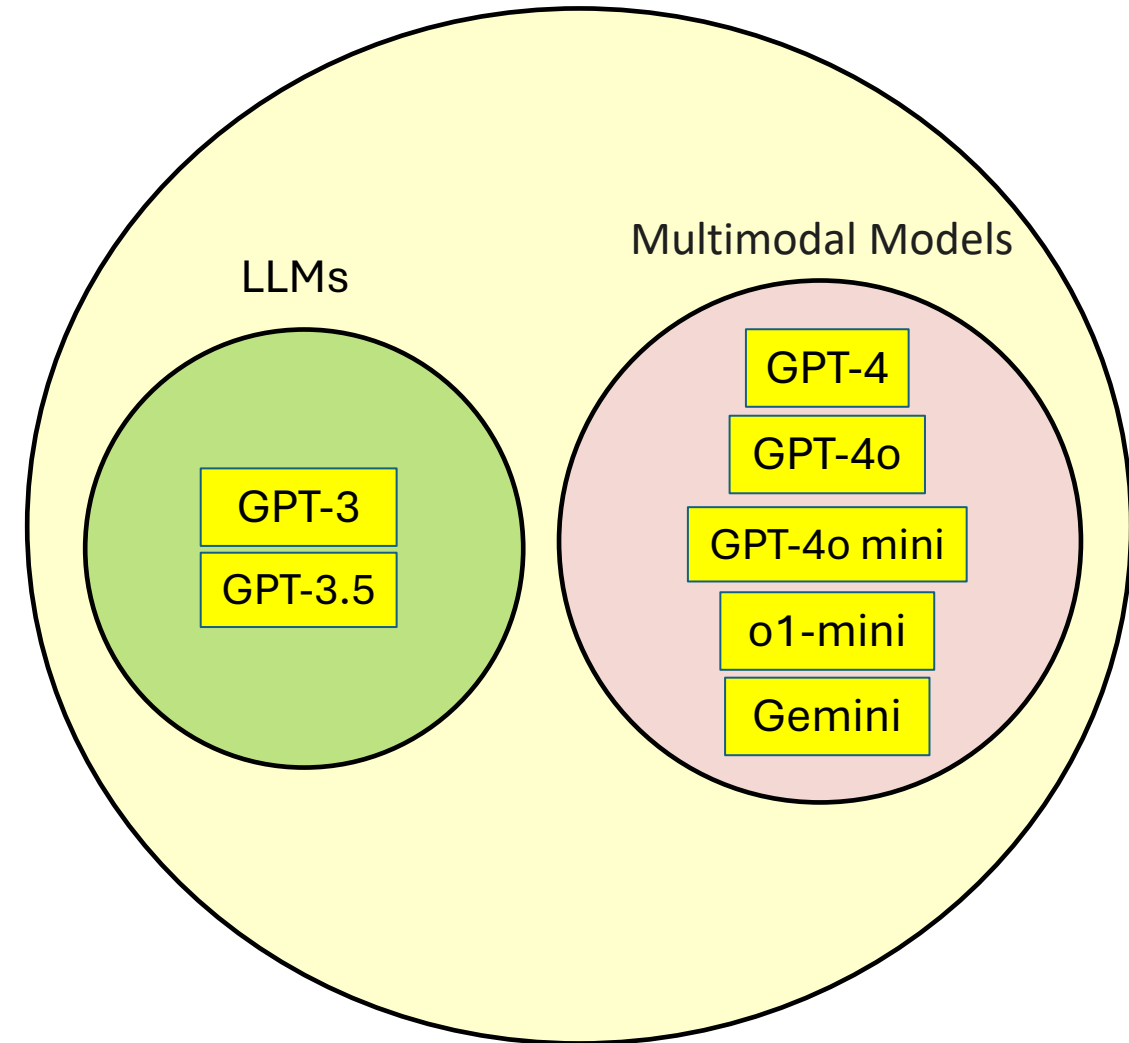


Data Generation

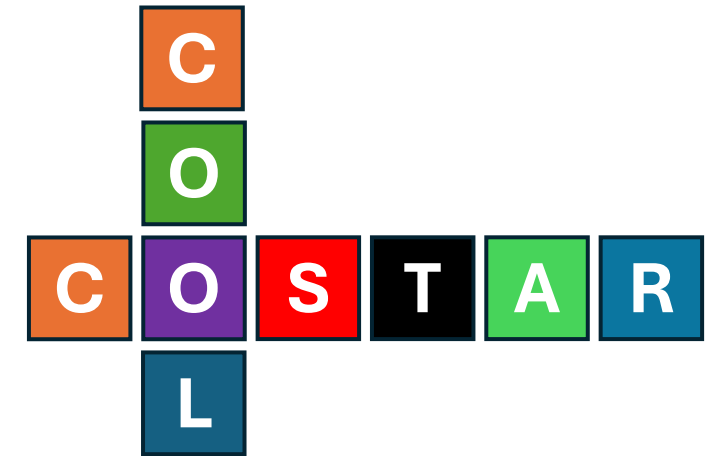
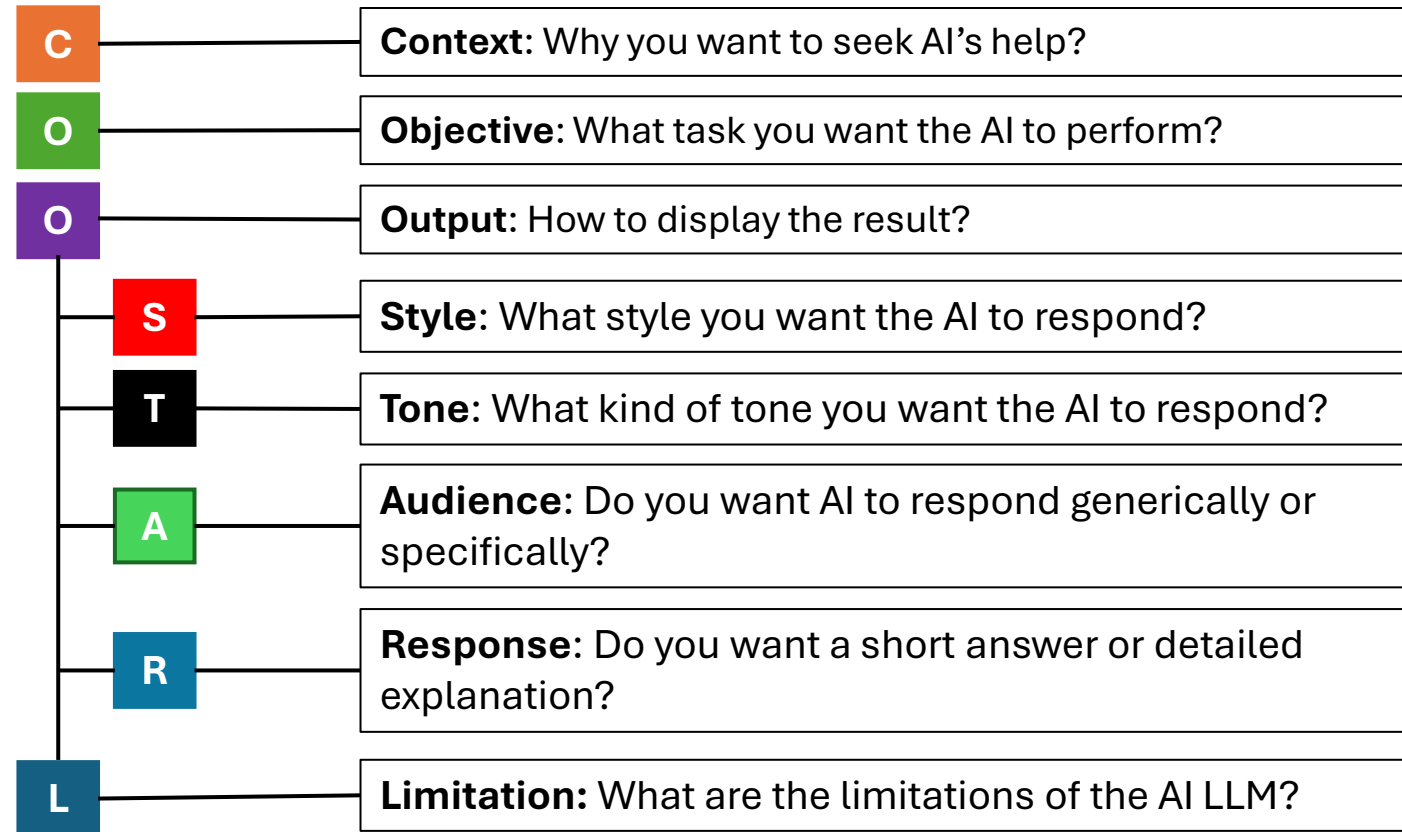
# Large Language Models (LLMs) vs Multimodal Models

Generative AI Models

Aspect	LLMs	Multimodal Models
Data Type	Primarily text-based	Multiple types (text, images, audio, video)
Focus	Understanding and generating text	Integrating and understanding multiple data types
Applications & Example Tasks	Translation, summarization, text completion, sentiment analysis	Image captioning, visual question answering, speech recognition, object detection in images
Integration with Other Modalities	Generally not designed to handle non-text data	Specifically designed to handle and integrate various data types



# Factors of Effective Prompts – COOL-STAR Principles



\*LLM: Large Language Model



# Focus on COO for Now

- Sufficient **Context**: (Why)
  - **Why** do you want to seek AI's help?
  - Provide the relevant background/subject information and focus of the writing.
- Clear **Objective**: (What)
  - **What** task do you want the AI to do? (e.g., write an article, craft a story)
  - **What** question do you want the AI to answer?
- Specific **Output**: (How)
  - **How** to display the result?
  - The format, style, tone, and length?

# STAR – Relating to Output

- **S-Style**
  - The style you want the response to be in. For example, generic response or a particular person's style of writing.
- **T-Tone**
  - Do you want the response to be causal, professional or humorous so that it resonates better with the audience.
- **A-Audience**
  - Are you writing to a general audience or specific group?
- **R-Response**
  - Text, numbered form or tabular form?

# Sample Prompt

I am a lecturer that teaches blockchain. I want to test my second-year polytechnic students' understanding of the following topics:

- Blockchain Structure
- Smart Contracts

Create 3 multiple choice questions for me.

For each question, provide the correct answer. Then write feedback to students about the correct and incorrect options.

# Breakdown of the Components

I am a lecturer that teaches blockchain. I want to test my second-year polytechnic students' understanding of the following topics:

- Blockchain Structure

- Smart Contracts

Create 3 multiple choice questions for me.

For each question, provide the correct answer. Then write feedback to students about the correct and incorrect options.

Sufficient **Context**

Clear **Objective**

Specific **Output**

# Crafting Effective Prompts - Practical Tips

- Engage the Model
  - **Not Engaging**: “List all the countries in the world.”
  - **Not Engaging**: “What's 1 + 1?”
  - **Engaging**: “Create Python code to retrieve the pricing info from XYZ website.”
- Be Conversational<sup>1, 2</sup>
  - Good example: “Hey ChatGPT, can you help me understand how blockchain technology can improve supply chain management? Feel free to include examples.”
  - Bad example: “Blockchain. Supply chain.”
- Experiment and iterate

1. [Hard Evidence That Please And Thank You In Prompt Engineering Counts When Using Generative AI](#)
2. [Does Being Nice to ChatGPT Make It Work Better? We Tested It Out.](#)

# Understanding of AI's



Source: <https://www.picpedia.org/highway-signs/l/limitation.html>

# Model Limitations

- Data bias
- Lack of contextual understanding
- Lack of memory
- Lack of common sense
- Ethics and safety concerns
- Limited ability to learn
  - GPT-3.5: Up to Sep 2021
  - GPT-4: Up to Apr 2023
  - Gemini: Early 2023
  - Claude 3.5 Sonnet: Apr 2024
- Inability to verify information



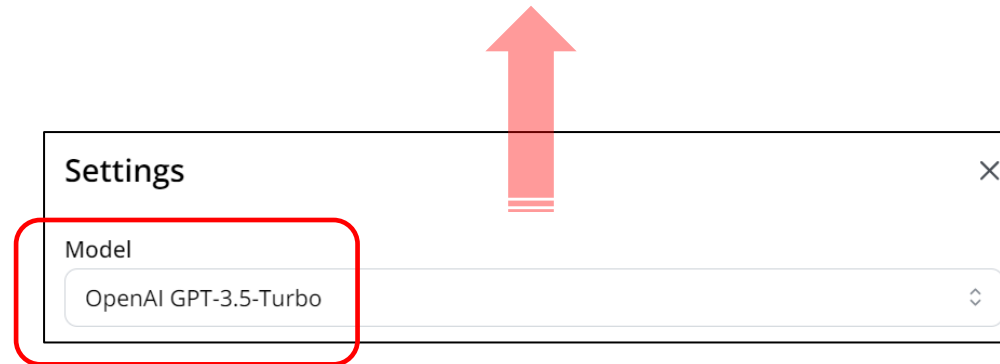
# Lack of Common Sense



The trophy doesn't fit into the brown suitcase because it's too small. What is too small?



In the given sentence, "it's too small" refers to the trophy. The sentence means that the trophy is too small to fit into the brown suitcase.



\*Latest LLM models reduce the effect significantly



# Limited Ability to Learn (Pre-trained model)

Example 1:

“Make up a plausible sounding scientific theory that hasn’t been discovered yet.” vs.  
“Explain the theory of relativity.”

Example 2:

“Tell me what I’m thinking.” vs. “Tell me about cognitive processes involved in thinking.”

1. A Large language model (LLM), has a limited ability to "learn" in the traditional sense because it is designed to work based on patterns and information it has been exposed to during its training.
2. LLMs rely on learned knowledge, so they can explain known theories but cannot "learn" new concepts in real time or invent them.

# Hallucination of LLM

- A phenomenon where the LLM generates text that is incorrect, nonsensical, or not real

Reason for Hallucination	Ways to Reduce Hallucination
Training data bias and noise	Careful selection and cleaning
Lack of specificity in prompting	More specific and clear prompts; Reducing the "temperature"
No access to external, real-world knowledge	Testing against ground truth; Injecting specific context to the prompt



# LangChain

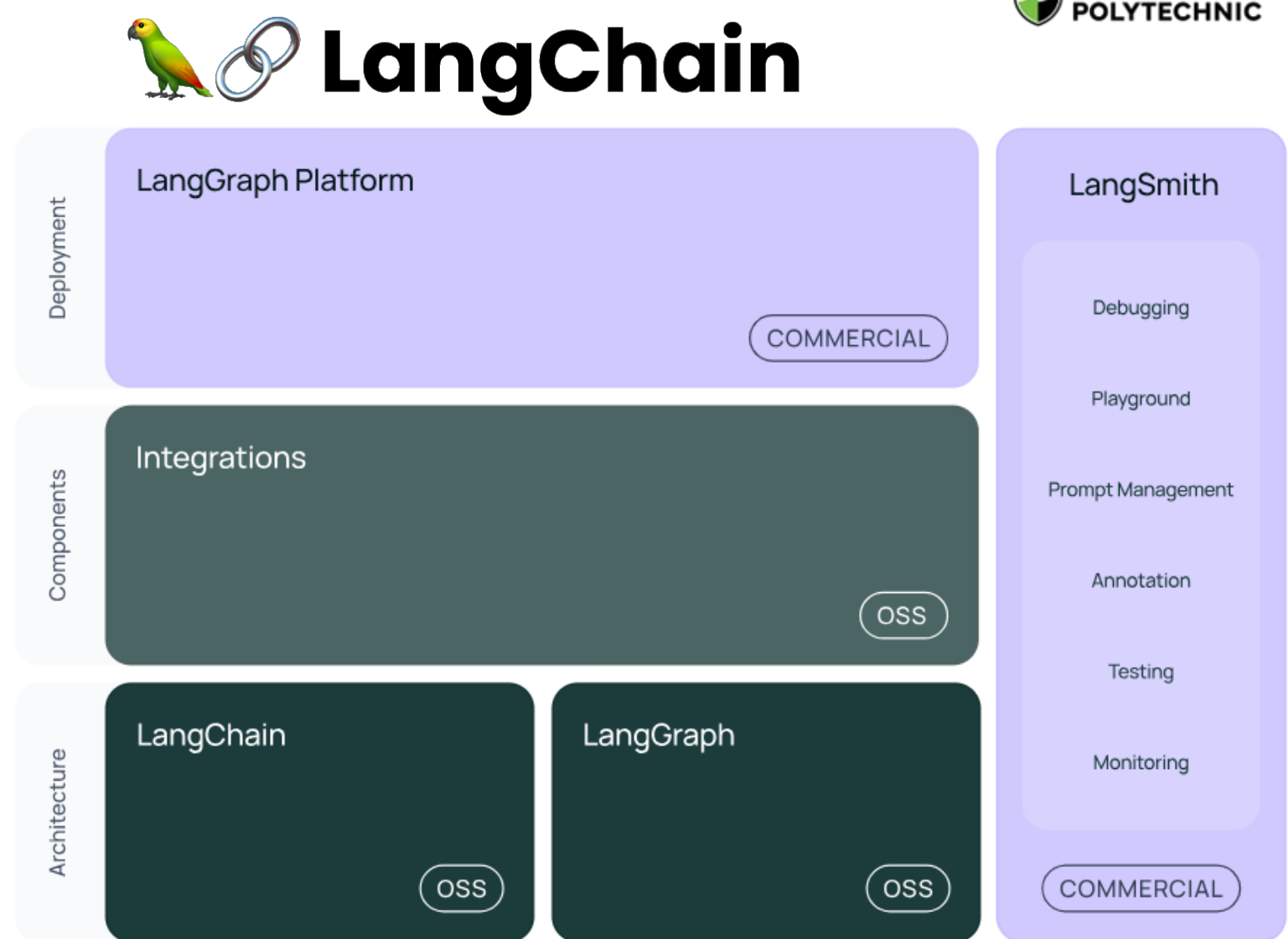
# Introduction

LangChain is a **framework** for developing applications powered by large language models. It enables applications that are context-aware and rely on a language model to reason.

Use LangGraph to build stateful agents with first-class streaming and human-in-the-loop support.

Use LangSmith to inspect, monitor and evaluate your chains, so that you can continuously optimize and deploy with confidence.

Turn your LangGraph applications into production-ready APIs and Assistants with LangGraph Platform

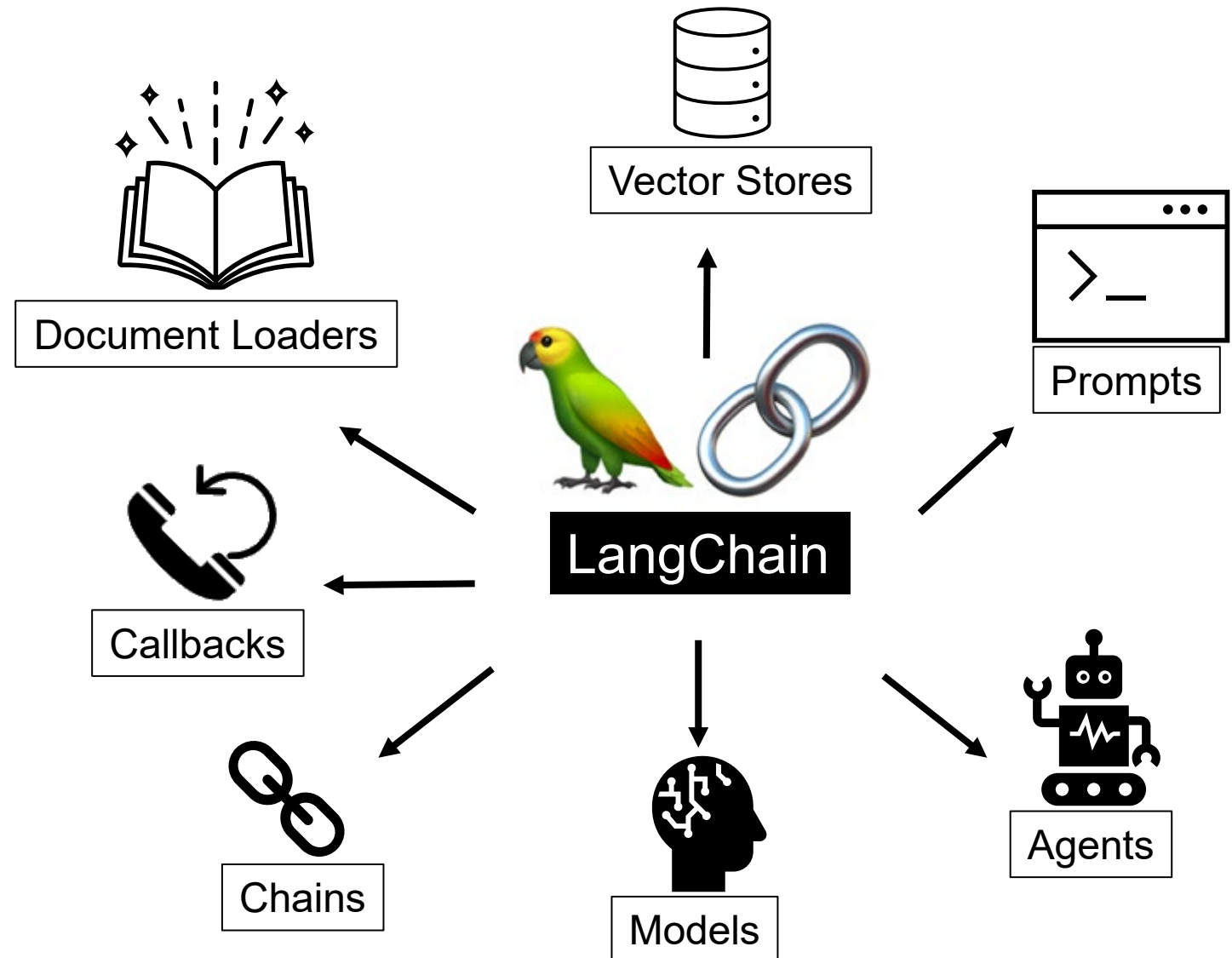


Source: <https://python.langchain.com/docs/introduction/>

# Components of LangChain



- In **LangChain**, components are the building blocks used to construct powerful applications that interact with language models (LLMs).
- These components provide modular, reusable, and customizable functionalities, making it easy to design workflows or pipelines for specific tasks.



# LangChain Ecosystem Packages

Any integrations that haven't been split out into their own packages will live in the langchain-community package

langchain-community

This package acts as a starting point to using LangChain

langchain

langchain-core

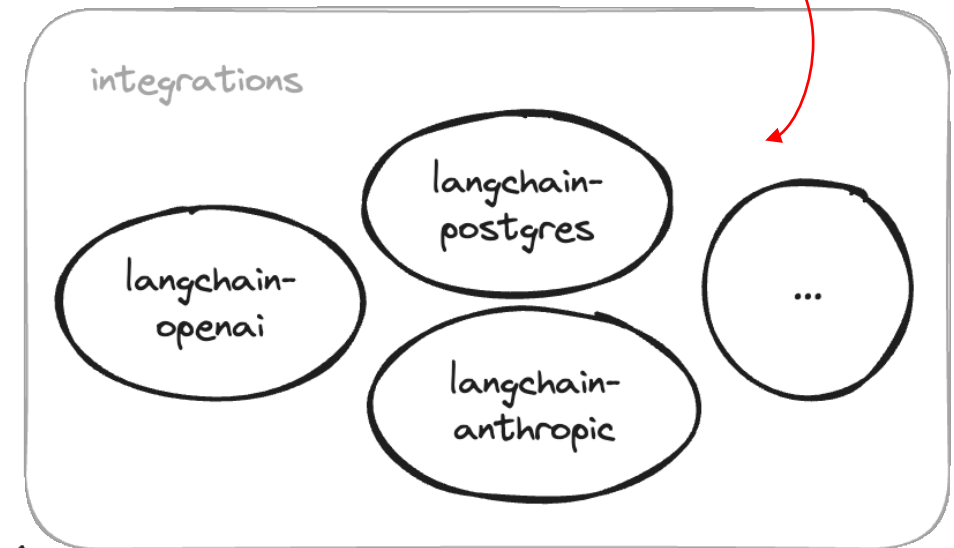
**Note:** A directed arrow (→) indicates that the source package depends on the target package

langgraph

**Langgraph** is a library for building stateful, multi-actor applications with LLMs

A directed arrow indicates that the source package depends on the target package:

Certain integrations like OpenAI and Anthropic have their own packages

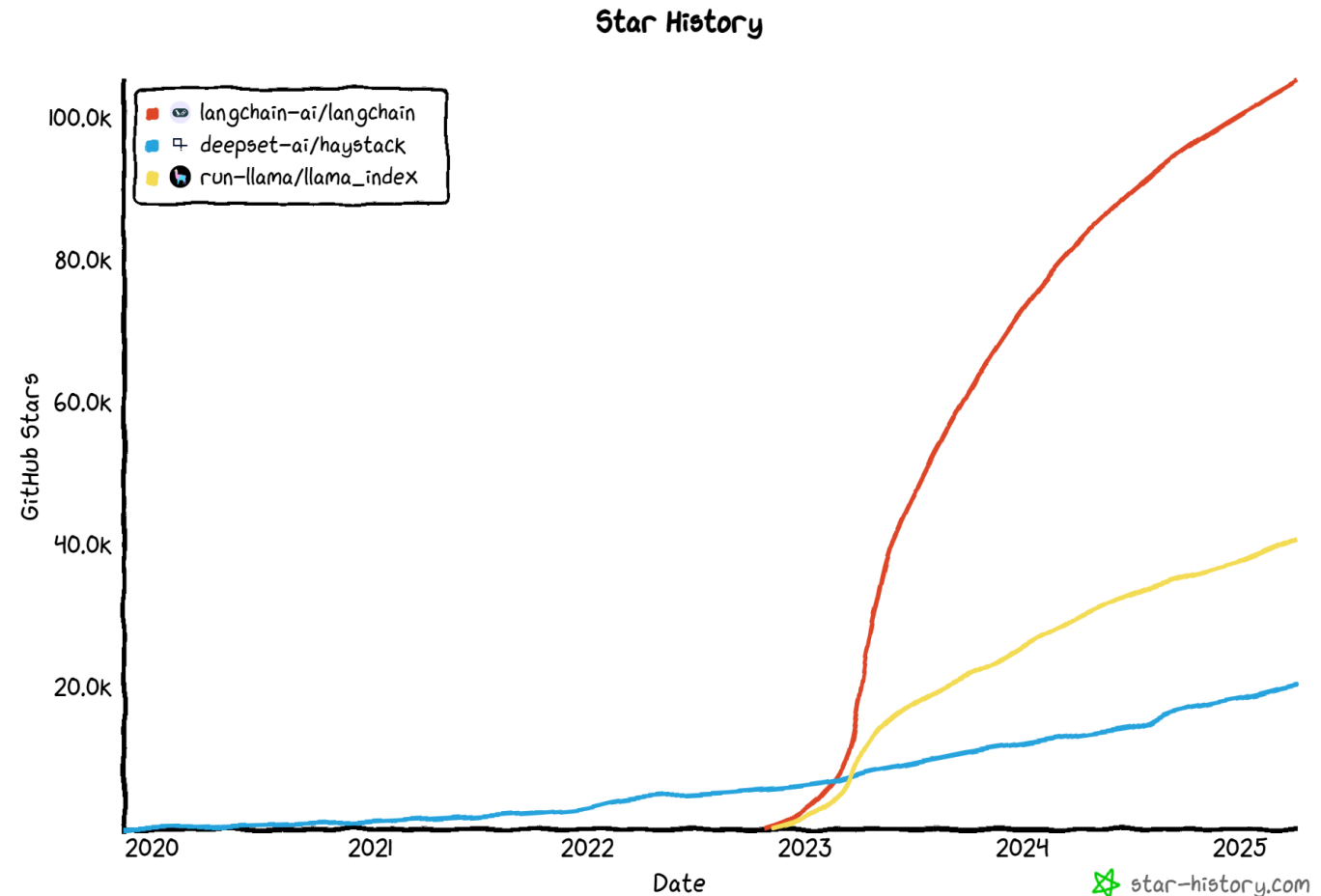


With the exception of the **langsmith** SDK, all packages in the LangChain ecosystem depends on langchain-core, which contains base classes and abstractions that other packages use.

Source: [https://python.langchain.com/docs/how\\_to/installation/](https://python.langchain.com/docs/how_to/installation/)

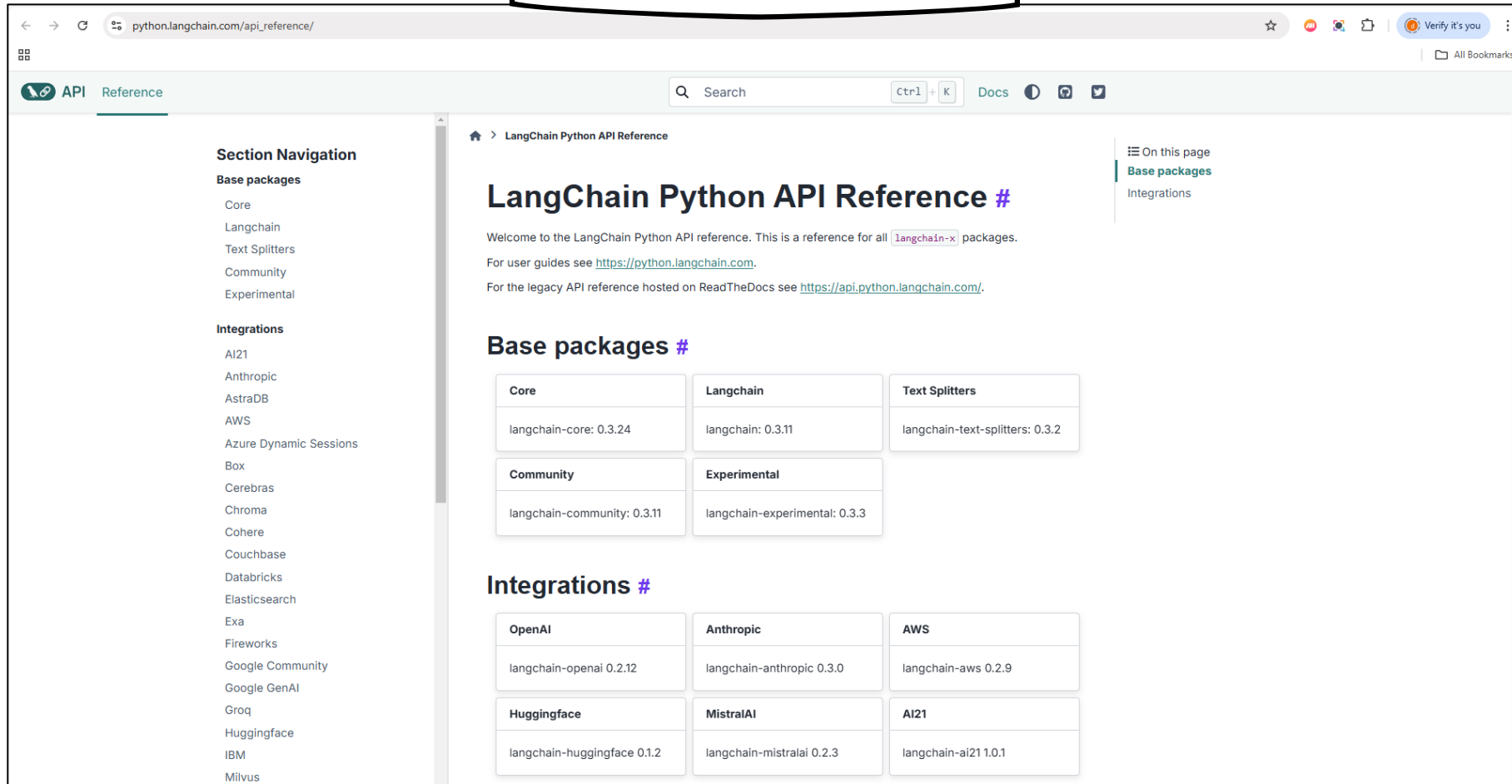
# Why LangChain?

- LangChain gains popularity in a short period of time
- Benefits:
  - Increased flexibility
  - Improved performance
  - Enhanced reliability
  - Open source



# LangChain API Documentation

Best Friend Forever



The screenshot shows the LangChain Python API Reference website. The browser address bar displays `python.langchain.com/api_reference/`. The page has a light green header with a search bar and navigation links. A left sidebar titled "Section Navigation" lists categories like "Base packages" and "Integrations". The main content area is titled "LangChain Python API Reference #" and includes a welcome message, user guides, and legacy API reference links. Below this, there are two sections: "Base packages #" and "Integrations #", each containing a grid of boxes listing specific packages and their versions.

**Section Navigation**

- Base packages**
  - Core
  - Langchain
  - Text Splitters
  - Community
  - Experimental
- Integrations**
  - AI21
  - Anthropic
  - AstraDB
  - AWS
  - Azure Dynamic Sessions
  - Box
  - Cerebras
  - Chroma
  - Cohere
  - Couchbase
  - Databricks
  - Elasticsearch
  - Exa
  - Fireworks
  - Google Community
  - Google GenAI
  - Groq
  - Huggingface
  - IBM
  - Milvus

**LangChain Python API Reference #**

Welcome to the LangChain Python API reference. This is a reference for all `langchain-x` packages.

For user guides see <https://python.langchain.com>.

For the legacy API reference hosted on ReadTheDocs see <https://api.python.langchain.com/>.

**Base packages #**

Core	Langchain	Text Splitters
langchain-core: 0.3.24	langchain: 0.3.11	langchain-text-splitters: 0.3.2

Community	Experimental
langchain-community: 0.3.11	langchain-experimental: 0.3.3

**Integrations #**

OpenAI	Anthropic	AWS
langchain-openai 0.2.12	langchain-anthropic 0.3.0	langchain-aws 0.2.9

Huggingface	MistralAI	AI21
langchain-huggingface 0.1.2	langchain-mistralai 0.2.3	langchain-ai21 1.0.1

[https://python.langchain.com/api\\_reference/](https://python.langchain.com/api_reference/)



# Best Practice for API Key(s) Handling

- OpenAI recommends that the API key(s) be handled with extreme care
- Never deploy your key in client-side environments like browsers or mobile apps
- Never commit your key to your repository (for example: GitHub)
- Load the API key via a text file or use environment variable in place of your API key
- Monitor your account usage and rotate your keys often or when needed



**Source:** <https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety>

# Getting Started with LangChain

- LangChain requires integrations with various model providers, data stores, APIs and other third-party components.
- You must provide relevant API keys for LangChain to function. Some methods to achieve this are:
  - Setting up key as an environment variable

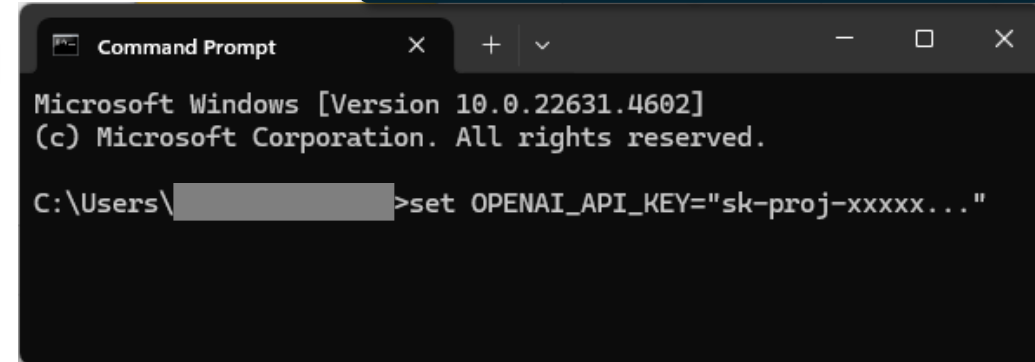
```
import getpass
import os

# setup the OpenAI API Key

# get OpenAI API key ready and enter it when ask
os.environ["OPENAI_API_KEY"] = getpass.getpass()
```

Enter the API key manually

Export the API key at Command Prompt



```
Microsoft Windows [Version 10.0.22631.4602]
(c) Microsoft Corporation. All rights reserved.

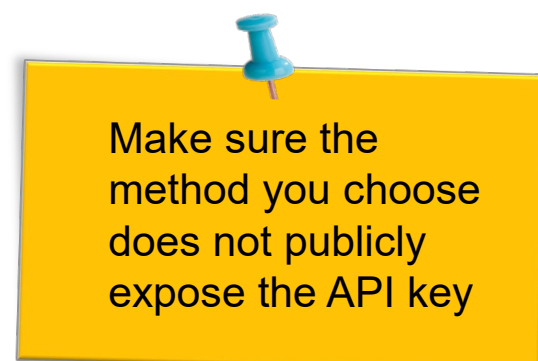
C:\Users\[redacted]>set OPENAI_API_KEY="sk-proj-xxxxx..."
```

- Load the key to an environment variable via a text file

```
import os
from dotenv import load_dotenv

# load OPENAI API key
load_dotenv('env.txt')
openai_api_key = os.getenv('OPENAI_API_KEY')
```

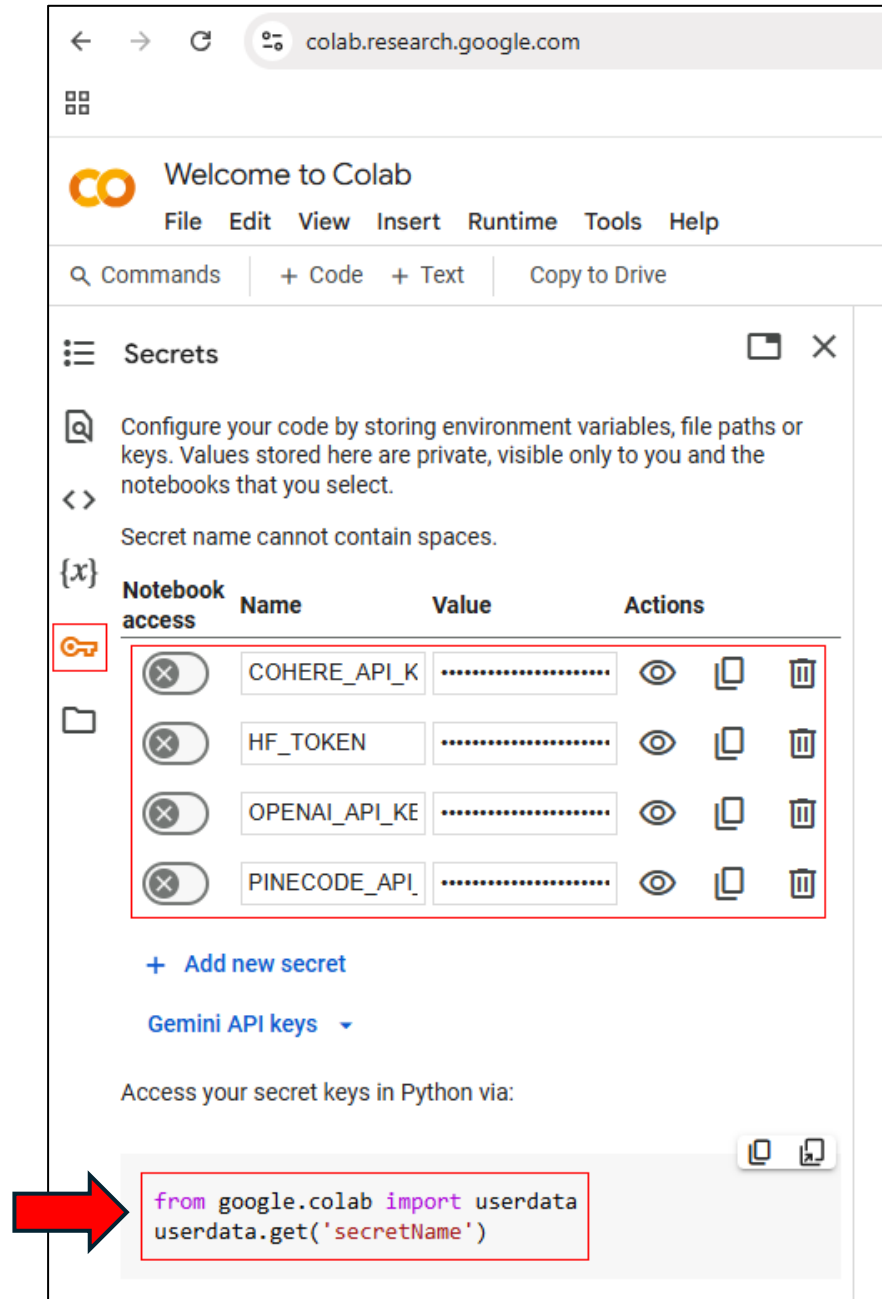
Load the API key from a file



Make sure the method you choose does not publicly expose the API key

# Google Colab

- Provides a convenient way to store and retrieve keys
- Remember: Do not hard code the API keys in your code and store it in GitHub.



Welcome to Colab













File Edit View Insert Runtime Tools Help

Q Commands + Code + Text Copy to Drive

Secrets

Configure your code by storing environment variables, file paths or keys. Values stored here are private, visible only to you and the notebooks that you select.

Secret name cannot contain spaces.

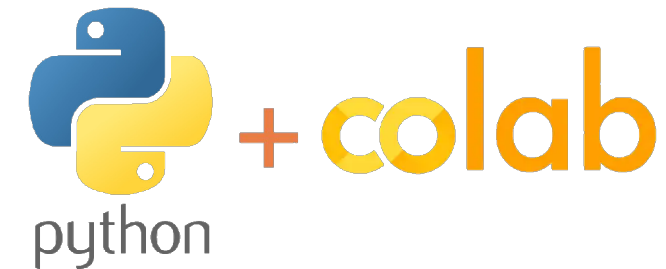
Notebook access	Name	Value	Actions
<input checked="" type="checkbox"/>	COHERE_API_K	.....	  
<input checked="" type="checkbox"/>	HF_TOKEN	.....	  
<input checked="" type="checkbox"/>	OPENAI_API_KE	.....	  
<input checked="" type="checkbox"/>	PINECODE_API_	.....	  

+ Add new secret

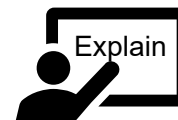
Gemini API keys ▾

Access your secret keys in Python via:

```
from google.colab import userdata
userdata.get('secretName')
```



# A Simple LLM Application



```
# load langchain libraries
from langchain_openai import ChatOpenAI
from langchain.schema import HumanMessage
```

Import the libraries

```
chat_model = ChatOpenAI(
    # don't need this if the OpenAI API Key is stored in the environment variable
    #openai_api_key="sk-proj-xxxxxxxxxx",
    model_name='gpt-4o-mini'
)
```

Setup chat model with model 'gpt-4o-mini'

```
# setup message prompt
text = "What date is Singapore National Day?"
messages = [HumanMessage(content=text)]
```

Setup the human message "What date is Singapore National Day"

```
# note that Chat Model takes in message objects as input and generate message object as output
```

```
response = chat_model.invoke(messages)
print(response.content)
```

## Output

Singapore National Day is celebrated on August 9th each year. It commemorates the country's independence from Malaysia in 1965.

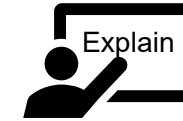


# Prompt Template

# Prompt Template

- Translate user input and parameters into instructions for an LLM
- Take as input a dictionary where each key represent a variable in the prompt template to fill in
- Is a string template we can pass variables to in order to generate the final prompt string
- LangChain documentation ➔ “A prompt template refers to a reproducible way to generate a prompt”

# A Simple LLM Application With Prompt Template



```
system_template = "You are a helpful assistant that translates {input_language} to  
{output_language}."
```

```
human_template = "{text}"
```

```
chat_prompt = ChatPromptTemplate.from_messages([  
    ("system", system_template),  
    ("human", human_template),  
)
```

```
# trnsnlate English to French
```

```
messages = chat_prompt.format_messages(  
    input_language="English",  
    output_language="French",  
    text="I love programming."  
)
```

```
response = chat_model.invoke(messages).content  
print(response)
```

-Setup system and human templates  
-Create the chat prompt template

Create the prompt and  
populate the values

**Output**  
J'adore la programmation.

# LLM API Provider Interface

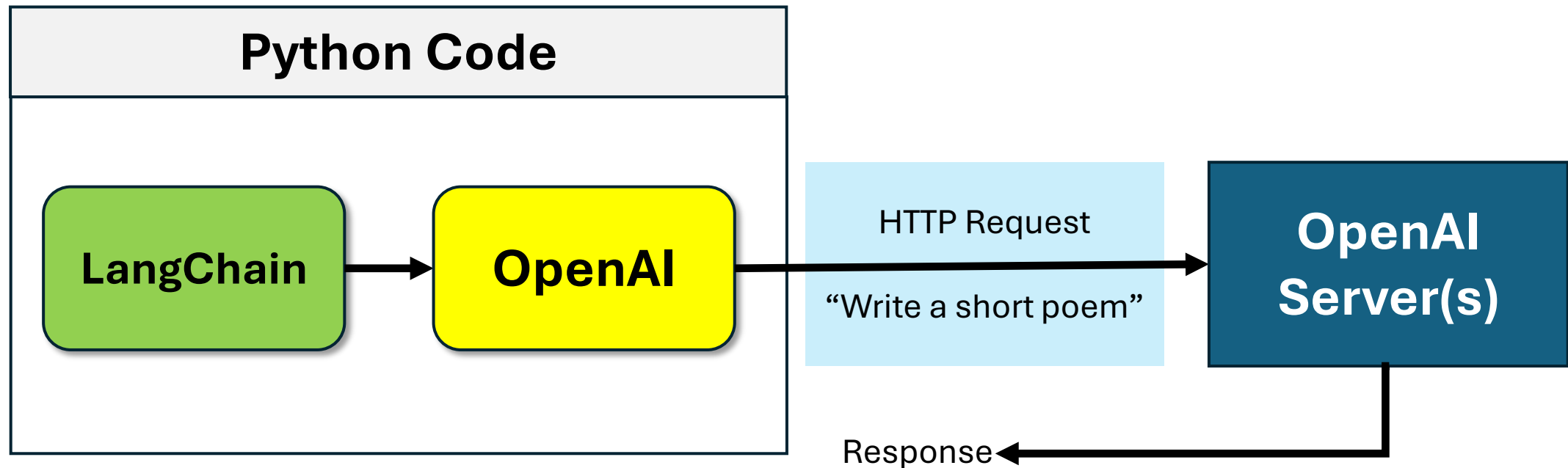
- LangChain provides integration for different types of model
- **LLM**: the model takes a text string as input and returns a text string
- **Chat models**: the model takes a list of chat messages as input and return a chat message. Specifically it has 3 major components:
  - HumanMessage – the prompt given by the user
  - SystemMessage – A context that can be passed to the ChatModel about its role
  - AIMessage – the response given by LLM



# Message Types

Role	Purpose	Usage Pattern
System	<ul style="list-style-type: none"> <li>Helpful background context that guides AI</li> <li>Set the behaviour or context for the conversation. It defines how the AI should respond and what role it should play</li> </ul>	<pre>{   "role": "system", "content": "You are a helpful assistant   that specializes in programming and technology." }</pre>
Human (User)	<ul style="list-style-type: none"> <li>Message representing the input from the user. It is what the AI receives as a query or prompt to generate a response</li> </ul>	<pre>{   "role": "human", "content": "Can you explain the   difference between a list and a tuple in Python?" }</pre>
AI (Assistant)	<ul style="list-style-type: none"> <li>The response generated by the model. It addressed the user's query based on the context provided by the system message</li> </ul>	<pre>{   "role": "assistant", "content": "Sure! In Python, a list is a   mutable sequence, meaning you can modify its elements   after creation, while a tuple is immutable, meaning it   cannot be changed once defined. Tuples are generally   used for fixed collections of items." }</pre>

# How it work?



\*The LLM by OpenAI does not reside in your local machine.

\*LangChain also works with open source LLM model like Llama from Meta. You can host the LLM model locally.

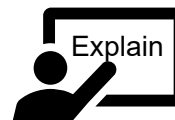


# LangChain Expression Language

# LangChain Expression Language (LCEL)

- LangChain provides a declarative way to compose chains that is more intuitive and productive than directly write code
- The different components of LCEL are placed in a sequence which is separated by a pipe symbol ( `|` )
- The chain or LCEL is executed from left to right
  - For example: `chain = prompt | model | output_parser`
  - The prompt output is piped to the LLM `model`. The output of the LLM `model` is then piped to `output_parser` which extracts the text in the output
- LCEL is a method to create arbitrary custom chains. It is built on the Runnable protocol.

# Prompt Template & LCEL



```
from langchain_core.output_parsers.string import StrOutputParser

llm = ChatOpenAI(
    model_name='gpt-4o-mini',
    temperature=0.7,
)

output_parser = StrOutputParser()

human_template = "Write {lines} sentences about {topic}."
prompt = ChatPromptTemplate.from_template(human_template)

lines_topic_dict = {
    "lines": "3",
    "topic": "Sir Stamford Raffles"
}

lcel_chain_02 = prompt | llm | output_parser

lcel_chain_02.invoke(lines_topic_dict)
```

Parse the output of a language model into a string format

Create the prompt template

Setup the template parameter dictionary

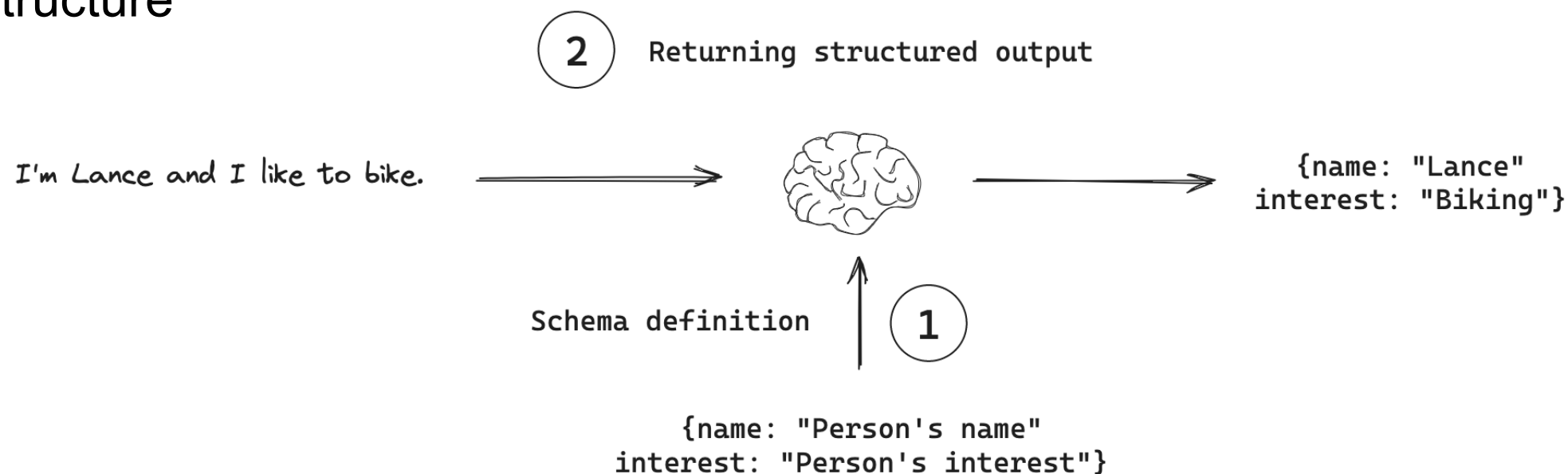
Use LCEL notation to pipe from prompt to output parser

## Output

Sir Stamford Raffles was a British statesman ...

# Structured Output

- LLM model responds to user directly in natural language
- There are scenarios where the outputs needs to be in some structured format
- In LangChain, models can be instructed to output a particular output structure



Source: [https://python.langchain.com/docs/concepts/structured\\_outputs/](https://python.langchain.com/docs/concepts/structured_outputs/)



# Persistence

# Memory

- LLMs are stateless – each incoming query is processed independently of the other interactions i.e. in other words, LLMs don't save anything
- Memory allows a LLM to remember previous interactions with the user
- There are many applications where remembering previous interactions is important such as chatbots. Memory persistence allows us to do that
- There are few pros and cons using memory persistence

Pros	Cons
Maximum information due to storing everything during conversation exchanges	Increase in token counts will slow down response times and higher costs
API is simple and intuitive	Limited by LLM context window limit 4096 token for text-davinci-003 and gpt-3.5-turbo

Source: [https://python.langchain.com/api\\_reference/langchain/memory.html#](https://python.langchain.com/api_reference/langchain/memory.html#)



# Memory Persistence

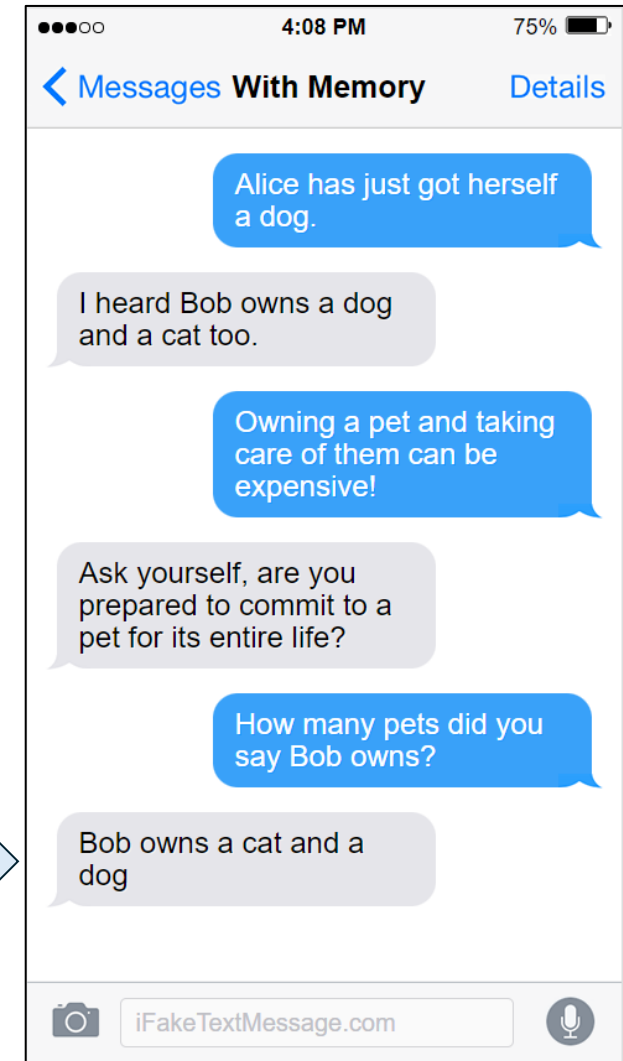
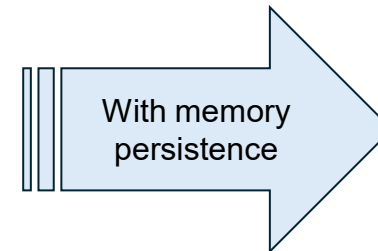
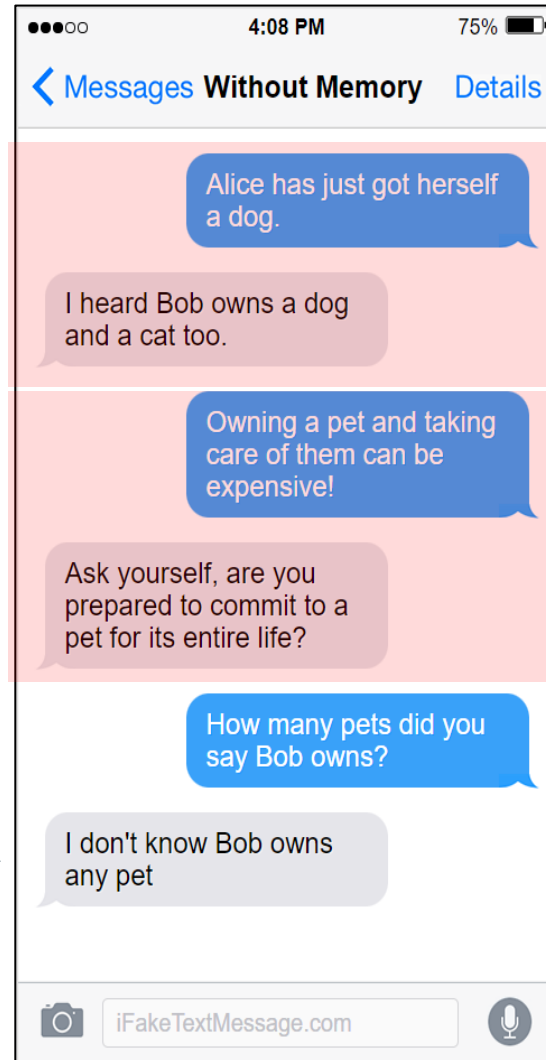
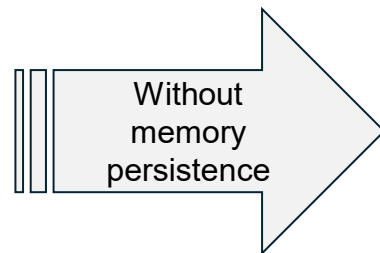
- The basic implementation is to simply stores the conversation history
- Additional processing may be required in some situations when the conversation history is too large to fit in the content window of the model
- One method is to summarize each conversation before storing it
- Starting from v0.3 release of LangChain, LangChain is recommending user to take advantage of LangGraph persistence to incorporate memory into new LangChain applications

# Context Window (Recap)

$\leq$  Max Tokens



# Conversation History





# Activity 01



# Activity

- In this activity, we will be creating chat model that takes in a sequences of message and returns chat messages as outputs
- LangChain does not host any of the chat models. It depends on [third party](#) LLM model providers
- Prompt template serves as a flexible framework for creating dynamic and reusable prompts
- LCEL is the preferred way to define workflows
- LangChain Parsers are components to process or transform the output generated by an LLM
- LLMs are inherently stateless. To enable a more seamless and context-aware user experience, memory persistence is crucial when working with LLMs

# Reference

- LangChain  
<https://www.langchain.com/>
- LangChain Documentation  
<https://python.langchain.com/docs/introduction/>
- LangChain Tutorial  
[https://python.langchain.com/docs/tutorials/llm\\_chain/](https://python.langchain.com/docs/tutorials/llm_chain/)
- LangGraph  
<https://langchain-ai.github.io/langgraph/tutorials/introduction/>



# Pydantic

# Introduction to Python Type Hints

- Python is a dynamically-typed language where the type of a variable is determined at runtime.
- **PEP 484** (Python Enhancement Proposal) introduces type hints to Python, laying the foundation for static type checking.
- Type hints introduce a layer of optional static typing allowing developer to check for type consistency without enforcing types at runtime.
- Static type checking verifies the type safety of a program based on analysis of a source code. Static type checking is done at compile-time.
- Python type hints do not enforce type checking at runtime but provide guidance for developers.
- It can be applied to function parameters and return types, which aids in understanding what is expected when calling a function.
- It supports complex types, generic and user-defined type.



# Introduction to Python Type Hints

- Features like Optional, Union and Any enable more flexible type annotations while Generics allow coding functions and classes that work with multiple types.
- Static type checking like `pyright` analyses code for type consistency before execution.
- It helps to catch potential bugs early by using the provided type hints to check for mismatches in types during code development.
- Type hints are an invaluable tool for adding readability and reliability to your Python code, especially in larger projects or collaborative environment.
- Introduced in Python starting from version 3.5, type hints don't enforce types at runtime. They acts as optional metadata for development tool.

# Why Use Type Hints?

## 1. Enhance Code Readability

- Easier to understand what the type of data each function/variable expects and each function returns.

## 2. Debugging Facilitation

- Aids in identifying type related errors.

## 3. Development Experience

- IDEs (Visual Studio Code/PyCharm etc.) use type hints for better code completion and error detection.

## 4. Static Type Checking

Tools like `mypy` use type hints to perform type checking.

# Pydantic – An Introduction

- Pydantic and Python type hints share similarities but serve different purposes in Python development.
- A data validation library that extends Python type hints with **runtime** type-checking and data validation.
- Define models with types, and it ensures that the data matches these types by performing validation when data is passed in.
- Performs both type hints and automatic type enforcement and validation, which can raise errors if the data doesn't match the specified structure.
- Performs runtime validation based on the types you specify, ensuring that input data matches the expected types and constraints.
- Pydantic is widely used across various domains in the Python ecosystem, particularly for data validation and settings management.

# Why Pydantic?

Learning Pydantic for LLM applications with frameworks like **LangChain** or **LlamaIndex** can bring substantial benefits, particularly for handling and validating data efficiently.

Here are some key reasons why Pydantic is valuable in these contexts:

- Large Language Model (LLM) applications often involve complex data pipelines, including inputs from users, external APIs, and databases.
- Pydantic makes it easy to define clear data models with strict validation, ensuring that inputs to your LLMs are accurate and reducing the risk of runtime errors.
- With Pydantic, you can define exact data structures for LLM inputs, intermediate processing, and outputs. This is especially helpful when managing prompts, responses, and metadata in frameworks like LangChain, where data consistency is crucial.

# Pydantic Benefits

- Pydantic's type-based modeling encourages clearer and more maintainable code by using Python type hints. This makes data structures in LangChain or LlamaIndex pipelines easier to read and manage.
- It allows you to model prompt templates and response schemas explicitly. Pydantic can ensure each template and prompt step conforms to an expected structure.
- For response parsing, Pydantic's schema validation can help ensure that outputs from LLMs match a predictable structure, which is critical when chaining multiple LLM calls or interacting with other services.
- As LLM applications grow in complexity, Pydantic helps to create scalable, reusable code components. The well-defined data models can act as clear documentation for how different parts of your application communicate, making it easier to onboard new developers or extend existing workflows.

# Pydantic Vs Python Type Hints

Feature	Python Type Hints	Pydantic
Validation Time	Static type checking only	Runtime validate
Enforcement Type	No runtime enforcement	Strict runtime enforcement
Dependencies	Built into Python 3.5 and later versions	Requires external package
Serialization	Not supported	Built0in serialization (dict, JSON)
Error Handling	Relies on static analysis	Raises detailed errors at runtime
Data coercion	None	Automatic type conversion when possible

Python type hints are great for type-checking during development, while Pydantic provides powerful runtime validation and coercion, making it ideal for situations where data consistency and integrity are crucial, such as with external data sources or complex LLM applications.

# Summary

- Integrating Pydantic into LangChain or LlamaIndex workflows will allow you to streamline data handling, enhance robustness, and ensure type-safe operations across your LLM application pipeline.
- This structured approach can be invaluable for both development efficiency and long-term maintainability.

# References

- Pydantic  
<https://docs.pydantic.dev/latest/>
- Pydantic (v2) – In-depth Starter Guide  
<https://www.youtube.com/watch?v=ok8bF8M7gjk>
- Pydantic V2 – Full course – Learn the BEST library for Data Validation and Parsing  
[https://www.youtube.com/watch?v=7aBRk\\_JP-qY](https://www.youtube.com/watch?v=7aBRk_JP-qY)
- Python Pydantic Tutorial – Learn how to write advanced classes in Python  
<https://www.youtube.com/watch?v=pK6us8n9cLk>
- Pydantic – Validators (Root Validators, Pre-Item Validators, Each-Item Validators)  
<https://www.youtube.com/watch?v=nQJKkY8XnRg>

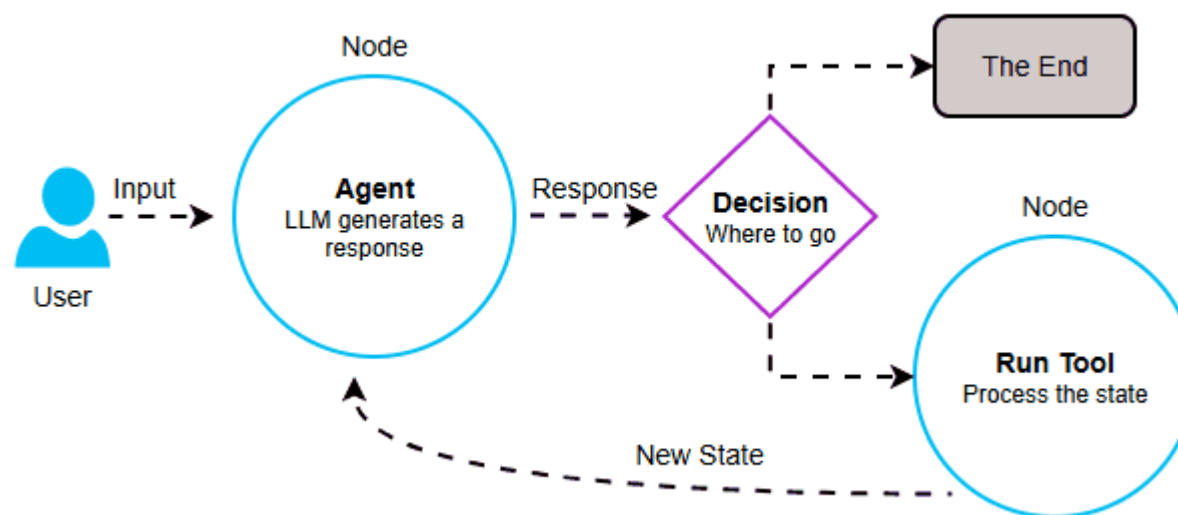




# LangGraph and Graph

# Introduction To LangGraph

- LangGraph is an advanced library built on top of LangChain
- It is designed to enhanced Large Language Model (LLM) application by implementing cyclic computation capabilities i.e. a graph
- LangChain allows the creation of Directed Acyclic Graph (DAGs) for linear workflows (sequential chain), LangGraph takes this a step further by enabling the addition of cycles
- This addition is essential for developing complex, agent-like behaviours allowing LLM to continuously loop through a process, dynamically deciding what action to take next based on changing conditions

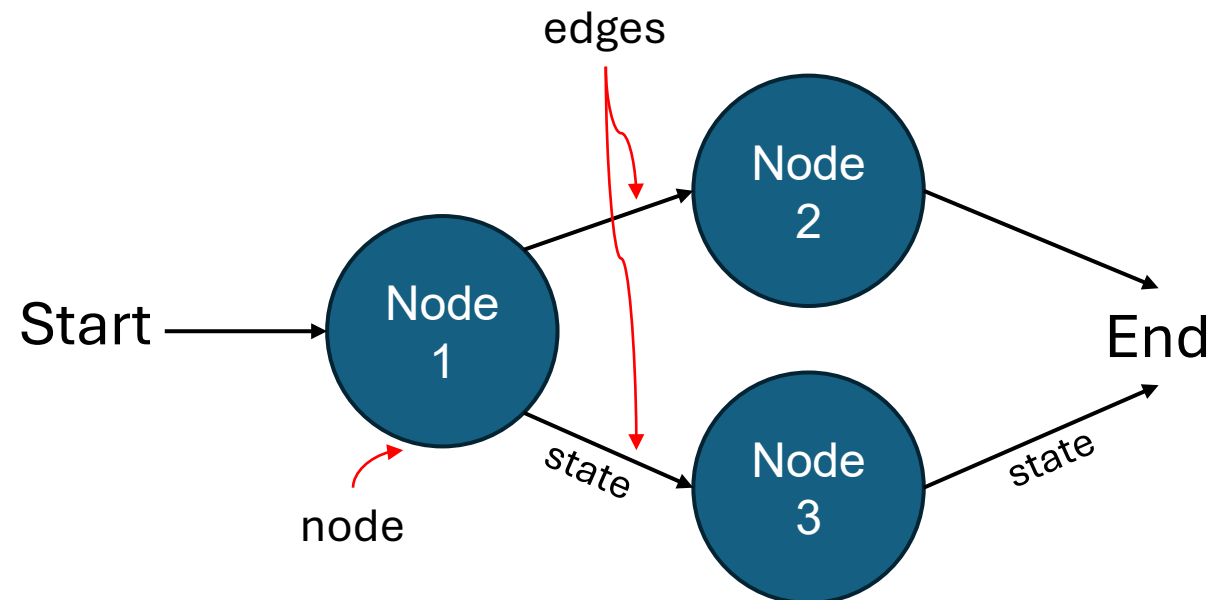


# Graph

- A graph is a structured representation of **nodes** (or vertices) and **edges** (or connections) used to model workflows, data flows or relationship within a Large Language Model (LLM)-based system.
- The components in a graph are:
  - Nodes:
    - Represent discrete tasks, **functions** or prompts
    - Example:
      - Send a prompt to an LLM
      - Perform a specific computation or operation
  - Edges
    - Represent the flow of data or control between nodes
    - Define how the output of one node is passed as the input to another, creating a logical sequence of tasks
    - Conditional edges are used when want to optionally route between nodes. It is implemented as a function

# Graph

- State
  - The state of the graph is defined using a state schema
  - It is defined either using **TypedDict** class from Python's typing module or **Pydantic** model
  - Maintains and updates the memory as the process advances
  - Contains relevant information from earlier steps for decision making
- Directed Graph
  - LangGraph workflows are typically represented as directed graphs, where the direction of edges indicates the flow of execution or data (e.g., from one LLM query to a subsequent operation)



# LangGraph Architecture

- In a LangGraph-style architecture, **nodes** represent individual tasks or operations
- **Edges** define the flow of information between these tasks. Every node is a function.
- Each node in the graph encapsulates a specific piece of functionality. It takes an input, performs a computation or action and produces an output.
- Edge as data flow. The connections between nodes represent the flow of data or the results from one function to the next.
- A **state** schema serves as a blueprint or structure for defining and validating the data that a workflow state holds during execution. It ensures consistency and prevents errors by enforcing rules on the data that each state in the workflow processes.

# Schema Definitions

## Pydantic

```
from pydantic import BaseModel
from typing import List, Optional

class StateSchema(BaseModel):
    user_query: str
    current_step: str
    results: List[str] = []
    error: Optional[str] = None
```

- user\_query: The input provided by the user (required, type str)
- current\_step: Tracks which node in the graph is currently processing the data
- results: A list to accumulate intermediate or final results (default is an empty list)
- error: An optional field to handle errors

# Vs

## TypedDict

```
from typing import TypedDict, List,
Optional

class StateSchema(TypedDict):
    user_query: str
    current_step: str
    results: List[str]
    error: Optional[str]
```

- user\_query: A required string representing the user's input
- current\_step: A string indicating the current stage of the workflow
- results: A list of strings to store intermediate or final results
- error: An optional string to capture error messages

# Building A Simple Graph

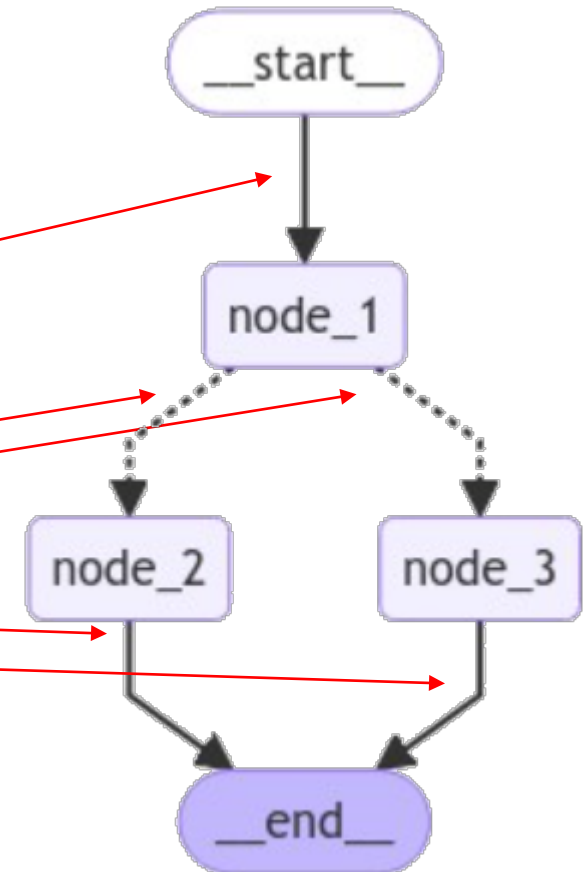
```
from langgraph.graph import StateGraph, START, END

# Build graph
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# Logic
builder.add_edge(START, "node_1")
builder.add_conditional_edges("node_1", decide_mood)
builder.add_edge("node_2", END)
builder.add_edge("node_3", END)

# Add
graph = builder.compile()

# View
display(Image(graph.get_graph().draw_mermaid_png()))
```



# Build A Simple Graph

```
import random
from typing import Literal

def decide_mood(state) -> Literal["node_2", "node_3"]:

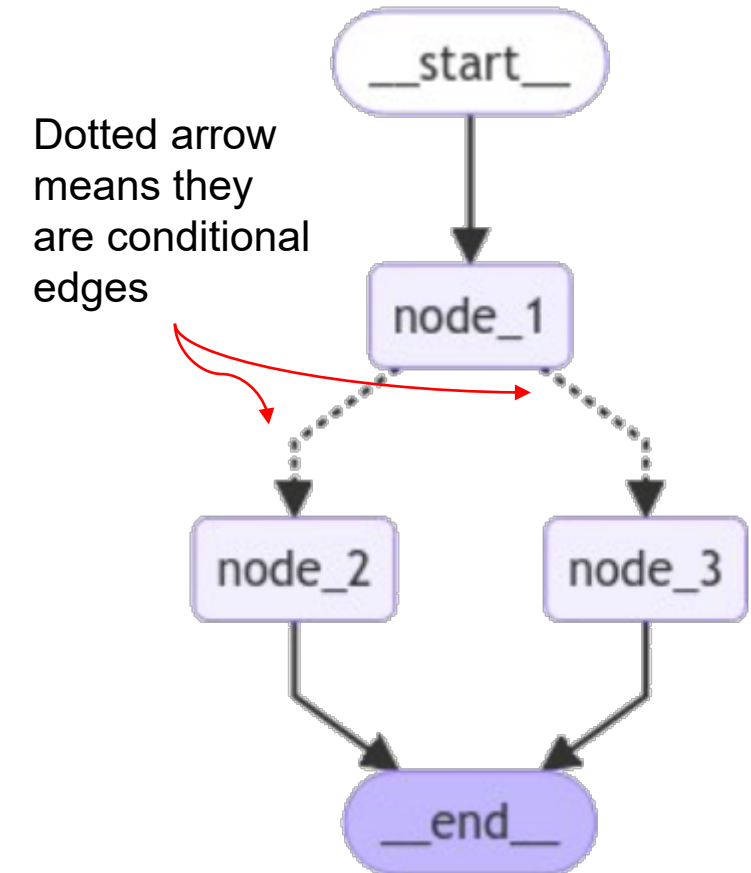
    # Often, we will use state to decide on the next node to visit
    user_input = state['graph_state']

    # Here, let's just do a 50 / 50 split between nodes 2, 3
    if random.random() < 0.5:

        # 50% of the time, we return Node 2
        return "node_2"

    # 50% of the time, we return Node 3
    return "node_3"
```

Arbitrarily example of a conditional node



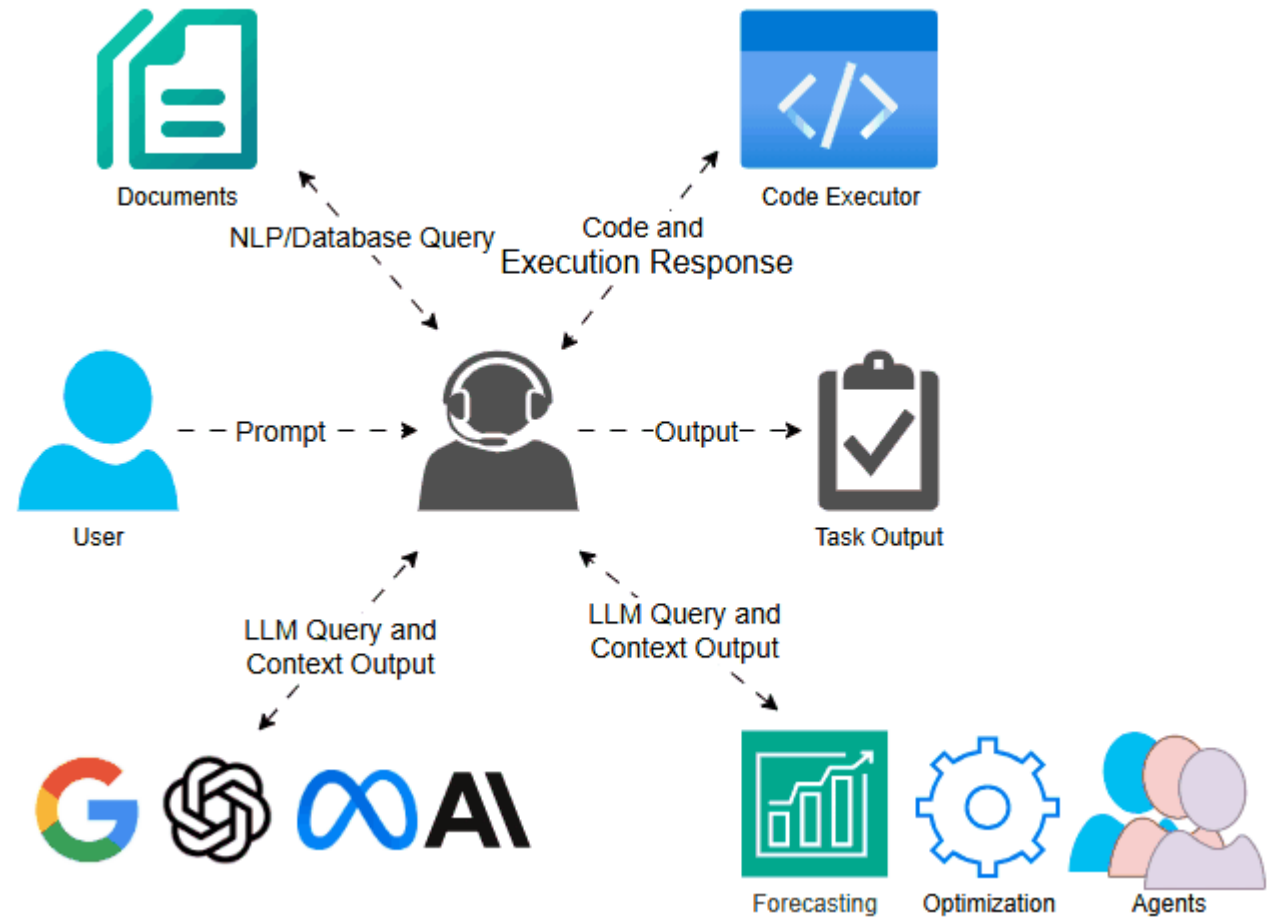




# Agent

# From Monolithic Models ...

- An AI agent is a system that uses LLM to decide the control flow of an LLM application.



# Agent

- An agent is an autonomous entity that can execute tasks, interact with various components, and make decisions based on the given instructions or data
  - Agents can perform tasks independently, such as querying data, executing code, or interacting with APIs
  - An agent typically works to achieve a specific objective, such as answering a question, processing input, or generating content
  - Based on the data they receive and the rules they are programmed with, agents can decide on the next steps in a workflow
- The ReAct agent is an advanced AI agent design pattern that integrates **Reasoning and Acting** in a unified workflow
  - Reasoning: The agent generates thoughts to reason through a problem step-by-step.
  - Acting: The agent performs actions to interact with the environment or retrieve necessary information.

# Agentic AI

- A concept first introduced by Andrew Ng (a prominent figure in AI)
- Agentic AI surpasses the reactive capabilities of Generative AI by acting autonomously, planning ahead and adapting to user's needs
- Key features of Agentic AI
  - Autonomy:
    - Operates independently, making decisions based on goals, constraints, and environmental inputs
    - Requires minimal human oversight for task execution
  - Goal-Oriented Behaviour:
    - Works toward specific objectives or goals defined by the user or its programming
    - Breaks down complex tasks into smaller, manageable steps
  - Contextual Understanding
    - Uses natural language understanding (NLU) to process user instructions or environmental data
    - Maintains context over time, especially in multi-step processes or dialogues

# Agentic AI

- Reasoning and Problem-Solving
  - Employs logic, heuristics, or LLMs for decision-making
  - Can evaluate multiple solutions or approaches to select the most effective one
- Tool Integration and Action Execution
  - Interacts with external tools, APIs, databases, and environments
  - Capable of performing actions like querying a database, making an API call, or writing and executing code
- Dynamic Planning and Adaptation
  - Plans actions dynamically, updating its strategy based on real-time feedback or new information
  - Adapts to unexpected challenges or changes in the environment
- Multi-agent collaboration
  - Multiple AI agents collaborate to achieve better outcomes than a single agent could



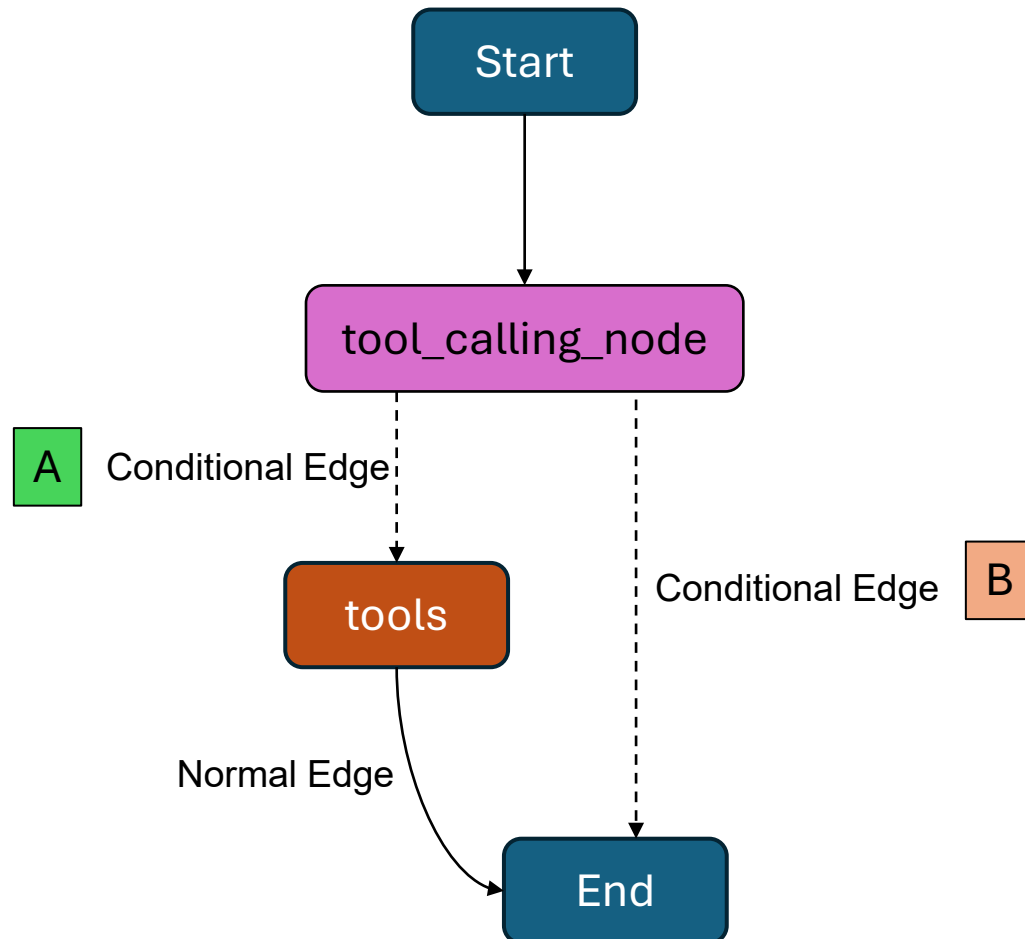
# Router

# Router

- Router is to route logic
- For example, to create a helpful assistant, the assistant knows when to use tools and when to simply respond
- It is like a worker at desk – sometimes they need to use tool to help complete the tasks (like a calculator or reference guide) and other times, he/she can just answer directly
- Main components are:
  - The model decides whether a tool is required
  - A special node called ToolNode that knows how to use different tools when needed
  - A routing system that looks at each response and decide:
    - Let's use the tool and then continue the task execution
    - No tool is needed. Continue other tasks till end

# Tool Calling Node

- Tool calling node decides whether to go with route A or route B



```

# Tool node
def tool_calling_node(state: MessagesState):
    return {"messages": [model_with_tools.invoke(state["messages"])]}

builder = StateGraph(MessagesState)

builder.add_node("tool_calling_node", tool_calling_node)
builder.add_node("tools", ToolNode([addition]))

# Set the entry point - start with the tool calling node
builder.add_edge(START, "tool_calling_node")

builder.add_conditional_edges(
    "tool_calling_node",
    tools_condition
)

builder.add_edge("tools", END)

graph = builder.compile()
  
```



# Custom Node Type

- A user defined component in the workflow chain
- Unlike pre-build nodes, a custom node implements your own logic, tailored to your application needs which implement complex agent logic
- There are two primary ways to include custom logic:
  - Define a Python function and add it as a node to the graph
  - Use a “custom” node type

Scenario	Python Function Node	Custom Node
Simple logic that doesn't require additional features	<input checked="" type="checkbox"/> Preferred	<input type="checkbox"/> Not Recommend
Workflow with potential for debugging or error handling	<input type="checkbox"/> Limited support	<input checked="" type="checkbox"/> Preferred
Need to reuse the function independently	<input checked="" type="checkbox"/> Easy to reuse	<input type="checkbox"/> Tie to the graph node

# Custom Node Type

- LangGraph allows user to create custom node types to implement complex agent logic

```
class MyCustomNode:
    def __init__(self, llm):
        self.llm = llm

    def __call__(self, state):
        # Implement your custom logic here
        # Access the state and perform actions
        messages = state["messages"]
        response = self.llm.invoke(messages)
        return {"messages": [response]}

graph_builder = StateGraph(State)

llm = ChatOpenAI(model="...")

custom_node = MyCustomNode(llm)

graph_builder.add_node("custom_node", custom_node)
```



# Activity



# Activity

- Activities to code in LangGraph. LangGraph is a framework designed to model workflow as directed graph
- In a LangGraph-style architecture, nodes represent individual tasks or operations
- Edges define the flow of information between these tasks. Every node is a function
- Each node in the graph encapsulates a specific piece of functionality. It takes an input, performs a computation or action and produces an output
- You will be creating:
  - Basic Chatbot
  - Tools whenever you want a model to interact with external systems
  - Agent
  - Human intervention

# Summary

- To build a graph, here are the steps:
  1. First define the State of the graph.
    - The State schema serves as input schema for all Nodes and Edges in the graph, You can use either TypedDict class or Pydantic library to define the schema
  2. Create the node.
    - The nodes are just Python functions or it can be a custom node
  3. Connect the nodes using Edges.
    - Normal edges are used when it always go from, for example Node A to Node B. Conditional edges are used to route between 2 or more nodes
  4. Construct the graph from StateGraph class.
    - Initialise the StateGraph with the State class defined earlier
    - Build the graph by compiling it
  5. Invoke the graph
    - When invoke is called, the graph starts execution from the START node
    - The execution continues until it reaches the END node

# Reference

- LangChain – LangGraph Quick Start  
<https://langchain-ai.github.io/langgraph/tutorials/introduction/>
- LangChain Academy – LangGraph  
<https://academy.langchain.com/courses/intro-to-langgraph>
- LangGraph Glossary  
[https://langchain-ai.github.io/langgraph/concepts/low\\_level/#multiple-schemas](https://langchain-ai.github.io/langgraph/concepts/low_level/#multiple-schemas)

# Thank you!