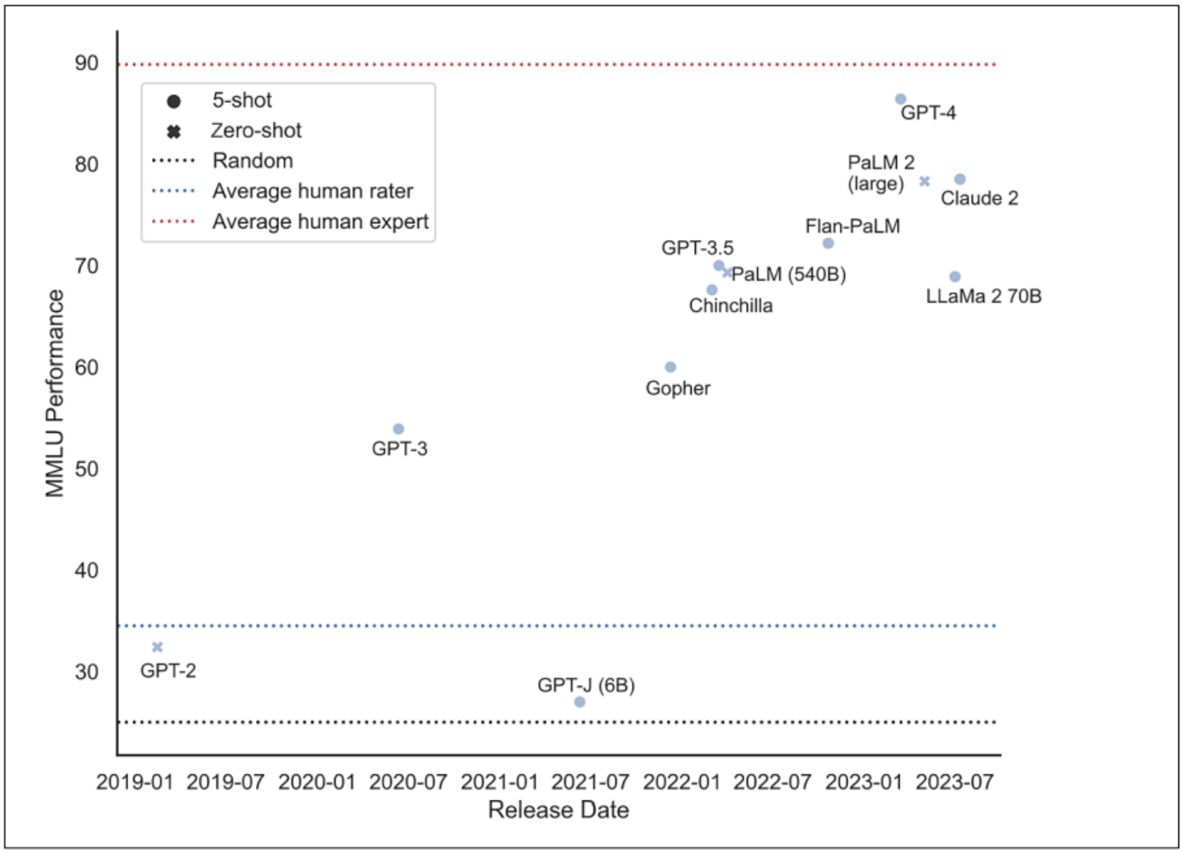
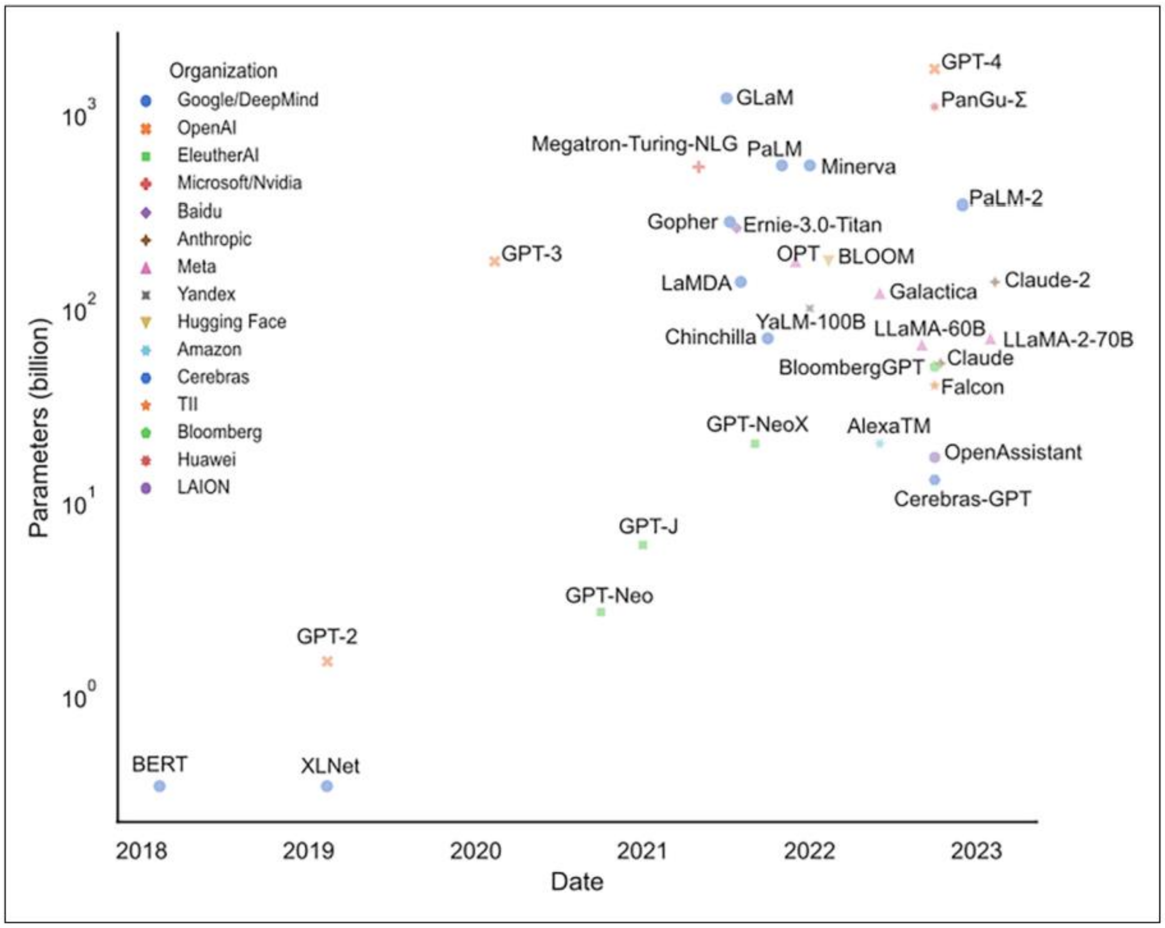
Generative AI:

Chapter 1: What Is Generative AI?





LLMs:

Generative AI: **Generative AI** refers to algorithms that can generate novel content, as opposed to analyzing or acting on existing data like more traditional, predictive machine learning or AI systems.

Zero-shot settings: Zero-shot means the models were prompted with the question.

5-shot settings: 5-shot settings, models were additionally given 5 question-answer examples.

GPT-1:

- Introduced by OpenAI in 2018
- trained on BookCorpus with 985 million words.

Generative Pre-trained Transformer (GPT-2): 1.5 billion

(GPT-3): 175 billion

- Trained on 300 billion tokens
- GPT-3 has 175 billion parameters

(GPT-4): a trillion

- Advancement in March 2023
- Trained on about 13 trillion tokens.
- About 1.8 trillion parameters

Palm:

- 540B

Palm2:

- the model behind Google’s chatbot Bard
- released in May 2023
- consisting of 340 billion parameters
- trained with the focus of improving multilingual and reasoning capabilities while being more compute efficient.
- PaLM 2 has significantly improved quality on downstream tasks, including multilingual common sense and mathematical reasoning, coding, and natural language generation, compared to its predecessor PaLM.
- PaLM 2 was also tested on various professional language-proficiency exams. The exams used were for Chinese Japanese, Italian, French, and Spanish. Across these exams, which were designed to test C2-level proficiency, considered mastery or advanced professional level according to the **CEFR** (**Common European Framework of Reference for Languages**), PaLM 2 achieved mostly high-passing grades.

Llama:

- LLaMa was released under a non-commercial license,
- one of the first large open-source models, which was released by Meta
- `llama.cpp` is a C++ toolkit that executes models based on architectures based on or like LLaMA
- One of the main use cases of `llama.cpp` is to run models efficiently on the CPU; however, there are also some options for GPU.

Llama2:

- **Released by** by Meta AI in February and July 2023
- up to 70B parameters
- Meta's LLaMa 2 model, with a size of up to 70 billion parameters, was trained on 1.4 trillion tokens
- LLaMa triggered the creation of models such as Vicuna, Koala, RedPajama, MPT, Alpaca, and Gorilla
- LLaMa 2 is an updated version of LLaMa 1 trained on a new mix of publicly available data.
- The pre-training corpus size has increased by 40% (2 trillion tokens of data), the context length of the model has doubled, and grouped-query attention has been adopted
- Variants of LLaMa 2 with different parameter sizes (7B, 13B, 34B, and 70B) have been released.
- LLaMa 2 are open to the general public for research and commercial use.

Claude:

- AI assistants created by Anthropic

Claude 2:

- released in July 2023
- best GPT-4 competitors in the market
- Key model improvements include an expanded context size of up to 200K tokens, far larger than most available models, and being commercial or open source. It also performs better on use cases like coding, summarization, and long document understanding.
- Claude 2 still has limitations in areas like confabulation, bias, factual errors, and potential for misuse, problems it has in common with all LLMs. Anthropic is working to address these through techniques like data filtering, debiasing, and safety interventions.
-

The importance of the number of parameters in an LLM: The more parameters a model has, the higher its capacity to capture relationships between words and phrases as knowledge. As a simple example of these higher-order correlations, an LLM could learn that the word "cat" is more likely to be followed by the word "dog" if it is preceded by the word "chase," even if there are other words in between. Generally, the lower a model's perplexity, the better it will perform, for example, in terms of answering questions.

OpenAI is a US AI research company that aims to promote and develop friendly AI. It was established in 2015 with the support of several influential figures and companies, who pledged over $1 billion to the venture. The organization initially committed to being non-profit, collaborating with other institutions and researchers by making its patents and research open to the public. In 2018, Elon Musk resigned from the board citing a potential conflict of interest with his role at Tesla. In 2019, OpenAI transitioned to become a for-profit organization, and subsequently Microsoft made significant investments in OpenAI, leading to the integration of

OpenAI systems with Microsoft’s Azure-based supercomputing platform and the Bing search engine. The most significant achievements of the company include OpenAI Gym for training reinforcement algorithms, and – more recently – the GPT-n models and the DALL-E generative models, which generate images from text.

Generative Models are a type of ML model that can generate new data based on patterns learned from input data

- **Language Models** (**LMs**) are statistical models used to predict words in a sequence of natural language. Some language models utilize deep learning and are trained on massive datasets, becoming **large language models** (**LLMs**). Text generation models, such as GPT-4 by OpenAI, can generate coherent and grammatically correct text in different languages and formats.

applications of language models:

- **Question answering**: AI chatbots and virtual assistants can provide personalized and efficient assistance, reducing response times in customer support and thereby enhancing customer experience. These systems can be used in specific contexts like restaurant reservations and ticket booking.
- **Automatic summarization**: Language models can create concise summaries of articles, research papers, and other content, enabling users to consume and understand information rapidly.
- **Sentiment analysis**: By analyzing opinions and emotions in texts, language models can help businesses understand customer feedback and opinions more efficiently.
- **Topic modeling**: LLMs can discover abstract topics and themes across a corpus of documents. It identifies word clusters and latent semantic structures.
- **Semantic search**: LLMs can focus on understanding meaning within individual documents. It uses NLP to interpret words and concepts for improved search relevance.
- **Machine translation**: Language models can translate texts from one language into another, supporting businesses in their global expansion efforts. New

generative models can perform on par with commercial products (for example, Google Translate).

- **Text-to-text**: Models that generate text from input text, like conversational agents. Examples: LLaMa 2, GPT-4, Claude, and PaLM 2.
- **Text-to-image**: Models that generate images from text captions. Examples: DALL-E 2, Stable Diffusion, and Imagen.
- **Text-to-audio**: Models that generate audio clips and music from text. Examples: Jukebox, AudioLM, and MusicGen.
- **Text-to-video**: Models that generate video content from text descriptions. Example: Phenaki and Emu Video.
- **Text-to-speech**: Models that synthesize speech audio from input text. Examples: WaveNet and Tacotron.
- **Speech-to-text**: Models that transcribe speech to text [also called **Automatic Speech Recognition** (**ASR**)]. Examples: Whisper and SpeechGPT.
- **Image-to-text**: Models that generate image captions from images. Examples: CLIP and DALL-E 3.
- **Image-to-image**: Applications for this type of model are data augmentation such as super-resolution, style transfer, and inpainting.
- **Text-to-code**: Models that generate programming code from text. Examples: Stable Diffusion and DALL-E 3.
- **Video-to-audio**: Models that analyze video and generate matching audio. Example: Soundify.

LLMs are deep neural networks adept at understanding and generating human language.
The current generation of LLMs such as ChatGPT are deep neural network architectures that utilize the transformer model and undergo pre-training using unsupervised learning on extensive text data, enabling the model to learn language patterns and structures.

companies and organizations developing generative AI in general, as well as LLMs:

- **OpenAI** have released GPT-2 as open source; however, subsequent models have been closed source but open for public usage on their website or through an API.

- **Google** (including Google's **DeepMind** division) have developed a number of LLMs, starting from BERT and – more recently – Chinchilla, Gopher, PaLM, and PaLM2. They previously released the code and weights (parameters) of a few of their models under open-source licensing, even though recently they have moved toward more secrecy in their development.
- **Anthropic** have released the Claude and Claude 2 models for public usage on their website. The API is in private beta. The models themselves are closed source.
- **Meta** have released models like RoBERTa, BART, and LLaMa 2, including parameters of the models (although often under a non-commercial license) and the source code for setting up and training the models.
- **Microsoft** have developed models like Turing-NLG and Megatron-Turing NLG but have focused on integrating OpenAI models into products over releasing their own models. The training code and parameters for phi-1 have been released for research use.
- **Stability AI**, the company behind Stable Diffusion, released the model weights under a non-commercial license.
- The French AI startup **Mistral** has unveiled its free-to-use, open-license 7B model, outperforming similar-sized models, generated from private datasets, and developed with the intent to support the open generative AI community, while also offering commercial products.
- **EleutherAI** is a grassroots collection of researchers developing open-access models like GPT-Neo and GPT-J, fully open source and available to the public.
- **Aleph Alpha**, **Alibaba**, and **Baidu** are providing API access or integrating their models into products rather than releasing parameters or training code.

Transformers:

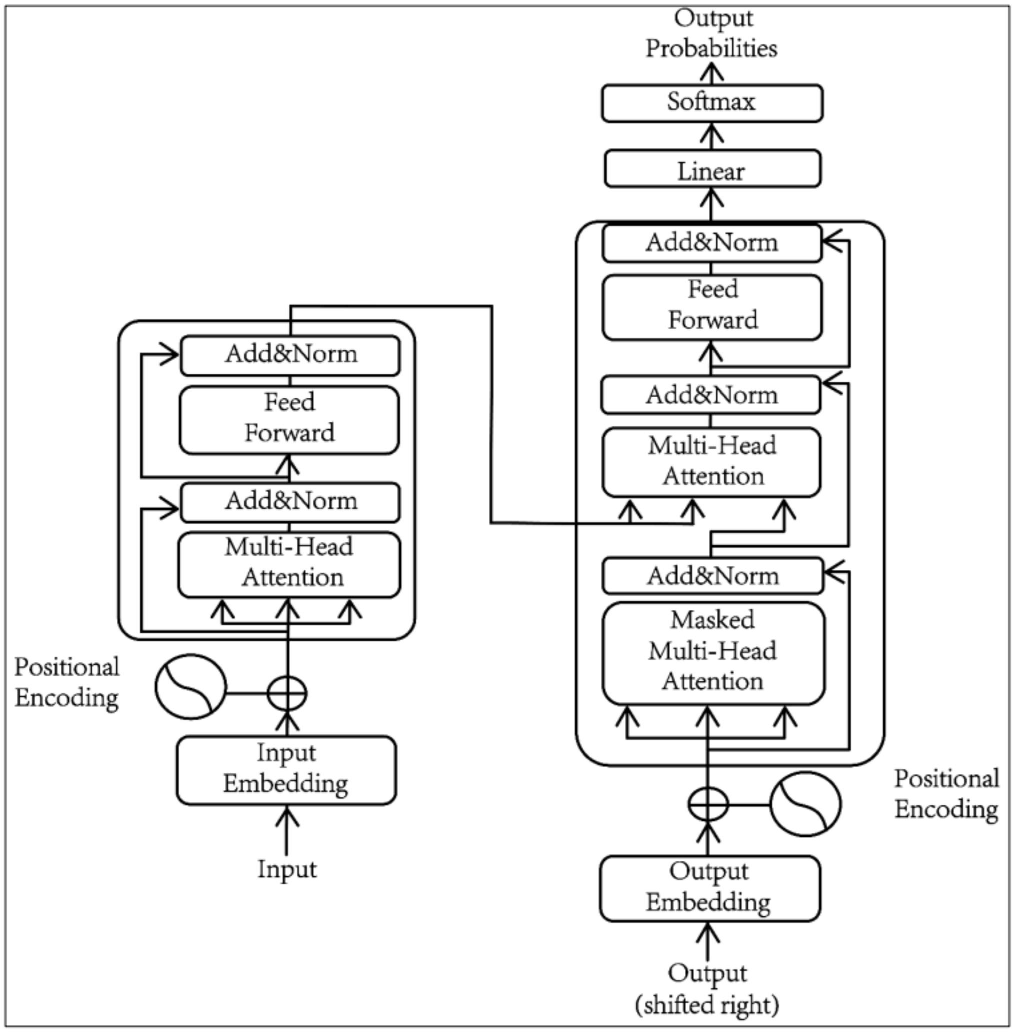
A **transformer** is a DL architecture, first introduced in 2017 by researchers at Google and the University of Toronto (in an article called *Attention Is All You Need*; Vaswani and colleagues), that comprises self-attention and feed-forward neural networks, allowing it to effectively capture the word relationships in a sentence. The attention mechanism enables the model to focus on various parts of the input sequence.

The transformer model architecture has an encoder-decoder structure, where the encoder maps an input sequence to a sequence of hidden states, and the decoder maps the hidden states to an output sequence. The hidden state representations consider not only the inherent meaning of the words (their semantic value) but also their context in the sequence.

The encoder is made up of identical layers, each with two sub-layers. The input embedding is passed through an attention mechanism, and the second sub-layer is a fully connected feed-forward network. Each sub-layer is followed by a residual connection and layer normalization. The output of each sub-layer is the sum of the input and the output of the sub-layer, which is then normalized.

The decoder uses this encoded information to generate the output sequence one item at a time, using the context of the previously generated items. It also has identical modules, with the same two sub-layers as the encoder.

In addition, the decoder has a third sub-layer that performs **Multi-Head Attention** (**MHA**) over the output of the encoder stack. The decoder also uses residual connections and layer normalization. The self-attention sub-layer in the decoder is modified to prevent positions from attending to subsequent positions.



- **Positional encoding**: Since the transformer doesn't process words sequentially but instead processes all words simultaneously, it lacks any notion of the order of words. To remedy this, information about the position of words in the sequence is injected into the model using positional encodings. These encodings are added to the input embeddings representing each word, thus allowing the model to consider the order of words in a sequence.
- Word embeddings, specifically, are dense vector representations that encode the meaning of a word based on its context in a large corpus of text. In simpler terms, they map words to numerical vectors in a high-dimensional space, where similar words are located closer to each other. This is done in a vector database. Creating these embeddings is done by an embedding model. E.g OpenAI-embeddings model *text-embedding-ada-002*. we will create a GPT-4 chatbot that can retrieve information from this database. The question asked to the chatbot will be embedded as well, and on the basis of similarity search, the retriever will return the embeddings with the data to answer the question. After this, the LLM will return a coherent and well-structured answer.
-
- **Layer normalization**: To stabilize the network's learning, the transformer uses a technique called layer normalization. This technique normalizes the model's inputs across the features dimension (instead of the batch dimension as in batch normalization), thus improving the overall speed and stability of learning.
- **Multi-head attention**: Instead of applying attention once, the transformer applies it multiple times in parallel – improving the model's ability to focus on different types of information and thus capturing a richer combination of features.

A key reason for the success of transformers has been their ability to maintain performance across longer sequences better than other models, for example, recurrent neural networks.

The basic idea behind attention mechanisms is to compute a weighted sum of the values (usually referred to as values or content vectors) associated with each position in the input sequence, based on the similarity between the current position and all other positions. This weighted sum, known as the context vector, is then used as an input to the subsequent layers of the model, enabling the model to selectively attend to relevant parts of the input during the decoding process.

To enhance the expressiveness of the attention mechanism, it is often extended to include multiple so-called *heads*, where each head has its own set of query, key, and value vectors, allowing the model to capture various aspects of the input representation. The individual context vectors from each head are then concatenated or combined in some way to form the final output.

Early attention mechanisms scaled quadratically with the length of the sequences (context size), rendering them inapplicable to settings with long sequences. Different mechanisms have been tried out to alleviate this. Many LLMs use some form of **Multi-Query Attention** (**MQA**), including OpenAI's GPT-series models, Falcon, SantaCoder, and StarCoder.

MQA is an extension of MHA, where attention computation is replicated multiple times. MQA improves the performance and efficiency of language models for various language tasks. By removing the heads dimension from certain computations and optimizing memory usage, MQA allows for 11 times better throughput and 30% lower latency in inference tasks compared to baseline models without MQA.

LLaMa 2 and a few other models used **Grouped-Query Attention** (**GQA**), which is a practice used in autoregressive decoding to cache the key (K) and value (V) pairs for the previous tokens in the sequence, speeding up attention computation. However, as the context window or batch sizes increase, the memory costs associated with the KV cache size in MHA models also increase significantly. To address this, the key and value projections can be shared across multiple heads without much degradation of performance.

There have been many other proposed approaches to obtain efficiency gains, such as sparse, low-rank self-attention, and latent bottlenecks, to name just a few. Other work has tried to extend sequences beyond the fixed input size; architectures such as transformer-XL reintroduce recursion by storing hidden states of already encoded sentences to leverage them in the subsequent encoding of the next sentences.

The combination of these architectural features allows GPT models to successfully tackle tasks that involve understanding and generating text in human language and other domains. The overwhelming majority of LLMs are transformers, as are many other state-of-the-art models we will encounter in the different sections of this chapter, including models for image, sound, and 3D objects.

As the name suggests, a particularity of GPTs lies in pre-training. Let's see how these LLMs are trained!

Pre-training:

The transformer is trained in two phases using a combination of unsupervised pre-training and discriminative task-specific fine-tuning. The goal during pre-training is to learn a general-purpose representation that transfers to a wide range of tasks.

The unsupervised pre-training can follow different objectives. In **Masked Language Modeling** (**MLM**), introduced in *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* by Devlin and others (2019), the input is masked out, and the model attempts to predict the missing tokens based on the context provided by the non-masked portion. For example, if the input sentence is "The cat [MASK] over the wall," the model would ideally learn to predict "jumped" for the mask.

In this case, the training objective minimizes the differences between predictions and the masked tokens according to a loss function. Parameters in the models are then iteratively updated according to these comparisons.

Negative Log-Likelihood (NLL) and **Perplexity (PPL)** are important metrics used in training and evaluating language models. NLL is a loss function used in ML algorithms, aimed at maximizing the probability of correct predictions. A lower NLL indicates that the network has successfully learned patterns from the training set, so it will accurately predict the labels of the training samples. It's important to mention that NLL is a value constrained within a positive interval.

PPL, on the other hand, is an exponentiation of NLL, providing a more intuitive way to understand the model's performance. Smaller PPL values indicate a well-trained network that can predict accurately while higher values indicate poor learning performance. Intuitively, we could say that a low perplexity means that the model is less surprised by the next word. Therefore, the goal in pre-training is to minimize perplexity, which means the model's predictions align more with the actual outcomes.

In comparing different language models, perplexity is often used as a benchmark metric across various tasks. It gives an idea about how well the language model is performing, where a lower perplexity indicates the model is more certain of its predictions. Hence, a model with lower perplexity would be considered better performing in comparison to others with higher perplexity.

After pre-training, a major step is how models are prepared for specific tasks either by fine-tuning or prompting.

Conditioning/Adapting models for specific tasks:

Conditioning LLMs refers to adapting the model for specific tasks. It includes fine-tuning and prompting:

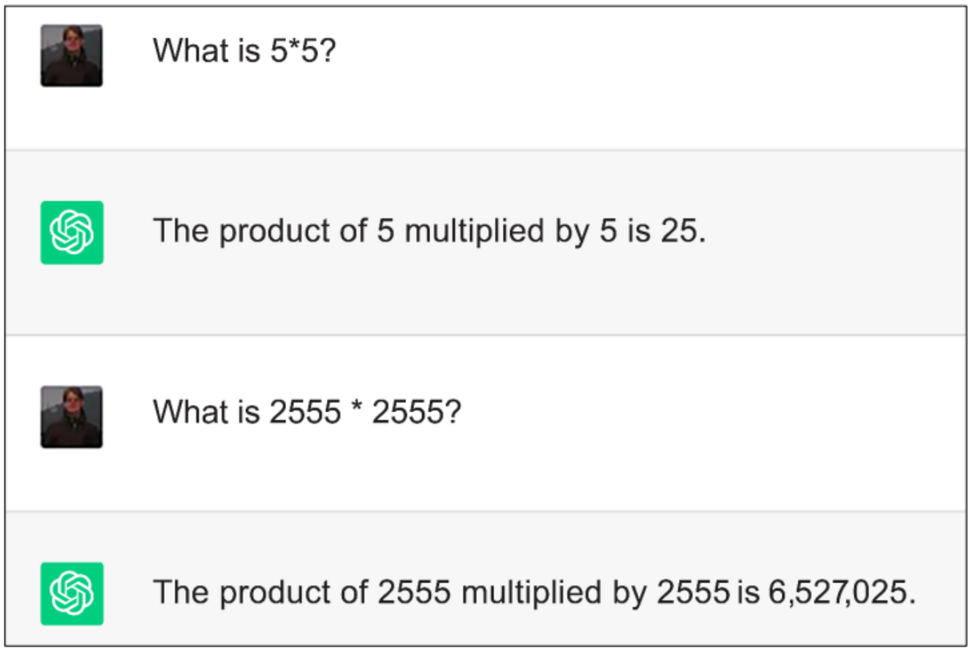
- **Fine-tuning** involves modifying a pre-trained language model by training it on a specific task using supervised learning. For example, to make a model more amenable to chats with humans, the model is trained on examples of tasks formulated as natural language instructions (instruction tuning). For fine-tuning, pre-trained models are usually trained again using **Reinforcement Learning from Human Feedback** (**RLHF**) to be helpful and harmless.
- **Prompting techniques** present problems in text form to generative models. There are a lot of different prompting techniques, starting from simple questions to detailed instructions. Prompts can include examples of similar problems and their solutions. Zero-shot prompting involves no examples, while few-shot prompting includes a small number of examples of relevant problem and solution pairs.

GPTs:

Generative Pre-Trained Transformers (**GPTs**), on the other hand, were introduced by researchers at OpenAI in 2018 together with the first of their eponymous GPT models, GPT-1 (*Improving Language Understanding by Generative Pre-Training*; Radford and others). The pre-training process involves predicting the next word in a text sequence, enhancing the model’s grasp of language as measured in the quality of the output. Following pre-training, the model can be fine-tuned for specific language processing tasks like sentiment analysis, language translation, or chat. This combination of unsupervised and supervised learning enables GPT models to perform better across a range of NLP tasks and reduces the challenges associated with training LLMs.

The size of the training corpus for LLMs has been increasing drastically

Chat GPT issues without including Langchain:



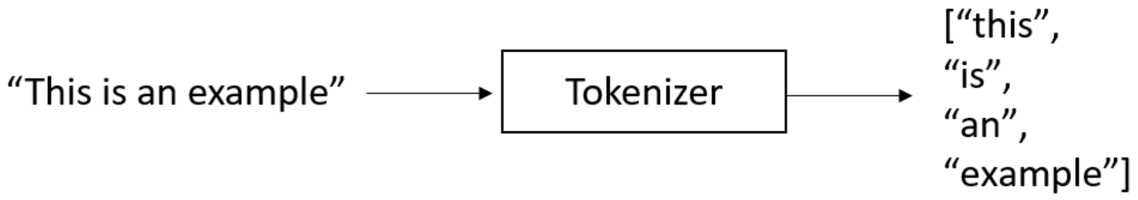
What is 5*5?

The product of 5 multiplied by 5 is 25.

What is 2555 * 2555?

The product of 2555 multiplied by 2555 is 6,527,025.

Tokenization:



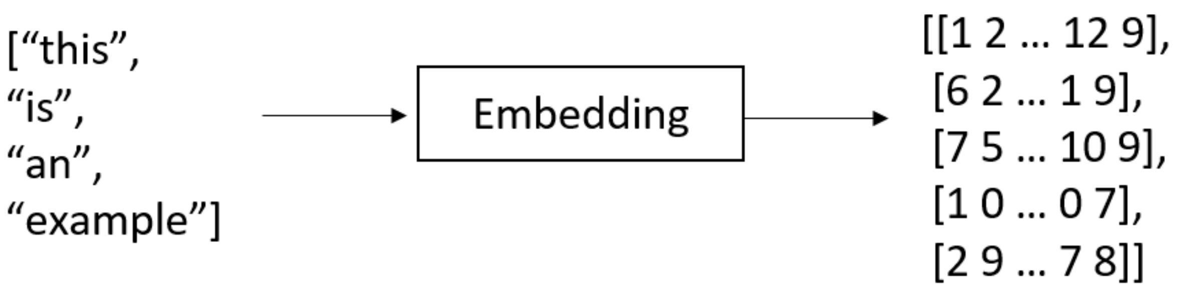
"The quick brown fox jumps over the lazy dog!"

["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "!"]

Tokenization: This refers to the process of splitting text into tokens. A tokenizer splits on whitespace and punctuation to break text into individual tokens. It is the process of breaking down a piece of text (a sentence, paragraph, or document) into smaller units called tokens.

- **Token**: A token is an instance of a sequence of characters, typically forming a word, punctuation mark, or number.
- Each word is an individual token, as is the punctuation mark
- There are a lot of tokenizers that work according to different principles, but common types of tokenizers employed in models are **Byte-Pair Encoding** (**BPE**), **WordPiece,** and **SentencePiece**. For example, LLaMa 2's BPE tokenizer splits numbers into individual digits and uses bytes to decompose unknown UTF-8 characters. The total vocabulary size is 32K tokens.

- **Embedding**->Once the text has been tokenized, each token is converted into a dense numerical vector called an embedding. Embeddings are a way to represent words, subwords, or characters in a continuous vector space. These embeddings are learned during the training of the language model and capture semantic relationships between tokens. The numerical representation allows the model to perform mathematical operations on the tokens and understand the contexst in which they appear.



Text-to-image Models:

Text-to-image models are a powerful type of generative AI that creates realistic images from textual descriptions.

The main applications are:

- **Text-conditioned image generation**: Creating original images from text prompts like “a painting of a cat in a field of flowers.” This is used for art, design, prototyping, and visual effects.
- **Image inpainting**: Filling in missing or corrupted parts of an image based on the surrounding context. This can restore damaged images (denoising, dehazing, and deblurring) or edit out unwanted elements.
- **Image-to-image translation**: Converting input images to a different style or domain specified through text, like “make this photo look like a Monet painting.”
- **Image recognition**: Large foundation models can be used to recognize images, including classifying scenes, but also object detection, for example, detecting faces.

Models like **Midjourney, DALL-E 2, and Stable Diffusion** provide creative and realistic images derived from textual input or other images. These models work by training deep neural networks on large datasets of image-text pairs. The key technique used is diffusion models, which start with random noise and gradually refine it into an image through repeated denoising steps.

Popular models like **Stable Diffusion and DALL-E 2** use a text encoder to map input text into an embedding space. This text embedding is fed into a series of conditional diffusion models, which denoise and refine a latent image in successive stages. The final model output is a high-resolution image aligned with the textual description.

Two main classes of models are used: **Generative Adversarial Networks** (**GANs**) and diffusion models. GAN models like StyleGAN or GANPaint Studio can produce highly realistic images, but training is unstable and computationally expensive. They consist of two networks that are pitted against each other in a game-like setting – the generator, which generates new images from text embeddings and noise, and the discriminator, which estimates the probability of the new data being real. As these two networks compete, GANs get better at their task, generating realistic images and other types of data.

Diffusion models have become popular and promising for a wide range of generative tasks, including text-to-image synthesis. These models offer advantages over previous approaches, such as GANs, by reducing computation costs and sequential error accumulation. Diffusion models operate through a process like diffusion in physics. They follow a **forward diffusion process** by adding noise to an image until it becomes uncharacteristic and noisy. This process is analogous to an ink drop falling into a glass of water and gradually diffusing.

The unique aspect of generative image models is the **reverse diffusion process**, where the model attempts to recover the original image from a noisy, meaningless image. By iteratively applying noise removal transformations, the model generates images of increasing resolutions that align with the given text input. The final output is an image that has been modified based on the text input

A text encoder first maps the input text to a sequence of embeddings. A cascade of conditional diffusion models takes the text embeddings as input and generates images.

Denoising Diffusion Implicit Model (**DDIM**) sampling method, which repeatedly removes Gaussian noise, and then decodes the denoised output into pixel space.

Stable Diffusion was developed in 2022. The Stable Diffusion model significantly cuts training costs and sampling time compared to previous (pixel-based) diffusion models. The model can be run on consumer hardware equipped with a modest. By creating high-fidelity images from text on consumer GPUs, the Stable Diffusion model democratizes access.

Significantly, Stable Diffusion introduced operations in latent (lower-dimensional) space representations, which capture the essential properties of an image, in order to improve computational efficiency. A VAE provides latent space compression (called *perceptual compression* in the paper), while a U-Net performs iterative denoising.

Stable Diffusion generates images from text prompts through several clear steps:

1. It starts by producing a random tensor (random image) in the latent space, which serves as the noise for our initial image.
2. A noise predictor (U-Net) takes in both the latent noisy image and the provided text prompt and predicts the noise.
3. The model then subtracts the latent noise from the latent image.
4. *Steps 2* and *3* are repeated for a set number of sampling steps, for instance, 40 times, as shown in the plot.
5. Finally, the decoder component of the VAE transforms the latent image back into pixel space, providing the final output image.

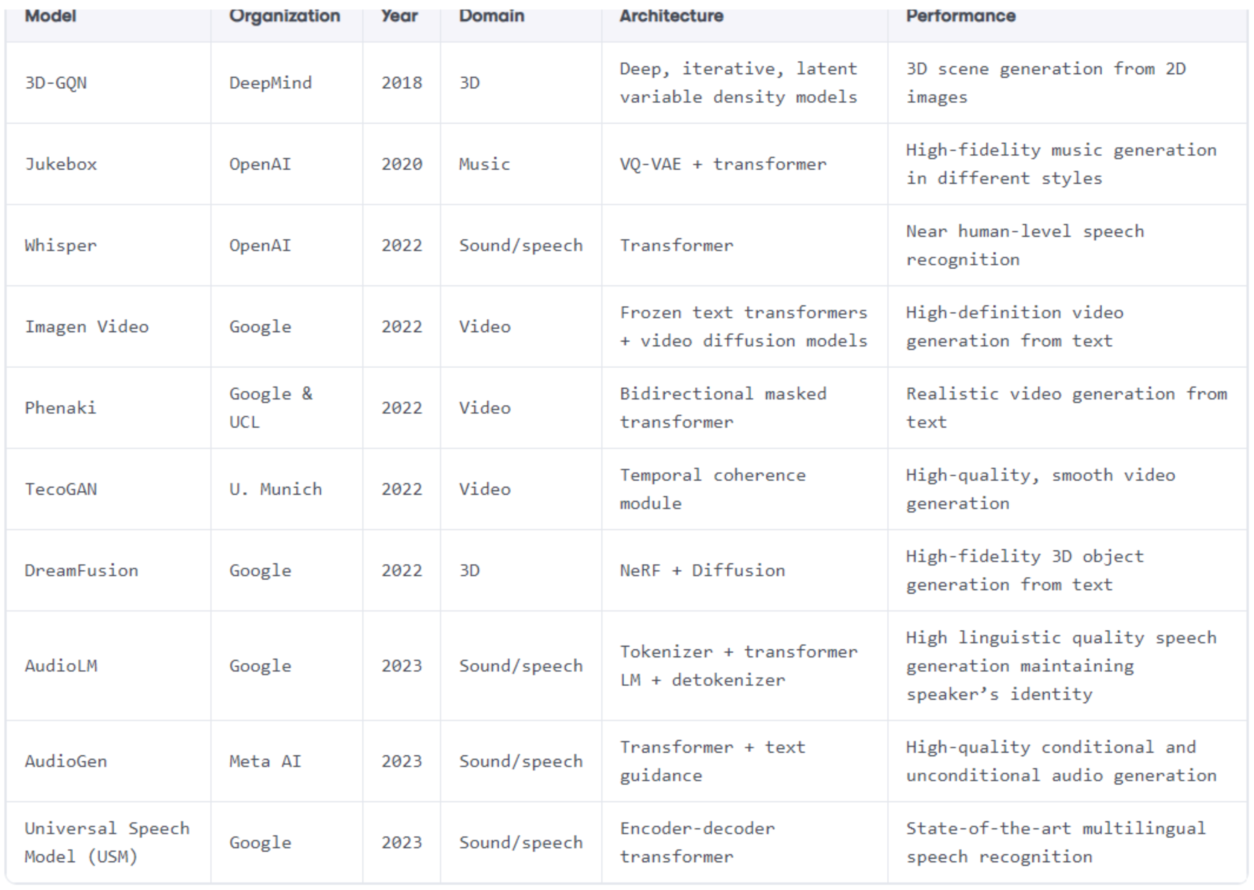
A **VAE** is a model that encodes data into a learned, smaller representation (encoding). These representations can then be used to generate new data similar to that used for training (decoding). This VAE is trained first.

A **U-Net** is a popular type of **convolutional neural network** (**CNN**) that has a symmetric encoder-decoder structure. It is commonly used for image segmentation tasks, but in the context of Stable Diffusion, it can help to introduce and remove noise in the image. The U-Net takes a noisy image (seed) as input and processes it through a series of convolutional layers to extract features and learn semantic representations.

For training the image generation model in the latent space itself (**latent diffusion model**), a loss function is used to evaluate the quality of the generated images. One commonly used loss function is the **Mean Squared Error** (**MSE**) loss, which quantifies the difference between the generated image and the target image. The model is optimized to minimize this loss, encouraging it to generate images that closely resemble the desired output.

This training was performed on the LAION-5B dataset, consisting of billions of image-text pairs, derived from Common Crawl data, comprising billions of image-text pairs from sources such as Pinterest, WordPress, Blogspot, Flickr, and DeviantArt.

Overall, image generation models such as Stable Diffusion and Midjourney process textual prompts into generated images, leveraging the concept of forward and reverse diffusion processes and operating in a lower-dimensional latent space for efficiency.



Model	Organization	Year	Domain	Architecture	Performance
3D-GQN	DeepMind	2018	3D	Deep, iterative, latent variable density models	3D scene generation from 2D images
Jukebox	OpenAI	2020	Music	VQ-VAE + transformer	High-fidelity music generation in different styles
Whisper	OpenAI	2022	Sound/speech	Transformer	Near human-level speech recognition
Imagen Video	Google	2022	Video	Frozen text transformers + video diffusion models	High-definition video generation from text
Phenaki	Google & UCL	2022	Video	Bidirectional masked transformer	Realistic video generation from text
TecoGAN	U. Munich	2022	Video	Temporal coherence module	High-quality, smooth video generation
DreamFusion	Google	2022	3D	NeRF + Diffusion	High-fidelity 3D object generation from text
AudioLM	Google	2023	Sound/speech	Tokenizer + transformer LM + detokenizer	High linguistic quality speech generation maintaining speaker's identity
AudioGen	Meta AI	2023	Sound/speech	Transformer + text guidance	High-quality conditional and unconditional audio generation
Universal Speech Model (USM)	Google	2023	Sound/speech	Encoder-decoder transformer	State-of-the-art multilingual speech recognition

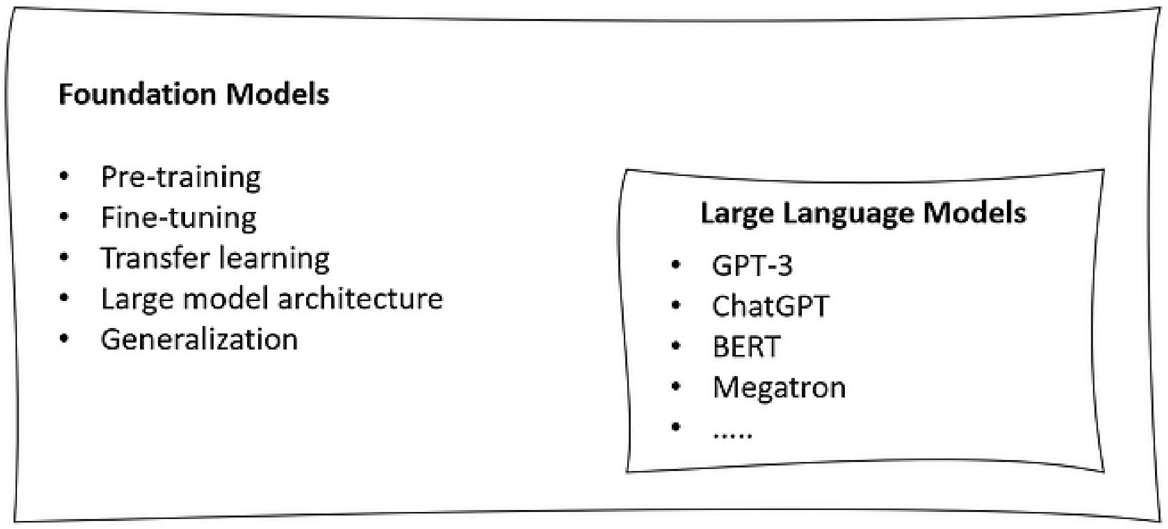
Multi-layer Neural Network:

Once we have the vectorized input, we can pass it into the multi-layered neural network. There are three main types of layers:

- **Input Layer**: The first layer of the neural network receives the input data. Each neuron in this layer corresponds to a feature or attribute of the input data.
- **Hidden Layers**: Between the input and output layers, there can be one or more hidden layers. These layers process the input data through a series of mathematical transformations and extract relevant patterns and representations from the data.

- **Output Layer**: The final layer of the neural network produces the desired output, which could be predictions, classifications, or other relevant results depending on the task the neural network is designed for.

LLM:



an LLM is a type of foundation model specifically designed for natural language processing tasks. These models, such as ChatGPT, BERT, Llama and many others, are trained on vast amounts of text data and can generate human-like text, answer questions, perform translations, and more.

The LLM uses Bayes' theorem and the probabilities learned during training to generate text that is contextually relevant and meaningful, capturing patterns and associations from the training data to complete sentences in a coherent manner.

For example, let's consider the following prompt: *"the cat is on the…."* and we want the model to complete this sentence. However, the LLM may generate multiple candidate words, and we want to use Bayes' theorem to select the most likely word given the context. Let's assume the LLM has three candidate words: "table," "chair," and "roof".

- Using Bayes' theorem, we can calculate the posterior probability for each candidate word based on the prior probability and the likelihood.

$$P(\text{"table"}|\text{"The cat is on the..."}) = \frac{P(\text{"table"})P(\text{"The cat is on the table"})}{P(\text{"}The\ cat\ is\ on\ the\ \ldots\text{"})}$$

$$P(\text{"chair"}|\text{"The cat is on the..."}) = \frac{P(\text{"chair"})P(\text{"The cat is on the chair"})}{P(\text{"}The\ cat\ is\ on\ the\ \ldots\text{"})}$$

$$P(\text{"roof"}|\text{"The cat is on the..."}) = \frac{P(\text{"roof"})P(\text{"The cat is on the roof"})}{P(\text{"}The\ cat\ is\ on\ the\ \ldots\text{"})}$$

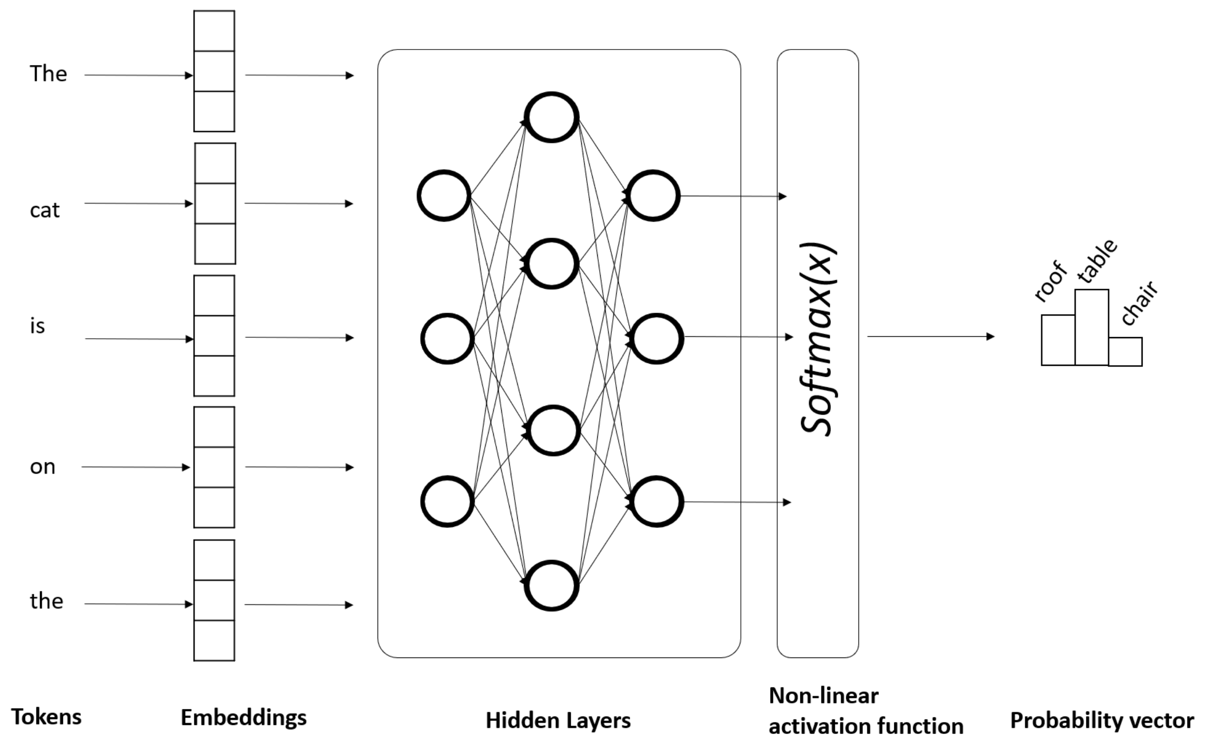


Fig: Predicting the next most likely word in a Large Language Model (LLM)

LLMs are trained using unsupervised learning on massive datasets, which often consist of billions of sentences collected from diverse sources on the internet. The transformer architecture, with its self-attention mechanism, allows the model to efficiently process long sequences of text and capture intricate dependencies between words. Training such models necessitates vast computational resources, typically employing distributed systems with multiple Graphical Processing Units (GPUs) or Tensor Processing Units (TPUs).

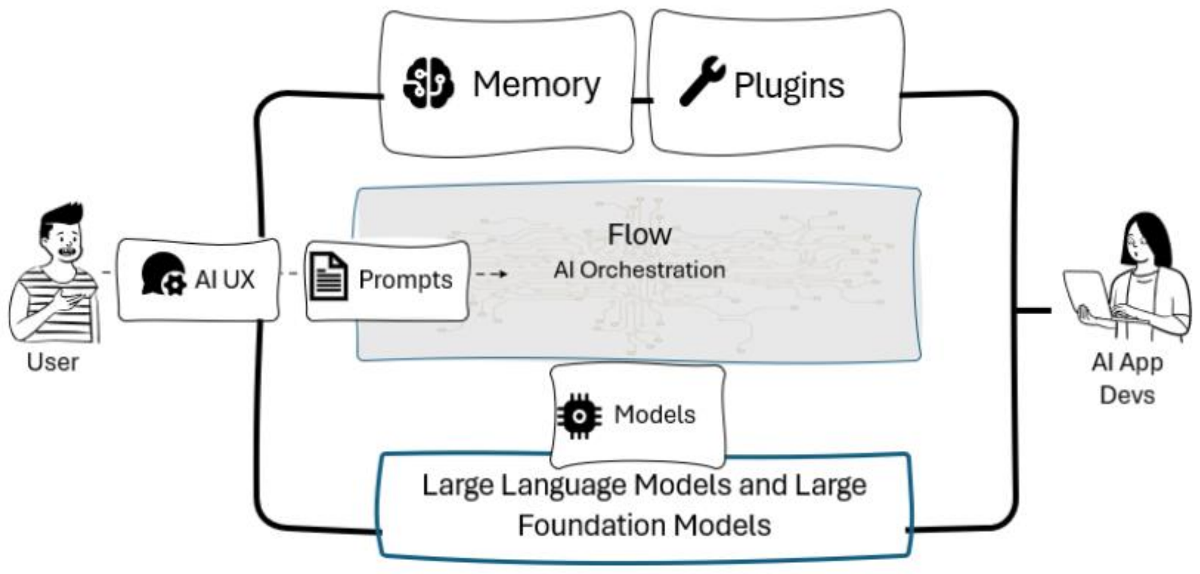


Fig: High-level architecture of LLM-powered applications.

Bayes Therorem:

According to Bayes' theorem, given two events A and B, we can define the conditional probability of A given B as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

With:

P(B|A) = probabilty of B occurring given A, also known as likelihood of A given a fixed B.

P(A|B) = probabilty of A occurring given B, also known as posterior probability, also known as posterior probability of A given B.

P(A) and P(B) = probability of observing A or B without any conditions.

Bayes' theorem is particularly useful when we want to make inferences about unknown parameters or events in the presence of uncertainty.

RNN limitations:

The vanishing gradient problem occurs when the gradient becomes extremely small during training. As a result, the RNN learns very slowly and struggles to capture long-term patterns in the data. Conversely, the exploding gradient problem happens when the gradient becomes extremely large. This leads to unstable training and prevents the RNN from converging to a good solution.

RNN is mostly suitable for tasks like language modelling, machine translation, and text generation.

LSTM limitations:

LSTM had limitations in handling long-range dependencies, scalability, and overall efficiency, especially when dealing with large-scale natural language processing tasks that would need massive parallel processing.

LSTMs became popular for various sequential tasks, **including text generation, speech recognition, and sentiment analysis.**

Transformer:

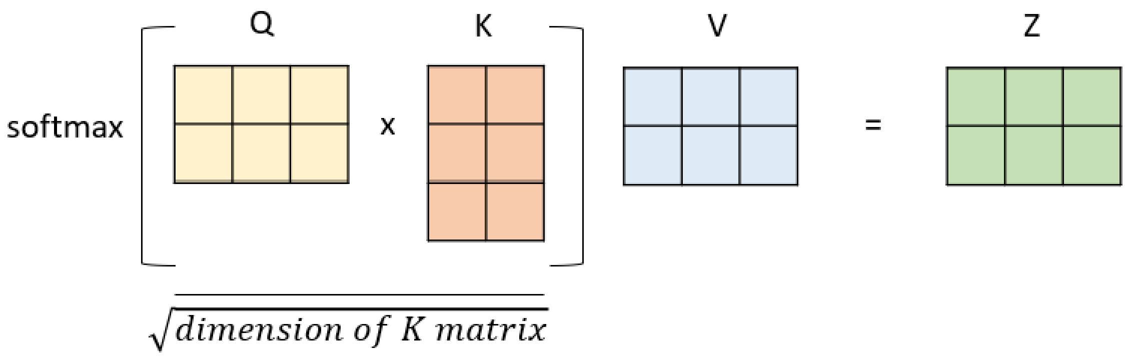
Attention:

In the Transformer architecture, "attention" is a mechanism that enables the model to focus on relevant parts of the input sequence while generating the output. It calculates attention scores between input and output positions, applies softmax to get weights, and takes a weighted sum of the input sequence to obtain context vectors. Attention is crucial for capturing long-range dependencies and relationships between words in the data.

Attnetion layers are responsible for determining the importance of each input token in generating the output.

Self-attention vector for a sentence:

In order to obtain the **self-attention vector for a sentence**, the elements we need are “value”, “query”, and “key”. These matrices are used to calculate attention scores between the elements in the input sequence and are the **three weight matrices that are learned during the training process (typically initialized with random values)“Query” is used to represent the current focus of the attention mechanism, while “key” is used to determine which parts of the input should be given attention and “value” is used to compute the context vectors**. Those matrices are then multiplied and passed through a non-linear transformation (thanks to a softmax function). The output of the self-attention layer represents the input values in a transformed, context-aware manner, which allows the transformer to attend to different parts of the input depending on the task at hand.



Transformer:

From an architectural point of view, the transformer consists of two main components: an encoder and a decoder.

- The **encoder** takes the input sequence and produces a sequence of hidden states, each of which is a **weighted sum of all the input embeddings**.
- The **decoder** takes the output sequence (shifted right by one position) and produces a sequence of predictions, **each of which is a weighted sum of all the encoder hidden states and the previous decoder hidden states.**

Tensor:

A tensor is a multi-dimensional array used in mathematics and computer science. It holds numerical data and is fundamental in fields like machine learning.

A Tensor Processing Unit (TPU) is a specialized hardware accelerator created by Google for deep learning tasks. TPUs are optimized for tensor operations, making them highly efficient for training and running neural networks. They offer fast processing while consuming less power, enabling faster model training and inference in data centers.

zero-shot evaluation:

The concept of zero-shot evaluation is a method of evaluating a language model without any labeled data or fine-tuning. It measures how well the language model can perform a new task by using natural language instructions or examples as prompts, and computing the likelihood of the correct output given the input. **It the probability that a trained model will produce a particular set of tokens without needing any labeled training data**.

CoPilot System:

This concept was coined by Microsoft and already introduce into its applications, such as M365 Copilot or the new Bing, now powered by GPT-4

copilots are meant to be AI assistant that work side-by-side with users and support them in various activities, from information retrieval to blog writing and posting, from ideas brainstorming to code review and generation

A copilot is powered by LLMs or, more generally, LFMs, meaning that these re the reasoning engines that make the copilot “intelligent”

A copilot is designed to have a conversational user interface, allowing users to interact with it using natural language

A copilot is **grounded** to domain-specific data, so that it is entitled to answer only within the perimeter of the application or domain

Grounding:

Grounding is the process of using large language models (LLMs) with information that is use-case specific, relevant, and not available as part of the LLM’s trained knowledge. It is crucial for ensuring the quality, accuracy and relevance of the output.

Grounding also helps Copilot to be restricted or scoped to application-specific data, so that it can support only within the boundaries of the provided knowledge base.

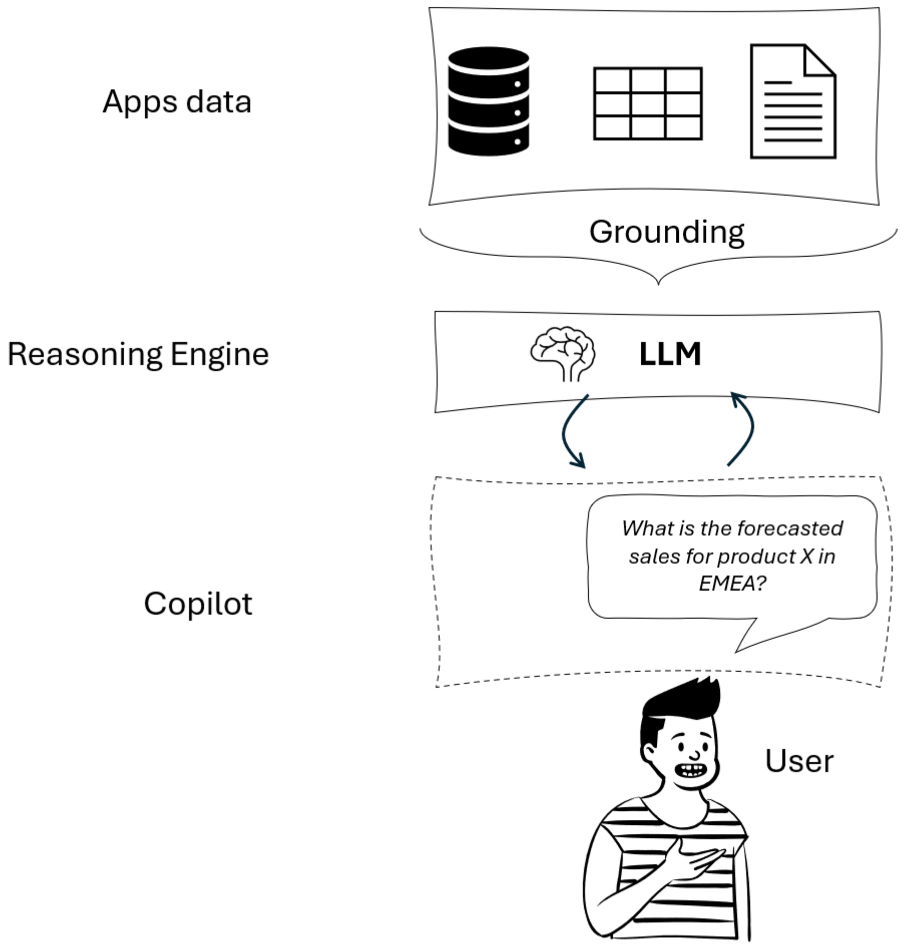


Fig: Example of grounding a Copilot

Categories of LLM Models:

- **Proprietary LLMs->**models that are owned by specific companies or organizations. Examples include GPT-3 and GPT-4 developed by OpenAI or Bard, developed by Google. As their source code and architecture is not available, those models cannot be re-trained from scratch on custom data, yet they can be fine-tuned if needed.

- **Open-source**->models with code and architecture freely available and distributed, hence they can also be trained from scratch on custom data. Examples include Falcon LLM, developed by Abu Dhabi’s Technology Innovation Institute (TII), or LLaMA, developed by Meta.

VectorDB:

A VectorDB is a type of database that stores and retrieves information based on vectorized embeddings, the numerical representations that capture the meaning and context of text. By using VectorDB, you can perform semantic search and retrieval based on the similarity of meanings rather than keywords. VectorDB can also help LLMs generate more relevant and coherent text by providing contextual understanding and enriching generation results. **Some examples of vectorDB are Chroma, FAISS, Elastic Search, Milvus, Pinecone, Qdrant, and Weaviate**.

For example, if we want to limit our application to answer only to questions related to the documentatios we provided in the VectorDB, we will specify in our meta-promts to the model: *“Answer only if the question is related to the provided documentation”*.

LLM Libraries:

1. **LangChain**
2. **HayStack**
3. **Semantic Kernel**

LangChain:

LangChain was launched as an open-source project by Harrison Chase, in October 2022. It can be used both in Python and JS/TS. LangChain is a framework for developing applications powered by language models, making them data-aware (with grounding) and agentic – that means, able to interact with external environments.

LangChain provides modular abstractions for the components necessary to work with language models that we previously mentioned, such as prompts, memory and plug-ins. Alongside those components, LangChain also offer pre-built **chains**, that are structured contatenations of components

while LangChain and Haystack are mainly based on Python (even thought LangChain also introduced JS/TS support)

Core Modules of LangChain:

- **Models**. These are the Large Language Models or Large Foundation Models that will be the engine of the application. LangChain supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the **Hugging Face Hub**.

- **Data connections**. These refer to the building blocks needed to retrieve the additional non-parametric knowledge we want to provide the model with. Examples of data connetctions are document loaders or text embedding models.
- **Memory**. It allows the application to keep references to user's interactions, both in the short and long-term. It is typically based on vectorized embeddings stored into a VectorDB.
- **Chains**. These are predetermined sequences of actions and calls to LLMs that make it easier to build complex applications that require chaining LLMs with each other or with other components. An example of chain might be: take the user query, chunk it into smaller pieces, embed those chunks, search for similar embeddings in a VectorDB, use the top three most similar chunks in the VectorDB as context to provide the answer, generate the answer.
- **Agents**. Agents are entities that drive decision-making within LLMs-powered applications. They have access to a suite of tools and can decide which tool to call based on the user input and the context. Agents are dynamic and adaptive, meaning that they can change or adjust their actions based on the situation or the goal.

Hugging Face:

Hugging Face is a company and a community that builds and shares state-of-the-art models and tools for natural language processing and other machine learning domains. It developed the Hugging Face Hub, a platform where people can create, discover, and collaborate on machine learning models and LLMs, datasets, and demos. Hugging Face Hub hosts over 120k models, 20k datasets, and 50k demos in various domains and tasks, such as audio, vision, and language.

Hugging Face is an American company that develops tools for building machine learning applications. Its employees develop and maintain the Transformers Python library, which is used for NLP tasks, includes implementations of state-of-the-art and popular models like Mistral 7B, BERT, and GPT-2, and is compatible with PyTorch, TensorFlow, and JAX.

Hugging Face also provides the Hugging Face Hub, a platform for hosting Git-based code repositories, machine learning models, datasets, and web applications, which provides over 120k models, 20k datasets, and 50k demo apps (spaces) for machine learning. It is an online platform where people can collaborate and facilitate machine learning development.

These tools allow users to load and use models, embeddings, and datasets from Hugging Face. The `HuggingFaceHub` integration, for example, provides access to different models for tasks like text generation and text classification. The `HuggingFaceEmbeddings` integration allows users to work with sentence-transformer models.

Hugging Face offer various other libraries within their ecosystem, including `Datasets` for dataset processing, `Evaluate` for model evaluation, `Simulate` for simulation, and `Gradio` for machine learning demos.

Hugging Face released an open LLM called BLOOM with 176 billion parameters. Hugging Face has also formed partnerships with companies like Graphcore and Amazon Web Services to optimize their offerings and make them available to a broader customer base.

To use Hugging Face as a provider for your models, you can create an account and API keys at https://huggingface.co/settings/profile. Additionally, you can make the token available in your environment as `HUGGINGFACEHUB_API_TOKEN`.

Let’s see an example, where we use an open-source model developed by Google, the Flan-T5-XXL model:

```
from langchain.llms import HuggingFaceHub
llm = HuggingFaceHub(
    model_kwargs={"temperature": 0.5, "max_length": 64},
    repo_id="google/flan-t5-xxl"
)
prompt = "In which country is Tokyo?"
```

```
completion = llm(prompt)
print(completion)
```

CopyExplain

We get the response `"japan"`.

The LLM takes a text input, a question in this case, and returns a completion. The model has a lot of knowledge and can come up with answers to knowledge questions.

Google Cloud Platform

There are many models and functions available through **Google Cloud Platform** (**GCP**) and Vertex AI, GCP's machine learning platform. GCP provides access to LLMs like LaMDA, T5, and PaLM. Google has also updated the Google Cloud **Natural Language** (**NL**) API with a new LLM-based model for **Content Classification**. This updated version offers an expansive pre-trained classification taxonomy to help with ad targeting and content-based filtering. The NL API's improved v2 classification model is enhanced with over 1,000 labels and supports 11 languages with improved

For models with GCP, you need to have the `gcloud` **command-line interface** (**CLI**) installed. You can find the instructions here: https://cloud.google.com/sdk/docs/install.

You can then authenticate and print a key token with this command from the terminal:

```
gcloud auth application-default login
```

You also need to enable Vertex AI for your project. To enable Vertex AI, install the Google Vertex AI SDK with the `pip install google-cloud-aiplatform` command. If you've followed the instructions on GitHub as indicated in the previous section, you should already have this installed.

Then we have to set up the Google Cloud project ID. You have different options for this:

- Using `gcloud config set project my-project`
- Passing a constructor argument when initializing the LLM

- Using `aiplatform.init()`
- Setting a GCP environment variable

The GCP environment variable works well with the `config.py` file that I mentioned earlier. I found the `gcloud` command very convenient though, so I went with this. Please make sure you set the project ID before you move on.

If you haven't enabled it, you should get a helpful error message pointing you to the right website, where you click **Enable**.

Let's run a model!

```
from langchain.llms import VertexAI
from langchain import PromptTemplate, LLMChain
template = """Question: {question}
Answer: Let's think step by step."""
prompt = PromptTemplate(template=template, input_variables=["question"])
llm = VertexAI()
llm_chain = LLMChain(prompt=prompt, llm=llm, verbose=True)
question = "What NFL team won the Super Bowl in the year Justin Beiber
was born?"
llm_chain.run(question)
```

We should see this response:

```
[1m> Entering new chain...[0m
Prompt after formatting:
[[Question: What NFL team won the Super Bowl in the year Justin Beiber
was born?
Answer: Let's think step by step.[0m
```

```
[1m> Finished chain.[0m
Justin Beiber was born on March 1, 1994. The Super Bowl in 1994 was won
by the San Francisco 49ers.
```

I've set `verbose` to `True` to see the model's reasoning process. It's quite impressive that it produces the right response even given a misspelling of the name. The step-by-step prompt instruction is key to the correct answer.

Vertex AI offers a range of models tailored for tasks like following instructions, conversation, and code generation/assistance:

- **text-bison** is fine-tuned to follow natural language instructions, with a max input of 8,192 tokens and an output of 1,024.
- **chat-bison** is optimized for multi-turn conversation with a max input of 4,096 tokens, an output of 1,024 tokens, and up to 2,500 turns.
- **code-bison** generates code from natural language descriptions, with a max input of 4,096 tokens and an output of 2,048 tokens.
- **codechat-bison** is a chatbot that is fine-tuned to help with code-related questions. It has an input limit of 4,096 tokens and an output limit of 2,048 tokens.
- **code-gecko** suggests code completions. It has a max input length of 2,048 tokens and an output of 64 tokens.

We can also generate code. Let's see if the code-bison model can solve **FizzBuzz**, a common interview question for entry-level software developer positions:

```
question = """
Given an integer n, return a string array answer (1-indexed) where:
answer[i] == "FizzBuzz" if i is divisible by 3 and 5.
answer[i] == "Fizz" if i is divisible by 3.
answer[i] == "Buzz" if i is divisible by 5.
answer[i] == i (as a string) if none of the above conditions are true.
```

```
"""
llm = VertexAI(model_name="code-bison")
llm_chain = LLMChain(prompt=prompt, llm=llm)
print(llm_chain.run(question))
```

We are getting this response:

~~~
```python
answer = []
for i in range(1, n + 1):
    if i % 3 == 0 and i % 5 == 0:
        answer.append("FizzBuzz")
    elif i % 3 == 0:
        answer.append("Fizz")
    elif i % 5 == 0:
        answer.append("Buzz")
    else:
        answer.append(str(i))
return answer
```
~~~

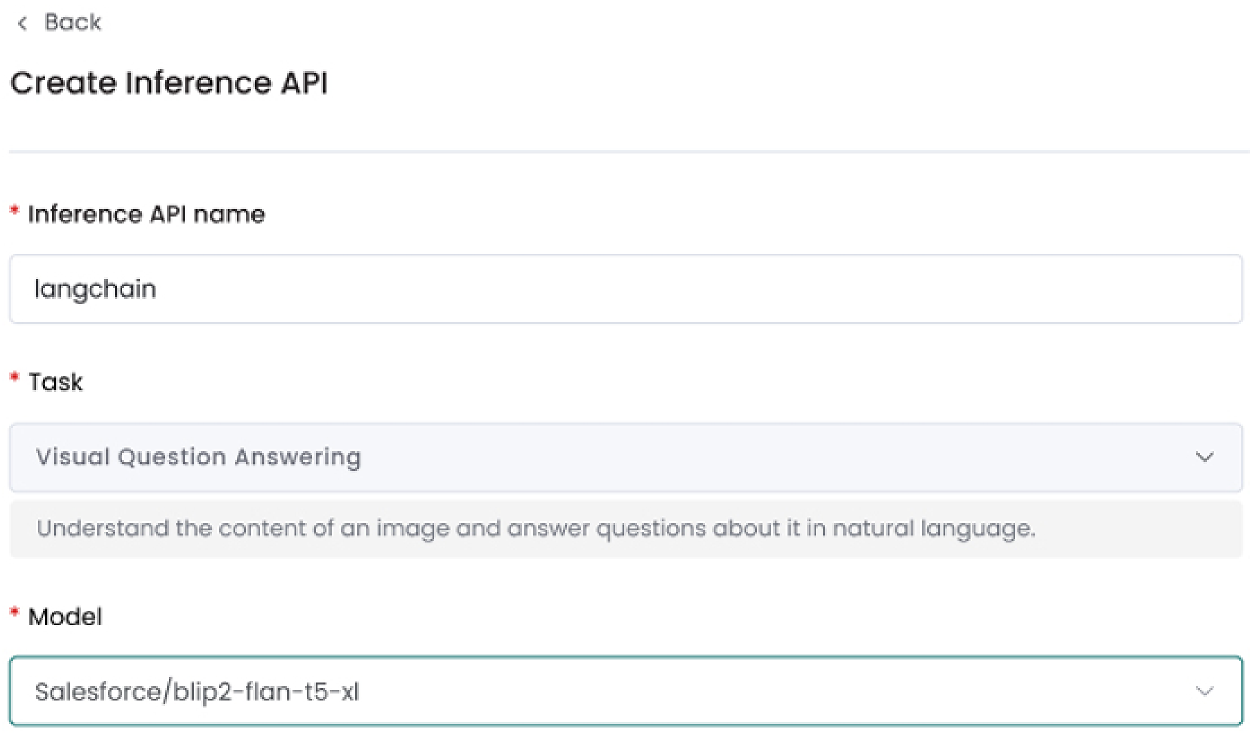
Jina AI

Jina AI, founded in February 2020, is a German AI company based in Berlin that specializes in providing cloud-native neural search solutions with models for text, image, audio, and video. Recently, Jina AI launched **Finetuner**, a tool that enables the fine-tuning of any deep neural network to specific use cases and requirements.

You can set up a login at https://chat.jina.ai/api.

On the platform, we can set up APIs for different use cases such as image caption, text embedding, image embedding, visual question answering, visual reasoning, image upscale, or Chinese text embedding.

Here, we are setting up a Visual Question Answering API with the recommended model:


< Back

Create Inference API

* Inference API name

langchain

* Task

Visual Question Answering

Understand the content of an image and answer questions about it in natural language.

* Model

Salesforce/blip2-flan-t5-xl

Figure 3.3: Visual Question Answering API in Jina AI

We get examples for client calls in Python and cURL, and a demo, where we can ask a question. This is cool, but unfortunately, these APIs are not available yet through LangChain. We can implement such calls ourselves by subclassing the `LLM` class in LangChain as a custom LLM interface.

Let’s set up another chatbot, this time powered by Jina AI. We can generate the API token, which we can set as `JINACHAT_API_KEY`, at https://chat.jina.ai/api.

Let’s translate from English to French here:

```
from langchain.chat_models import JinaChat
from langchain.schema import HumanMessage
chat = JinaChat(temperature=0.)
messages = [
    HumanMessage(
        content="Translate this sentence from English to French: I love
generative AI!"
    )
]
chat(messages)
```
CopyExplain

We should be seeing :

```
AIMessage(content="J'adore l'IA générative !", additional_kwargs={},
example=False).
```
CopyExplain

We can set different temperatures, where a low temperature makes the responses more predictable. In this case, it makes only a minor difference. We are starting the conversation with a system message clarifying the purpose of the chatbot.

Let's ask for some food recommendations:

```
from langchain.schema import SystemMessage
chat = JinaChat(temperature=0.)
chat(
    [
        SystemMessage(
            content="You help a user find a nutritious and tasty food to eat in one word."
        ),
        HumanMessage(
            content="I like pasta with cheese, but I need to eat more vegetables, what should I eat?"
        )
    ]
)
```
CopyExplain

I get this response in Jupyter – your answer could vary:

```
AIMessage(content='A tasty and nutritious option could be a vegetable pasta dish. Depending on your taste, you can choose a sauce that complements the vegetables. Try adding broccoli, spinach, bell peppers, and zucchini to your pasta with some grated parmesan cheese on top. This way, you get to enjoy your pasta with cheese while incorporating some veggies into your meal.', additional_kwargs={}, example=False)
```
CopyExplain

It ignored the one-word instruction, but I liked reading the ideas. I think I should try this for my son. With other chatbots, I got `Ratatouille` as a suggestion.

It's important to understand **the difference in LangChain between LLMs and chat models. LLMs are text completion models that take a string prompt as input and output a string completion. As mentioned, chat models are like LLMs but are specifically designed for conversations**. They take a list of chat messages as input, labeled with the speaker, and return a chat message as output.

Both LLMs and chat models implement the base language model interface, which includes methods such as `predict()` and `predict_messages()`.

Replicate:

Established in 2019, Replicate Inc. is a San Francisco-based start-up that presents a streamlined process to AI developers, where they can implement and publish AI models with minimal code input through the utilization of cloud technology. The platform works with private as well as public models and enables model inference and fine-tuning..

Replicate has lots of models available on their platform: https://replicate.com/explore.

You can authenticate with your GitHub credentials at https://replicate.com/. If you then click on your user icon at the top left, you'll find the API tokens – just copy the API key and make it available in your environment as `REPLICATE_API_TOKEN`. To run bigger jobs, you need to set up your credit card (under **billing**).

Here is a simple example for creating an image:

```
from langchain.llms import Replicate
text2image = Replicate(
    model="stability-ai/stable-
diffusion:db21e45d3f7023abc2a46ee38a23973f6dce16bb082a930b0c49861f96d1e5
bf",
    input={"image_dimensions": "512x512"},
)
image_url = text2image("a book cover for a book about creating
generative ai applications in Python")
```
CopyExplain

I got this image:

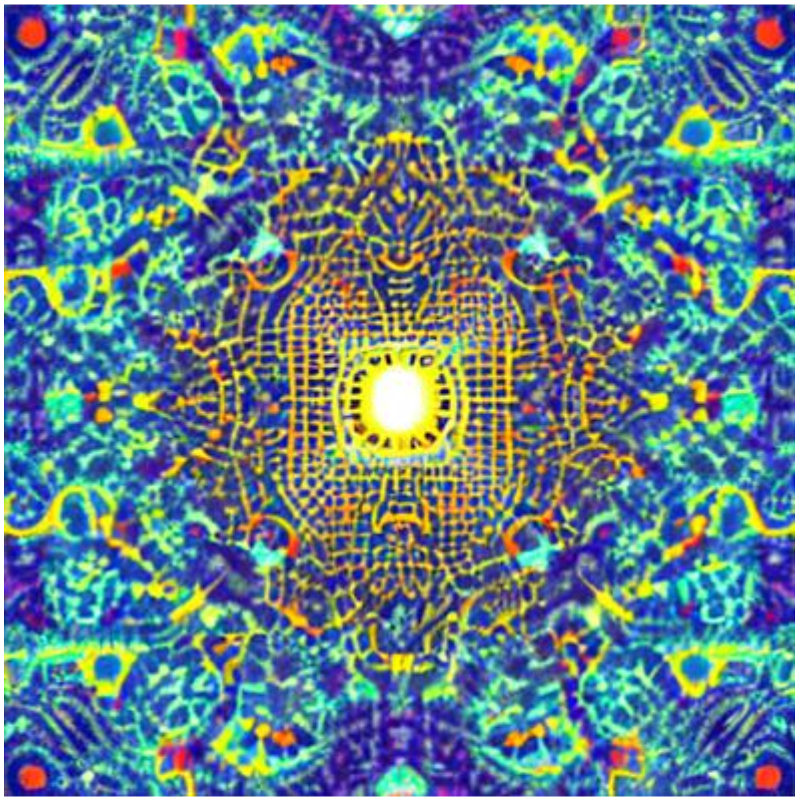


Figure 3.4: A book cover for a book about generative AI with Python – Stable Diffusion

Azure:

Azure, the cloud computing platform run by Microsoft, integrates with OpenAI to provide powerful language models like **GPT-3, Codex, and Embeddings**. It offers access, management, and development of applications and services through its global data centers for use cases such as writing assistance, summarization, code generation, and semantic search. It provides capabilities like **software as a service** (**SaaS**), **platform as a service** (**PaaS**), and **infrastructure as a service** (**IaaS**).

By authenticating either through GitHub or Microsoft credentials, we can create an account on Azure at https://azure.microsoft.com/.

We can then create new API keys under **Cognitive Services | Azure OpenAI**.

After setting up, the models should be accessible through the `AzureOpenAI()` class interface in LangChain.

Anthropic

Anthropic is an AI start-up and public-benefit corporation based in the United States. It was founded in 2021 by former members of OpenAI.

Unfortunately, Claude is not available to the general public (yet). You need to apply for access to use Claude and set the `ANTHROPIC_API_KEY` environment variable.

HayStack:

Haystack is a Python-based framework developed by deepset, a startup founded in 2018 in Berlin by Milos Rusic, Malte Pietsch, Timo Möller. Deepset provides developers with the tools to build Natural Language Processing (NLP)-based application, and with the introduction of Haystack they are making it to the next level.Haystack has the following core components:

- **Nodes**. These are components that perform a specific task or function, such as a retriever, a reader, a generator, a summarizer, etc. Nodes can be LLMs or other utilities that interact with LLMs or other resources. Among LLMs, Haystack supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the Hugging Face Hub.
- **Pipelines**. These are sequences of calls to nodes that perform natural language tasks or interact with other resources. Pipelines can be querying pipelines or indexing pipelines, depending on whether they perform searches on a set of documents or prepare documents for search. Pipelines are predetermined and hardcoded, meaning that they do not change or adapt based on the user input or the context.
- **Agent**. This is an entity that uses LLMs to generate accurate responses to complex queries. An agent has access to a set of tools, which can be pipelines or nodes, and it can decide which tool to call based on the user input and the context. An agent is dynamic and adaptive, meaning that it can change or adjust its actions based on the situation or the goal.

- **Tools**. There are functions that an agent can call to perform natural language tasks or interact with other resources. Tools can be pipelines or nodes that are available to the agent and they can be grouped into toolkits, which are sets of tools that can accomplish specific objectives.
- **DocumentStores**. Those are backends that store and retrieve documents for search. DocumentStores can be based on different technologies, also including VectorDB (such as FAISS, Milvus, or Elasticsearch).

A nice thing about HayStack is that you can deploy it as a REST API and be consumed directly

Semantic Kernel:

Semantic Kernel is the third open-shource SDK we are going to explore in this chapter. It's been developed by Microsoft, originally in C# and now available also in Python.This framework takes its name over the concept of "kernel" that, generally speaking, refers to the core or essence of a system. In the context of this framework, a kernel is meant to act as the engine that addresses user's input by chaining and concatenating a series of components into pipelines, encouraging **function composition.**

Semantic Kernel supports C#, Python, and Java

Semantic Kernel has the following main components:

- **Models**. These are the Large Language Models or Large Foundation Models that will be the engine of the application. Semantic Kernel supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the Hugging Face Hub.
- **Memory**. It allows the application to keep references to user's interactions, both in the short and long-term. Within the framework of Semantic Kernel, memories can be accessed in three ways:
    - **Key-value pairs**-> it consists in saving environment variables that store simple information, such as names or dates.
    - **Local-storage**->it consists in saving information to a file that can be retrieved by its filename, such as a CSV or JSON file.

    - **Semantic memory search**-> it is similar to LangChain's and HayStack's memory, as it uses embeddings to represent and search for text information based on its meaning.
- **Functions**. Functions can be seen as skills that mix LLM prompts and code, with the goal of making user's asks interpretable and actionable. There are two types of functions:
    - **Semantic functions**-> it is basically a templated prompt, which is a natural language query that specifies the input and output format for the LLM, also incorporating prompt configuration, which sets the parameters for the LLM.
    - **Native functions**->those refer to the native computer code that can route the intent captured by the semantic function and permorm the related task.

Feature	LangChain	HayStack	Semantic Kernel
LLM support	Proprietary and Open-source	Proprietary and Open-source	Proprietary and Open-source
Supported Languages	Python and JS/TS	Python	C#, Java and Python
Process orchestration	Chains	Pipelines of nodes	Pipelines of functions
Deployment	No REST API	REST API	No REST API
Memory	Key-value paris, local storage, semantic memory search (embeddings)	Embeddings	Embeddings

Most Promising LLMs Models:

1. Proprietary models: GPT-4, Palm2, Claude 2
2. Open Source models:

GPT-4:

Released in March 2023, GPT-4 is the latest model developed by OpenAI, as well as the top performer in the market right now. It belongs to the class of Generative Pretrained Transformer (GPT), a decoder-only transforber-based architecture.GPT-4, as the previous models in the GPT-series, has been trained on both publicly available and OpenAI licensed dataset (OpenAI didn't disclose the exact composition of the training set). **Additionally, to make the model more aligned with user's intent, the training process also involved a Reinforcement Learning with Human Feedback (RLHF) training.**

On MMLU, GPT-4 outperformed previous models not only in English, but also in other languages.

GPT-4 is a multi-modal models, meaning that it can take as input also images, in addition to text.

Another great improvement of GPT-4 with respect to its predecessors (GPT-3.5 and GPT-3) is its **noticeable reduction in the risk of hallucination**.

Finally, with GPT-4 OpenAI made an additional effort to make it safer and more aligned.

RLHF:

Reinforcement Learning with Human Feedback (RLHF) is a technique aims at using human feedbacks as an evaluating metric for LLMs' generated output, and then using that feedback to further optimize the model. There are two main steps to achieve that goal:

(1) Training a reward model based on human preferences.

(2) Optimizing the LLM with respect to the reward model. This step is done via **Reinforcement Learning**, is a type of machine learning paradigm where an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions, and its goal is to

maximize the cumulative reward over time by continuously adapting its behavior through trial and error.

With RLHF, thanks to the reward model the LLM is able to learn from human preferences and be more aligned with users' intents.

Hallucination:

Hallucination is a term that describes a phenomenon where large language models (LLMs) generate text that is incorrect, nonsensical, or not real, but appears to be plausible or coherent.

Hallucination can happen because LLMs are not databases or search engines that store or retrieve factual information. Rather, they are statistical models that learn from massive amounts of text data and produce outputs based on the patterns and probabilities they have learned. However, these patterns and probabilities may not reflect the truth or the reality, as the data may be incomplete, noisy, or biased. Moreover, LLMs have limited contextual understanding and memory, as they can only process a certain number of tokens at a time and abstract them into latent representations. Therefore, LLMs may generate text that is not supported by any data or logic, but is the most likely or correlated from the prompt.

Alignment:

Alignment is a term that describes the degree to which large language models (LLMs) behave in ways that are useful and harmless for their human users. For example, an LLM may be aligned if it generates text that is accurate, relevant, coherent, and respectful. An LLM may be misaligned if it generates text that is false, misleading, harmful, or offensive.

PaLM 2:

PaLM 2 is a large language model developed by Google that aims to improve the state of the art in natural language understanding and generation. **PaLM 2 stands for Pathways Language Model 2**, and it builds on the previous version of PaLM that was released in 2022.

PaLM 2 was trained on around 100 spoken languages and over 20 programming languages. The model was perfectioned by merging three disting areas of development in the field of Large Language Models, that cover the topic, respectively, of optimized compute, high-quality training dataset and state-of-the-art architecture. By doing so, Google was able to come up with a model that have the following features:

- **Compute-optimal scaling. PaLM 2 uses a technique called compute-optimal scaling, which balances the model size and the training data size in proportion to each other.** This makes PaLM 2 more efficient and faster than its predecessor PaLM, with fewer parameters and lower serving cost.
- **Improved dataset mixture. PaLM 2 is trained on a diverse and multilingual corpus of text, which includes hundreds of human and programming languages, mathematical equations, scientific papers, and web pages.** This gives PaLM 2 a broader knowledge base and better performance on multilingual tasks, such as translation, cross-lingual transfer, and language proficiency.
- On top of that, Google also put a lot of effort in improving PaLM 2 capabilities in its toxicity classification capabilities, adding a built-in control over toxic generation.

- **Updated model architecture and objective.** PaLM 2 has an improved architecture that made it possible to train it on a variety of tasks simultaneously. This allowed the model to learn different aspects of language from different data sources, as well as learn general and transferable skills that can be applied to new tasks.

Overall, PaLM-2 is capable of performing advanced reasoning, coding, and mathematics tasks that require logic, common sense, and creativity.

A peculiarity of PaLM-2 is its capability at solving mathematical task.

Google is already powering its products with PaLM 2, including Google Workspace (for example, email summarization in Gmail and content generation in Docs) and Bard (an AI assistant, similar to OpenAI’s ChatGPT).As of today, you can consume PaLM 2 via API, yet you first need to enroll the waitlist at https://makersuite.google.com/waitlist to get access to it. Alternatively, you can try PaLM directly with a chat interface via Bard.

CLAUDE 2:

CLAUDE 2, which stands for Constitutional Large-scale Alignment via User Data and Expertise, is an LLM developed by Anthropic, a research company founded by former OpenAI researchers and focused on AI safety and alignment. It was announced in July 2023.CLAUDE 2 is a transformer-based LLM that has been trained on a mix of publicly available information from the internet and proprietary data, via unsupervised learning, Reinforcement Learning with Human Feedback (RLHF) and **Constitutional AI.**This latter – Constitutional AI (CAI) - is a real peculiarity of CLAUDE

Anthropic developed this unique technique called Constitutional AI (CAI), disclosed in December 2022 in the paper "Constitutional AI: Harmlessness from AI Feedback*".* Constitutional AI involves using a set of principles to guide the model's behavior and outputs, rather than relying on human feedback or data alone. The principles are derived from various sources, such as the UN Declaration of Human Rights, trust and safety best practices, principles proposed by other AI research labs, non-western perspectives, and empirical research.Constitutional AI aims to make the model more safe and aligned with human values and intentions, by avoiding toxic or discriminatory outputs, avoiding helping a human engage in illegal or unethical activities, and broadly creating an AI system that is helpful, honest, and harmless.

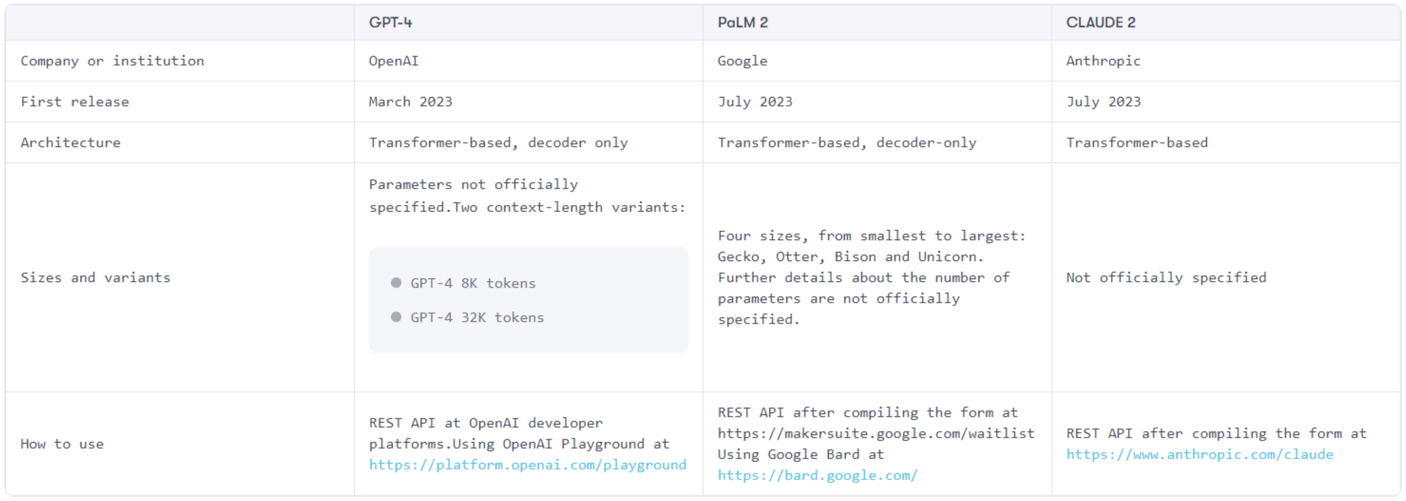
Constitutional AI uses the principles in two stages of the training process:

- First, the model is trained to critique and revise its own responses using the principles and a few examples.
- Second, the model is trained via reinforcement learning, but rather than using human feedback, it uses AI-generated feedback based on the principles to choose the more harmless output.

Another peculiarity of CLAUDE 2 is the context length, that reaches the 100k tokens. This means that users can input longer prompts, namely pages of technical documentations or even a book, which do not need to be embedded. Plus, the model can also generates longer output compared to other LLMs.Finally, CLAUDE 2 demontrates relevant capabilities also when working with code, scoring 71.2% on the HumanEval benchmark.

HumanEval benchmark:

HumanEval is a benchmark for evaluating the code generation ability of large language models (LLMs). It consists of 164 human-crafted coding problems in Python, each with a prompt, a solution, and a test suite. The problems cover various topics, such as data structures, algorithms, logic, math, and string manipulation. The benchmark can be used to measure the functional correctness, syntactic validity, and semantic coherence of the LLM's outputs.



	GPT-4	PaLM 2	CLAUDE 2
Company or institution	OpenAI	Google	Anthropic
First release	March 2023	July 2023	July 2023
Architecture	Transformer-based, decoder only	Transformer-based, decoder-only	Transformer-based
Sizes and variants	Parameters not officially specified.Two context-length variants: ● GPT-4 8K tokens ● GPT-4 32K tokens	Four sizes, from smallest to largest: Gecko, Otter, Bison and Unicorn. Further details about the number of parameters are not officially specified.	Not officially specified
How to use	REST API at OpenAI developer platforms.Using OpenAI Playground at https://platform.openai.com/playground	REST API after compiling the form at https://makersuite.google.com/waitlist Using Google Bard at https://bard.google.com/	REST API after compiling the form at https://www.anthropic.com/claude

Open-source models:

In addition to proprietary models, there is a huge market of open-source LLMs available today. The advantage of an open-source models is that, by definition, developers have full visibility and access to the source code.

LlaMA-2:

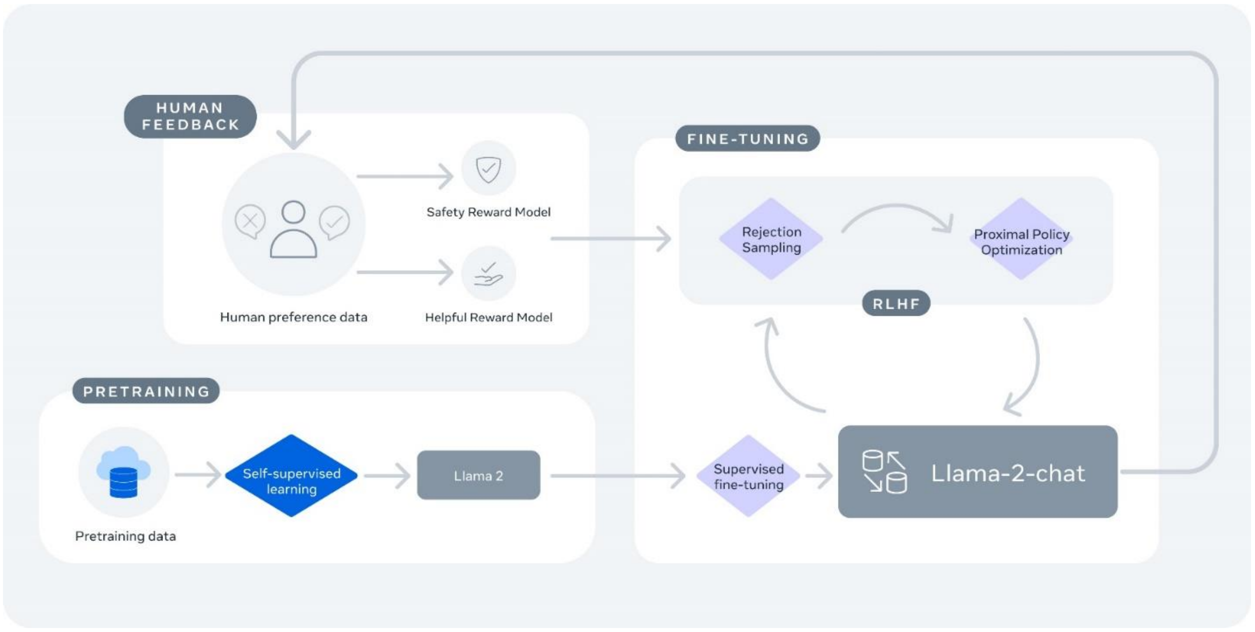
LLaMA-2 is a new family of models developed by Meta and unveiled to the public on 18 July, 2023, open-source and for free. It is an **auto-regressive** model with an optimized, decoder-only transformer architecture.

LlaMA-2 models come in three sizes: 7, 13, and 70 billion parameters. All the version have been trained on 2 trillion tokens and have a context lengths of 4092 tokens.On top of that, all model sizes come with a "chat" version, called LlaMA-2-chat. LlaMA-2-chat was developed with a fine-tuning process that consisted into two main steps:

- **Supervised fine-tuning**. This step involves fine-tuning the model on publicly available instruction datasets and over 1 million human annotations, to make them more helpful and safe for conversational use cases. The fine-tuning process uses a selected list of prompts to guide the model outputs, and a loss function that encourages diversity and relevance (that's the reason why it is "supervised").

- **Reinforcement Learning with Human Feedback (RLHF).** As we saw while introducing GPT-4, RFHF is a technique aims at using human feedbacks as an evaluating metric for LLMs' generated output, and then using that feedback to further optimize the model.

Below an illustration of how the training process of LlaMA work:



auto-regressive :

The concept of auto-regressive in the context of transformers refers to the fact that the model predicts the next token in the sequence, conditioned on all the previous tokens. This is done by masking the future tokens in the input, so that the model can only attend to the past tokens. For example, if the input sequence is "The sky is blue", the model would predict "The" first, then "sky", then "is", and finally "blue", using a mask to hide the tokens that come after each prediction.

Falcon LLM:

Falcon LLM

Falcon LLM is a representation of a new trend of LLMs, consisting on building lighter models (with fewer parameters) focusing rather on the quality of the training dataset. Indeed, it is a matter of fact that complex models like GPT-4 with trillion of parameters are extremely heavy, both in the training phase and inference phase. This implies the need for high and expensive computational power (GPU and TPU-powered) as well as long training time.Falcon LLM, is an open-source model launched by Abu Dhabi's Technology Innovation Institute (TII) in May 2023. It is an auto-regressive, decoder-only transformer, trained on 1 trillion tokens and has 40 billion parameters (even though it has also been released a lighter version with 7 billion parameters). Similarly to what we saw for LlaMA, also Falcon LLM comes with a fine-tuned variant, called "instruct", which is tailored towards following user's instructions.

So the question might be: how can a model with "only" 40 billion parameters perform so well? In fact, the answer is in the quality of the dataset.Falcon was developed using specialized tools and incorporates a unique data pipeline capable of extracting valuable content from web data. The pipeline was designed to extract high-quality content by employing extensive filtering and deduplication techniques.The resulting dataset, called *RefinedWeb*, has been released by TII under the Apache-2.0 license and can be found at https://huggingface.co/datasets/tiiuae/falcon-refinedweb.

By combining superior data quality with these optimizations, Falcon achieves remarkable performance while utilizing around 75% of the training compute budget of the GPT-3 and 80% of PaLM-62B's.

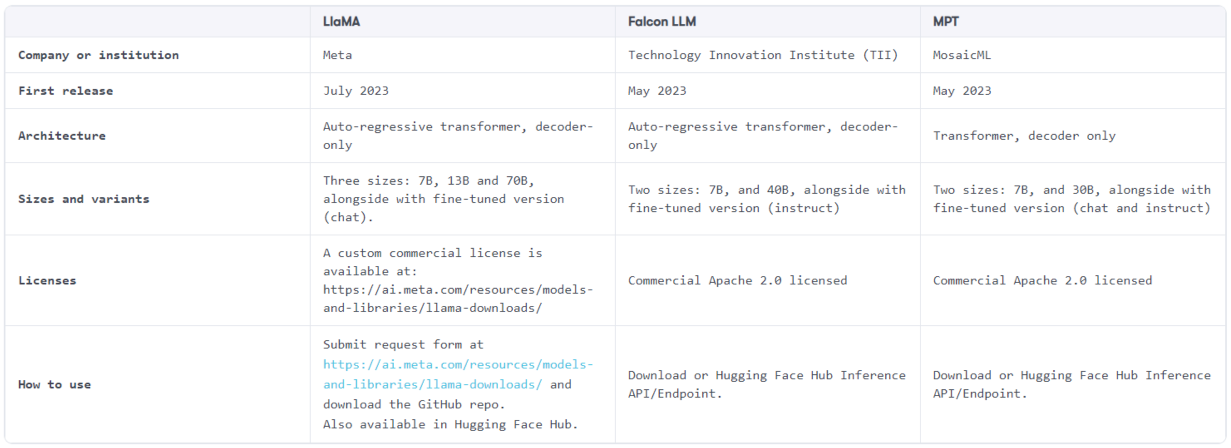
MPT:

The third and last open-source model series we are going to cover is MosaciML Pretrainer Transformer (MPT), developed by MosaicML, a company that provides a platform for training and deploying LLMs. The first version with 7 billion parameters, MPT-7B, was launched in May 2023, alongside some fine-tuned variants (as we saw for the LlaMA series):

- **MPT-7B-Instruct**. A model for short-form instruction following.
- **MPT-7B-Chat**. A chatbot-like model for dialogue generation.
- **MPT-7B-StoryWriter-65k+.** A model designed to read and write stories with super long context lengths, with a context length of 65k tokens. This noticeable

length of tokens is achieved thanks to the replacing of positional embedding swith Attention with Linear Biases (ALiBI).

Then, in June 2023 a new, more complex version with 30 billion parameters was launched: MPT-30B. This latest version has been trained on an initial chunk of 1 trillion tokens with 2k-long sequences, and then an additional 50 billion chunk of tokens with 8k-long sequence. The 8k tokens-long context window allows the model to handle 8k tokens context when consumed, longer than LlaMA or Falcon LLM.



	LlaMA	Falcon LLM	MPT
Company or institution	Meta	Technology Innovation Institute (TII)	MosaicML
First release	July 2023	May 2023	May 2023
Architecture	Auto-regressive transformer, decoder-only	Auto-regressive transformer, decoder-only	Transformer, decoder only
Sizes and variants	Three sizes: 7B, 13B and 70B, alongside with fine-tuned version (chat).	Two sizes: 7B, and 40B, alongside with fine-tuned version (instruct)	Two sizes: 7B, and 30B, alongside with fine-tuned version (chat and instruct)
Licenses	A custom commercial license is available at: https://ai.meta.com/resources/models-and-libraries/llama-downloads/	Commercial Apache 2.0 licensed	Commercial Apache 2.0 licensed
How to use	Submit request form at https://ai.meta.com/resources/models-and-libraries/llama-downloads/ and download the GitHub repo. Also available in Hugging Face Hub.	Download or Hugging Face Hub Inference API/Endpoint.	Download or Hugging Face Hub Inference API/Endpoint.

Beyond Language Models:

Samples of Large Foundation Models available in the market today.

- **Whisper**. It is is a general-purpose speech recognition model developed by OpenAI that can transcribe and translate speech in multiple languages. It is trained on a large dataset of diverse audio and is also a multitasking model that can perform multilingual speech recognition, speech translation, spoken language identification, and voice activity detection.
- **Midjourney**. Developed by the homonymous independent research lab, Midjourney is based on a Transformer sequence-to-sequence model that takes

text prompts and outputs a set of four images that match the prompts. Midjourney is designed to be a tool for artists and creative professionals, who can use it for rapid prototyping of artistic concepts, inspiration, or experimentation.

- **DALL-E**. Similarly to the previous one, DALL-E, developed by OpenAI, generates images from natural language descriptions, using a 12-billion parameter version of GPT-3 trained on a dataset of text-image pairs.

The idea is that we can combine and orchestrate multiple LFMs within our applications to achieve extraordinary results. For example, lets say we want to write a review about an interview with a young chef and post it on Instagram. The involved models might be the following:

- Whisper->it will convert the interview audio into a transcript.
- An LLM, such as Falcon-7B-instruct, with a web plug-in->it will extrapolate the name of the young chef and search it in the internet to retrieve the biography.
- An LLM, such as LlaMA->it will process the transcript and generate a review with an Instagram post-like style. We can ask the same model also to generate a prompt that will ask the following model to generate a picture based on the post content.
- Dall-E->it will generate an image based on the prompt generated by the LLM.

A decision framework to pick the right LLM:

There are many factors to consider when choosing a large language model (LLM) for your application.Below you can find some factors and trade-offs you might want to consider while choosing your LLMs:

- **Size and performance**. We saw that complex models (that means, with high number of parameters) tend to have better performance, especially in terms of parametric knowledge and generalization capabilities. Nevertheless, the larger the model, the more computation and memory it requires to process the input and generate the output. Which can result in higher latency and, as we will see, in higher costs.

- **Cost and hosting strategy**. When incorporating LLMs within our applications, there are two type of costs we have to have in mind:
- **Cost for model consumption**->it refers to the fee we pay to consume the model. Proprietary models like GPT-4 or CLAUDE 2 require a fee, which is typically proportional to the number of tokens processed. On the other hand, open-source models like LlaMA or Falcon LLM are free to use.
- **Cost for model hosting**->it refers to your hosting strategy. Typically, proprietary models are hosted in private or public hyperscaler, so that they can be consumed via REST API and you don't have to worry about the underlying infrastructure (for example, GPT-4 is hosted in a super-computer built in Microsoft Azure cloud). With open-source models, we typically need to provide our own infrastructure, since those models can be downloaded locally. Of course, the larger the model, the more powerful the computational power needed.

Pick CLAUDE 2 if you are looking for exceptional coding capabilities, or PaLM 2 if analytical reasoning is what you are looking for. On the other hand, if you need a model which encompass all the above capabilities, GPT-4 might be the right choice for you.

LLM'S Performance:

The most popular way of evaluating LLMs' performance is that of averaging different benchmarks across domains. However, there are benchmarks that are tailored towards specific capabilities: if **MMLU** measure LLMs' generalized culture and common sense reasoning, **TruthfulQA** is more concerned around LLMs' alignment, while **HumanEval** is tailored towards LLMs' coding capabilities.

Prompt Engineering:

A prompt is a text input that guides the behaviour of an LLM to generate a text output. Prompt engineering is the process of designing effective prompts that elicit high-quality and relevant outputs from LLMs. Prompt engineering requires creativity, understanding of the LLM, and precision.

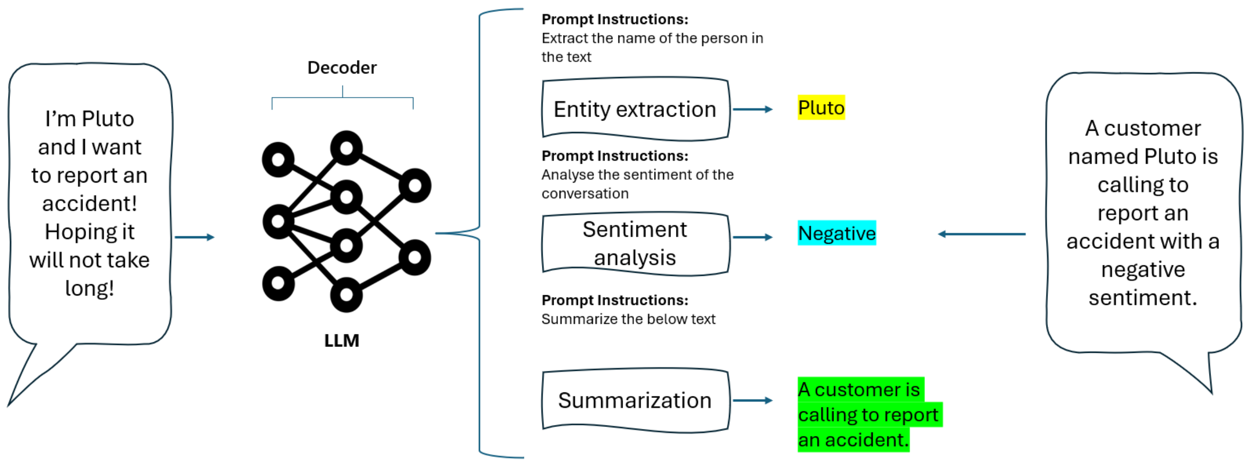

I'm Pluto and I want to report an accident! Hoping it will not take long!
Decoder
LLM
Prompt Instructions: Extract the name of the person in the text
Entity extraction
Pluto
Prompt Instructions: Analyse the sentiment of the conversation
Sentiment analysis
Negative
Prompt Instructions: Summarize the below text
Summarization
A customer is calling to report an accident.
A customer named Pluto is calling to report an accident with a negative sentiment.

Figure 1: Example of prompts engineering to specialize LLMs.

Prompt engineering involves selecting the right words, phrases, symbols, and formats that elicit the desired response from the LLM. Prompt engineering also involves using other controls, such as parameters, examples, or data sources, to influence the LLM's behavior.

Principles of prompt engineering:

1. **Clear Instructions:**
   The principle of giving clear instructions is to provide the model with enough information and guidance to perform the task correctly and efficiently.

2. **Split complex tasks into subtasks:**
   Here are some examples of splitting complex tasks into subtasks:
   Text summarization:
   Machine translation:
   Poem generation:
   Code generation:

3. **Ask for justification:**
4. **Generate many outputs, then use the model to pick the best one.**

splits the job into two sub-jobs for our LLM:

- Generating multiple responses to user's query;
- Comparing those responses and picking the best one, according to some criteria we can specify in the metaprompt.

5. **Repeat instructions at the end:**
 repeating the main instruction at the end of the prompt can help the model to overcome its inner **recency bias.**

Recency bias:

Recency bias is the tendency of large language models to give more weight to the information that appears near the end of a prompt and ignore or forget the information that appears earlier. This can lead to inaccurate or inconsistent responses that do not take into account the whole context of the task. For example, if the prompt is a long conversation between two people, the model may only focus on the last few messages and disregard the previous ones.

One possible way to overcome recency bias is to break down the task into smaller steps or subtasks and provide feedback or guidance along the way.

Another way to overcome recency bias with prompt engineering techniques is to repeat the instructions or the main goal of the task at the end of the prompt.

6. **Use delimiters:**

   A delimiter can by any sequence of characters or symbols that is clearly mapping a schema rather than a concept.

7. **Few-shot approach:**

   GPT-3 can achieve strong performance on many NLP tasks in a few-shot setting.This means that for all tasks, GPT-3 is applied without any fine-tuning, with tasks and few-shot demonstrations specified purely via text interaction with the model.

   This is an example and evidence of how the concept of few-shot learning – which means, providing the models with examples of how we would like it to respond – is a powerful techniques that enables models’ customization without interfering with their overall architecture.

   few shot learning, most of the time, is powerful enough to customize a model even if extremely specialized scenarios, where we could think about fine tuning as the proper tool. In fact, a proper few shot learning could be as effective as a fine tuning process.

The model was able to correctly classify all the reviews, without even fine tuning! This is just an example of what you can achieve – in terms of model specialization – with the technique of few shot learning.

8. Chain of thoughts:

   Chain of thought (CoT) is a technique that enables complex reasoning capabilities through intermediate reasoning steps. It also encourages the model to explain its reasoning, “forcing” it not to be too fact and risking to give the wrong response (as we saw in previous sections).

9. ReAct:

   ReAct (Reason and Act) is a general paradigm that combines reasoning and acting with large language models. ReAct prompts the language model to generate verbal reasoning traces and actions for a task, and also receives observations from external sources such as web search or databases. This allows the language model to perform dynamic reasoning, and quickly adapt its acting plan based on external information.

   ReAct prompts the language model to generate intermediate reasoning steps, actions, and observations for a task.

LangChain:

Created in 2022 by Harrison Chase, LangChain is an open-source Python framework for building LLM-powered applications.

In particular, LangChain’s support for chains, agents, tools, and memory allows developers to build applications that can interact with their environment in a more sophisticated way and store and reuse information over time.

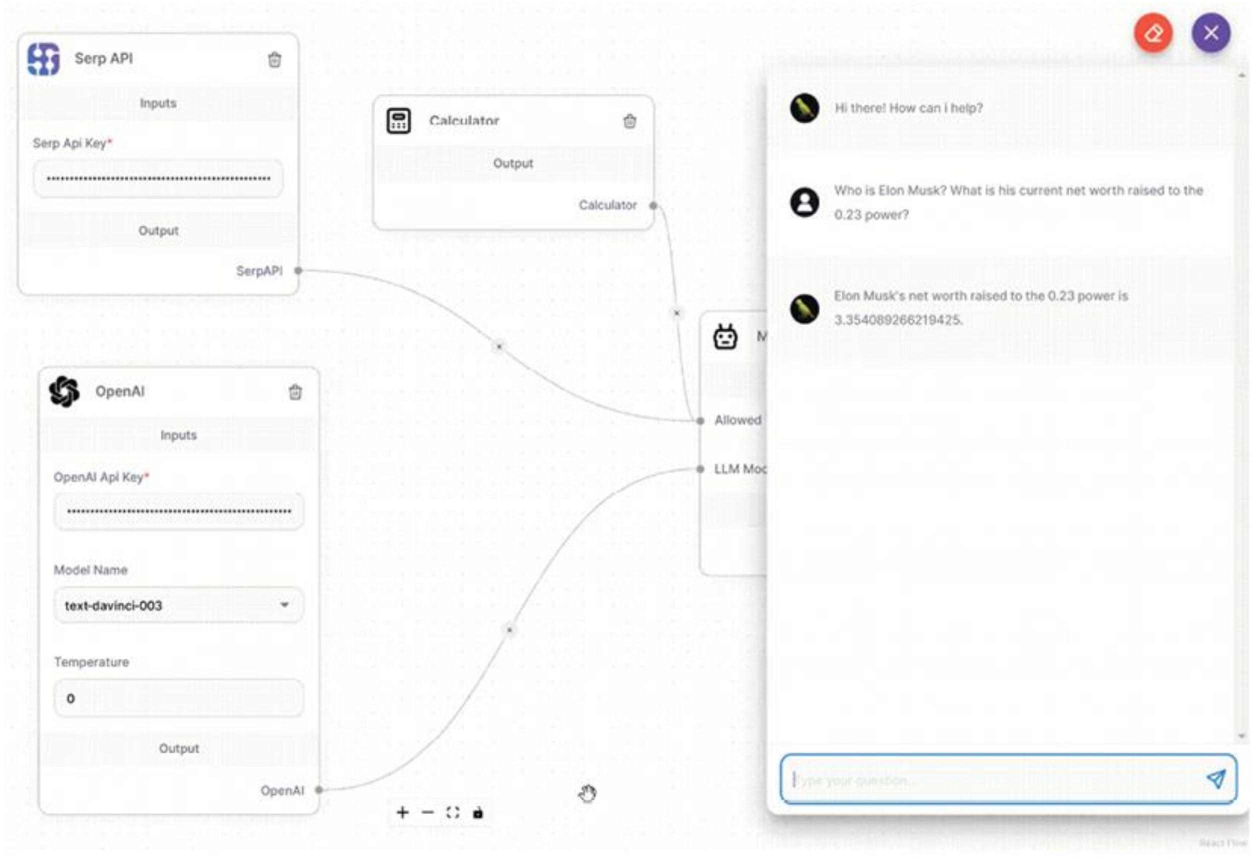
The key benefits LangChain offers developers are:

- **Modular architecture** for flexible and adaptable LLM integrations.
- **Chaining together** multiple services beyond just LLMs.
- Goal-driven agent interactions instead of isolated calls.
- **Memory and persistence** for statefulness across executions.
- **Open-source access** and community support.

LlamaHub is a library of data loaders, readers, and tools created by the LlamaIndex community. It provides utilities to easily connect LLMs to diverse knowledge sources. The loaders ingest data for retrieval, while tools enable models to read/write to external data services. LlamaHub simplifies the creation of customized data agents to unlock LLM capabilities.

LangChainHub is a central repository for sharing artifacts like prompts, chains, and agents used in LangChain. Inspired by the Hugging Face Hub, it aims to be a one-stop resource for discovering high-quality building blocks to compose complex LLM apps. The initial launch focuses on a collection of reusable prompts. Future plans involve adding support for chains, agents, and other key LangChain components.

LangFlow and **Flowise** are UIs that allow chaining LangChain components in an executable flowchart by dragging sidebar components onto the canvas and connecting them together to create your pipeline. This is a quick way to experiment and prototype pipelines and is illustrated in the following screenshot of Flowise (source: https://github.com/FlowiseAI/Flowise):



Serp Api Key*
••

OpenAI Api Key*
••

Model Name
text-davinci-003

Temperature
0

Hi there! How can i help?

Who is Elon Musk? What is his current net worth raised to the 0.23 power?

Elon Musk's net worth raised to the 0.23 power is 3.354089266219425.

Figure 2.9: Flowise UI with an agent that uses an LLM, a calculator, and a search tool

LangChain and LangFlow can be deployed locally, for example, using the Chainlit library, or on different platforms, including Google Cloud. The `langchain-serve` library helps to deploy both LangChain and LangFlow on the Jina AI cloud as LLM-apps-as-a-service with a single command.

LangChain unlocks more advanced LLM applications via its combination of components like memory, chaining, and agents.

LangChain as a way to overcome LLM limitations and build innovative language-based applications. We aim to demonstrate the potential of combining recent AI advancements with a robust framework like LangChain.

some challenges faced when using LLMs on their own, like the lack of external knowledge, incorrect reasoning, and the inability to take action. LangChain provides solutions to these issues through different integrations and off-the-shelf components for specific tasks.

LangChain is a lightweight framework meant to make it easier to integrate and orchestrate LLMs' and their components within applications. It is mainly Python based, yet it recently extended its support to JavaScript and TypeScript.In addition to Large Language Models integration (which we will cover in the upcoming dedicated paragraph), we saw that LangChain offer the following main components:

- Models and prompt templates
- Data Connections
- Memory
- Chains
- Agents

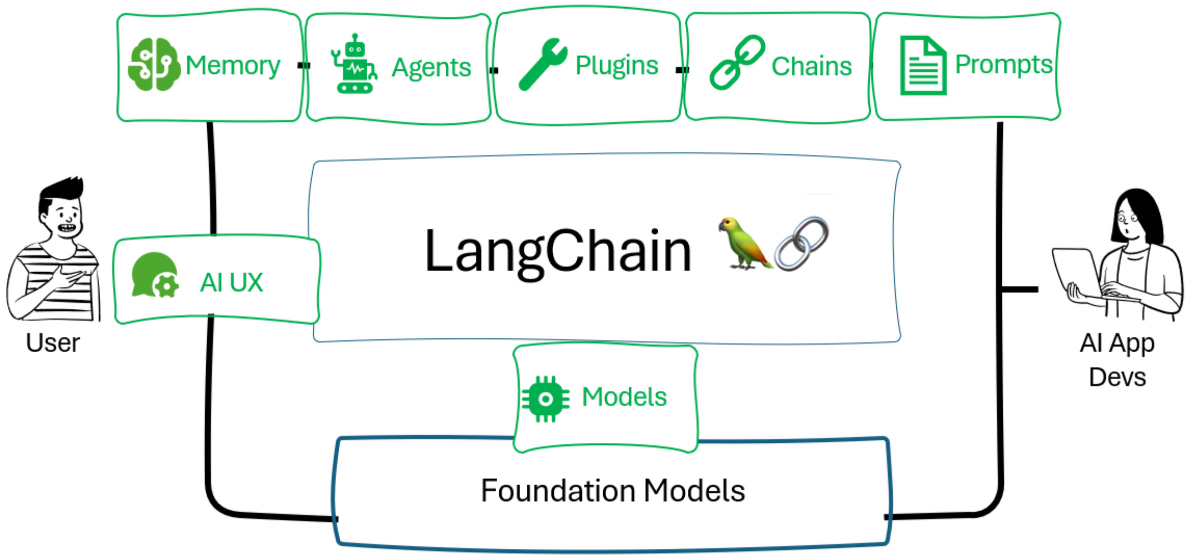


Fig: LangChain components

Stochastic Parrots:

Stochastic parrots refers to LLMs that can produce convincing language but lack any true comprehension of the meaning behind words. Without being grounded in the real world, models can produce responses that are inaccurate, irrelevant, unethical, or make little logical sense.

Raw model scale alone cannot transform stochastic parroting into beneficial systems. Innovations like prompting, chain-of-thought reasoning, retrieval grounding, and others are needed to educate models.

Limitations of LLMs:

- **Outdated knowledge**: LLMs rely solely on their training data. Without external integration, they cannot provide recent real-world information.
- **Inability to take action**: LLMs cannot perform interactive actions like searches, calculations, or lookups. This severely limits functionality.
- **Lack of context**: LLMs struggle to incorporate relevant context like previous conversations and the supplementary details that are needed for coherent and useful responses.
- **Hallucination risks**: Insufficient knowledge on certain topics can lead to the generation of incorrect or nonsensical content by LLMs if not properly grounded.
- **Biases and discrimination**: Depending on the data they were trained on, LLMs can exhibit biases that can be religious, ideological, or political in nature.
- **Lack of transparency**: The behavior of large, complex models can be opaque and difficult to interpret, posing challenges to alignment with human values.
- **Lack of context**: LLMs may struggle to understand and incorporate context from previous prompts or conversations. They may not remember previously mentioned details or may fail to provide additional relevant information beyond the given prompt.

LLMs face significant limitations in their lack of real-time knowledge and inability to take actions themselves, which restricts their effectiveness in many real-world contexts

An LLM would have zero awareness of current events that occurred after its training data cut-off date. Asking an LLM about breaking news or the latest societal developments would leave it unable to construct responses without external grounding.

LLMs cannot interact dynamically with the world around them. They cannot check the weather, look up local data, or access documents.

Mitigating LLM limitations:

- **Retrieval augmentation**: This technique accesses knowledge bases to supplement an LLM’s outdated training data, providing external context and reducing hallucination risk.
- **Chaining**: This technique integrates actions like searches and calculations.
- **Prompt engineering**: This involves the careful crafting of prompts by providing critical context that guides appropriate responses.
- **Monitoring, filtering, and reviews**: This involves ongoing and effective oversight of emerging issues regarding the application’s input and output to detect issues. Both manual reviews and automated filters then correct potential problems with the output. This includes the following:
    a. **Filters**, like block lists, sensitivity classifiers, and banned word filters, can automatically flag issues.
    b. **Constitutional principles** monitor and filter unethical or inappropriate content.
    c. **Human reviews** provide insight into model behavior and output.
- **Memory**: Retains conversation context by persisting conversation data and context across interactions.
- **Fine-tuning**: Training and tuning the LLM on more appropriate data for the application domain and principles. This adapts the model’s behavior for its specific purpose.

LLM app:

an **LLM app** is an application that utilizes an LLM to understand natural language prompts and generate responsive text outputs. LLM apps typically have the following components:

- A client layer to collect user input as text queries or decisions.

- A prompt engineering layer to construct prompts that guide the LLM.
- An LLM backend to analyze prompts and produce relevant text responses.
- An output parsing layer to interpret LLM responses for the application interface.
- Optional integration with external services via function APIs, knowledge bases, and reasoning algorithms to augment the LLM's capabilities.

In the simplest possible cases, the frontend, parsing, and knowledge base parts are sometimes not explicitly defined, leaving us with just the client, the prompt, and the LLM:

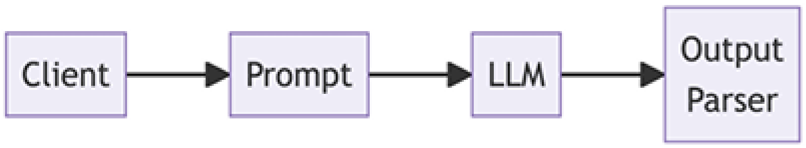


Figure 2.6: A simple LLM application

LLM apps can integrate external services via:

- Function APIs to access web tools and databases.
- Advanced reasoning algorithms for complex logic chains.
- Retrieval augmented generation via knowledge bases

RAG:

Retrieval augmented generation (RAG**), which we will discuss in *Chapter 5*, *Building a Chatbot like ChatGPT*, enhances the LLM with external knowledge**. These extensions expand the capabilities of LLM apps beyond the LLM's knowledge alone. For instance:

- Function calling allows parameterized API requests.
- SQL functions enable conversational database queries.
- Reasoning algorithms like chain-of-thought facilitate multi-step logic.

1. Model and Prompts:

   LangChain offers more than 50 integrations towards third-party vendor and platforms, including OpenAI, Azure OpenAI, Databricks and MosaicML, as well as the integration with Hugging Face Hub and the world of open-source LLMs.

   `from langchain.llms import OpenAI`

   `llm = OpenAI(openai_api_key="your-api-key")`

   `print(llm('tell me a joke'))`

   By default, the module `OpenAI` use the GPT-3 (or `text-davinci-003`) as base model.

   There are two main components related to LLM's prompts and prompts' design/engineering:

   a. **Prompt templates**. A prompt template is a component that defines how to generate a prompt for a language model. It can include variables, placeholders, prefixes, suffixes, and other elements that can be customized according to the data and the task.

      `from langchain import PromptTemplate`

      Generally speaking, prompt templates tend to be agnostic with respect to the LLM you might decide to use, and it is adaptable to both completion and chat models

      Completion models:

A completion model is a type of LLM that takes a text input and generates a text output, called a completion. The completion model tries to continue the prompt in a coherent and relevant way, according to the task and the data it was trained on. For example, a completion model can generate summaries, translations, stories, code, lyrics, and more, depending on the prompt.

Chat models:

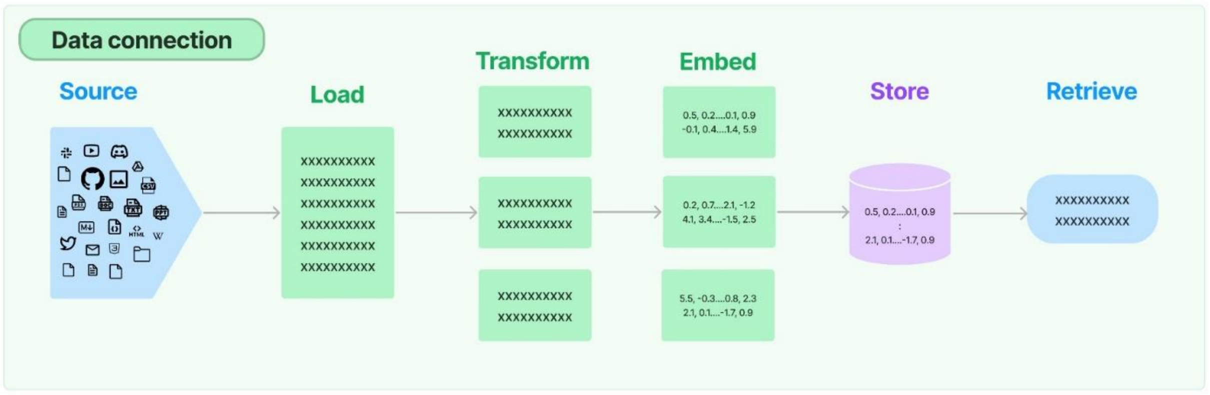
A chat model is a special kind of completion model that is designed to generate conversational responses. A chat model takes a list of messages as input, where each message has a role (either system, user, or assistant) and content. The chat model tries to generate a new message for the assistant role, based on the previous messages and the system instruction.

The main difference between completion and chat models is that completion models expect a single text input as prompt, while chat models expect a list of messages as input.

b. **Example selector.** An example selector is a component in LangChain that allows you to choose which examples to include in a prompt for a language model. A prompt is a text input that guides the language model to produce a desired output. Examples are pairs of inputs and outputs that demonstrate the task and the format of the output.

2. Data Connections:

- Data connections refer to the building blocks needed to retrieve the additional non-parametric knowledge we want to provide the model with. The idea is that of covering the typical flow of incorporating user-specific data into applications that made of five main blocks, as illustrated in the following picture:



- a. **Document loaders**. They are in charge of loading documents from different sources such as CSV, File Directory, HTML, JOSN, Markdown and PDF. Document loaders expose a `.load` method for loading data as documents from a configured source. The output is a `Document` object that contains a piece of text and associated metadata.

```
from langchain.document_loaders.csv_loader import CSVLoader

loader = CSVLoader(file_path='sample.csv')

data = loader.load()

print(data)
```

b. **Document transformers.** After importing your documents, it's common to modify them to better match your needs. A basic instance of this is breaking down a lengthy document into smaller chunks that fit your model's context window. Within LangChain, there are various pre-built document transformers available called **Text splitters**. The idea of Text splitters is to make it easier to split documents into chunks that are semantically related, so that we do not lose context or relevant information.

- With Text splitters, you can decide how to split the text (for example, by character, heading, token…) and how to measure the length of the chunk (for example, by number of characters).

```
with open('mountain.txt') as f:

    mountain = f.read()

from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(

    chunk_size = 100,

    chunk_overlap  = 20,

    length_function = len

)

texts = text_splitter.create_documents([mountain])

print(texts[0])

print(texts[1])

print(texts[2])
```

c. **Text embedding models**.

The concept of embedding as are a way to represent words, subwords, or characters in a continuous vector space.

- Embeddings are the key step to incorporate non-parametric knowledge into LLMs. In fact, once properly stored into a VectorDB (which will be covered in next section), they become the non-parametric knowledge against which we can measure the distance of user's query.

- To get started with embedding, you will need an embedding model:

Then, LangChain offer the Embedding class with two main modules, that address the embedding of, respectively, the non-parametric knowledge (multiple input text) and the user query (single input text).

```
from langchain.embeddings import OpenAIEmbeddings

from dotenv import load_dotenv

load_dotenv()

os.environ["OPENAI_API_KEY"]

embeddings_model = OpenAIEmbeddings(model ='text-embedding-
ada-002' )

embeddings = embeddings_model.embed_documents(

    [

        "Good morning!",

        "Oh, hello!",

        "I want to report an accident",

        "Sorry to hear that. May I ask your name?",

        "Sure, Mario Rossi."

    ]

)

print("Embed documents:")
print(f"Number of vector: {len(embeddings)}; Dimension of each vector: {len(embeddings[0])}")

embedded_query = embeddings_model.embed_query("What was the name mentioned in the conversation?")

print("Embed query:")

print(f"Dimension of the vector: {len(embedded_query)}")

print(f"Sample of the first 5 elements of the vector: {embedded_query[:5]}")
```

- Once we have both documents and query embedded, the next step will be that of computing the similarity between the two elements and retrieve the most suitable information from the documents embedding. We will see the details of this passage when talking about Vector stores.

d. **Vector stores.** A vector store (or VectorDB) is a type of database that can store and search over unstructured data, such as text, images, audio, or video, by using embeddings. By using embeddings, vector stores can perform fast and accurate similarity search, which means finding the most relevant data for a given query.

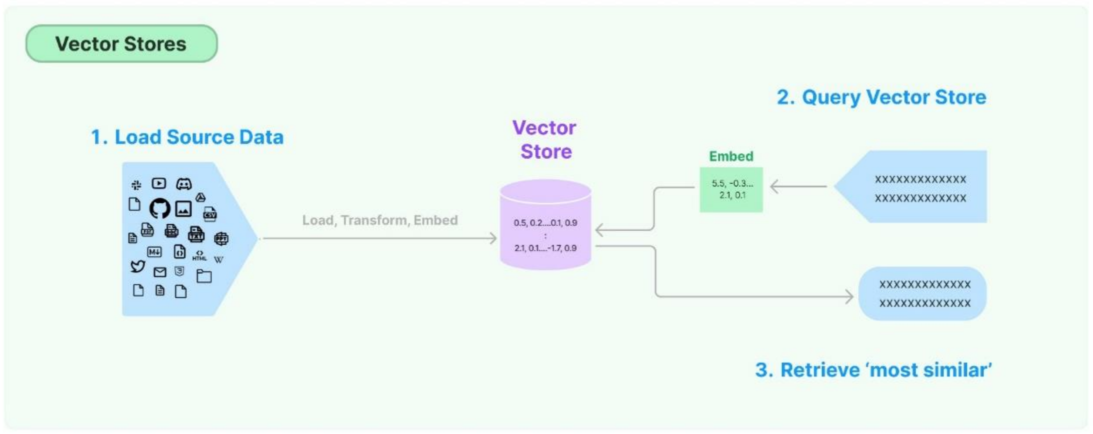
Similarity:

Similarity is a measure of how close or related two vectors are in a vector space. In the context of Large Language Models, vectors are numerical representations of sentences, words or documents that capture their semantic meaning, and the distance between those vector should be representative of their semantic similarity.

There are different ways to measure similarity between vectors, and while working with LLMs, one of the most popular measure in use is the cosine similarity.

This is the cosine of the angle between two vectors in a multidimensional space. It is computed as the dot product of the vectors divided by the product of their lengths. Cosine similarity is insensitive to scale and location, and it ranges from -1 to 1, where 1 means identical, 0 means orthogonal, and -1 means opposite.

The following is an illustration of the typical flow while using a vector store.



- LangChain offers more than 40 integrations towards 3 party vector stores. Some examples are FAISS, ElasticSearch, MongoDB Atlas, and Azure Search
- Facebook AI Similarity Search (FAISS) vectore store, which has been developed by Meta AI research for efficient similarity search and clustering of dense vectors

```
from langchain.document_loaders import TextLoader

from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS

from dotenv import load_dotenv load_dotenv()
os.environ["OPENAI_API_KEY"]
```

e. **Retrievers**. A retriever is a component in Langchain that can return documents relevant to an unstructured query, such as a natural language question or a keyword. A retriever does not need to store the documents itself, but only to retrieve them from a source. A retriever can use different methods to find the relevant documents, such as keyword matching, semantic search, or ranking algorithms

- The difference between a retriever and a vector store is that a retriever is more general and flexible than a vector store. A retriever can use any method to find relevant documents, while a vector store relies on embeddings and similarity metrics. A retriever can also use different sources of documents, such as web pages, databases, or files, while a vector store needs to store the data itself.
- However, a vector store can also be used as the backbone of a retriever, if the data is embedded and indexed by a vector store. In that case, the retriever can use the vector store to perform similarity search over the embedded data and return the most relevant documents. This is one of the main types of retrievers in Langchain, called a vector store retriever.

```
from langchain.chains import RetrievalQA from
langchain.llms import OpenAI retriever = db.as_retriever()
```

Overall, data connections modules offer a plethora of integrations and pre-built templates that make it easier to manage the flow of your LLM-powered application.

3. Memory:

In LangChain, memory refers to the persisting state between executions of a chain or agent. For example, storing chat history context in memory improves the coherence and relevance of LLM responses over time.

Rather than treating each user input as an isolated prompt, chains can pass conversational memory to models on each call to provide consistency. Agents can also persist facts, relationships, and deductions about the world in memory. This knowledge remains available even as real-world conditions change, keeping the agent contextually informed.

Retaining information in memory reduces the number of calls to LLMs for repetitive information. This lowers API usage and costs, while still providing the agent or chain with the needed context.

LangChain provides a standard interface for memory, integrations with storage options like databases, and design patterns for effectively incorporating memory into chains and agents.

- `ConversationBufferMemory` stores all messages in model history. This increases latency and costs.
- `ConversationBufferWindowMemory` retains only recent messages.
- `ConversationKGMemory` summarizes exchanges as a knowledge graph for integration into prompts.
- `EntityMemory` backed by a database persists agent state and facts.

Moreover, LangChain integrates many database options for durable storage:

- SQL options like Postgres and SQLite enable relational data modeling.
- NoSQL choices like MongoDB and Cassandra facilitate scalable unstructured data.
- Redis provides an in-memory database for high-performance caching.
- Managed cloud services like AWS DynamoDB remove infrastructure burdens.

Beyond databases, purpose-built memory servers like Remembrall and Motörhead offer optimized conversational context.

In the context of LLM-powered applications, memory allows the application to keep references to user’s interactions, both in the short and long-term. LangChain offers several modules for designing your memory system within your applications, enabling it with both reading and writing skills.The first step to do with your memory system is that of actually store your human interactions somewhere. To do so, you can leverage more than x built-in memory integrations with 3rd party provider, including Redis, Cassandra, Postgres.Then, when it comes to define how to query your memory system, there are varioys memory types you can leverage:

- **Conversation buffer memory.** This is the “plain vanilla” memory type available in LangChain. It allows you to store your chat messages and extract them in a variable.
- **Conversation buffer window memory**. It is identical to the previous one, with the only difference of allowing for a sliding window over only K interactions, so that you can manage longer chat history over time.
- **Entity memory**. Entity memory is a feature of Langchain that allows the language model to remember given facts about specific entities in a conversation. An entity is a person, place, thing, or concept that can be identified and distinguished from others. For example, in the sentence “Deven & Sam are working on a hackathon in Italy”, Deven and Sam are entities (person), as well as hackathon (thing) and Italy (place).
- Entity memory works by extracting information on entities from the input text using a LLM. It then builds up its knowledge about that entity over time by storing the extracted facts in a memory store. The memory store can be accessed and updated by the language model whenever it needs to recall or learn new information about an entity.
- **Conversation Knowledge Graph memory**. This type of memory uses a knowledge graph to recreate memory.

Knowledge graph:

A knowledge graph is a way of representing and organizing knowledge in a graph structure, where nodes are entities and edges are relationships between them. A knowledge graph can store and integrate data from various sources, and encode the semantics and context of the data. A knowledge graph can also support various tasks, such as search, question answering, reasoning, and generation.

Another example of a knowledge graph is DBpedia, which is a community project that extracts structured data from Wikipedia and makes it available on the web. DBpedia covers topics such as geography, music, sports, films, and more, and provides links to other datasets like GeoNames and WordNet.

- **Conversation summary memory**. When it comes to longer conversations to be stored, this type of memory can be very useful, since it takes creates a summary of the conversation over time (leveraging an LLM).
- **Conversation summary buffer memory**. This type of memory combines the ideas behind Buffer Memory and Conversation Summary Memory. It keeps a buffer of recent interactions in memory, but rather than just completely flushing old interactions (as occurs for the conversation buffer memory) it compiles them into a summary and uses both.
- **Conversation token buffer memory**. It is similar to the previous one, with the difference that, to determine when to start summarizing the interactions, this type of memory use token lengths rather than the number of interactions (as occurs in summary buffer memory).
- **Vector store-backed memory**. This type of memory leverages the concepts of embeddings and vectore stores previously covered. It is different from all the previous memories since it stores interactions as vectors, and then retrieve the top-K most similar texts every time it is queried, using a retriever.

```
from langchain.memory import ConversationSummaryMemory,
ChatMessageHistory from langchain.llms import OpenAI
```

A knowledge graph memory is useful for applications that need to access information from a large and diverse corpus of data and generate responses based on semantic relationships, while a Conversation summary buffer memory could be suitable for creating conversational agents that can maintain a coherent and consistent context over multiple turns, while also being able to compress and summarize the previous dialogue history.

4. **Chains:**

A chain is a sequence of calls to components, which can include other chains. The most innocuous example of a chain is probably the `PromptTemplate`, which passes a formatted response to a language model.

Prompt chaining is a technique that can be used to improve the performance of LangChain applications, which involves chaining together multiple prompts to autocomplete a more complex response. More complex chains integrate models with tools like `LLMMath`, for math-related queries, or `SQLDatabaseChain`, for querying databases. These are called **utility chains**, because they combine language models with specific tools.

LangChain implements chains to make sure the content of the output is not toxic, does not otherwise violate OpenAI's moderation rules (`OpenAIModerationChain`), or that it conforms to ethical, legal, or custom principles (`ConstitutionalChain`).

An `LLMCheckerChain` verifies statements to reduce inaccurate responses using a technique called self-reflection. The `LLMCheckerChain` can prevent hallucinations and reduce inaccurate responses by verifying the assumptions underlying the provided statements and questions. this strategy has been found to improve task performance by about 20% on average across a benchmark including dialogue responses, math reasoning, and code reasoning.

Like agents, router chains can decide which tool to use based on their descriptions. A `RouterChain` can dynamically select which retrieval system, such as prompts or indexes, to use.

Chains are predetermined sequences of actions and calls to LLMs that make it easier to build complex applications that require combining LLMs with each other or with other components. LangChain offer four main type of chain to get started with:

- **LLMChain**. This is the most common type of chain. It consists of a PromptTemplate, an LLM, and an optional **output parser**.

Output Parser:

An output parser is a component that helps structure language model responses. It is a class that implements two main methods: `get_format_instructions` and `parse`. The `get_format_instructions` method returns a string containing instructions for how the output of a language model should be formatted. The `parse` method takes in a string (assumed to be the response from a language model) and parses it into some structure, such as a dictionary, a list, or a custom object.

This chain takes multiple input variables, uses the PromptTemplate to format them into a prompt, passes it to the model, and then uses the OutputParser (if provided) to parse the output of the LLM into a final format

```
from langchain import PromptTemplate
```

```
from langchain import OpenAI, LLMChain
```

- **RouterChain**. This is a type of chain that allows you to route the input variables to different chains based on some conditions. You can specify the conditions as functions or expressions that return a boolean value. You can also specify the default chain to use if none of the conditions are met.
- For example, you can use this chain to create a chatbot that can handle different types of requests, such as booking a restaurant reservation or planning an itinerary. To achieve this goal, you might want to differentiate two different prompts, depending on the type of query the user will make:

Thanks to the RouterChain, we can build a chain which is able to activate a different prompt depending on the user's query.

- **SequentialChain.** This is a type of chain that allows you to execute multiple chains in a sequence. You can specify the order of the chains and how they pass their outputs to the next chain. The simplest module of sequential chain, namely, takes by default the output of one chain as the input of the next chain, however you can also use a more complex module to have more flexibility to set input and output among chains.

```
from langchain.llms import OpenAI

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate
```

- **TransformationChain.** This is a type of chain that allows you to transform the input variables or the output of another chain using some functions or expressions. You can specify the transformation as a function that takes the input or output as an argument and returns a new value, as well as specifying the output format of the chain.
- For example, let's say we want to summarize a text, but before that we want to rename one of the protagonist of the story (a cat) as "Silvester the Cat". As a sample text, I asked Bing Chat to generate a story about Cats and Dogs

```
from langchain.chains import
TransformChain,LLMChain,SimpleSequentialChain

from langchain.llms import OpenAI

from langchain.prompts import PromptTemplate
```

Overall, Langchain chains are a powerful way to combine different language models and tasks into a single workflow. Chains are flexible, scalable, and easy to use, and they enable users to leverage the power of language models for various purposes and domains.

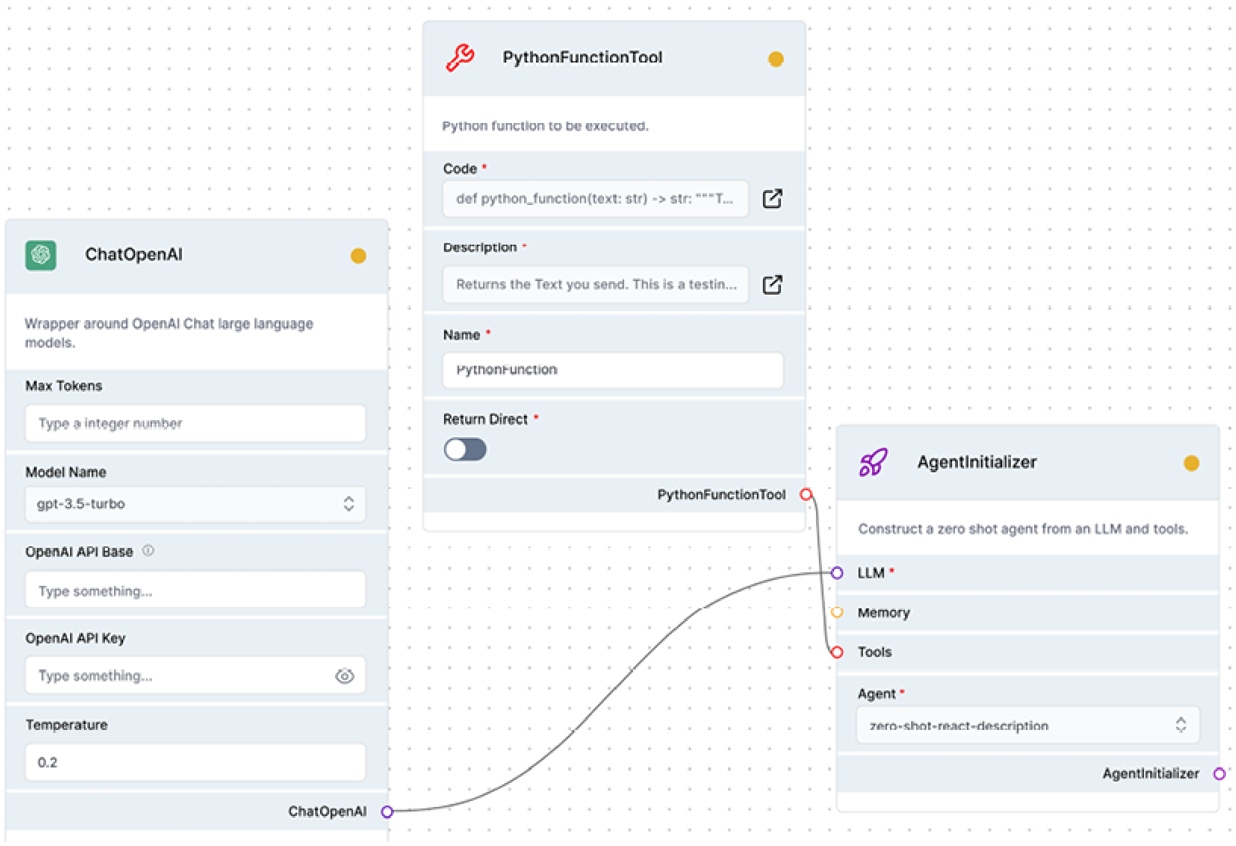
5. **Agents:**

   Agents are a key concept in LangChain for creating systems that interact dynamically with users and environments over time. An agent is an autonomous software entity that is capable of taking actions to accomplish goals and tasks.

Agents combine and orchestrate chains. The agent observes the environment, decides which chain to execute based on that observation, takes the chain’s specified action, and repeats.

Agents decide which actions to take using LLMs as reasoning engines. The LLM is prompted with available tools, user input, and previous steps. It then selects the next action or final response.

for calculations, a simple calculator outperforms a model consisting of billions of parameters. In this case, an agent can decide to pass the calculation to a calculator or to a Python interpreter. We can see a simple app here, where an agent is connected to both an OpenAI model and a Python function:



PythonFunctionTool
Python function to be executed.
Code *: def python_function(text: str) -> str: """T...
Description *: Returns the Text you send. This is a testin...
Name *: PythonFunction
Return Direct *

ChatOpenAI
Wrapper around OpenAI Chat large language models.
Max Tokens
Model Name: gpt-3.5-turbo
OpenAI API Base
OpenAI API Key
Temperature: 0.2

AgentInitializer
Construct a zero shot agent from an LLM and tools.
Agent *: zero-shot-react-description

Figure 2.10: A simple LLM app with a Python function visualized in LangFlow

Based on the input, the agent can decide to run a Python function. Each agent also decides which tool to use and when. We'll look more at the mechanics of how this works in *Chapter 4*, *Building Capable Assistants*.

A key limitation of agents and chains is their statelessness – each execution occurs in isolation without retaining prior context. This is where the concept of memory becomes critical. Memory in LangChain refers to persisting information across chain executions to enable statefulness.

Agents are entities that drive decision-making within LLMs-powered applications. They have access to a suite of tools and can decide which tool to call based on the user input and the context. Agents are dynamic and adaptive, meaning that they can change or adjust their actions based on the situation or the goal.

A core concept while talking about agents is that of tools. In fact, an agent might be good at planning all the right actions to fulfill user’s query, but what if it cannot actually executing them, since it is missing information or executive power?

For example, imagine I want to build an agent which is capable of answering my questions searching in the web. By itself, the agent has no access to the Web, so I need to provide it with this tool. I will do so by using the SerpApi (the Google Search API) integration provided by LangChain

```
from langchain import SerpAPIWrapper

from langchain.agents import AgentType, initialize_agent

from langchain.llms import OpenAI

from langchain.tools import BaseTool, StructuredTool, Tool, tool

import os

from dotenv import load_dotenv

load_dotenv()

os.environ["SERPAPI_API_KEY"]
```

while initializing agent, I set agent type as ZERO_SHOT_REACT_DESCRIPTION. This is one of the configurations we can pick and, specifically, it configures the agent to decide which tool to pick based solely on the tool’s description with a ReAct approach.

ReAct approach:

The ReAct approach is a way of using large language models (LLMs) to solve various language reasoning and decision making tasks. It was introduced in the paper “ReAct: Synergizing Reasoning and Acting in Language Models” by Shunyu Yao et al., back to October 2022.

The ReAct approach prompts LLMs to generate both verbal reasoning traces and text actions in an interleaved manner, allowing for greater synergy between the two. Reasoning traces help the model to plan, track, and update its actions, as well as handle exceptions. Actions allow the model to interact with external sources, such as knowledge bases or environments, to gather additional information.

On top of this configuration, LangChain offers also the following types of agents:

- **Structured input ReAct**. This is an agent type that uses the ReAct framework to generate natural language responses based on structured input data. The agent can handle different types of input data, such as tables, lists, or key-value pairs. The agent uses a language model and a prompt to generate responses that are informative, concise, and coherent.
- **OpenAI Functions.** This is an agent type that uses the OpenAI Functions API to access various language models and tools from OpenAI. The agent can use different functions, such as GPT-3, Codex, DALL-E, CLIP, or ImageGPT. The agent uses a language model and a prompt to generate requests to the OpenAI Functions API and parse the responses.
- **Conversational**. This is an agent type that uses a language model to engage in natural language conversations with the user. The agent can handle different types of conversational tasks, such as chit-chat, question answering, or task completion. The agent uses a language model and a prompt to generate responses that are relevant, fluent, and engaging.
- **Self ask with search**. This is an agent type that uses a language model to generate questions for itself and then search for answers on the web. The agent can use this technique to learn new information or test its own knowledge.
- **ReAct document store**. This is an agent type that uses the ReAct framework to generate natural language responses based on documents stored in a database.

The agent can handle different types of documents, such as news articles, blog posts, or research papers.

- **Plan-and-execute agents**. This is an experimental agent type that uses a language model to choose a sequence of actions to take based on the user's input and a goal. The agent can use different tools or models to execute the actions it chooses. The agent uses a language model and a prompt to generate plans and actions, and then uses an `AgentExecutor` to run them.

LangChain agents are pivotal whenever you want to let your LLMs interact with the external world. Plus, it is interesting to see how agents leverage LLMs not only to retrieve and generate responses, but also as reasoning engines to plan an optimized sequence of actions.Together with all the LangChain components covered in this paragraph, agents can be the core of LLM-powered applications

Tools:

Tools provide modular interfaces for agents to integrate external services like databases and APIs. LangChain offers tools like document loaders, indexes, and vector stores, which facilitate the retrieval and storage of data for augmenting data retrieval in LLMs. There are many tools available, and here are just a few examples:

- **Machine translator**: A language model can use a machine translator to better comprehend and process text in multiple languages. This tool enables non-translation-dedicated language models to understand and answer questions in different languages.
- **Calculator**: Language models can utilize a simple calculator tool to solve math problems. The calculator supports basic arithmetic operations, allowing the model to accurately solve mathematical queries in datasets specifically designed for math problem-solving.
- **Maps**: By connecting with the Bing Map API or similar services, language models can retrieve location information, assist with route planning, provide driving distance calculations, and offer details about nearby points of interest.

- **Weather**: Weather APIs provide language models with real-time weather information for cities worldwide. Models can answer queries about current weather conditions or forecast the weather for specific locations within varying time periods.
- **Stocks**: Connecting with stock market APIs like Alpha Vantage allows language models to query specific stock market information such as opening and closing prices, highest and lowest prices, and more.
- **Slides**: Language models equipped with slide-making tools can create slides using high-level semantics provided by APIs such as the `python-pptx` library or image retrieval from the internet based on given topics. These tools facilitate tasks related to slide creation that are required in various professional fields.
- **Table processing**: APIs built with pandas DataFrames enable language models to perform data analysis and visualization tasks on tables. By connecting to these tools, models can provide users with a more streamlined and natural experience for handling tabular data.
- **Knowledge graphs**: Language models can query knowledge graphs using APIs that mimic human querying processes, such as finding candidate entities or relations, sending SPARQL queries, and retrieving results. These tools assist in answering questions based on factual knowledge stored in knowledge graphs.
- **Search engine**: By utilizing search engine APIs like Bing Search, language models can interact with search engines to extract information and provide answers to real-time queries. These tools enhance the model's ability to gather information from the web and deliver accurate responses.
- **Wikipedia**: Language models equipped with Wikipedia search tools can search for specific entities on Wikipedia pages, look up keywords within a page, or disambiguate entities with similar names. These tools facilitate question-answering tasks using content retrieved from Wikipedia.
- **Online shopping**: Connecting language models with online shopping tools allows them to perform actions like searching for items, loading detailed information about products, selecting item features, going through shopping pages, and making purchase decisions based on specific user instructions.

Additional tools include AI Painting, which allows language models to generate images using AI image generation models; 3D Model Construction, enabling language models to create 3D models using a sophisticated 3D rendering engine;

LLMs via Hugging Face Hub:

If you want to use open-source LLMs, leveraging the Hugging Face Hub integration is extremely versatile. In fact, with just one access token you can leverage all the open-source LLMs available in Hugging Face's repos. As it being a non-production scenario, I will be using the free Inference API, however if you are meant to build production-ready applications, you can easily scale to the Inference Endpoint, which grants you a dedicated and fully managed infrastructure to host and consume your LLMs.

The nice thing about the Hugging Face Hub integration is that you can navigate on its portal and decide, within the model catalog, what to use. Models are also clustered per category (Computer Vision, Natural Language Processing, Audio…) and, within each category, per capabilities (within NLP, we have summarization, classification, Q&A…).

```
from langchain import HuggingFaceHub
```

How does LangChain work?

The LangChain framework simplifies building sophisticated LLM applications by providing modular components that facilitate connecting language models with other data and services. The framework organizes capabilities into modules spanning from basic LLM interaction to complex reasoning and persistence.

These components can be combined into pipelines also called chains that sequence the following actions:

- Loading documents

- Embedding for retrieval
- Querying LLMs
- Parsing outputs
- Writing memory

Chains match modules to application goals, while agents leverage chains for goal-directed interactions with users. They repeatedly execute actions based on observations, plan optimal logic chains, and persist memory across conversations.

The modules, ranging from simple to advanced, are:

- **LLMs and chat models**: Provide interfaces to connect and query language models like GPT-3. Support async, streaming, and batch requests.
- **Document loaders**: Ingest data from sources into documents with text and metadata. Enable loading files, webpages, videos, etc.
- **Document transformers**: Manipulate documents via splitting, combining, filtering, translating, etc. Help adapt data for models.
- **Text embeddings**: Create vector representations of text for semantic search. Different methods for embedding documents vs. queries.
- **Vector stores**: Store embedded document vectors for efficient similarity search and retrieval.
- **Retrievers**: General interface to return documents based on a query. Can leverage vector stores.
- **Tools**: Interfaces that agents use to interact with external systems.
- **Agents**: Goal-driven systems that use LLMs to plan actions based on environment observations.
- **Toolkits**: Initialize groups of tools that share resources like databases.
- **Memory**: Persist information across conversations and workflows by reading/writing session data.
- **Callbacks**: Hook into pipeline stages for logging, monitoring, streaming, and others. Callbacks enable monitoring chains.

LangChain offers interfaces to connect with and query LLMs like GPT-3 and chat models. This provides a flexible API for integrating different language models.

Although LangChain doesn't supply models itself, it supports integration through LLM wrappers with various language model providers, enabling the app to interact with chat models as well as text embedding model providers. Supported providers include OpenAI, HuggingFace, Azure, and Anthropic.

A core building block of LangChain is the prompt class, which allows users to interact with LLMs by providing concise instructions or examples. Prompt engineering helps optimize prompts for optimal model performance. Templates give flexibility in terms of input and the available collection of prompts is battle-tested in a range of applications..

Document loaders allow ingesting data from various sources into documents containing text and metadata. This data can then be manipulated via document transformers – splitting, combining, filtering, translating, etc. These tools adapt external data for use in LLMs.

Data loaders include modules for storing data and utilities for interacting with external systems, like web searches or databases, and most importantly data retrieval. Examples are Microsoft Word documents (`.docx`), **HyperText Markup Language** (**HTML**), and other common formats such as PDF, text files, JSON, and CSV.

Text embedding models create vector representations of text that capture semantic meaning. This enables semantic search by finding text with the most similar vector representations. Vector stores build on this by indexing embedded document vectors for efficient similarity-based retrieval.

Vector stores come in when working with large documents, where the document needs to be chunked up in order to be passed to the LLM. These parts of the document would be stored as embeddings, which means that they are vector representations of the information. All these tools enhance the LLMs' knowledge and improve their performance in applications like question answering and summarization.

There are numerous integrations for vector storage. These include Alibaba Cloud OpenSearch, AnalyticDB for PostgreSQL, Meta AI's Annoy library for **Approximate**

Nearest Neighbor (**ANN**) search, Cassandra, Chroma, Elasticsearch, **Facebook AI Similarity Search** (**Faiss**), MongoDB Atlas Vector Search, PGVector as a vector similarity search for Postgres, Pinecone, scikit-learn (`SKLearnVectorStore` for k-nearest neighbor search), and many more.

Comparing LangChain with other frameworks:

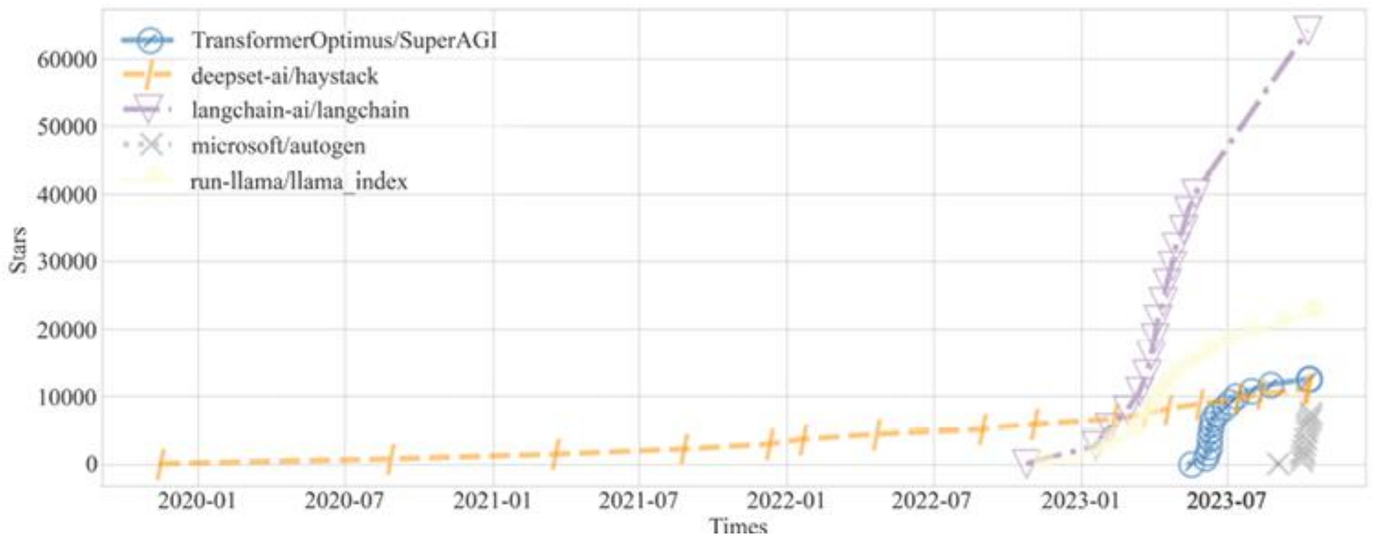


Figure 2: Comparison of popularity between different frameworks in Python

Conversational applications:

A conversational application is a type of software that can interact with users using natural language. It can be used for various purposes, such as providing information, assistance, entertainment, or transactions. Generally speaking, a conversational application can use different modes of communication, such as text, voice, graphics, or even touch. A conversational application can also use different platforms, such as messaging apps, websites, mobile devices, or smart speakers.

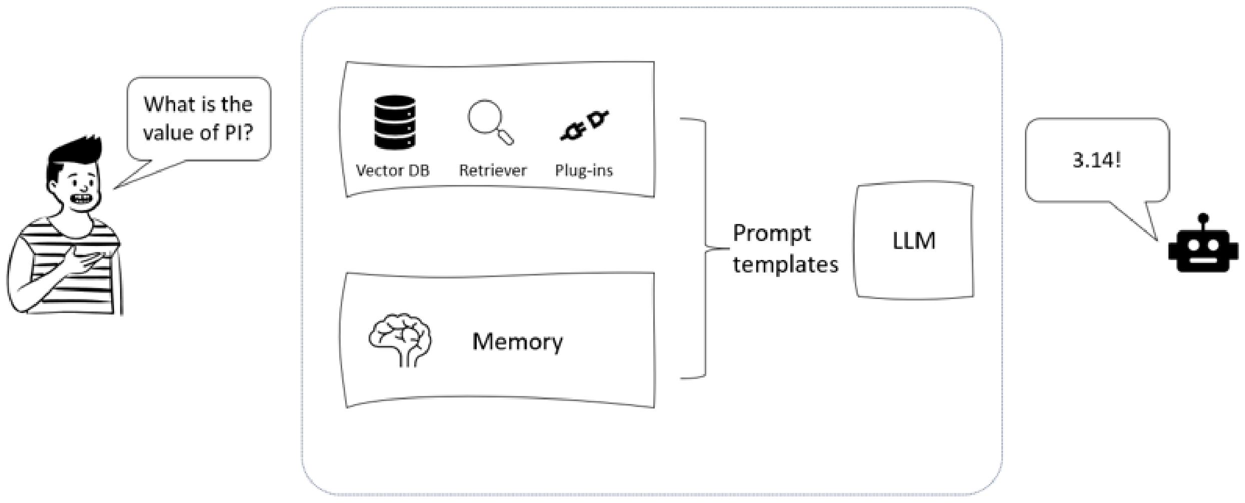


Fig: Sample architecture of a conversational bot

a. Plain Vanilla bot:

Set the schema for bot. The schema refers to the type of messages the bot is able to receive.

- **System message**the instructions we give the bot so that it behave as a travel assistant.
- **AI Message**the message generated by the LLM
- **Human Message**the user’s query

```
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage
)
from langchain.chains import LLMChain, ConversationChain
from langchain.chat_models import ChatOpenAI
from langchain.chat_models import ChatOpenAI
chat = ChatOpenAI()
messages = [
    SystemMessage(content="You are a helpful assistant that help the user to plan an optimized
itinerary."),
HumanMessage(content="I'm going to Rome for 2 days, what can I visit?")
```

```
output = chat(messages) print(output.content)
```

b. Adding Memory to bot:

Initializing our memory and chain using `ConversationBufferMemory and ConversationChain` to combine the LLM and the memory components.

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain
memory = ConversationBufferMemory()
conversation = ConversationChain(
    llm=chat, verbose=True, memory=memory
)

conversation.run("Hi there!")
conversation.run("what is the most iconic place in Rome?")
conversation.run("What kind of other events?")
```

The bot was pretty able to understand that our request was related to its previous answer. We can also retrieve the messages history with the memory.load_memory_variables() method

c. Adding non-parametric knowledge:

Imagine that you also want your GlobeBotter to have access to exclusive documentations about itineraries, that are not part of its parametric knowledge. To do so, we can either embed the documentation in a VectorDB, or directly use a retriever to do the job. In this case, we will use a vector store-backed retriever using the a particular chain, `ConversationalRetrievalChain.` This type of chain leverage a retriever over the provided knowledge base keeping the chat history, which can be passed as a parameter using the desired type of memory previously seen.
With this goal in mind, we will use a sample Italy travel guide PDF downloaded from https://www.minube.net/guides/italy. The following Python code shows how to initialize all the ingredients we need, that are:

- **Document Loader**since the document is in PDF format, we will use `PyPDFLoader`.
- **Text splitter**we will use a `RecursiveCharacterTextSplitter`, which splits text by recursively look at characters to find one that works.
- **Vector store**we will use the `FAISS` vectorDB.
- **Memory**we will use a `ConversationBufferMemory`.

- **LLMs**we will use the `gpt-3.5-turbo` model for conversations.
- **Embeddings**we will use the `text-embedding-ada-002`.

```
from langchain.llms import OpenAI from langchain.chat_models
import ChatOpenAI from langchain.embeddings.openai import
OpenAIEmbeddings from langchain.text_splitter import
RecursiveCharacterTextSplitter from langchain.vectorstores
import FAISS from langchain.document_loaders import
PyPDFLoader from langchain.chains import
ConversationalRetrievalChain from langchain.memory import
ConversationBufferMemory
```

- `create_retriever_tool`this method create a custom tool which act as a retriever for an agent.
- `create_conversational_retrieval_agent`this method initialize a conversational agent which is configured to work with retrievers and chat models.

```
from langchain.agents.agent_toolkits import
create_retriever_tool
```

d. Adding external tools:

The tool we are going to add here is the Google SerpAPI tool, so that our bot will be able to navigate the internet. Since we don't want our GlobeBotter to be focused only to the web, we will add the SerpAPI tool to the previous one, so that the agent will be able to pick the most useful tool to answer the question – or rather using no tool since not necessary.

```
from langchain import SerpAPIWrapper
```

e. `Front-end with Streamlit:`

Streamlit is a Python library that allows you to create and share web apps.
You can write your app in pure Python, using simple commands to add widgets, charts, tables, and other elements.
In addition to its native capabilities, in July 2023 Streamlit announced an initial integration and future plans with LangChains. At the core of this initial integration there is the ambition of making it easier to build GUI for conversational applications, as well as showing all the streps LangChain's agents take before producing the final response.
To achieve this goal, the main module that Streamlit introduced is the Streamlit callback handler. This module provides a class
called `StreamlitCallbackHandler` that implements

the `BaseCallbackHandler` interface from LangChain. This class can handle various events that occur during the execution of a LangChain pipeline, such as tool start, tool end, tool error, LLM token, agent action, agent finish, etc. The class can also create and update Streamlit elements such as containers, expanders, text, progress bars, etc. to display the output of the pipeline in a user-friendly way.

You can use the streamlit callback handler to create Streamlit apps that showcase the capabilities of LangChain and interact with the user through natural language. For example, you can create an app that takes a user prompt and runs it through an agent that uses different tools and models to generate a response. You can use the streamlit callback handler to show the agent's thought process and the results of each tool in real time. To start building your application, you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `globebotter.py`.Those are the main building blocks of the application:

```
import streamlit as st
```

Recommendation Systems:

A recommendation system is a computer program that recommends items for users of digital platforms such as e-commerce websites and social networks. It uses large data sets to develop models of users' likes and interests, and then recommends similar or recommended items to individual users.There are different types of recommendation systems, depending on the methods and data they use. Some of the common types are:

- **Collaborative filtering**: This type of recommendation system uses the ratings or feedback of other users who have similar preferences to the target user. It assumes that users who liked similar items in the past will like similar items in the future. For example, if user A and user B both liked movies X and Y, then the algorithm may recommend movie Z to user A if user B also liked it.
- Collaborative filtering can be further divided into two subtypes: user-based and item-based:

- **User-based collaborative filtering** finds similar users to the target user and recommends items that they liked.

- **Item-based collaborative filtering** finds similar items to the ones that the target user liked and recommends them
- **Content-based filtering**: This type of recommendation system uses the features or attributes of the items themselves to recommend items that are similar to the ones that the target user has liked or interacted with before. It assumes that users who liked certain features of an item will like other items with similar features. For example, if user A liked movie X, which is a comedy with actor Y, then the algorithm may recommend movie Z, which is also a comedy with actor Y.
- **Hybrid filtering**: This type of recommendation system combines both collaborative and content-based filtering methods to overcome some of their limitations and provide more accurate and diverse recommendations. For example, YouTube uses hybrid filtering to recommend videos based on both the ratings and views of other users who have watched similar videos, and the features and categories of the videos themselves.
- **Knowledge-based filtering**: This type of recommendation system uses explicit knowledge or rules about the domain and the user's needs or preferences to recommend items that satisfy certain criteria or constraints. It does not rely on ratings or feedback from other users, but rather on the user's input or query. For example, if user A wants to buy a laptop with certain specifications and budget, then the algorithm may recommend a laptop that satisfies those criteria. Knowledge-based recommender systems work well when there is no or little rating history available, or when the items are complex and customizable.

Modern recommender systems:

Modern recommendation systems uses Machine Learning (ML) techniques to make better predictions about user's preferences, based on the available data. some of the most popular ML rechniques are K-nearest neighbors, dimensionality reduction and Neural networks used for recommendation systems.

1. KNN:

- K-nearest neighbor (KNN) is a machine learning algorithm that can be used for both classification and regression problems. It works by finding the k closest data points to a new data point and using their labels or values to make a prediction.

- KNN assumption is that the similar data points are likely to have similar labels or values.
- KNN can be applied to recommendation systems in the context of collaborative filtering, both user-based and content-based:User-based.

- KNN is a type of collaborative filtering, which uses the ratings or feedback of other users who have similar tastes or preferences to the target user.

- For example, if user A and user B both liked movies X and Y, then the algorithm may recommend movie Z to user A if user B also liked it. User-based KNN works by finding the k most similar users to the target user based on their ratings, and then taking the weighted average of their ratings for the items that the target user has not rated yet.
- Item-based KNN is another type of collaborative filtering, which uses the attributes or features of the items to recommend similar items to the target user. For example, if user A liked movie X, which is a comedy with actor Y, then the algorithm may recommend movie Z, which is also a comedy with actor Y. Item-based KNN works by finding the k most similar items to the items that the target user has rated, and then taking the weighted average of their ratings for the items that the target user has not rated yet.

K-Nearest Neighbors (KNN) is a popular technique in recommendation systems, but it has some pitfalls:

- **Scalability**. KNN can become computationally expensive and slow when dealing with large datasets, as it requires calculating distances between all pairs of items or users.
- **Cold Start Problem.** KNN struggles with new items or users that have limited or no interaction history, as it relies on finding neighbors based on historical data.
- **Data Sparsity**. KNN performance can degrade in sparse datasets where there are many missing values, making it challenging to find meaningful neighbors.
- **Feature Relevance.** KNN treats all features equally and assumes that all features contribute equally to similarity calculations. This may not hold true in scenarios where some features are more relevant than others.

- **Choice of K.** Selecting the appropriate value of K (number of neighbors) can be subjective and impact the quality of recommendations. A small K may result in noise, while a large K may lead to overly broad recommendations.

To address these pitfalls, techniques like distance weighting, dimensionality reduction, and hybrid methods (combining KNN with other recommendation techniques) have been explored. Additionally, further techniques are widely used in the file of recommendation systems, such as matrix factorization.

Matrix factorization:
Matrix factorization is a technique used in recommendation systems to analyze and predict user preferences or behaviors based on historical data. It involves decomposing a large matrix into two or more smaller matrices to uncover latent features that contribute to the observed data patterns and address the so-called"curse of dimensionality".

Curse of dimensionality:

The curse of dimensionality refers to challenges that arise when dealing with high-dimensional data. It leads to increased complexity, sparse data, and difficulties in analysis and modeling due to the exponential growth of data requirements and potential overfitting.

In the context of recommendation systems, this technique is employed to predict missing values in the user-item interaction matrix, which represents users' interactions with various items (such as movies, products, or books).

Let's consider the following example. Imagine you have a matrix where rows represent users, columns represent movies, and the cells contain ratings (from 1 as lowest to 5 as highest). However, not all users have rated all movies, resulting in a matrix with many missing entries:

	Movie 1	Movie 2	Movie 3	Movie 4
User 1	4	-	5	-
User 2	-	3	-	2

User 3	5	4	-	3

Matrix factorization aims to break down this matrix into two matrices: one for users and another for movies, with a reduced number of dimensions (latent factors). These latent factors could represent attributes like genre preferences or specific movie characteristics. By multiplying these matrices, you can predict the missing ratings and recommend movies that the users might enjoy.There are different algorithms for matrix factorization, among which:

- **Singular Value Decomposition (SVD).** It decomposes a matrix into three separate matrices, where the middle matrix contains singular values that represent the importance of different components in the data. It's widely used in data compression, dimensionality reduction, and collaborative filtering in recommendation systems.
- **Principal Component Analysis (PCA).** PCA is a technique to reduce the dimensionality of data by transforming it into a new coordinate system aligned with the principal components. These components capture the most significant variability in the data, allowing for efficient analysis and visualization.
- **Non-Negative Matrix Factorization (NMF).** NMF decomposes a matrix into two matrices with non-negative values. It's often used for topic modeling, image processing, and feature extraction, where the components represent non-negative attributes.

In the context of recommendation system, probably the most popular technique is SVD, so let's use this one to go on with our example. We will use the Python `numpy` module to apply SVD as follows:

```
import numpy as np
# Your user-movie rating matrix (replace with your actual data)
user_movie_matrix = np.array([
    [4, 0, 5, 0],
    [0, 3, 0, 2],
    [5, 4, 0, 3]
])
# Apply SVD
U, s, V = np.linalg.svd(user_movie_matrix, full_matrices=False)
# Number of latent factors (you can choose this based on your
preference)
num_latent_factors = 2
# Reconstruct the original matrix using the selected latent factors
reconstructed_matrix = U[:, :num_latent_factors] @
np.diag(s[:num_latent_factors]) @ V[:num_latent_factors, :]
# Replace negative values with 0
reconstructed_matrix = np.maximum(reconstructed_matrix, 0)
print("Reconstructed Matrix:")
print(reconstructed_matrix)
```
CopyExplain

Output:

```
Reconstructed Matrix:
[[4.2972542  0.         4.71897811 0.        ]
 [1.08572801 2.27604748 0.         1.64449028]
 [4.44777253 4.36821972 0.52207171 3.18082082]]
```
CopyExplain

In this example, the `U` matrix contains user-related information, the `s` matrix contains the singular values, and the `V` matrix contains movie-related information. By selecting a certain number of latent factors (`num_latent_factors`), you can reconstruct the original matrix with reduced dimensions, while setting the `full_matrices=False` parameter in the `np.linalg.svd` function ensures that the decomposed matrices are truncated to have dimensions consistent with the selected number of latent factors.These predicted ratings can then be used to recommend movies with higher predicted ratings to users. Matrix factorization enables recommendation systems to uncover hidden patterns in user preferences and make personalized recommendations based on those patterns.Matrix factorization has been a widely used technique in recommendation systems, but it has some pitfalls (some similar to the KNN's technique):

- **Cold Start Problem**. Similar to KNN, matrix factorization struggles with new items or users that have limited or no interaction history. Since it relies on historical data, it can't effectively provide recommendations for new items or users.
- **Data Sparsity**. As the number of users and items grows, the user-item interaction matrix becomes increasingly sparse, leading to challenges in accurately predicting missing values.
- **Scalability**. For large datasets, performing matrix factorization can be computationally expensive and time-consuming.
- **Limited Context**. Matrix factorization typically only considers user-item interactions, ignoring contextual information like time, location, or additional user attributes.

Neural Networks:

Neural networks are used in recommendation systems to improve the accuracy and personalization of recommendations by learning intricate patterns from data. Here's how neural networks are commonly applied in this context:

- **Collaborative Filtering with Neural Networks**. Neural networks can model user-item interactions by embedding users and items into continuous vector spaces. These embeddings capture latent features that represent user preferences and item characteristics. Neural collaborative filtering models combine these embeddings with neural network architectures to predict ratings or interactions between users and items.

- **Content-Based Recommendations**. In content-based recommendation, neural networks can learn representations of item content, such as text, images, or audio. These representations capture item characteristics and user preferences. Neural networks like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are used to process and learn from item content, enabling personalized content-based recommendations.
- **Sequential Models**. In scenarios where user interactions have a temporal sequence, such as clickstreams or browsing history, recurrent neural networks (RNNs) or variants like Long Short-Term Memory (LSTM) networks can capture temporal dependencies in the user behavior and make sequential recommendations.
- **Autoencoders and Variational Autoencoders (VAEs).** Autoencoders and VAEs can be used to learn low-dimensional representations of users and items.

Autoencoders and Variational Autoencoders:

Autoencoders are a type of neural network architecture used for unsupervised learning and dimensionality reduction. They consist of an encoder and a decoder. The encoder maps the input data into a lower-dimensional latent space representation, while the decoder attempts to reconstruct the original input data from the encoded representation.

Variational Autoencoders are an extension of traditional autoencoders that introduce probabilistic elements. VAEs not only learn to encode the input data into a latent space but also model the distribution of this latent space using probabilistic methods. This allows for the generation of new data samples from the learned latent space. VAEs are used for generative tasks like image synthesis, anomaly detection, and data imputation.

In both autoencoders and variational autoencoders, the idea is to learn a compressed and meaningful representation of the input data in the latent space, which can be useful for various tasks including feature extraction, data generation, and dimensionality reduction.

- These representations can then be used to make recommendations by identifying similar users and items in the latent space.

- **Side Information Integration**. Neural networks can incorporate additional user and item attributes, such as demographic information, location, or social connections, to improve recommendations by learning from diverse data sources.
- **Deep Reinforcement Learning.** In certain scenarios, deep reinforcement learning can be used to optimize recommendations over time, learning from user feedback to suggest actions that maximize long-term rewards.

Neural networks pitfalls:
first one being their being task-specific. For example, a rating-prediction recommendation system will not be able to tackle a task where we need to recommend the top k items that are likely matching user's taste. Actually, if we extend this limitation to other "pre-LLMs" AI solutions, we might see some similarities: it is indeed the task-specific situation that Large Languare Models and, more generally, Large Foundation Models are revolutionizing, being highly generalized and adaptable to various tasks, depending on user's prompts and instructions.Henceforth, several researches in the field of recommendation systems are experimenting to what extent LLMs can enhance the current models.

How LLMs are changing recommendation systems:

We saw in previous chapters how LLMs can be customized in three main ways: pre-training, fine-tuning and prompting. According to to the paper "Recommender systems in the Era of Large Language Models (LLMs)" from Wenqi Fan et al., these are also the ways you can tailor an LLM to be a recommender system.

- **Pre-training**. Pre-training LLMs for recommender systems is an important step to enable LLMs to acquire extensive world knowledge and user preferences, and to adapt to different recommendation tasks with zero or few shots.

Note

An example of a recommendation system LLM is P5, introduced by Shijie Gang et al. in their paper "Recommendation as Language Processing (RLP).

P5 is a unified text-to-text paradigm for building recommender systems using large language models (LLMs). It stands for Pretrain, Personalized Prompt & Predict Paradigm and consists of three steps:

Pretrain. A foundation language model based on T5 architecture is pretrained on a large-scale web corpus and fine-tuned on recommendation tasks.

Personalized Prompt. A personalized prompt is generated for each user based on their behavior data and contextual features.

Predict. The personalized prompt is fed into the pretrained language model to generate recommendations.

P5 is based on the idea that LLMs can encode extensive world knowledge and user preferences, and can be adapted to different recommendation tasks with zero or few shots.

- **Fine-tuning.** Training an LLM from scratch is a highly computational-intensive activity. An alternative and less intrusive approach to customize an LLM for recommendation systems might be fine-tuning.

More specifically, the authors of the paper review two main strategies for fine-tuning LLMs:

- **Full-model fine-tuning**it involves changing the entire model weights based on task-specific recommendation datasets.
- **Parameter-efficient fine-tuning**it aims to change only a small part of weights or develop trainable adapters to fit specific tasks.
- **Prompting**. The third and “lightest” way of tailoring LLMs to be recommender systems is prompting. According to the authors, there are three main techniques for prompting LLMs:
- **Conventional prompting**it aims to unify downstream tasks into language generation tasks by designing text templates or providing a few input-output examples.
- **In-context learning**it enables LLMs to learn new tasks based on contextual information without fine-tuning.

- **Chain-of-thought**it enhances the reasoning abilities of LLMs by providing multiple demonstrations to describe the chain of thought as examples within the prompt. The authors also discuss the advantages and challenges of each technique, and provide some examples of existing methods that adopt them.
- Regardless of the typology, prompting is the fastest way to test whether an general-purpose LLM can tackle recommendations systems' tasks.

we are going to implement our own recommendation application using the prompting approach and leveraging the capabilities of LangChain as AI orchestrator.

Implementing an LLM-powered recommendation system:

let's start building our recommendation app, which will be a movie recommender system called MovieHarbor. The goal will be to make it as general as possible, meaning that we want our app to be able to address various recommendations tasks with a conversational interface. The scenario we are going to simulate will be that of the so called "cold-start", that means the first interaction of an user with the recommendation system, so that we do not have user's preference history.

```
import pandas as pd

import tiktoken

import os

import openai

openai.api_key = os.environ["OPENAI_API_KEY"]

from openai.embeddings_utils import get_embedding

embedding_encoding = "cl100k_base" #this the encoding for text-embedding-ada-002

max_tokens = 8000 # the maximum for text-embedding-ada-002 is 8191
encoding = tiktoken.get_encoding(embedding_encoding)
```

Cl100k_base:

Cl100k_base is the name of a tokenizer that is used by OpenAI’s embeddings API. A tokenizer is a tool that splits a text string into smaller units called tokens, which can then be processed by a neural network. Different tokenizers have different rules and vocabularies for how to split the text and what tokens to use.

The cl100k_base tokenizer is based on the Byte Pair Encoding (BPE) algorithm, which learns a vocabulary of subword units from a large corpus of text. The cl100k_base tokenizer has a vocabulary of 100,000 tokens, which are mostly common words and word pieces, but also include some special tokens for punctuation, formatting, and control. The cl100k_base tokenizer can handle texts in multiple languages and domains, and can encode up to 8,191 tokens per input.

Great, now that we have our final dataset, we need to store it into a VectorDB. For this purpose, we are going to leverage **LanceDB,** an open-source database for vector-search built with persistent storage, which greatly simplifies retrieval, filtering and management of embeddings, which also offers a native integration with LangChain. You can easily install LanceDB via `pip install lancedb`.

```
import lancedb
```

Cold Start:

In the cold start scenario where the system knew nothing about the user

the cold start scenario – that means, interacting with an user for the first time without its backstory- often represents a problem to face for recommendation systems. By definition, the less information we have about a user, the harder is it to match its preferences.In this section, we are going to simulate a cold start scenario with LangChain and OpenAI’s LLMs with the following high-level architecture:

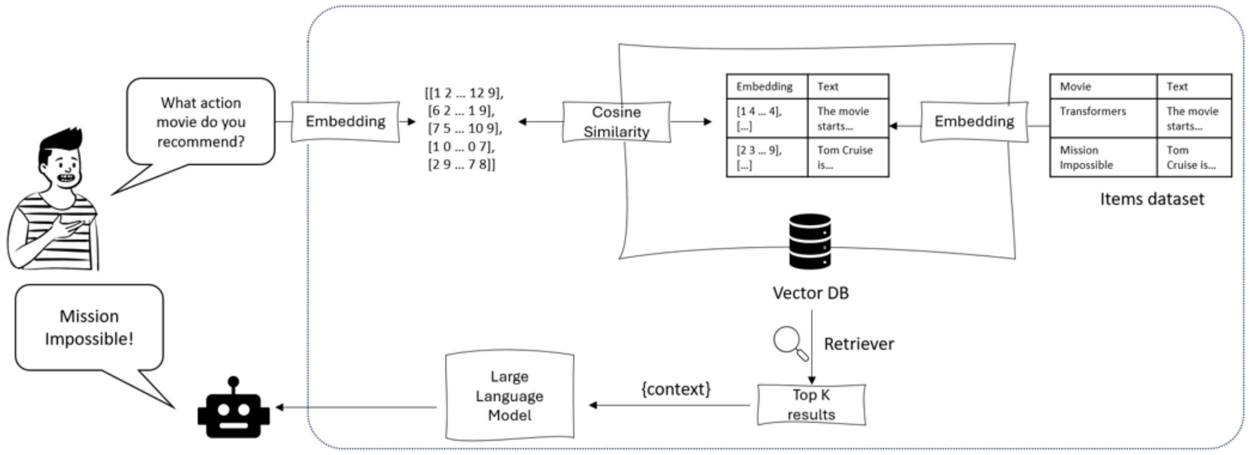


Figure 2: High-level architecture of recommendation system in a cold start scenario.
In the previous section, we've already saved our embeddings in LanceDB. Now, we are going to build a LangChain RetrievalQA using the vector store as retriever, so that it returns the top k most similar movies upon user's query, using cosine similarity as distance metric (which is the default).

So let's start building the chain using only the movie overview as information input:

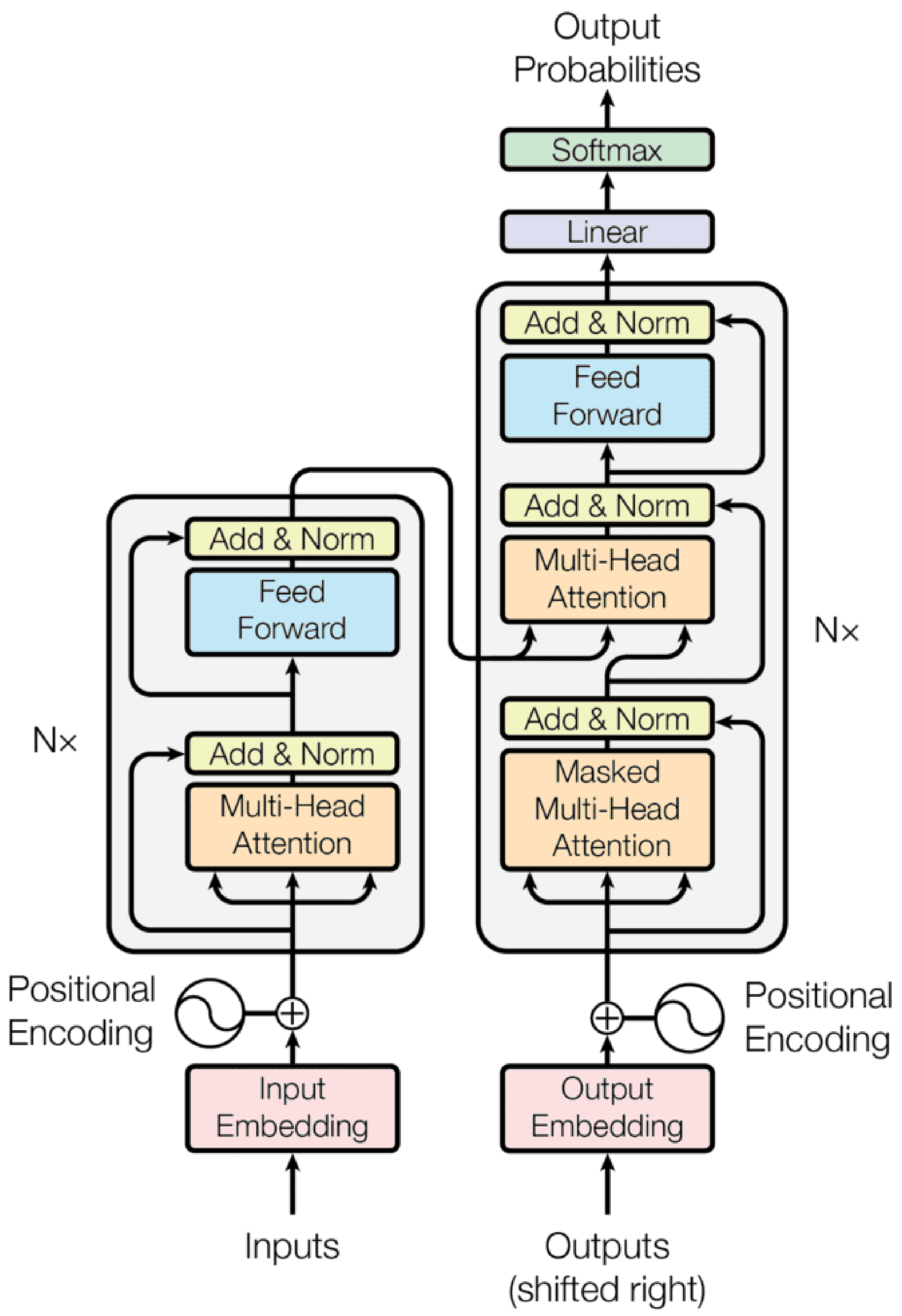
```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import LanceDB
os.environ["OPENAI_API_KEY"]
embeddings = OpenAIEmbeddings()
docsearch = LanceDB(connection = table, embedding = embeddings)
```

One of the advantages of using LangChain's pre-built components, such as the RetrievalQA chain, is that they come with a pre-configured, well-curated prompt template. Before overriding the existing prompt, it's a good practice to inspect it, so that you can also see which are the variables (within {}) that are already expectd from the component.

Front-end with Streamlit:

Now that we have seen the logic behind an LLM-powered recommendation system, it is now time to give a GUI to our MovieHarbor. To do so, we will once again leverage Streamlit, and we will assume the cold start scenario. As always, you can find the whole

Python code in the GitHub book repository at https://github.com/PacktPublishing/Building-Large-Language-Model-Applications.As per the Globebotter application, also in this case you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `movieharbor.py`.


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

- **Input embedding**->those are the vector representation of tokenized input text;
- **Positional encoding**-> positional encodings are added to the input embeddings. These encodings provide information about the positions of words in the input sequence, allowing the model to understand the order of tokens;
- **Multi-head attention layer**-> a mechanism in which multiple self-attention mechanisms operate in parallel on different parts of the input data, producing multiple representations. This allows the Transformer model to attend to different parts of the input data in parallel and aggregate information from multiple perspectives.
- **Add and Norm layer**-> it combines element-wise addition and layer normalization. It adds the output of a layer to the original input and then applies layer normalization to stabilize and accelerate training. This technique helps mitigate gradient-related issues and improves the model's performance on sequential data;
- **Feed-forward layer**-> it is responsible for transforming the normalized output of attention layers into a suitable representation for the final output, using non-linear activation function such as the ReLU.

The decoding part of the Transformer starts with a similar process as the encoding part, where the target sequence (output sequence) undergoes input embedding and positional encoding.

- **Output embedding (shifted right)**-> For the decoder, the target sequence is "shifted right" by one position. This means that at each position, the model tries to predict the token that comes after the token in the original target sequence. This is achieved by removing the last token from the target sequence and padding it with a special start-of-sequence token (start symbol). This way, the decoder learns to generate the correct token based on the preceding context during autoregressive decoding.
- **Decoders layers**->similarly to the encoder block, also here we have positional encoding, multi-head attention, Add&Norm, and feed-forward layers, whose role is the same as above;

- **Linear and SoftMax**->those layers apply, respectively, a linear and non-linear transformation to the output vector. The non-linear transformation (softmax) convey the output vector into a probability distribution, corresponding to a set of candidate word. The word corresponding to the greatest element of the probability vector will be the output of the whole process.

Some models use only the encoder part, such as **BERT** (Bidirectional Encoder Representations from Transformers), which is designed for natural language understanding tasks such as text classification, question answering and sentiment analysis.

Other models use only the decoder part, such as **GPT-3** (Generative Pre-trained Transformer 3), which is designed for natural language generation tasks such as text completion, summarization and dialogue.

Finally, there are models that use both the encoder and the decoder parts, such as **T5** (Text-to-Text Transfer Transformer), which is designed for various natural language processing tasks that can be framed as text-to-text transformations, such as translation, paraphrasing and text simplification.

the core component of Transformer – the attention mechanism – remain a constant within LLMs architecture.