EXPERT INSIGHT

Early Access

Generative AI with LangChain

Build large language model (LLM) apps with Python, ChatGPT and other models

2023/11/03
朱覲镳
NEU建筑B324

Ben Auffarth

Generative AI with LangChain

Early Access Publication: Generative AI with LangChain

Early Access Production Reference: B21269

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-83508-346-8

www.packt.com

Table of Contents

1. Generative AI with LangChain: Build large language model (LLM) apps with Python, ChatGPT and other models
2. 1 What Are Generative Models?
    I. Join our book community on Discord
    II. Why the hype?
        i. What are Generative Models?
        ii. Text
        iii. Images
        iv. Sound and Music
        v. Video
        vi. Multi-Modal
    III. What is a GPT?
        i. Why now?
        ii. How does it work?
        iii. How to give it a go?
    IV. What is a Stable Diffusion model?
        i. How does it work?
        ii. Conditioning
    V. Summary
    VI. Questions
3. 2 Introduction to LangChain
    I. Join our book community on Discord
    II. What are the limitations of LLMs?
    III. What's an LLM app?
        i. Framework Comparison
    IV. What is LangChain?
        i. Why is LangChain relevant?
        ii. What can I build with LangChain?
    V. How does LangChain work?
        i. What is an agent?
        ii. What's a chain?
        iii. What is memory?
        iv. What kind of tools are there?

VI. Summary
VII. Questions
4. 3 Getting Started with LangChain
I. Join our book community on Discord
II. How to Set Up LangChain?
i. Python installation
ii. Jupyter Notebook and JupyterLab
iii. Environment management
III. Model Integrations
i. Fake LLM
ii. OpenAI
iii. Hugging Face
iv. Azure
v. Google Cloud
vi. Anthropic
vii. Jina AI
viii. Replicate
ix. Local Models
IV. Customer Service Helper
V. Summary
VI. Questions
5. 4 Querying with Tools
I. Join our book community on Discord
II. What are hallucinations?
III. How to summarize long documents?
i. Token usage
IV. Extracting information from documents
V. Answer questions with tools
VI. Reasoning strategies
VII. Summary
VIII. Questions
6. 5 Building a Chatbot like ChatGPT
I. Join our book community on Discord
II. What is a chatbot?
i. What's the state-of-the-art?
ii. Context and Memory
iii. Intentional vs Proactive

III. Retrieval and vectors
  i. Embeddings
  ii. How can we store embeddings?
  iii. Vector indexing
  iv. Vector libraries
  v. Vector databases
  vi. Document loaders
  vii. Retrievers in LangChain
IV. Implementing a chatbot!
  i. Memory mechanisms
V. Don’t say anything stupid!
VI. Summary
VII. Questions

7. 6 Developing Software
  I. Join our book community on Discord
  II. Software development and AI
    i. Present day
    ii. Code LLMs
    iii. Outlook
  III. Writing code with LLMs
  IV. Automated software development
  V. Summary
  VI. Questions
8. 7 LLMs for Data Science
  I. Join our book community on Discord
  II. Automated data science
    i. Data collection
    ii. Visualization and EDA
    iii. Pre-processing and feature extraction
    iv. AutoML
    v. The impact of generative models
  III. Agents can answer data science questions
  IV. Data exploration with LLMs
  V. Summary
  VI. Questions
9. 8 Customizing LLMs and their output
  I. Join our book community on Discord

II. Conditioning and alignment
  i. Methods for alignment
  ii. Evaluations
III. Fine-Tuning
IV. Prompt Engineering
  i. Structure of Prompts
  ii. Templating
  iii. Advanced Prompt Engineering
V. Summary
VI. Questions
10. 9 Generative AI in Production
  I. Join our book community on Discord
  II. Introduction
    i. Terminology
  III. How to evaluate your LLM apps?
    i. Comparing two outputs
    ii. Comparing against criteria
    iii. String and semantic comparisons
    iv. Benchmark dataset
  IV. How to deploy your LLM apps?
    i. Fast API webserver
    ii. Ray
  V. How to observe LLM apps?
    i. Tracking and Tracing
    ii. Observability tools
    iii. LangSmith
    iv. PromptWatch
  VI. Summary
  VII. Questions
11. 10 The Future of Generative Models
  I. Join our book community on Discord
  II. Current State of Generative AI
    i. Technical Challenges
  III. Possible Future Capabilities
  IV. Societal Implications
    i. Creative industries and advertising
    ii. Economic

iii. Education
iv. Jobs
v. Law
vi. Manufacturing
vii. Medicine
viii. Military
ix. Misinformation and cybersecurity
V. Practical Implementation Challenges
VI. The Road Ahead

Generative AI with LangChain: Build large language model (LLM) apps with Python, ChatGPT and other models

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: What are Generative Models?
2. Chapter 2: LangChain: Core Fundamentals
3. Chapter 3: Getting started with LangChain
4. Chapter 4: Question Answering over Docs
5. Chapter 5: Building a Chatbot like ChatGPT/Bard
6. Chapter 6: Developing Software with LangChain Coder
7. Chapter 7: LLM for Data Analysis
8. Chapter 8: Prompt Engineering
9. Chapter 9: LLM applications in Production
10. Chapter 10: The Future of Generative Models

1 What Are Generative Models?

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

Artificial Intelligence (**AI**) has made significant advancements, impacting businesses, societies, and individuals. For about the last decade, deep learning has evolved to process and generate unstructured data like text, images, videos, and more. These advanced AI models, which are based on deep learning, have gained popularity in various industries, and include **large language models** (**LLMs**).There is currently a significant level of hype in both the media and the industry surrounding AI. This is driven by various factors, including advancements in technology, high-profile applications, and the potential for transformative impacts across multiple sectors.In this chapter, we'll discuss generative models, in particular LLMs, and their application to domains such as text, image, sound, and video. We'll go through some of the technical background that makes them tick, and how they are trained.We'll start with an introduction clarifying where we are at in terms of the state-of-the-art, and what the hype is about.

Why the hype?

In the media, there is substantial coverage of AI-related breakthroughs and their potential implications. These range from advancements in natural language processing and computer vision to the development of sophisticated language models like GPT-3. Media outlets often highlight AI's capabilities and its potential to revolutionize industries such as healthcare, finance, transportation, and more.Particularly, generative models have received a lot of attention due to their ability to generate text, images, and other creative content that is often indistinguishable from human-generated content. These same models also provide a wide functionality including semantic search, content manipulation, and classification. This allows cost-savings by automation and can allow humans to leverage their creativity by an unprecedented level.This graph, inspired by a blog post about GPT-4 Predictions by Stephen McAleese on LessWrong, shows the improvements of LLMs in the **Massive Multitask Language Understanding** (**MMLU**) benchmark, which was designed to quantify knowledge and problem-solving ability in elementary mathematics, US history, computer science, law, and more:

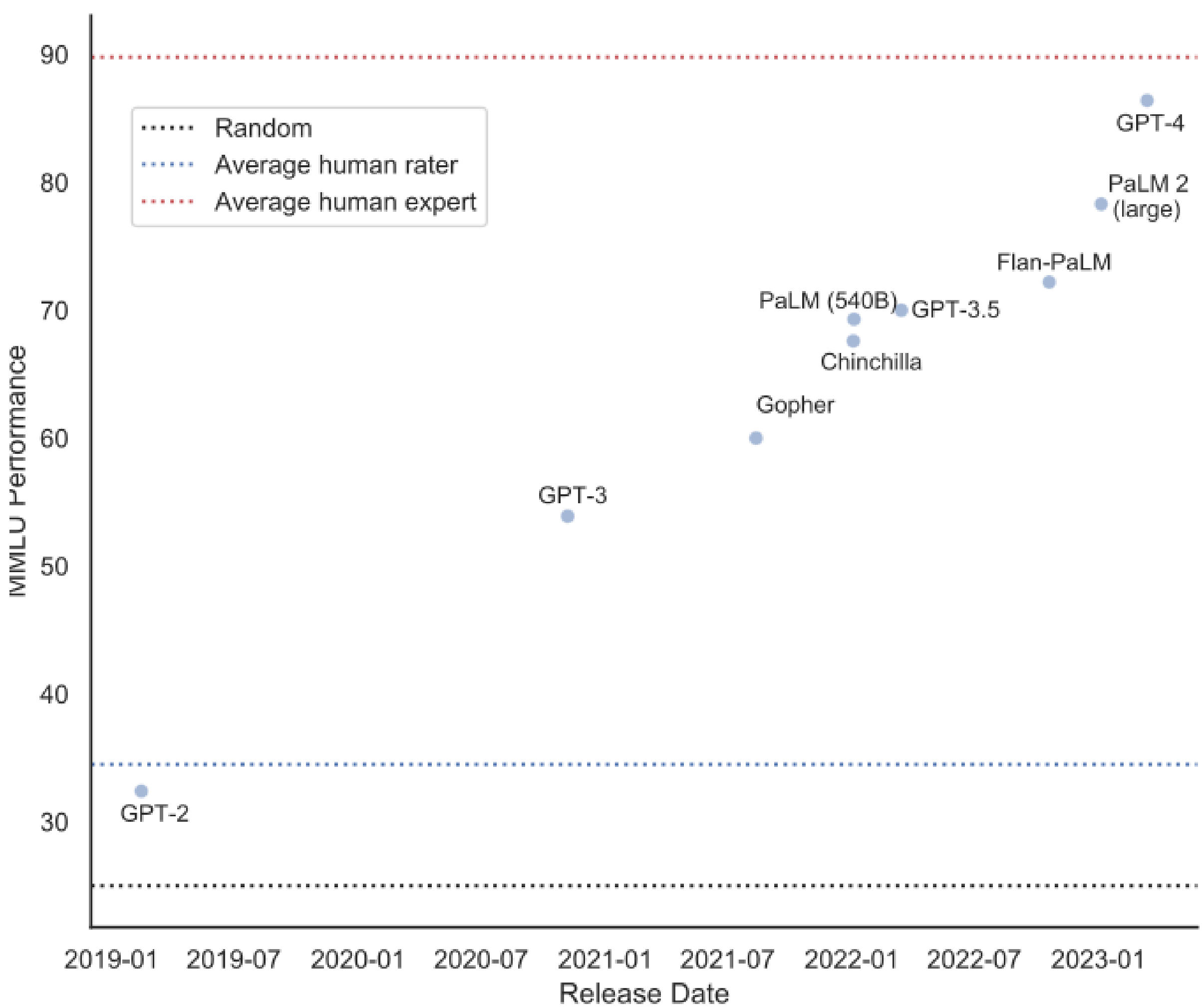


Figure 1.1: Average performance on the Massive Multitask Language Understanding (MMLU) benchmark of Large Language Models (LLM). Please note that while most benchmark results come from 5-shot, a few the GPT-2, PaLM, and PaLM-2 results refer to fine-tuned models.

You can see the progress in recent years in the benchmark. Particularly to highlight is the progress of the models provided through a public user interface by OpenAI, especially the improvements between releases, from GTP-2 to GPT-3 and GPT-3.5 to GPT-4. These models only recently started to perform better than an average human rater, but still haven't reached the performance of a human expert. These achievements of human engineering are impressive; however, it should be noted that performance of these models depends on the field; most models are still performing poorly for on the GSM8K benchmark of grade school math word problems.

> **OpenAI** is an American AI research laboratory that aims to promote and develop friendly AI. II 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 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 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 into the Bing search engine. The most significant achievements of the company include the OpenAI Gym for training reinforcement algorithms, and - more recently - the GPT-n models and DALL-E, another deep learning model that generates images from text.

Generative Pre-training Transformer (**GPT**) models, like the recently launched OpenAI's ChatGPT, are prime examples of AI advancements in the sphere of LLMs. ChatGPT has greatly improved chatbot capabilities by training at a much bigger scale and by being much bigger than previous models. These AI-based chatbots can generate human-like responses as real-time feedback to customers and can be applied to a wide range of use cases, from software development and testing to poetry and business communication. Within the industry, there is a growing sense of excitement around AI's capabilities and its potential impact on business operations. We'll look at this more in *Chapter 10*, *The Future of Generative Models*.As AI models like OpenAI's GPT continue to improve, they could become indispensable assets to teams in need of diverse knowledge and skills. For example, GPT-4 could be considered a "polymath" that could work tirelessly without demanding compensation (beyond subscription or API fees), providing assistance in subjects like Math, Verbal, Stats, Macroeconomics, Biology, and even passing the Bar exam. As these AI models become more proficient and easily accessible, they are likely to play a significant role in shaping the future of work and learning.By making knowledge more accessible and adaptable, these models have the potential to level the playing field and create new opportunities for people from all walks of life. These models have shown potential in areas that require higher levels of reasoning and understanding, although progress varies depending on the complexity of the tasks involved.As for generative models with images, we could expect models with better capabilities to assist in creating visual content, and possibly improvements in computer vision tasks such as object detection, segmentation, captioning, and much more.Let's get our terminology a bit clearer, and explain more in detail what is meant by generative model, artificial intelligence, deep learning, and machine learning.

What are Generative Models?

In the media, the term artificial intelligence is used a lot when referring to these new models. It's worth distinguishing a bit more clearly how term generative model differs from artificial intelligence, deep learning, machine learning, and language model:

- **Artificial intelligence** (**AI**) is a broad field of computer science that deals with the creation of intelligent agents, which are systems that can reason, learn, and act autonomously.
- **Machine learning** (**ML**) is a subset of AI that deals with the development of algorithms that can learn from data. ML algorithms are trained on a set of data, and then they can use that data to make predictions or decisions.

- **Deep learning** (DL) is a subset of ML that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain, and they are able to learn complex patterns from data.
- **Generative models** are a type of ML model that can generate new data. Generative models are trained on a set of data, and then they can use that data to create new data that is similar to the training data.
- **Language models** are statistical models that predict tokens (typically words) in a sequence. Some of these models, which are capable of more complex tasks, consist of many parameters (on the order of billions or even trillions), therefore they are called **large language model**.

The main difference between generative models and other types of ML models is that generative models do not just make predictions or decisions. They can actually create new data. This makes generative models very powerful, and they can be used for a variety of tasks, such as generating images, text, music, and video.Here is a table that summarizes the differences between AI, ML, DL, language model, and generative models:

Term	**Definition**
Artificial intelligence	A broad field of computer science that deals with the creation of intelligent agents.
Machine learning	A subset of AI that deals with the development of algorithms that can learn from data.
Deep learning	A subset of ML that uses artificial neural networks to learn from data.
Generative model	A type of ML model that can generate new data.
Language model	A type of model, nowadays mostly a deep learning model, that predicts tokens in context.

Figure 1.2: Terminology - Artificial Intelligence, Machine Learning, Deep Learning, and Generative Model.Generative models are a powerful type of AI that can generate new data samples that resemble the training data. Generative AI models have come a long way, enabling the generation of new examples from scratch using patterns in data. These models can handle

different types of data and are employed across various domains, including text generation, image generation, music generation, and video generation.For language models, it's important to note that some of them, particularly newer generations, are generative, in the sense that they can produce language (text), others are not. These generative models facilitate the creation of **synthetic data** to train AI models when real data is scarce or restricted. This type of data generation reduces labeling costs and improves training efficiency. Microsoft Research took this approach ("Textbooks Are All You Need", June 2023) for training their phi-1 model, where they created synthetic text books and exercises with GPT-3.5 as their training dataset.In the following sections, we'll look at the different domains of generative models such as text, images, sound, video. The applications mostly revolve around content generation, editing, and processing (recognition). Let's start with text!

Text

Text generation, such as GPT-4 by OpenAI, can generate coherent and grammatically correct poems, or code in different languages and extract features like keywords and topics. These models have practical applications in fields like content creation and **natural language processing** (**NLP**), where the ultimate goal is to create algorithms capable of interpreting human language.Language modeling aims to predict the next word, character, or even sentence based on the previous ones in a sequence. In this sense, language modeling serves as a way of encoding the rules and structures of a language in a way that can be understood by a machine. Large language models capture the structure of human language in terms of grammar, syntax, and semantics. These models are important as they form the backbone of a number of larger NLP tasks such as content creation, translation, summarization, machine translation, and text editing tasks such as spelling correction.At its core, language modeling, and more broadly Natural Language Processing, rely heavily on the quality of the representation learning. A well-trained language model encodes information about the text it's trained on and generates new text based on those learnings, thereby taking on the task of text generation.Recently, large language models have found application for tasks like essay generation, code development, translation, and understanding genetic sequences. More broadly applications of language models involve multiple areas, such as:

- **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 for solving specific problems 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**: Analyzing opinions and emotions in texts, language models can help businesses understand customer feedback and opinions more efficiently.
- **Topic modeling** and **semantic search**: These models can identify, categorize by topics, and compress documents into concise vectors, making content management and discovery easier for organizations.
- **Machine translation**: AI-powered language models can translate texts from one language into another, supporting businesses in their global expansion efforts. New generative models can perform competitively with commercial products (for example Google translate).

Despite the remarkable achievements, language models still face limitations when dealing with complex mathematical or logical reasoning tasks. It remains uncertain whether continually increasing the scale of language models will inevitably lead to new reasoning capabilities. As mentioned, we also have to consider the importance of data quality and scale, as these factors play a significant role in improving language model performance in different tasks and areas.Another domain for generative models is image generation, let's see what that's about!

Images

Generative AI is extensively used in generating 3D images, avatars, videos, graphs, illustrations for virtual or augmented reality, video games graphics design, logo creation, image editing or enhancement.This graph illustrates image generation from a text prompt with stable diffusion (Source: "Restart Sampling for Improving Generative Processes" by Yilun Xu and others at MIT and Google Research, June 2023; https://arxiv.org/pdf/2306.14878.pdf):

(c) Heun (Steps=101) (d) DDPM (Steps=100)

Figure 1.3: Image generation from a text prompt "A transparent sculpture of a duck made out of glass".

With stable diffusion models, you can see a wide variety of outcomes using only minimal changes to the initial setting of the model or - as in this case - numeric solvers and samplers. Although, they sometimes produce striking results, this instability and inconsistency is a significant challenge to applying these models more broadly.Services like **MidJourney**, **DALL-E 2**, and **Stable Diffusion** provide creative and realistic images derived from textual input or other images. Services like **DreamFusion**, **Magic3D**, and **Get3D** enable users to convert textual descriptions into 3D models and scenes, driving innovation in design, gaming, and virtual experiences.There are three main applications:

1. **Image generation**: models can generate images, such as paintings, photographs, and sketches. This can be used for a variety of purposes, such as creating art, designing products, and generating realistic visual effects.
2. **Image editing**: Models can perform tasks such as removing objects, changing colors, and adding effects. This can be used to improve the quality of images, and to make them more visually appealing.

3. **Image recognition**: large foundation models can be used to recognize images including classifying scenes, but also object detection, for example detecting faces.

Models like Generative Adversarial Networks (GANs) and DALL-E. GANs generate realistic images that have numerous business applications, while DALL-E creates images from textual descriptions, which is helpful in creative industries for designing advertisements, products, and fashion.Image editing involves modifying an image's semantics by changing content or style attributes using techniques like facial attribute editing or image morphing. Optimization- and learning-based approaches generate images with styles obtained via latent representations of pre-trained GAN models like StyleGAN. Diffusion models have recently been used for advanced image editing tasks such as connecting manually designed masked regions seamlessly or generating 3D object manipulations through text guidance. These techniques enable flexible image generation but face limited diversity issues that can be mitigated by incorporating other text inputs into the process.Into the category of image editing fall also tasks such as image restoration, which means restoring clean images from their degraded versions, which involves tasks like image super-resolution, inpainting, denoising, dehazing, and deblurring. Deep learning-based methods using CNN and transformer architectures are prevalent due to superior visual quality compared to traditional approaches. Generative models like GANs and Diffusion Models (DMs) are used for restoration but can suffer from complex training processes and mode collapse. Multi-distortion datasets and single-network approaches with attention modules or guiding sub-networks improve effectiveness for handling multiple degradation types.We'll see next what models can do with sound and music.

Sound and Music

Generative models can develop songs and audio clips based on text inputs, recognize objects in videos and create accompanying audio, and create custom music. We can classify applications again roughly into generation, editing, and recognition:

- **Music generation**: Generative models can be used to generate music, such as songs, beats, and melodies. This can be used for a variety of purposes, such as creating new music, composing soundtracks, and generating personalized playlists.
- **Sound editing**: Generative models can be used to edit sound, such as removing noise, changing pitch, and adding effects. This can be used to improve the quality of sound, and to make it more sonically appealing.
- **Sound recognition**: Generative models can be used to recognize sound, such as identifying instruments, classifying genres, and detecting speech. This can be used for a variety of purposes, such as music analysis, search, and recommendation systems.

Music generation algorithms started with algorithmic composition in the 1950s and have seen recent innovations like Google's WaveNet and OpenAI's Jukebox. These models have led to AI composer assistants, which can generate music in various styles and enable newer applications like speech synthesis.As a special case, speech-to-text generation, also known as **automatic speech recognition** (**ASR**), is the process of converting spoken language into text. They are trained on sounds and texts. ASR systems are becoming increasingly accurate, and are now used in a wide variety of applications. However, there are still some challenges that need to be addressed, such as the ability to handle noisy environments and different accents. With many potential applications such as voice dialing and computer-assisted personal assistance like Alexa and Siri, the technology behind ASR evolved from Markov Models to rely on GPTs.We'll see videos next.

Video

Video generation models like **DeepMind's** Motion to Video and **NVIDIA's Vid2Vid** rely on **GANs** for high-quality video synthesis. They can convert videos between different domains, modify existing videos, and animate still images, showing great potential for video editing and media production.Tools like Make-a-Video and Imagen Video convert natural language prompts into video clips, simplifying video production and content creation processes. The broad classes of applications are these:

- **Video generation**: Generative models can be used to generate videos, such as short films, animations, and commercials. This can be for creating new content, advertising products, and generating realistic visual effects.
- **Video editing**: We can edit videos, such as removing objects, changing colors, and adding effects. This can help to improve the quality of videos, and to make them more visually appealing.
- **Video recognition**: Models can recognize video, such as identifying objects, classifying scenes, and detecting faces. This can help for applications such as security, search, and recommendation systems.

Some models can generate content in more than one domain or modality. These are called multi-modal models.

Multi-Modal

Multi-modal generative models can generate **text**, **images**, **sound**, and **video**. This allows them to create more realistic and immersive experiences. Multi-modal models are still in their early stages of development, but they have the potential to revolutionize the way we interact with computers and the way we experience the world. For example, these advancements have significantly improved performance in image captioning tasks, the process of describing an image's content through natural language.Multi-modal models adopt generative architectures that fuse images and captions into a single model for shared learning space. The process involves a two-step encoder-decoder architecture: visual encoding and language decoding.We can distinguish these potential use cases:

- **Virtual reality**: These models can be used to create virtual reality experiences that are more realistic and immersive. This can help in gaming, education, and training.
- **Augmented reality**: They can create augmented reality experiences that overlay digital content on the real world. This is useful for navigation, shopping, and entertainment.

In the next section, we'll discuss the technical background of large language models.

What is a GPT?

Large Language Models (**LLMs**) are deeply trained 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 it to learn language patterns and structures.The notable strength of the latest generation of LLMs as conversational interface (ChatBot) lies in their ability to generate coherent and contextually appropriate responses, even in open-ended conversations. This generating the next word based on the preceding words repeatedly, the model produces fluent and coherent text often indistinguishable from text produced by humans. However, ChatGPT has been observed to "sometimes write plausible-sounding but incorrect or nonsensical answers" as expressed in a disclaimer by **OpenAI**. This is referred to as hallucination and is just one of the concerns around **LLMs**.A **Transformer**, is a deep learning architecture, first introduced in 2017 by researchers at Google and the University of Toronto (in an article called "Attention is All You Need"), which 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 different parts of the input sequence.**Generative Pre-Trained Transformers** (**GPTs**) were introduced by researchers at OpenAI in 2018 together with the first of their eponymous GPT models, GPT-1, and published as "Improving Language Understanding by Generative Pre-Training". 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. GPT-1, introduced by OpenAI in 2018, was trained on BookCorpus with 985 million words. BERT, released in the same year, was trained on a combined corpus of **BookCorpus** and **English Wikipedia**, totalling **3.3 billion words**. Now, training corpora for LLMs reach up to trillions of tokens.This graph illustrates how LLMs have been growing very large:

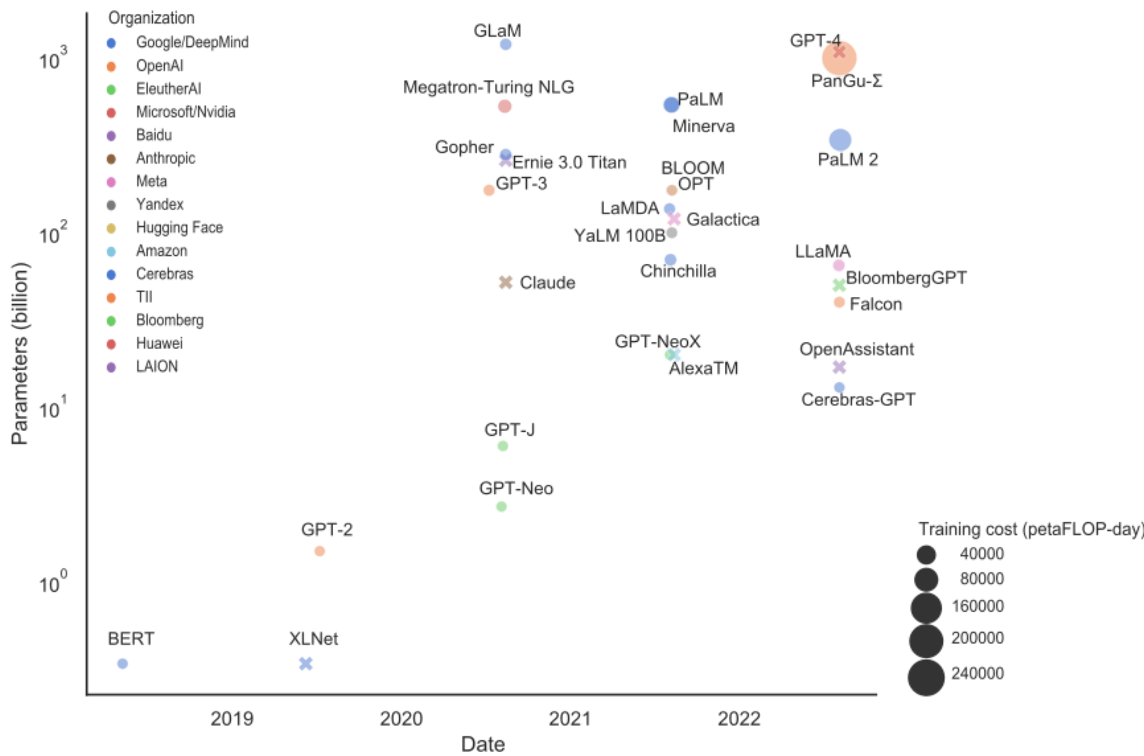


Figure 1.4: Large Language Models from BERT to GPT-4 - size, training budget, and organizations.

The size of the data points indicates training cost in terms of petaFLOP days. For some models, especially for proprietary and closed-source models this information is not known - in these cases, I've placed a cross. For example, for XLNet, the paper doesn't give the information about compute in flops, however, the training was done on 512 TPU v3 chips over 2.5 days. The colors of the data points show the company or organization developing the model - since these might not come out in the print or on the Kindle (unless you have a Kindle color), you can find a color version of this graph at this URL:The development of GPT models has seen significant progress, with OpenAI's GPT-n series leading the way in creating foundational AI models. The size of the training corpus for LLMs has been increasing drastically. GPT-1, introduced by OpenAI in 2018, was trained on BookCorpus with 985 million words. BERT, released in the same year, was trained on a combined corpus of BookCorpus and English Wikipedia, totaling 3.3 billion words. Now, training corpora for LLMs reach up to trillions of tokens.

> A **foundation model** (sometimes: base model) is a large model, which was trained on an immense quantity of data at scale so that it can be adapted to a wide range of downstream tasks. In GPT models, this pre-training is done self-supervised learning.

Trained on 300 billion tokens, **GPT-3** has **175 billion parameters**, an unprecedented size for deep learning models. **GPT-4** is the most recent in the series, though its size and training details have not been published due to competitive and safety concerns, however, different estimates were putting it between **200 and 500 billion parameters**. *Sam Altman*, the CEO of OpenAI has stated that the cost of training **GPT-4** was more than $100 million.ChatGPT, a conversation model, was released by **OpenAI** in November 2022. Based on prior **GPT** models (particularly **GPT-3**) and optimized for dialogues, it uses a combination of human-generated roleplaying conversations and a dataset of human labeler demonstrations of the desired model behavior. The model exhibits excellent capabilities such as wide-ranging knowledge retention and precise context tracking in multi-turn dialogues.Another substantial advancement came with **GPT-4**, which extends beyond text input to include multimodal signals, in March 2023. **GPT-4** provides superior performance on various evaluation tasks coupled with significantly better response avoidance to malicious or provocative queries due to six months of iterative alignment during training.Other notable foundational GPT models beside OpenAI's includes Google's **PaLM2**. Although GPT-4 leads most benchmarks in performance, these and other models demonstrate a comparable performance in some tasks, and have contributed to advancements in generative transformer-based language models. Meta's **LLaMA** was trained on **1.4 trillion tokens**, while **PaLM2**, the model behind Google's chatbot, **Bard**, consists of **340 billion parameters** - smaller than previous LLMs - appears to have a larger scale of training data in at least 100 languages.

> There are quite a few **companies and organizations developing LLMs**, and they are releasing them on different terms. OpenAI has released GPT-2 and subsequent models have been closed-source, but

open for usage on their website or through an API. Meta is releasing models from RoBERTa, BART to LLaMA including parameters (the weights) of the models, although under a non-commercial license, and the source code for setting up and training the models. Google AI and their DeepMind division have developed a number of large language models, including BERT, GPT-2, LaMDA, Chinchilla, Gopher, PaLM, and PaLM2. They've been releasing the code and weights of a few of their models under open-source licensing, even though recently they have moved towards more secrecy in their development. Microsoft has developed a number of large language models, including Turing NLG and Megatron-Turing NLG, however, they have integrated OpenAI models into Microsoft 365 and Bing. Technology Innovation Institute (TII), an Abu Dhabi government funded research institution, has open-sourced Falcon LLM for research and commercial usage.

GPT models can also work with modalities beyond text for input and output, as seen in GPT-4's ability to process image input alongside text. Additionally, they serve as a foundation for text-to-image technologies like diffusion and parallel decoding, enabling the development of **visual foundation models** (**VFMs**) for systems that work with images.In summary, GPT models have evolved rapidly, enabling the creation of versatile foundational AI models suitable for a wide range of downstream tasks and modalities, ultimately driving innovation across various applications and industries.In the next section, we'll review the progress deep learning and generative models have been making over recent years that leads up to the current explosion of apparent capabilities and the attention these models have been getting.

Why now?

The success of generative AI coming into the public spotlight in 2022 can be attributed to several interlinked drivers. The development and success of generative models have relied on improved algorithms, considerable advances in compute power and hardware design, the availability of large labelled datasets, and an active and a collaborative research community helping evolve a set of tools and techniques.The development of more sophisticated mathematical and computational methods has played a vital role in the advancement of generative models. The backpropagation algorithm introduced in the 1980s by Geoffrey Hinton, David Rumelhart, and Ronald Williams is one such example. It provided a way to effectively train multi-layer neural networks.In the 2000s, **neural networks** began to regain popularity as researchers developed more complex architectures. However, it was the advent of deep learning, a type of neural network with numerous layers, which marked a significant turning point in the performance and capabilities of these models. Interestingly, although the concept of deep learning had existed for some time, the development and expansion of generative models correlate with significant advances in hardware, particularly **graphics processing units** (**GPUs**), which have been instrumental in propelling the field forward.As mentioned, the availability of cheaper and more powerful hardware has been a key factor in the development of deeper models. This is because deep learning models require a lot of computing power to train and to run. This concerns all aspects of processing power, memory, and disk space. This graph shows the cost of computer storage over time for different mediums such as disks, solid state, flash, and internal memory in terms of price in dollars per terabyte (source: Our World in Data; https://ourworldindata.org/moores-law):

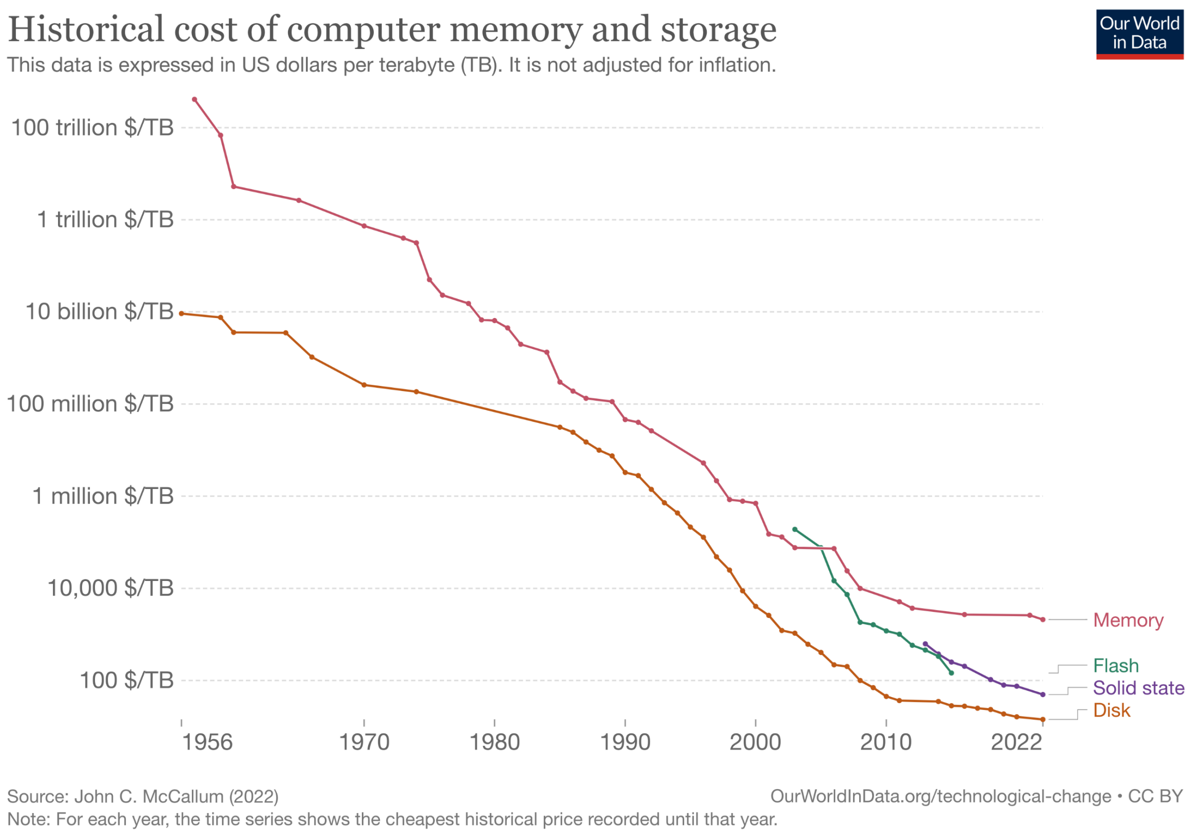


Figure 1.5: Cost of computer storage since the 1950s in dollars per terrabyte.

While the past, training a deep learning model was prohibitively expensive, as the cost of hardware has come down, it has become possible to train bigger models on much larger datasets. The model size is one of the factors determining how well a model can approximate (as measured in perplexity) the training dataset.

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 for 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. Particularly, it seems that in models consisting of in the range of between 2 to 7 billion parameters new capabilities emerge such as the ability to generate different creative text formats, like poems, code, scripts, musical pieces, email, letters, and answer questions in an informative way, even if they are open-ended and challenging.

This trend towards larger models started around 2009, when Nvidia catalyzed what is often called the "big bang" of deep learning. GPUs are particularly well-suited for the matrix/vector computations necessary to train deep-learning neural networks, therefore significantly increasing the speed and efficiency of these systems by several orders of magnitude, and reducing running times from weeks to days. In particular, Nvidia's **CUDA** platform, which allows direct programming of GPUs, has made it easier than ever for researchers and developers to experiment with and deploy complex generative models.In the 2010s, different types of generative models started getting traction. Autoencoders, a kind of neural network that can learn to compress data from the input layer to a representation, and then reconstructs the input, served as a basis for more advanced models like **Variational Autoencoders** (**VAEs**) first proposed in 2013. **VAEs**, unlike traditional autoencoders, use variational inference to learn the distribution of data, also called the latent space of input data.Around the same time, **Generative Adversarial Networks** (**GANs**) were proposed by Ian Goodfellow and others in 2014. The setup for training GANs is illustrated in this diagram (taken from "A Survey on Text Generation using Generative Adversarial Networks", G de Rosa and J P. Papa, 2022; https://arxiv.org/pdf/2212.11119.pdf):

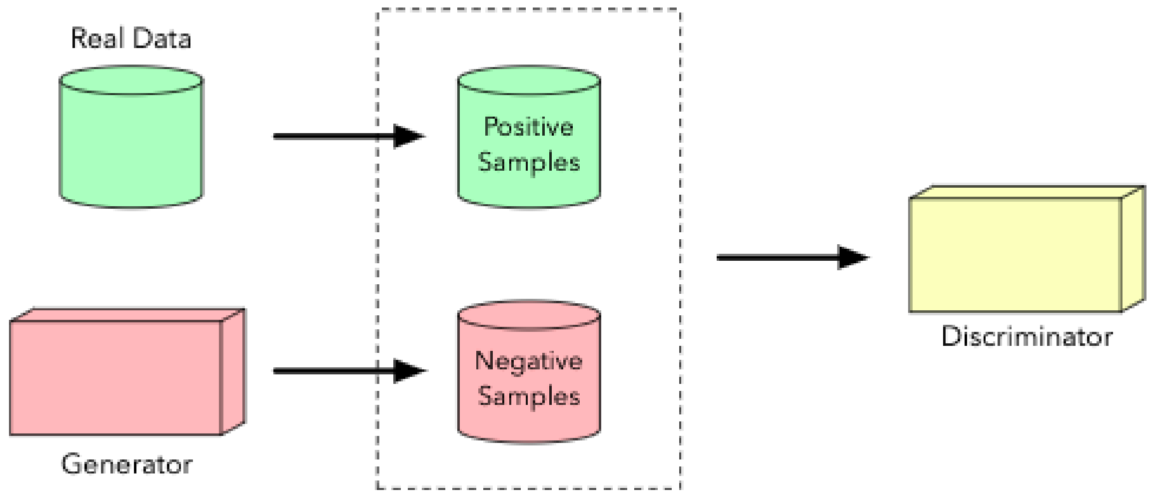


Figure 1.6: Generative Adversarial Network (GAN) training.

The GANs consist of two networks that are pitted against each other in a game-like setting - the generator that generates new data, often images, and the discriminator which estimates the probability of the new data being real. As they compete against each other, GANs gets better at its task, being able to generate realistic images and other types of data.Over the past decade, significant advancements have been made in the fundamental algorithms used in deep learning, such as better optimization methods, more sophisticated model architectures, and improved regularization techniques.Transformer models, introduced in 2017, built upon this progress and enabled the creation of large-scale models like GPT-3. Transformers rely on attention mechanisms, and resulted in a further leap in the performance of generative models. These models, such as Google's BERT and OpenAI's GPT series, can generate highly coherent and contextually relevant text.The development of transfer learning techniques, which allow a model pre-trained on one task to be fine-tuned on another, similar task, has also been significant. These techniques have made it more efficient and practical to train large generative models.Moreover, part of the rise of generative models can be attributed to the development of software libraries and tools (**TensorFlow**, **PyTorch**, **Keras**) specifically designed to work with these artificial neural networks, streamlining the process of building, training, and deploying them.To further drive the development of generative models, the research community has regularly held challenges like ImageNet for image classification, and has started to do the same for generative models, with competitions such as the Generative Adversarial Networks (GAN) Competition.In addition to the availability of cheaper and more powerful hardware, the availability of large datasets of labeled data has also been a key factor in the development of generative models. This is because deep learning models and generative models in particular require vast amounts of text data for effective training. The explosion of data available from the internet, particularly in the last decade, created the suitable environment for such models to thrive. As the internet has become more popular, it has become easier to collect large datasets of text, images, and other data. This has made it possible to train generative models on much larger datasets than would have been possible in the past.In summary, generative modelling is a fascinating and rapidly evolving field. It has the potential to revolutionize the way we interact with computers and the way we create new content. I am excited to see what the future holds for this field.Let's get into the nitty gritty - how does this work?

How does it work?

Models such as BERT and GPT were made possible by the **Transformer** deep neural network architecture, which has been a game changer for natural language processing. Designed to avoid recursion in order to allow parallel computation, the Transformer architecture, in different variations, is continuing to push the boundaries of what's possible within the field of Natural Language Processing and Generative AI.One of the defining features of Transformers is the attention mechanism. Traditional sequence-to-sequence models often suffered from the problem of handling long dependencies - they had difficulty remembering relevant information if the sequences were too long. Transformer models introduced attention mechanisms to navigate this problem. The Self-Attention mechanism, often referred to as the core of the Transformer model, assigns a score to each word in the sequence, determining how much focus should be given to that word.Transformers consist of modules, which can be stacked, thereby creating very large models that can learn massive datasets. These are indicated in the diagram here:

避免递归
允许并行计算

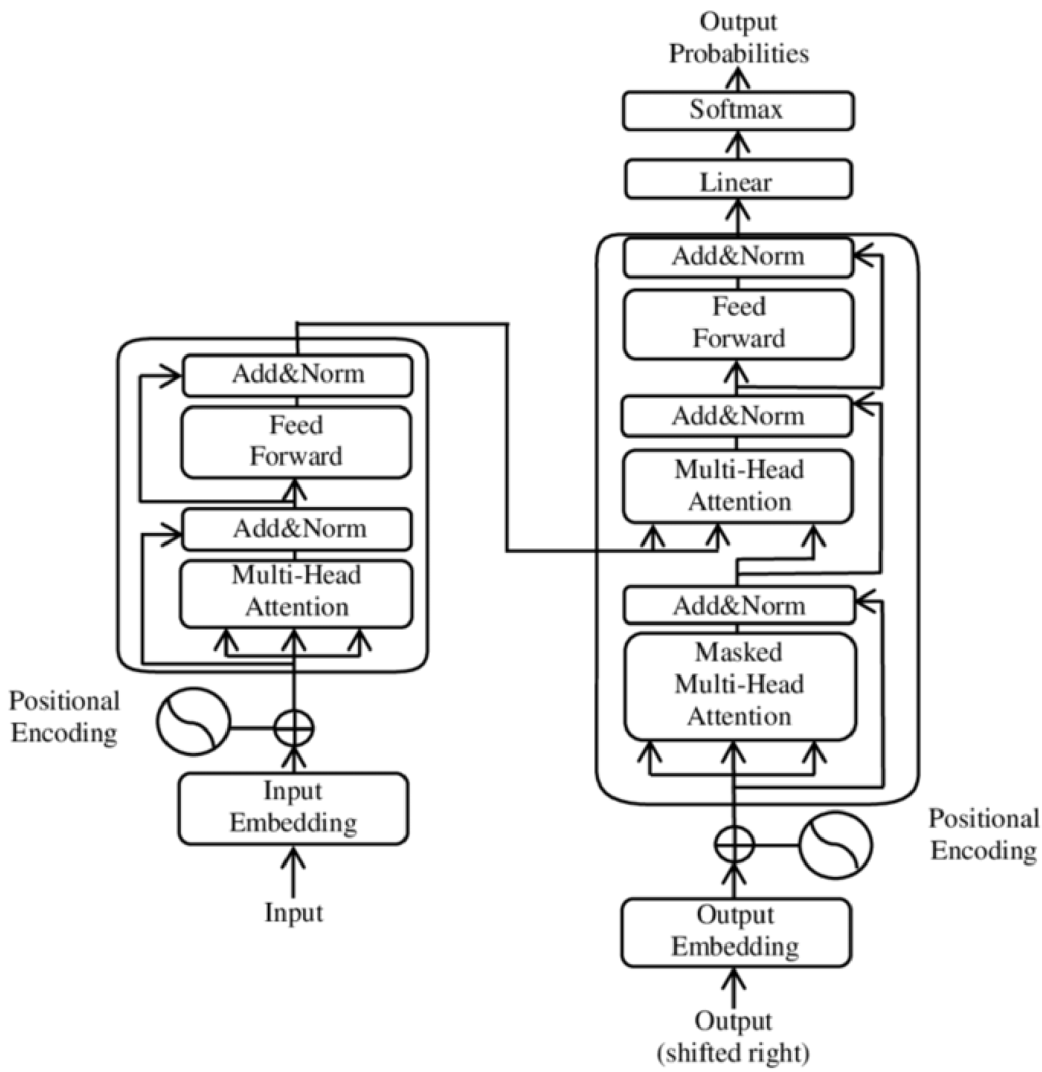


Figure 1.7: The Transformer architecture (credit: Yuening Jia, Wikimedia Commons)

The architectural features that have contributed to the success of Transformers:

- **Encoder-Decoder structure**: The Transformer model follows an Encoder-Decoder structure. The encoder takes the input sequence and computes a series of representations, (contextual embeddings), for each word. These representations consider not only the inherent meaning of the words (their semantic value) but also their context in the sequence. The decoder then uses this encoded information to generate the output sequence one item at a time, using the context of the previously generated items.

- **Positional Encoding**: Since 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.

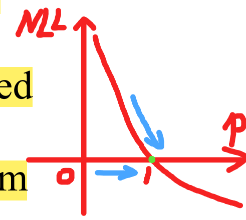
层归一化算法通过计算在一个训练样本上某一层所有的神经元的均值和方差来对输入进行归一化，像批归一化算法那样，同样也给每个神经元加入了增益 \gamma 和偏置 \beta 来实现线性变换，这在归一化后激活函数前使用。

- **Layer Normalization**: To stabilize the network's learning, 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.

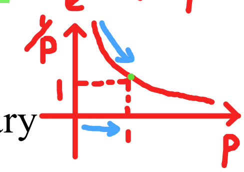
Another, optional architectural feature, which is not specific to Transformers are Skip Connections (also: Residual Connections). These were introduced to mitigate degradation problems as networks get deeper, Skip Connections are used. This allows gradients to flow unchanged across layers by shortcutting the inputs to deeper layers Transformers have pushed the envelope in NLP, especially in translation and language understanding. **Neural machine translation** (NMT) is a mainstream approach to machine translation that uses deep learning to capture long-range dependencies in the sentence. **NMT** based on Transformers outperformed previous approaches such as using recurrent neural networks, in particular **long short-term memory** (**LSTM**) networks. This can be attributed to this powerful architecture including first and foremost attention, which allows the Transformer model to treat word order in a flexible manner, no matter how far apart they are, that is optimal for each specific situation.Furthermore, the combination of these architectural features allows it to successfully tackle tasks that involve understanding and generating human language and other domains. OpenAI's powerful GPT models for language generation is a Transformer as well, as is **DeepMind's AlphaFold 2**, a model that predicts protein structure from their genetic sequences. Transformers have been able to maintain performance across longer sequences better than other models, for example recurrent neural networks. This has been contributing to their success, however, the transformer architecture means that they can capture only dependencies within a fixed input width. Early attention mechanisms scaled quadratically with the number of data points, rendering them inapplicable to settings with large amounts of inputs. There have been many proposed approaches to obtain efficiency gains such as sparse, low-rank self-attention, and latent bottlenecks to name just a few. Other work 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 particularity of GPTs and the origin of their name is the pre-training. Let's see how these LLMs are trained!

Pre-training

The first step in training a **Large Language Model** (**LLM**) is tokenization. This process involves converting words into numbers so that they can be processed by the model given that **LLMs** are mathematical functions that require numerical inputs and outputs.To carry out this tokenization, **LLMs** use a unique tokenizer. Tokenizers map words in text to corresponding lists of integers. Before training the **LLM**, the tokenizer is typically fitted to the entire training dataset and then frozen. A common type of tokenizer employed is byte pair encoding. It's important to note that tokenizers do not produce arbitrary integers. Instead, they output integers within a specific range - from to where represents the vocabulary size of the tokenizer.Now, considering outputs, when LLM receives a text, it mainly yields a vector falling within . This output vector is then passed through a softmax function to yield another vector, which is referred to as a probability vector. With its entries being non-negative and summing up to , this vector can be interpreted as the probability distribution over the vocabulary of the **LLM**.Also, it is necessary to point out that **LLMs** can only generate a token based on a sequence of tokens that does not exceed its context window. This context window refers to the length of the longest sequence of tokens that a **LLM** can use. If a sequence longer than this window is presented, the **LLM** would need to



$PPL = e^{NLL} = e^{-\log P} = \frac{1}{P}$



either truncate the sequence or employ algorithmic modifications to handle it. Typical context window sizes for **LLMs** can range from about 1,000 to 10,000 tokens.Training **LLMs** involves a specific process of tokenizing input data, feeding it into the model, and generating an output that is a probability distribution over the model's vocabulary. The specific mechanisms within this process, such as the softmax function and context window, help to facilitate **LLMs** understanding of and response to input data.**Negative Log-Likelihood** (**NLL**) and **Perplexity** (**PPL**) are important metrics used in the process of training and evaluating language models. **NLL** is a loss function used in machine learning algorithms, aimed at maximizing the probability of correct predictions. A lower **NLL** indicates that the network has successfully learned patterns from the training set, and therefore it will be able to accurately predict the labels of the training samples. It's important to mention that **NLL** is a value constrained within a positive interval.**Perplexity** (**PPL**), on the other hand, is an exponentiation of the **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.

Scaling

It is worth talking at least briefly about the choice of architecture, and why these models are as large as they are. In a paper from 2020 by researchers from OpenAI, Kaplan and others discussed scaling laws and choice of parameters.Interestingly, they compare lots of different architecture choices, and, among other things, show that transformers outperform **LSTMs** as language models in terms of perplexity in no small part due to improved use of long contexts - while these recurrent networks plateau after less than 100 tokens, transformers improve through the whole context. Therefore, transformers come not only with better training and inference speed with respect to the transformer, but also give better performance when looking at relevant contexts.Further, they found a power-law relationship of dataset size, model size (number of parameters), and the amount of compute for training, in the sense that in order to improve performance by a certain factor, one of these factors have to scaled-up by as the power of the factor, however, for optimal performance all three factors must be scaled in tandem in order not to get a bottleneck effect.Researchers at **DeepMind** (Hoffmann and others, 2022) analyzed training compute and dataset size of **LLMs**, and concluded that a **LLMs** are undertrained in terms of compute budget and dataset size as suggested by scaling laws. They predicted that large models would perform better if substantially smaller and trained much longer than, and - in fact - validated their prediction comparing a **70 billion parameter Chinchilla model** on a benchmark to their Gopher model, which consists of **280 billion parameters**.More recently, the team at has been found that longer training in terms of epochs or more compute in terms of petaflops didn't seem to improve performance anymore, and smaller networks and higher-quality datasets can give very competitive performance.

trained much longer

parameters

Conditioning

Conditioning large language models refers to adapting the model for specific tasks. Different methods of conditioning include fine-tuning, prompting, instruction tuning, and reinforcement learning:

- 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).
- Prompting techniques present problems as text prompts and expect model completions.

For fine-tuning, usually, reinforcement learning combines supervised fine-tuning with reinforcement learning using human feedback to train the model based on human preferences. LLMs can be trained on a training set of examples which are themselves generated by an LLM (bootstrapped from a small initial set of human-generated examples) such as in the training set for phi-1 by Microsoft Research ("Textbooks Are All You Need", June 2023).With prompting techniques text examples of similar problems and their solutions will be presented. Zero-shot prompting involves no solved examples, while few-shot prompting includes a small number of examples of similar (problem, solution) pairs.These conditioning methods continue to evolve, becoming more effective and useful for a wide range of applications. Prompt engineering and conditioning methods will be explored further in *Chapter 8*, *Prompt Engineering*.

How to give it a go?

You can access OpenAI's model through their website or through their API. If you want to try other large language models on your laptop, open-source LLMs are a good place to get started. There is a whole zoo of stuff out there!You can access these models through Hugginface or other providers. You can even download them, finetune them, or - if you are feeling really fancy - fully train a model. We'll look at using these models in more detail in *Chapter 9. LLM applications in Production*.In the next section, we'll look at stable diffusion and how it works.

What is a Stable Diffusion model?

Image generation models are a type of generative model that can be used to generate images. Image generation models are a powerful tool that can be used to generate realistic and creative images. They are still in their early stages of development, but they have the potential to revolutionize the way we create and consume images.One of the most popular image generation models is **Stable Diffusion**, another one is **Midjourney**. In simplest terms these are a deep learning models that creates images given text prompts. Google Brain announced the creation of two text-to-image models, **Imagen** and **Parti**, in 2022.

How does it work?

The **Stable Diffusion** model is a deep learning, text-to-image model developed by researchers from the CompVis Group at Ludwig Maximilian University of Munich and Runway. It generates detailed images conditioned on text descriptions and utilizes a latent diffusion model architecture. The model's source code and even the weights have been publicly released under the **CreativeML OpenRAIL-M License**, which "does not impose any restrictions on reuse, distribution, commercialization, adaptation." The model can be run on consumer hardware equipped with a modest GPU (for example the **GeForce 40** series).**Stable Diffusion** is a type of diffusion model that uses a Gumbel distribution to add noise to the image. The Gumbel distribution is a continuous probability distribution that is often used in machine learning because it is easy to sample from and it has the property to be more stable. Stability means that the model is less likely to get stuck in local minima, which can happen with other types of diffusion models.The model consists of a **variational autoencoder** (**VAE**), a **U-Net**, and a **text encoder**. The **VAE** has two parts, an encoder and a decoder, compressing the original high-dimensional image into a lower dimensional latent space and reconstructing it back into the image space. The latent space significantly decreases computational complexity, making the diffusion process faster.The **VAE** encoder compresses images into a latent space, while the **U-Net** performs denoising from forward diffusion to obtain a latent representation. The **VAE** decoder then generates the final image. The model can be flexibly conditioned on various modalities, including text, and leverages a cross-attention mechanism to incorporate conditioning information.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 is utilized for predicting noise in the image. The U-Net takes the noisy image as input and processes it through a series of convolutional layers to extract features and learn representations. These convolutional layers, typically organized in a contracting path, reduce the spatial dimensions while increasing the number of channels.Once the contracting path reaches the bottleneck of the U-Net, it then expands through a symmetric expanding path. In the expanding path, transposed convolutions (also known as upsampling or deconvolutions) are applied to progressively upsample the spatial dimensions while reducing the number of channels.During the diffusion process, the U-Net's expanding path takes the noisy image and reconstructs the latent representation from the forward diffusion. By comparing the reconstructed latent representation with the true latent representation, the U-Net predicts an estimation of the noise in the original image. This prediction of noise helps in the reverse diffusion process to recover the original image.Diffusion models operate through a process similar to diffusion in physics. It follows 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 here is the **reverse diffusion process**, where the model attempts to recover the original image from a noisy, meaningless image. This result is achieved by subtracting estimated noise from the noisy image step-by-step, ultimately restoring an image resembling the original image.The denoising process is demonstrated in this plot (source: User Benlisquare on Wikimedia Commons):

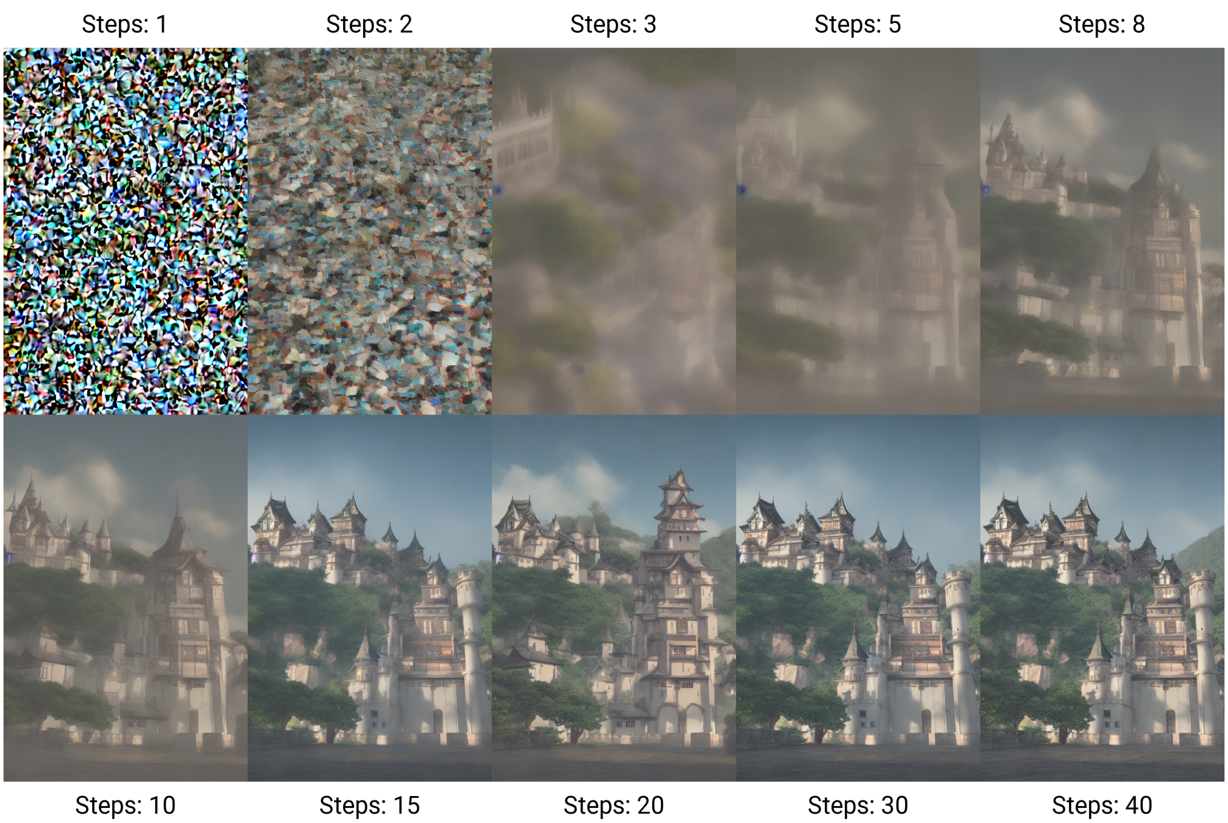


Figure 1.8: European-style castle in Japan, created using the Stable Diffusion V1-5 AI diffusion model. Only steps within a 40-step generation process are shown.

In the plot, you can see the image generation step by step, a U-Net denoising process using the DDIM sampling method, which repeatedly removes Gaussian noise, and then decodes the denoised output into pixel space.Stable Diffusion is a deep learning model that leverages a diffusion process to generate 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.

During the training of image generation models, 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.The model was trained on a dataset called **LAION-5B**, derived from Common Crawl data, comprising billions of image-text pairs. The training dataset was classified based on language, resolution, watermark likelihood, and aesthetic score. Stable Diffusion was trained on subsets of this dataset. The model's training data had a diverse range of sources, with a significant portion coming from websites such as **Pinterest**, **WordPress**, **Blogspot**, **Flickr**, **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.There are two main model versions so far of **Stable Diffusion**, **version 1** and **2**. Let's see how they differ.

Model differences

Stable Diffusion v1 and **v2** are different in terms of text processing, training data and their results. In terms of text processing, **Stable Diffusion v2** uses **OpenClip** for text embedding while **v1** uses **Open AI's CLIP ViT-L/14** for text embedding. **OpenClip** is five times larger than **CLIP**, which improves the image quality, and it also gives researchers more transparency in studying and optimizing the model.Regarding training data, **Stable Diffusion v1.4** is trained with three different datasets, while **Stable Diffusion v2** is trained on a subset of **LAION-5B** filtered for explicit pornographic material (**NSFW filter**) and an aesthetic score above a threshold. The **LAION 5B** dataset is a large-scale dataset consisting of 5.85 billion CLIP-filtered image-text pairs. Over 2.3 billion samples in the dataset contain English language, while **2.2 billion samples** originate from over 100 other languages. The remaining one billion samples do not allow a certain language assignment, such as names.The acquisition pipeline for the dataset was complex and required a lot of processing. It includes distributed processing of a petabyte-scale Common Crawl dataset, distributed download of images, and few **GPU** node post-processing of the data, which produces the final dataset. The filtering also removed duplicate samples and trimmed the dataset from 50 billion candidates to just below 6 billion **CLIP**-filtered image-text pairs.In terms of outcomes, Stable Diffusion v2 is harder to use for controlling styles and generating celebrities. This difference is likely due to the training data difference, as Open AI's proprietary data may have more artwork and celebrity photos, which are not included in Stable Diffusion v2 training data.In summary, **Stable Diffusion v2** used a different text embedding model and trained on a different subset of data, resulting in different outcomes compared to **Stable Diffusion v1**. While **Stable Diffusion v2** may be more transparent and better for long-term development, **Stable Diffusion v1** may have better results for particular use cases such as controlling styles or generating celebrities due to its training data.Now we'll look at the conditioning for the model in the text-to-image use case.

显著色情材料过滤器

Conditioning

The conditioning process allow these models to be influenced by the input textual prompts or other input types like depth maps or outlines for greater precision in order to create relevant images. During conditioning, the prompt is tokenized, and each token is converted to an embedding, a vector of a certain length, sometimes 768 values. These embeddings, which account for the semantic relationships between words, are then processed by a text transformer and fed to the noise predictor, steering it to produce an image that aligns with the text prompt.In the text-to-image process, the model uses a text prompt to generate a completely new image. The text prompt is encoded into the latent space, and a diffusion process gradually adds noise (controlled by denoising strength) to evolve the initial image towards the output image.Let's conclude the chapter!

Summary

Generative models like large language models (LLMs) have received considerable attention due to their potential to revolutionize numerous industries and tasks. Especially their applications in text generation and image synthesis have garnered significant media hype. Leading companies such as OpenAI are pushing the boundaries of LLMs, with their Generative Pre-trained Transformers (GPT) series capturing widespread attention for their impressive language generation capabilities. In this chapter, we've discussed the media attention the latest breakthroughs have been gathering, the recent history of deep learning and AI, generative models, and LLMs and Pre-trained Generative Models (GPTs) together with the theoretical ideas underpinning them, especially the Transformer architecture. We also discussed diffusion models for images, and applications for text, images, sound, and videos.The next chapter will explore the tooling of generative and particularly LLMs with the Langchain framework, focusing on the fundamentals, the implementation, and use of this particular tool in optimizing and enhancing LLMs.I think it's a good habit to check that you've digested the material when reading a technical book. I've created a few questions for this chapter.

Questions

If you've read and understood this chapter, you should be able to answer these questions:

1. What is a generative model?
2. Which applications exist for generative models?
3. What's a large language model (LLM) and what does it do?
4. How can we get bigger performance from LLMs?
5. What are the conditions that make these models possible?
6. Which companies and organizations are the big players in developing LLMs?
7. What is a transformer and what does it consist of?
8. What does GPT mean?
9. How does stable diffusion work?
10. How is stable diffusion trained?

If you struggle to answer these questions, please refer back to the corresponding sections in this chapter to make sure you've understood the material.

2 Introduction to LangChain

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

Keys: Agent
Agent executors
Chains
Memory
Tools

In this chapter, we discuss limitations of LLMs, and how combining LLMs with tools can overcome these challenges thereby building innovative language-based applications. There are a few powerful frameworks that empowers developers by providing robust tools for prompt engineering, chaining, data retrieval, and more. Whether you're a developer, data scientist or simply curious about technological advancements in natural language processing (NLP) or generative AI, you should learn about the most powerful and popular of these frameworks, LangChain.LangChain addresses pain points associated with working with LLMs and provides an intuitive framework to create customized NLP solutions. In LangChain, components like LLMs, internet searches, and database lookups can be chained together, which refers to executing different tasks one after another in a sequence based on requirements by the data or the tasks. By leveraging its features, developers can build dynamic and data-aware applications that harness the recent technological breakthroughs that we discussed in chapter 1. We'll include a few use cases to illustrate how the framework can help businesses and organizations in different domains.LangChain's support for agents and memory makes it possible to build a variety of applications that are more powerful and flexible than those that can be built by simply calling out to a language model via an API. We will talk about important concepts related to the framework such as agents, chains, action plan generation and memory. All these concepts are important to understand the how LangChain works.The main sections are:

important for robots

- What are the limitations of LLMs?
- What is an LLM app?
- What is LangChain?
- How does LangChain work?

We'll start off the chapter by going over the limitations of LLMs.

What are the limitations of LLMs?

Large language models (**LLMs**) have gained significant attention and popularity due to their ability to generate human-like text and understand natural language, which makes them useful in scenarios that revolve around content generation, text classification, and summarization. While **LLMs** offer impressive capabilities, they suffer from limitations that can hinder their effectiveness in certain scenarios. Understanding these limitations is crucial when developing applications. Some pain points associated with large language models include:

1. **Outdated Knowledge**: **LLMs** are unable to provide real-time or recent data as they rely solely on the training data provided to them.
2. **Inability to act**: **LLMs** cannot perform actions or interact with external systems, limiting their functionality. For example, they cannot initiate web searches, query databases in real-time, or use a calculator for multiplying numbers.
3. **Lack of context and additional information**: **LLMs** may struggle to understand and incorporate context from previous prompts or conversations. They may not remember previously mentioned details or fail to provide additional relevant information beyond the given prompt.

原始的LLM基本上是单次响应

4. **Complexity and Learning Curve**: Developing applications using large language models often requires a deep understanding of AI concepts, complex algorithms, and APIs. This can pose a challenge for developers who may not have expertise in these areas.

LLM
本质问题

5. **Hallucinations**: **LLMs** have a lot of general knowledge about the world implicit in their weights. However, they may have an insufficient understanding about certain subjects, and generate responses that are not factually correct or coherent. For example, they might produce information that does not exist or provide inaccurate details.
6. **Bias and Discrimination**: Depending on the data they were trained on, LLMs can exhibit biases, which can be of religious, ideological, political, and other nature.

LLMs don't have information on current events because they don't have a connection to the outside world and they wouldn't know about anything that they weren't trained on, such as anything after the cutoff date, which is when the training data were generated. More than that, they struggle with contextual understanding beyond training data limitations. For example, since the models cannot perform actions or interact with external systems directly, they wouldn't know the weather, don't have access to your documents.This cutoff day issue is illustrated here in the OpenAI ChatGPT chat interface asking about **LangChain**:

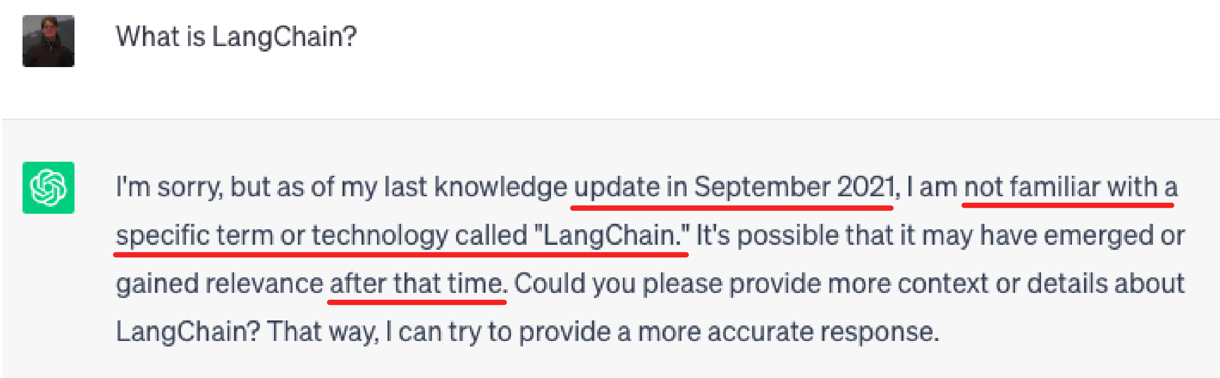


Figure 1.1: ChatGPT - lack of up-to-date information.

In this case, the model was able to correctly catch the problem and give the correct feedback. However, if we ask the same question in the **GPT-3** playground, we'll get this response:

USER What is LangChain?

ASSISTANT LangChain is a decentralized platform that aims to revolutionize the global language learning industry. It leverages blockchain technology to provide a secure and transparent learning environment, connecting language learners directly with native speakers and qualified tutors. The platform offers various language learning tools, including interactive lessons, practice exercises, and real-time conversations, all facilitated by its decentralized network. LangChain also incorporates a reward system using its native cryptocurrency, enabling learners to earn tokens by achieving language learning milestones and facilitating peer-to-peer transactions within the platform. Overall, LangChain aims to provide an efficient, affordable, and engaging language learning experience for users worldwide.

Figure 1.2: OpenAI playground with GPT 3.5 - Hallucination.

In this case, we can see that the model makes up the term and invents a decentralized platform by the name. This is a hallucination. It's important to watch out for these problems.This problem can be remedied by accessing external data, such as weather APIs, user preferences, or relevant information from the web, and this is essential for creating personalized and accurate language-driven applications.**LLMs** are proficient at generating text but lack true understanding and reasoning capability. However, they might struggle with logical reasoning. As an example, even advanced **LLMs** perform poorly at high-school level math, and can't perform simple math operations that they haven't seen before.Again, we can illustrate this with a simple demonstration:

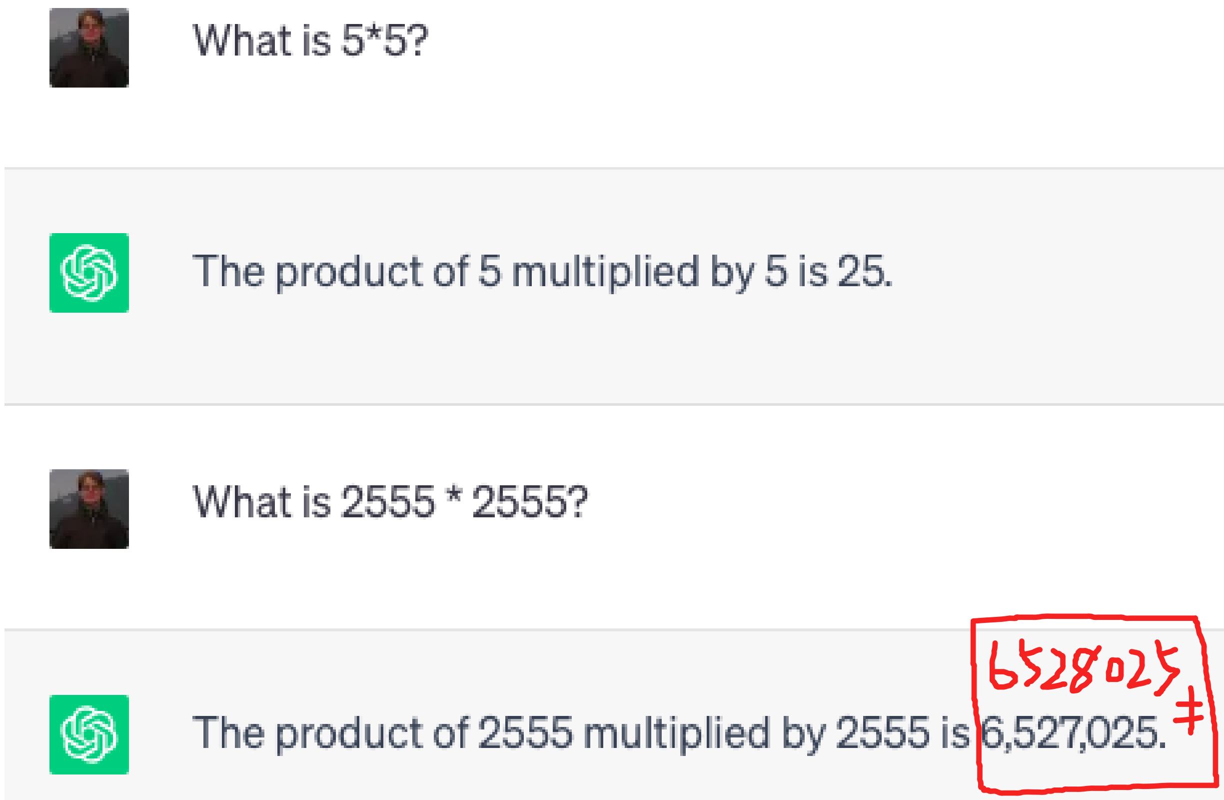
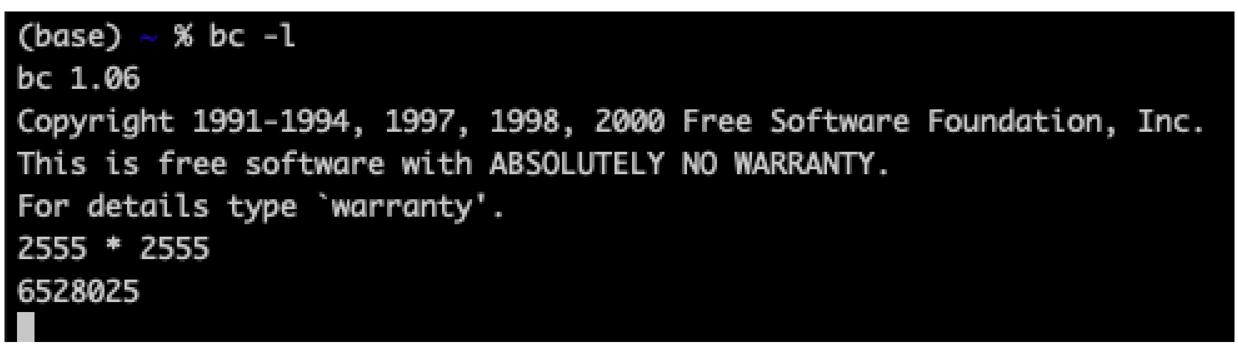


Figure 1.3: ChatGPT math solving.

So, the model comes up with the correct response for the first question, but fails with the second. Just in case if you were wondering what the true result is - if we use a calculator we get this:



```
(base) ~ % bc -l
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
2555 * 2555
6528025
```

Figure 1.4: Multiplication with a Calculator (BC).

The LLM hasn't stored the result of the calculation of hasn't encountered it often enough in the training data for it to be reliably remembered as in encoded in its weights. Therefore, it fails to correctly come up with the solution. A transformer-based LLM is not the suitable tool for the job in this case. The output of **LLMs** might need to be monitored and corrected for accuracy and for bias and inappropriate language before deployment of an app in domains such as customer service, education, and marketing. It's not hard to come up with examples for bias in Chatbots - just recall the Tay Chatbot, which turned a public relations disaster for Microsoft because of racial slurs and other xenophobic comments. For all of these concerns, **LLMs** need to be integrated with external data sources, memory, and capability in order to interact dynamically with their environment and respond appropriately based on the provided data. However, connecting large language models with different data sources and computations can be tricky and specific customized tools need to be developed and carefully tested. As a result, building data-responsive applications with Generative AI can be complex and can require extensive coding and data handling.Finally, working with **LLM** models directly can be challenging and time-consuming. This starts with the prompt engineering, but extends much further. The inherent challenge lies in navigating these sophisticated models, providing prompts that work, and parsing their output.

What's an LLM app?

To address the aforementioned challenges and limitations, **LLMs** can be combined with calls to other programs or services. The main idea is that the ability of LLMs can be augmented through the use of tools by connecting them together. Combining **LLMs** with other tools into applications using specialized tooling, **LLM**-powered applications have the potential to transform our digital world. Often this is done via a chain of one or multiple prompted calls to **LLMs**, but can also make use of other external services (such as APIs or data sources) in order to achieve particular tasks.

> An **LLM app** is an application that uses large language models (LLMs) like ChatGPT to assist with various tasks. It operates by sending prompts to the language models to generate responses, and it can also integrate with other external services, like APIs or data sources, to accomplish specific goals.

In order to illustrate how an **LLM** app can look like, here's a very simple **LLM** app that includes a prompt and an **LLM** (source: https://github.com/srush/MiniChain):

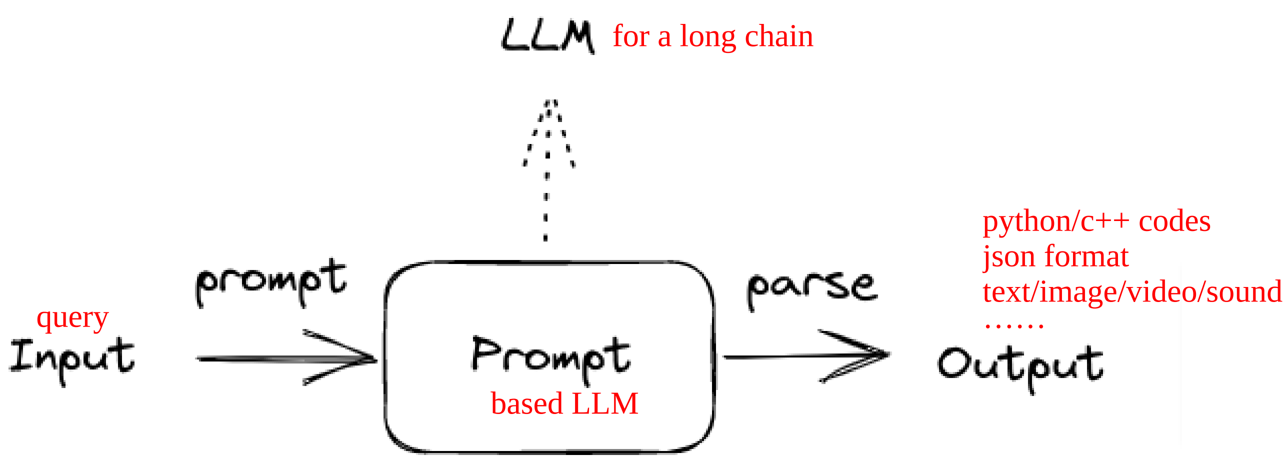


Figure 1.5: A simple LLM app that combines a prompt with an LLM.

LLM apps have significant potential for humans as they enhance our capabilities, streamline processes, and provide valuable assistance in various domains. Here are some key reasons why **LLM** apps are important:

- **Efficiency and Productivity**: **LLM** apps automate tasks, enabling faster and more accurate completion of repetitive or complex operations. They can handle data processing, analysis, pattern recognition, and decision-making with speed and accuracy that surpasses human capacity. This improves efficiency and productivity in areas such as data analysis, customer service, content generation, and more.
- **Task Simplification**: **LLM** apps simplify complex tasks by breaking them down into manageable steps or providing intuitive interfaces for users to interact with. These tools can automate complex workflows, making them accessible to a wider range of users without specialized expertise.
- **Enhanced Decision-Making**: **LLM** apps offer advanced analytics capabilities that enable data-driven decision-making. They can analyze large volumes of information quickly, identify trends or patterns that may not be apparent to humans alone, and provide valuable insights for strategic planning or problem-solving.
- **Personalization**: AI-powered recommendation systems personalize user experiences based on individual preferences and behavior patterns. These apps consider user data to provide tailored suggestions,

recommendations, and personalized content across various domains like e-commerce, entertainment, and online platforms.

A particular area of growth is the usage of company data, especially customer data, with **LLMs**. However, we have to be careful and consider implications for privacy and data protection. We should never feed **personally identifiable** (**PII**) data into public API endpoints. For these use cases, deploying models on in-house infrastructure or private clouds is essential, and where fine-tuning and even training specialized models provide important improvements. This is what we'll talk about in chapter 9, *LLM Apps in Production*.Let's compare a few frameworks that can help to build **LLM** apps.

API ---- deployed in houses/private cloud ---- fine-tune ---- learning from scratch

Framework Comparison

LLM application frameworks have been developed to provide specialized tooling that can harness the power of **LLMs** effectively to solve complex problems. A few libraries have emerged meeting the requirements of effectively combining generative AI models with other tools to build **LLM** applications.There are several open-source frameworks for **building** dynamic **LLM** applications. They all offer value in developing cutting-edge LLM applications. This graph shows their popularity over time (data source: github star history; https://star-history.com/):

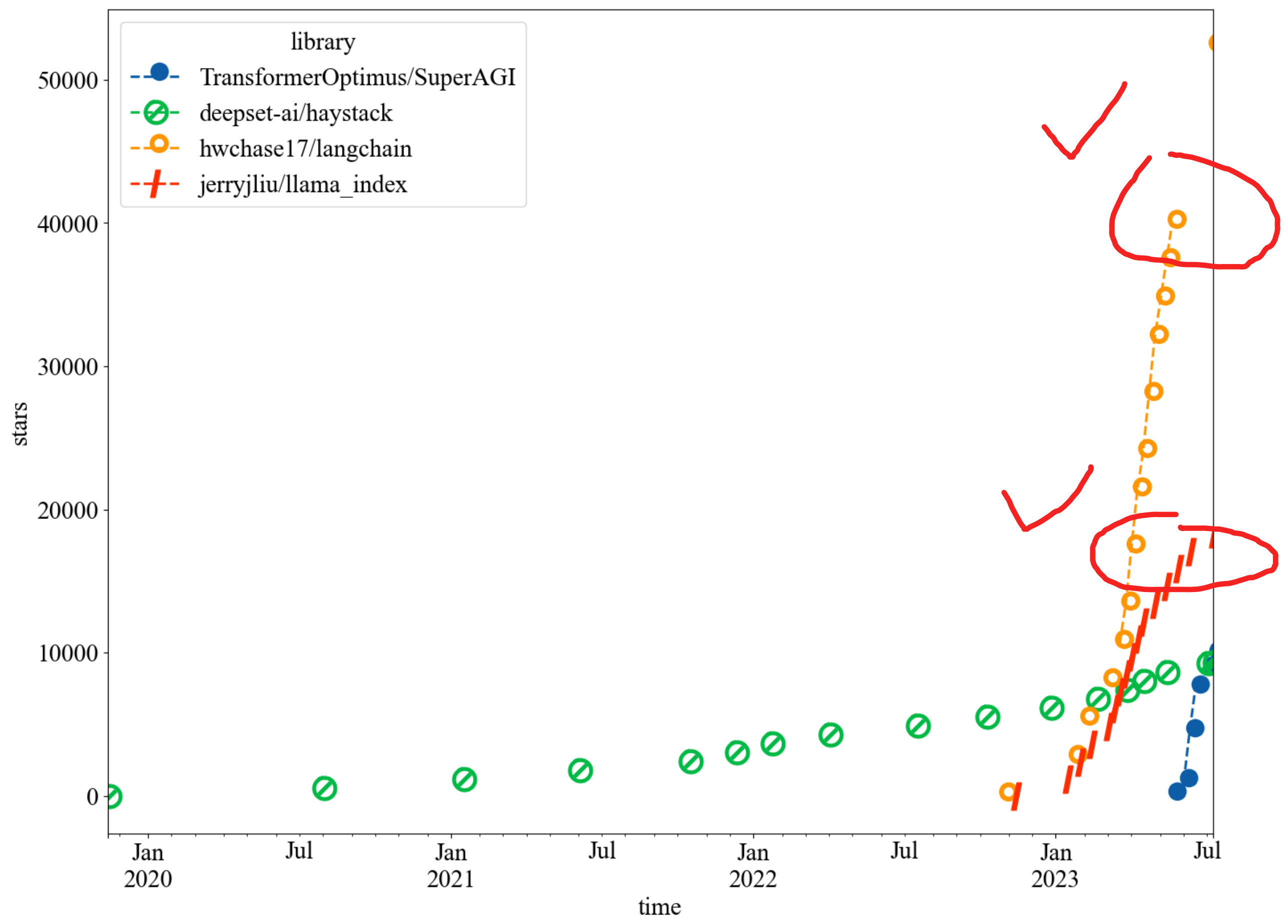


Figure 1.6: Comparison of popularity between different framework in Python. We can see the number of stars on github over time for each project.

We can see in the chart that Haystack is the oldest of the compared frameworks having been started early 2020 (as per github commits). It is also the least popular in terms of stars on github. Langchain, **LlamaIndex** (previously called GPTIndex), and **SuperAGI** were started late 2022 or early 2023, and they have all short to popularity in a very short time with **LangChain** growing most impressively. In this book, we'll see why its popularity is exploding right now.**LlamaIndex** focuses on advanced retrieval rather than on the broader aspects of **LLM** apps. Similarly, Haystack focuses on creating large-scale search systems with components designed specifically for scalable information retrieval using retrievers, readers, and other data handlers combined with semantic indexing via pre-trained models.**LangChain** excels at chaining **LLMs** together using agents for delegating actions to the models. Its use cases emphasize prompt optimization and context-aware information retrieval/generation, however with its Pythonic highly modular interface and its huge collection of tools, it is the number one tool to implement complex business logic.**SuperAGI** has similar features to **LangChain**. It even comes with a Marketplace, a repository for tools and agents. However, it's not as extensive and well-supported as **LangChain**.I haven't included **AutoGPT** (and similar tools like **AutoLlama**), a recursive application that breaks down tasks, because its reasoning capability, based on human and LLM feedback, is very limited compared to **LangChain**. As a consequence, it's often caught in logic loops and often repeats steps. I've also omitted a few libraries that concentrate on prompt engineering, for example Promptify. There are other LLM app frameworks in languages such as Rust, Javascript, Ruby, and Java. For example, Dust, written in Rust, focuses on the design of LLM Apps and their deployment.Let's look a bit more at **LangChain**.

What is LangChain?

LangChain is a framework for developing applications powered by language models and enables users to build applications using **LLMs** more effectively. It provides a standard interface for connecting language models to other sources of data, as well as for building agents that can interact with their environment. LangChain is designed to be modular and extensible, making it easy to build complex applications that can be adapted to a variety of domains. LangChain is open source, and is written in Python, although companion projects exist implemented in JavaScript or - more precisely - Typescript (LangChain.js), and the fledgling Langchain.rb project for Ruby, which comes with a Ruby interpretor for code execution. In this book, we focus on the Python flavor of the framework.

> **LangChain** is an open-source framework that allows AI developers to combine LLMs like ChatGPT with other sources of computation and information.

Started in October 2022 by Harrison Chase as an open-source project on github, it is licensed under the MIT license, a common license, which allows commercial use, modification, distribution, and private use, however, restricts liability and warranty. LangChain is still quite new, however, it already features 100s of integrations and tools. There are active discussions on a discord chat server, there's blog, and regular meetups are taking place in both San Francisco and London. There's even a Chatbot, ChatLangChain, that can answer questions about the LangChain documentation built with with LangChain and FastAPI, which is available online through the documentation website!The project has attracted millions in venture capital funding from the likes of Sequoia Capital and Benchmark, who provided funding to Apple, Cisco, Google, WeWork, Dropbox, and many other successful companies. LangChain comes with many extensions and a larger ecosystem that is developing around it. As mentioned, it has an immense number of integrations already, with many new ones every week. This screenshot showcases a few of the integrations (source: `integrations.langchain.com/trending`):

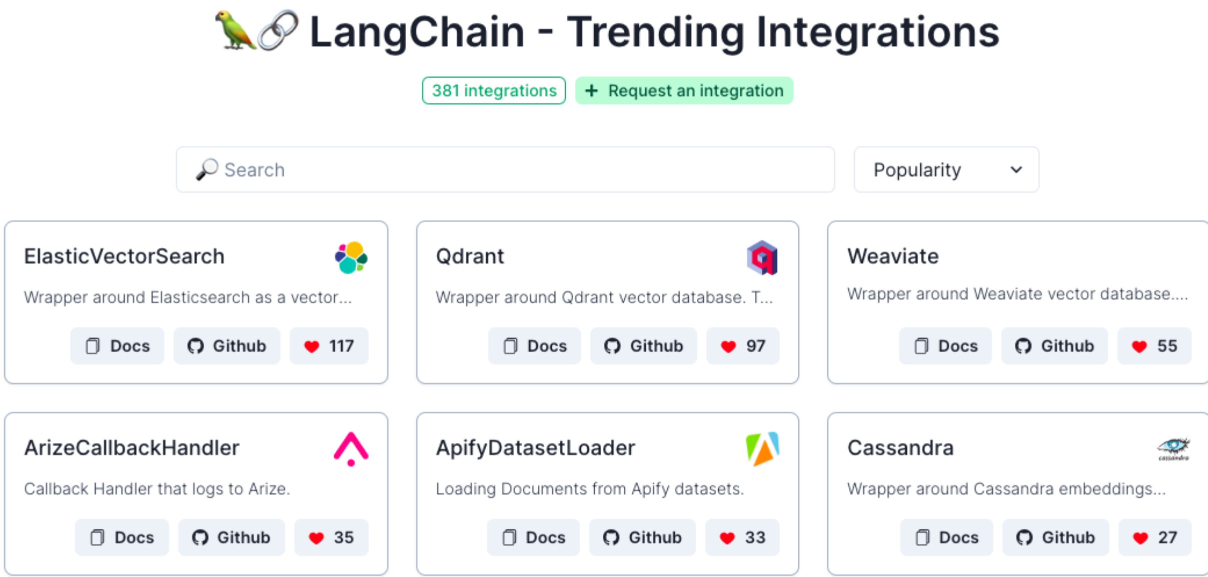


Figure 1.7: LangChain integrations.

For example, LangChainHub is a repository of artifacts that are useful for working with **LangChain** such as prompts, chains and agents, which combine together to form complex LLM applications. Taking inspiration from HuggingFace Hub, which is a collection of models, it aims repository is to be a central resource for sharing and discovering high quality **LangChain** primitives and applications. Currently, the hub solely contains a collection of prompts, but - hopefully - as the community is adding to this collection, you might be able to find chains and agents soon.Further, the **LlamaHub** library extends both LangChain and LlamaIndex with more data loaders and readers for example for **Google Docs**, **SQL Databases**, **PowerPoints**, **Notion**, **Slack**, and **Obsidian**. **LangFlow** is a UI, which allows 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.This is illustrated in the following screenshot of a basic chat pipeline with a prompt template and a conversation buffer as a memory:

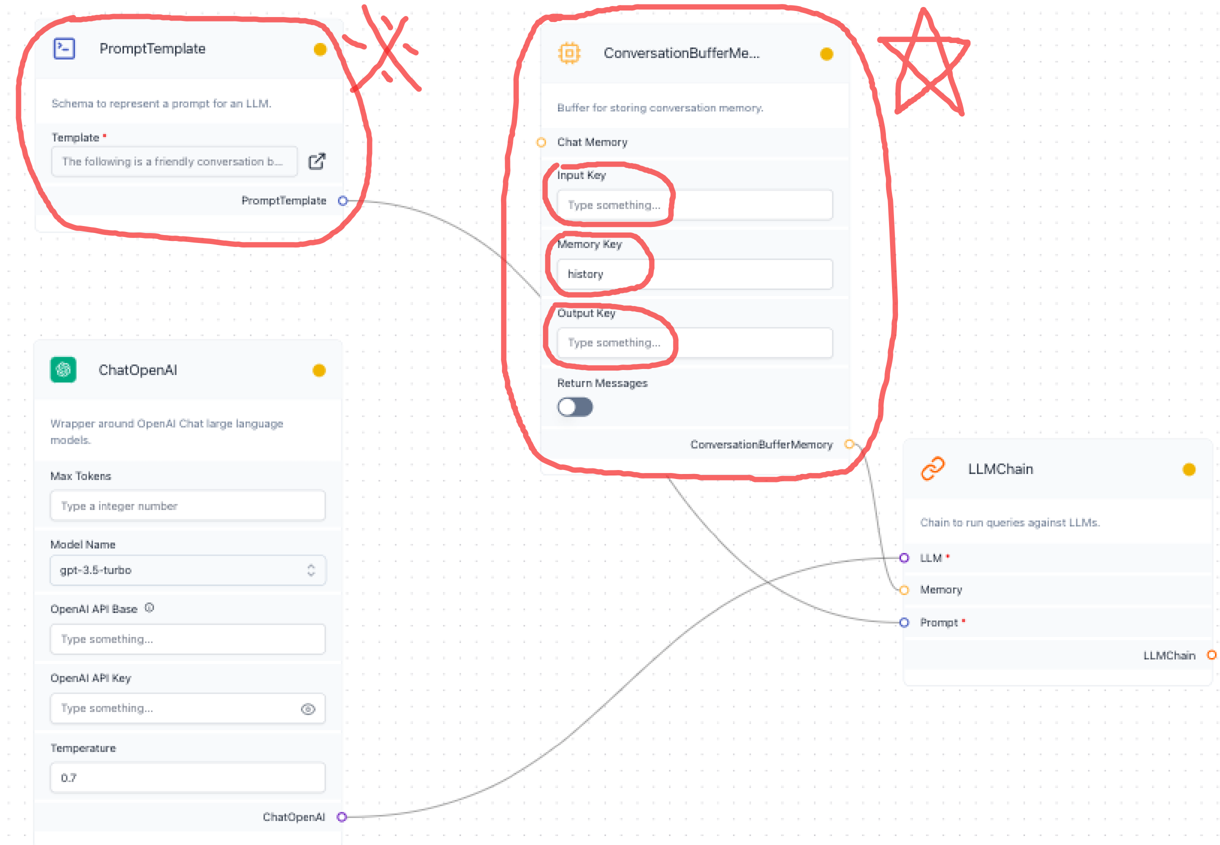


Figure 1.8: LangFlow UI with a basic chat.

In the sidebar of the browser interface (not shown here), you can see all the different **LangChain** components like a zero-shot prompt, data loaders, and language model wrappers. These flows can be either exported and loaded up in **LangChain** directly, or they can be called through API calls to the local server.**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 **Jina AI cloud** as LLM Apps as-a-service with a single command.**LangChain** provides an intuitive framework that makes it easier for developers, data scientists, and even those new to NLP technology to create applications using large language models. It's important to note that **LangChain** is neither a model nor a provider but essentially a framework that facilitates seamless interaction with diverse models. With **LangChain**, you don't need to be an expert in AI or complex algorithms — it simplifies the process and reduces the learning curve.

> Please note that although the main focus of LangChain is LLMs, and this is largely what we'll talk about in this book, there are also integrations for image generation.

By being data-aware and agentic, **LangChain** allows for easy integration with various data sources, including **Google Drive**, **Notion**, **Wikipedia**, **Apify Actors**, and more. This data-awareness enables applications to generate personalized and contextually relevant responses based on user preferences or real-time information from external sources.Let's explore why **LangChain** is important and then what it is used for.

Why is LangChain relevant?

LangChain fills a lot of the needs that we outlined before starting with the limitations of **LLMs** and the emergence of **LLM** apps. Simply put, it simplifies and streamlines the development process of applications using **LLMs**.It provides a way to build applications that are more powerful and flexible than those that can be built by simply calling out to a language model via an API. Particularly, **LangChain's** support for agents and memory allows developers to build applications that can interact with their environment in a more sophisticated way, and that can store and reuse information over time.**LangChain** can be used to improve the performance and reliability of applications in a variety of domains. In the healthcare domain, it can be used to build chatbots that can answer patient questions and provide medical advice. In this context, we have to be very careful with regulatory and ethical constraints around reliability of the information and confidentiality. In the finance domain, the framework can be used to build tools that can analyze financial data and make predictions. Here, we have to look at considerations around interpretability of these models. In the education domain, **LangChain** can be used to build tools that can help students learn new concepts. This is possibly one of the most exciting domains, where complete syllabi can be broken down by LLMs and delivered in customized interactive sessions, personalized to the individual learner.**LangChain's** versatility allows it to be used in several dynamic ways like building virtual personal assistants capable of recalling previous interactions; extracting analyzing structured datasets; creating Q&A apps providing interaction with APIs offering real-time updates; performing code understanding extracting interacting source codes from GitHub enriching developer experiences robustly enhanced codified performances.There are several benefits to using **LangChain**, including:

- **Increased flexibility**: It provides a wide range of tools and features for building powerful applications. Further, it's modular design makes it easy to build complex applications that can be adapted to a variety of domains.
- **Improved performance**: The support for action plan generation can help to improve the performance of applications.
- **Enhanced reliability**: LangChain's support for memory can help to improve the reliability of applications by storing and reusing information over time, and - by access to external information - it can reduce hallucinations.
- **Open source**: An open business-friendly license coupled with a large community of developers and users means that you can customize it to your needs and rely on broad support.

In conclusion: there are many reasons to use **LangChain**. However, I should caution that since **LangChain** is still quite new, there might be some bugs or issues that have not yet been resolved. The documentation is already relatively comprehensive and big, however, in construction in a few places.

What can I build with LangChain?

LangChain empowers various NLP use cases such as virtual assistants, content generation models for summaries or translations, question answering systems, and more. It has been used to solve a variety of real-world problems. For example, **LangChain** has been used to build chatbots, question answering systems, and data analysis tools. It has also been used in a number of different domains, including healthcare, finance, and education.You can build a wide variety of applications with **LangChain**, including:

- **Chatbots**: It can be used to build chatbots that can interact with users in a natural way.
- **Question answering**: **LangChain** can be used to build question answering systems that can answer questions about a variety of topics.
- **Data analysis**: You can use it for automated data analysis and visualization to extract insights.
- **Code generation**: You can set up software pair programming assistants that can help to solve business problems.
- And much more!

How does LangChain work?

With **LangChain**, you can build dynamic applications that harness the power of recent breakthroughs in **natural language processing** (**NLP**). By connecting components from multiple modules (chaining), you can create unique applications tailored around a large language model. From sentiment analysis to chatbots, the possibilities are vast.The principal value proposition of the LangChain framework consists of the following parts:

- **Components**:
  - **Model I/O**: This component provides LLM wrappers as a standardized interface for connecting to a language model.

- **Prompt Templates**: This allows you to manage and optimize prompts.
- **Memory**: Indexes are used to store and reuse information between calls of a chain/agent.

- **Agents**: Agents allow LLMs to interact with their environment. They decide the actions to take and take the action.
- **Chains**: These assemble components together in order to solve tasks. They can be comprised of sequences of calls to language models and other utilities.

Here's a visual representation of these parts:

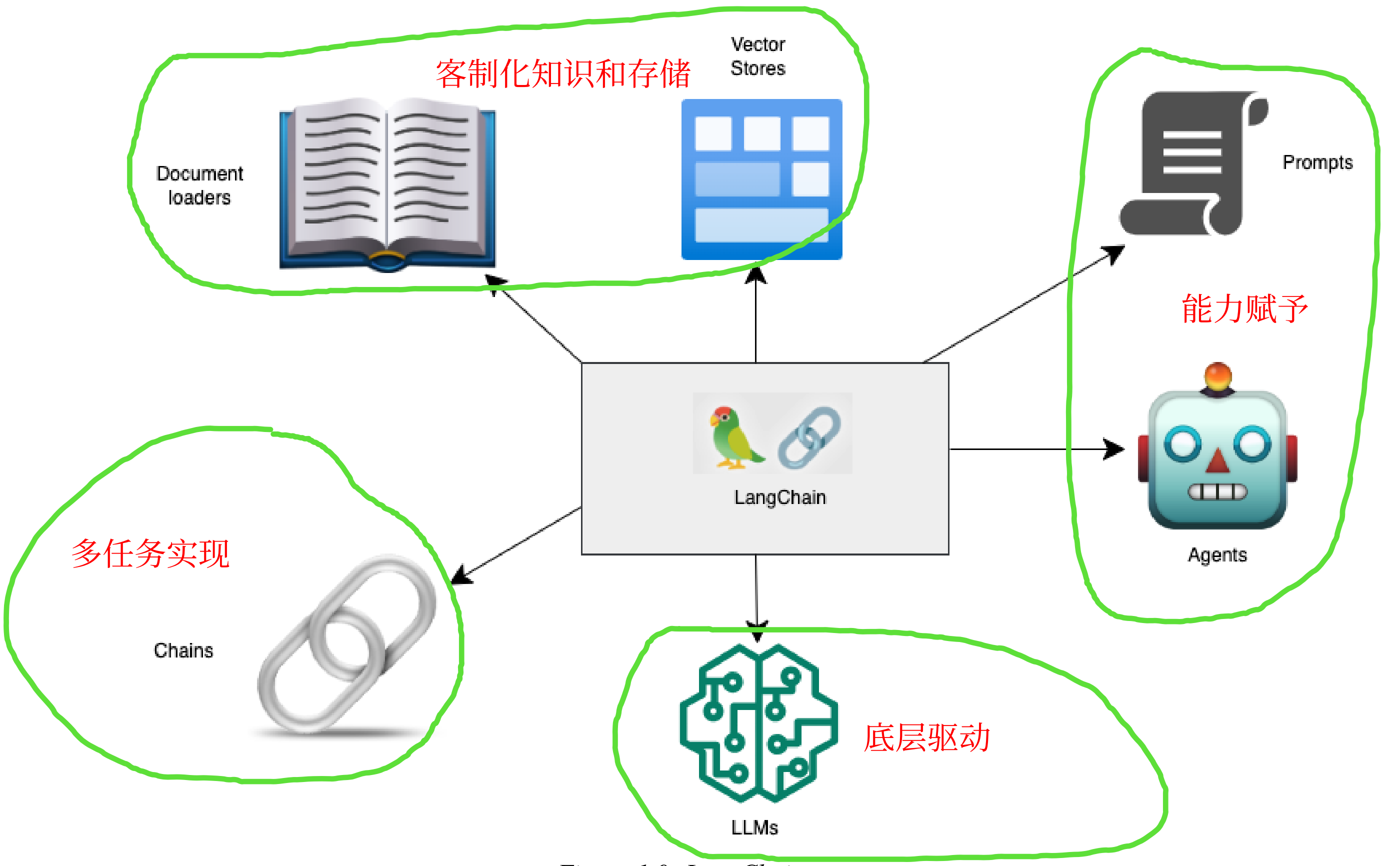


Figure 1.9: LangChain components.

There's a lot to unwrap about these parts. Let's go into a bit of detail!Although **LangChain** doesn't supply models itself, it supports integration through **LLM** wrappers with various different language model providers enabling the app to interact with chat models as well text embedding model providers. Supported providers include **OpenAI**, HuggingFace, Azure, and Anthropic. Providing a standardized interface, means being able effortlessly swap out models in order to save money and energy or get better performance. 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 the input and the available collection of prompts are battle-tested in a range of applications. 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 representation 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.Some other modules are these:

1. Alibaba Cloud OpenSearch
2. Chroma
3. Facebook AI Similarity Search
4. Pinecone

- **Data connectors and loaders**: These components provide interfaces for connecting to external data sources.
- **Callbacks**: Callbacks are used to log and stream intermediate steps of any chain.

Data connectors 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 Doc (docx), HyperText Markup Language (HTML), and other common formats such as PDF, text files, JSON, and CSV. Other tools will send emails to prospective customers, tweet funny puns to your followers, or send slack messages to your coworkers.Let's see a bit more in detail, what agents can be good for and how they make their decisions.

What is an agent?

Agents are used in **LangChain** to control the flow of execution of an application to interact with users, the environment, and other agents. Agents can be used to make decisions about which actions to take, to interact with external data sources, and to store and reuse information over time. Agents can transfer money, book flights, or talk to your customers.

> An **agent** is a software entity that can perform actions and tasks in the world and interact with its environment. In **LangChain**, agents take tools and chains and combine them for a task taking decisions on which to use.

Agents can establish a connection to the outside world. For example, a search engine or vector database can be utilized to find up-to-date and relevant information. This information can then be provided to models. This is called **retrieval augmentation**. By integrating external information sources, **LLMs** can draw from current information and extended knowledge. This is an example of how agents can overcome the weaknesses inherent in **LLMs** and enhance them by combining tools with the models.In the section about the limitations of **LLMs** we've seen that 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 connecting an OpenAI language model output to a Python function here:

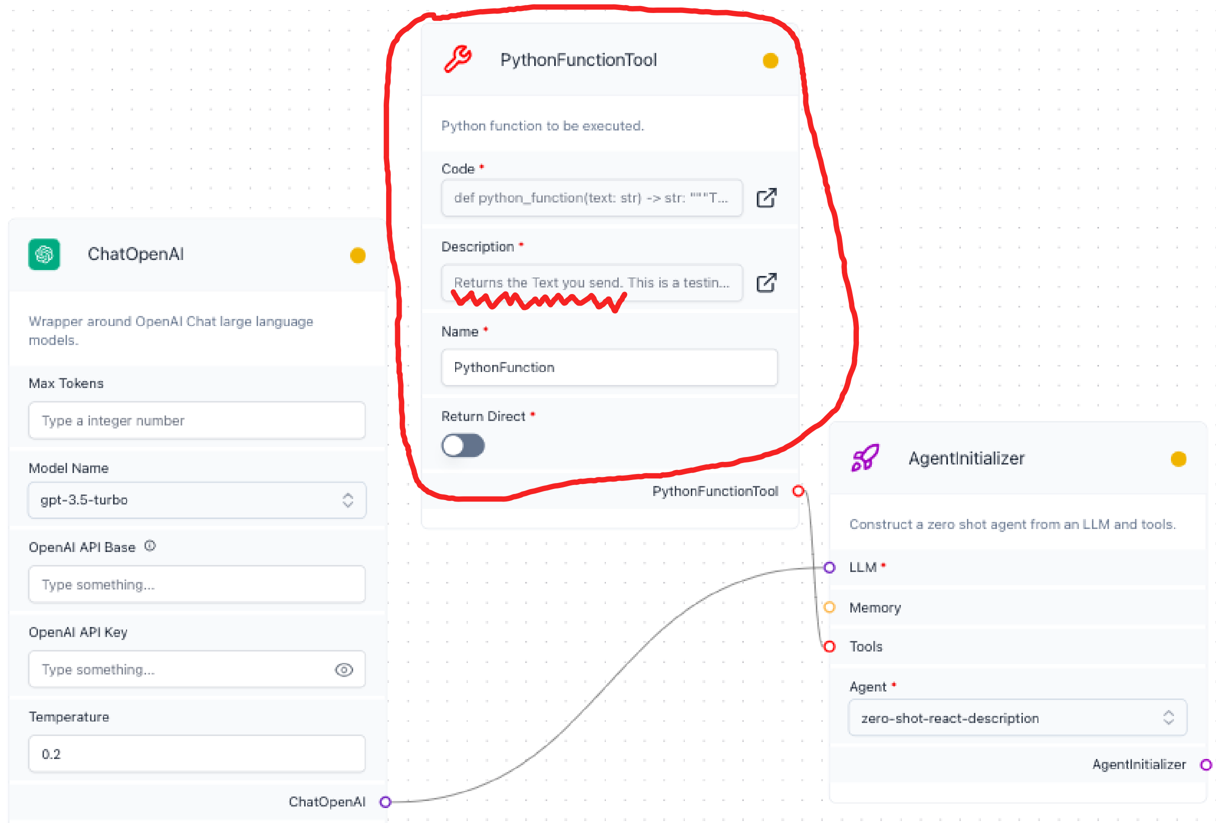


Figure 1.10: A simple LLM app with a Python function visualized in LangFlow.

We will see this in practice in *Chapter 3*, *Getting Started with LangChain*. Agents in **LangChain** can be used to perform a variety of tasks, such as, for example:

- Searching for information
- Calling APIs
- Accessing databases
- Code execution

Each agent can decide on which tool to use and when. Since this is crucial for understanding how **LangChain** works, let's see this in a bit of detail.

Action execution

Each agent is equipped with these subcomponents:

- Tools, which are functional components,
- Toolkits (these are collections of tools), and
- Agent Executors.

Toolset → Skill set → robot motions
Toolset → Tool set → perceptions

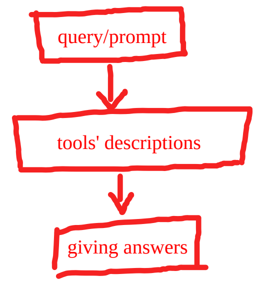
The **agent executor** is the execution mechanism that allows choosing between tools. The agent executor can be seen as the intermediary between the agent and the execution environment. It receives the requests or commands from the agent and translates them into actions that can be performed by the underlying system or software. It manages the execution of these actions and provides feedback or results back to the agent. We have different types of execution or decision patterns as we'll see.The **ReAct pattern** (published as "**ReACT**: Synergizing Reasoning and Acting in Language Models" by researchers at Princeton and Google DeepMind, May 2023), short for Reason and Act, where the agent actively assigns a task to an appropriate tool, customizes input for it, and parses its output in order to

事实上，agent和agent-executors可以是一样的

commands requests | actions → environments
feedbacks results | obser-vation ← environments

resolve the task. In the paper, a document store was utilized, where answers would be searched - this is implemented as the **ReAct document store pattern**. In **LangChain**, by default, agents follow the **Zero-shot ReAct pattern** (`ZERO_SHOT_REACT_DESCRIPTION`), where the decision is based only on the tool's description. This mechanism can be extended with memory in order to take into account the full conversation history. With **ReAct**, instead of asking an **LLM** to autocomplete on your text, you can prompt it to respond in a thought/act/observation loop. The prompt for the **LLM** is to respond step by step and associating actions with these steps. The result from these steps, for example search results, is then passed back into the **LLM** for its next deliberation as it iterates towards its goal. For the ZeroShot pattern, the prompt is really important, which is created from joining prefix, a string describing the tools and what they are good for, the format instructions, and the suffix:

```
PREFIX = """Answer the following questions as best you can. You have access to the following
FORMAT_INSTRUCTIONS = """Use the following format:
Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question"""
SUFFIX = """Begin!
Question: {input}
Thought:{agent_scratchpad}"""
```



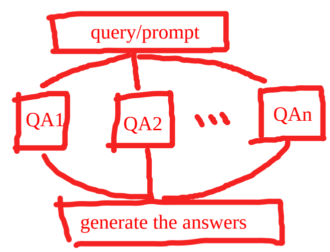
To see this in practice, for example, we can ask for the difference between **LangChain** agent executor and **LangChain** execution plan. Here's the log in **LangChain** - first the question goes to the language model:

```
I'm not familiar with these terms, so I should search for information about them.
Action: Search
Action Input: "difference between langchain agent executor and langchain execution plan"
Observation: The Concept of Agents in LangChain Action Agents decide an action to take and ex
Thought:Based on the observation, a langchain agent executor is an agent that decides and exe
Final Answer: A langchain agent executor executes actions one step at a time, while a langcha
```

There are a few more implemented mechanisms. Researchers at the University of Washington, **MosaicAI**, Meta AI Research, and Allen Institute (in the paper "Measuring and Narrowing the Compositionality Gap in Language Models" by in October 2022) found that **LLMs** might often not come up with the correct and complete answer for questions that require compositional reasoning, where multiple pieces of information have to be put together. The **self-ask with search** pattern decomposes a question into constituents and calls a search engine method in order to retrieve the necessary information in order to answer questions. An example for this powerful mechanism is discussed on LangChain's github by user nkov. The question is how lived longer, Muhammad Ali or Alan Turing, and the conversation develops thus:

```
Question: Who lived longer, Muhammad Ali or Alan Turing?
Are follow up questions needed here: Yes.
Follow up: How old was Muhammad Ali when he died?
Intermediate answer: Muhammad Ali was 74 years old when he died.
Follow up: How old was Alan Turing when he died?
Intermediate answer: Alan Turing was 41 years old when he died.

So the final answer is: Muhammad Ali
```



In each step, the **LLM** decides if follow-up searches are needed and this information is fed back to the **LLM**. Recently, OpenAI models (gpt-3.5-turbo-0613, gpt-4-0613) have been fine-tuned to detect when **function calls** should be executed and which input should be fed into the functions. For this to work, functions can also be described in API calls to these language models. This is also implemented in **LangChain**. There are a few strategies that are not (yet) implemented as execution mechanism in **LangChain**:

- **Recursively Criticizes and Improves** its output (**RCI**) methods ("Language Models can Solve Computer Tasks"; Kim and others, June 2023) use **LLM** as a planner to construct an agent, where the former uses an **LLM** to generate thoughts before executing the action, whereas the latter prompts an LLM to think up lessons learned for improving subsequent episodes.

- The **Tree of Thought** (**ToT**) algorithm (published as "Tree of Thoughts: Deliberate Problem Solving with Large Language Models" in May 2023 by researchers at Princeton and Google DeepMind) advances model reasoning by traversing a search tree. Basic strategies can be depth-first or breadth-first tree traversal, however many others can and have been tested such as Best First, Monte Carlo, and A*. These strategies have been found to significantly improve the success rate at problem solving.

These decisions can be planned out ahead or can be taken at each step. This process of creating a sequence of actions that an agent can take to achieve a goal is called the **action plan generation**. There are two different types of agents by action plan generation, which can be chosen based on the required dynamism of the task:

- **Action agents** decide at each iteration on the next action based on the outputs of all previous actions.
- **Plan-and-execute agents** decide on the full plan of actions at the start. They then execute all these actions without updating the sequence. This implementation in **LangChain** was inspired by **BabyAGI**.

Generally, action agents are more flexible, while plan-and-execute agents are better at maintaining long-term objectives. If we want to be as flexible as possible we can specify a Zero-shot **ReAct** mechanism for our agent to make decisions at every turn.Let's have a look at chains now!

What's a chain?

The core idea in **LangChain** is the compositionality of **LLMs** and other components to work together. For examples, users and developers can put together multiple **LLM** calls and other components in a sequence to create complex applications like chatbot-like social interactions, data extraction, and data analysis.

> In most generic terms, a **chain** is as a sequence of calls to components, which can include other chains.

For example, prompt chaining is a technique that can be used to improve the performance of LangChain applications. Prompt chaining involves chaining together multiple prompts to autocomplete a more complex response.Simply put, both chains and agents are wrappers around components. Both can also extend the functionality of LLMs by enabling them to interact with external systems and gather up-to-date information. This modularization of the applications into building blocks like chains and agents can make it easier to debug and maintain them.The most innocuous example for a chain is probably the `PromptTemplate` , which passes a formatted prompt ~~response~~ to a language model. More interesting examples for chains include `LLMMath` for math-related queries and `SQLDatabaseChain` for querying databases. These are called **utility chains**, because they combine language models with specific tools. A few chains can make autonomous decision. Similar to agents, router chains can make decisions on which tool from a selection to use based on their descriptions. A `RouterChain` can dynamically select which retrieval system such as prompts or indexes to use.**LangChain** implements chains to make sure the content of the output is not toxic or otherwise violates OpenAI's moderation rules (`OpenAIModerationChain`) or that it conforms to ethical, legal, or custom principles (`ConstitutionalChain`).The LLMCheckerChain can prevent hallucinations and reduce inaccurate responses by verifying assumptions underlying provided statements and questions. In a paper by researchers at Carnegie Mellon, Allen Institute, University of Washington, NVIDIA, UC San Diego, and Google Research in May 2023 ("SELF-REFINE: Iterative Refinement with Self-Feedback) this strategy has been found to improve task performance by about 20% absolute on average across a benchmark including dialogue responses, math reasoning, and code reasoning. Let's have a look at the memory strategies!

What is memory?

LLMs and tools are stateless in the sense that they don't retain any information about previous responses and conversations. Memory is a key concept in LangChain and can be used to improve the performance of LangChain applications by storing the results of previous calls to the language model, the user, the state of the environment that the agent is operating in, and the agent's goals. This can help to reduce the number of times that the language model needs to be called and can help to ensure that the agent can continue to operate even if the environment changes.

> **Memory** is a data structure that is used to store and reuse information over time.

Memory helps provide context to the application and can make the LLM outputs more coherent and contextually relevant. For example, we can store all the conversation (`ConversationBufferMemory`) or use a buffer to retain the last messages in a conversation using the `ConversationBufferWindowMemory` . The recorded messages are included in the model's history parameter during each call. We should note however, that this will increase the token usage (and therefore API fees) and the latency of the responses. It could also affect the token limit of the model. There is also a conversation summary memory strategy, where an LLM is used to summarize the conversation history - this might incur extra costs for the additional API calls.There are a few exciting nuances about these memory options. For example, an interesting feature is that the conversation with the LLM can be encoded as a Knowledge Graph (`ConversationKGMemory`), which can be integrated back into prompts or used to predict responses without having to go to the LLM.

> A **knowledge graph** is a representation of data that uses a graph-structured data model to integrate data typically in the shape of triplets, a subject, a predicate, and an object, for example subject=Sam, predicate=loves, object=apples. This graph stores information about entities like people, places, or events), and the connections between them.

In summary, memory in **LangChain** can be used to store a variety of information, including:

- The results of previous calls to the language model
- The state of the environment that the agent is operating in
- The goals that the agent is trying to achieve.

Now, we'll have a look at the different tools at our disposal.

What kind of tools are there?

Tools are components in **LangChain** that can be combined with models to extend their capability. **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 of you can do with tools:

- **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 word problems. The calculator supports basic arithmetic operations, allowing the model to accurately solve mathematical queries in datasets specifically designed for math problem-solving.
- **Map**: By connecting with 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 frames.
- **Stock**: 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 python-pptx library or image retrieval from the internet based on given topics. These tools facilitate tasks related to slide creation required in various professional fields.
- **Table Processing**: APIs built with pandas DataFrame 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 three-dimensional (3D) models using a sophisticated 3D rendering engine; Chemical Properties, assisting in resolving scientific inquiries about chemical properties using APIs like PubChem; Database tools facilitating natural language access to database data for executing SQL queries and retrieving results.These various tools provide language models with additional functionalities and capabilities to perform tasks beyond text processing. By connecting with these tools via APIs, language models can enhance their abilities in areas such as translation, math problem-solving, location-based queries, weather forecasting, stock market analysis, slides creation, table processing and analysis, image generation, text-to-speech conversion and many more specialized tasks. All these tools can give us advanced AI functionality, and there's virtually no limit to tools. We can easily build custom tools to extend the capability of LLMs as we'll see in the next chapter 3. The use of different tools expands the scope of applications for language models and enables them to handle various real-world tasks more efficiently and effectively.Let's summarize!

Summary

In today's world, understanding and processing language accurately is crucial in the development of smart applications and for creating personalized and effective user experiences. Therefore, **large language models** (**LLMs**) are ideally suited to lend that capability to applications. However, as we've discussed in this chapter, standalone **LLMs** have their limitations. If we supplement **LLMs** with tools, we can overcome some of these limitations and greatly augment their performance creating **LLM** applications. This is where LangChain comes in, which is a framework aimed at AI developers to set up applications of agents - these are composed of computing entities such as LLMs and other tools that can perform certain tasks autonomously. We've discussed its important concepts, first of all concepts such as agents and chains.In conclusion, LangChain is a valuable open-source framework for simplifying the development of applications using **large language models** (**LLMs**) from providers and platforms such as OpenAI and Hugging Face among many others. This framework offers immense value in unlocking the power of generative AI. In the following chapters, we'll build on these core principals of **LangChain** by building **LLM** applications. By leveraging **LangChain's** capabilities, developers can unlock the full potential of **LLMs**. In the *Chapter 3*, *Getting Started with LangChain*, we'll implement our first apps with **Langchain**!Let's see if you remember some of the key takeaways from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. What are limitations of LLMs?
2. What are LLM-applications?
3. What is LangChain and why should you use it?
4. What are LangChain's key features?
5. What is an agent in LangChain?
6. What is action plan generation?
7. What is a chain?
8. Why do you need memory for LangChain applications?
9. What kind of tools are available?
10. How does LangChain work?

3 Getting Started with LangChain

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

In this chapter, we'll first set up **LangChain** and the libraries needed for this book giving instructions for common dependency management tools such as **Docker**, **Conda**, **Pip**, and **Poetry**. Then we'll go through model integrations that we can use such as **OpenAI's Chatgpt**, models on Huggingface and Jina AI, and others. We'll introduce, set up, and work with a few providers in turn. We'll get an API key tokens and then do a short practical example. This will give us a bit more context at using **LangChain**, and introduce tips and tricks for using it effectively. As the final part, we'll develop a **LangChain** application, a practical example that illustrate a way that **LangChain** can be applied in a real-world business use case in customer service.The main sections are:

- How to Set Up **LangChain**?
- Model Integrations
- Customer Service Helper

We'll start off the chapter by setting up **LangChain** on your computer.

How to Set Up LangChain?

In this book, we are talking about LangChain. We can install LangChain by simply typing `pip install langchain` from a terminal however, in this book, we'll also be using a variety of other tools and integrations in a few different use cases. In order to make sure, all the examples and code snippets work as intended and they don't just work on my machine, but for anyone installing this, I am providing different ways to set up an environment.There are various approaches to setting up a Python environment. Here, we describe four popular methods for installing related dependencies: Docker, Conda, Pip, and Poetry. In case you encounter issues during the installation process, consult the respective documentation or raise an issue on the Github repository of this book. The different installations have been tested at the time of the release of this book, however, things can change, and we will update the Github readme online to include workarounds for possible problems that could arise.Please find a `Dockerfile` for docker, a `requirements.txt` for pip a `pyproject.toml` for poetry and a `langchain_ai.yml` file for **Conda** in the book's repository at https://github.com/benman1/generative_ai_with_langchainLet's set up our environment starting with Python.

Python installation

Before setting up a Python environment and installing related dependencies, you should usually have Python itself installed. I assume, most people who have bought this book will have Python installed, however, just in cases, let's go through it. You may download the latest version from python.org for your operating system or use your platform's package manager. Let's see this with Homebrew for MacOS and apt-get for Ubuntu.On MacOS, with Homebrew, we can do:

```
brew install python
```

For Ubuntu, with apt-get we can do:

```
sudo apt-get update
sudo apt-get install python3.10
```

> **Tip**: If you are new to programming or Python, it is advised to follow some beginner-level tutorials before proceeding with LangChain and the applications in this book.

An important tool for interactively trying out data processing and models is the Jupyter notebook and the lab. Let's have a look at this now.

Jupyter Notebook and JupyterLab

Jupyter Notebook and JupyterLab are open-source web-based interactive environments for creating, sharing, and collaborating on computational documents. They enable users to write code, display visualizations, and include explanatory text in a single document called a notebook. The primary difference between the two lies in their interface and functionality.

> **Jupyter Notebook** aims to support various programming languages like Julia, Python, and R - in fact, the project name is a reference to these three languages. Jupyter Notebook offers a simple user interface that allows users to create, edit, and run notebooks with a linear layout. It also supports extensions for additional features and customization.

>> **JupyterLab**, on the other hand, is an enhanced version of Jupyter Notebook. Introduced in 2018, JupyterLab offers a more powerful and flexible environment for working with notebooks and other file types. It provides a modular, extensible, and customizable interface where users can arrange multiple windows (for example, notebooks, text editors, terminals) side-by-side, facilitating more efficient workflows.

You can start up a notebook server on your computer from the terminal like this:

```
jupyter notebook
```

You should see your browser opening a new tab with the Jupyter notebook like this:

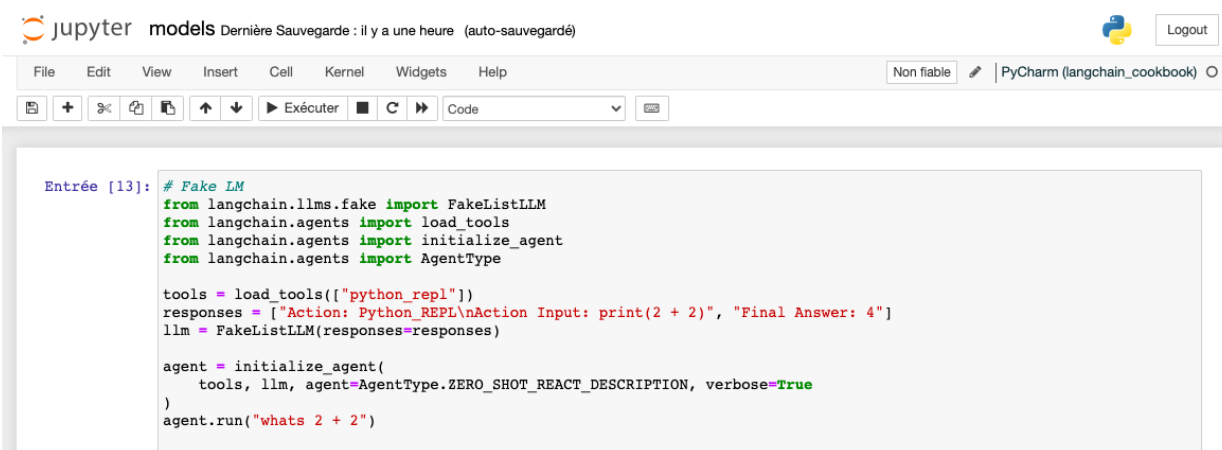


Figure 3.1: Jupyter Notebook with a LangChain Agent.

Alternatively, we can also use JupyterLab, the next-generation notebook server that brings significant improvements in usability. You can start up a JupyterLab notebook server from the terminal like this:

```
jupyter lab
```

We should see something like this:

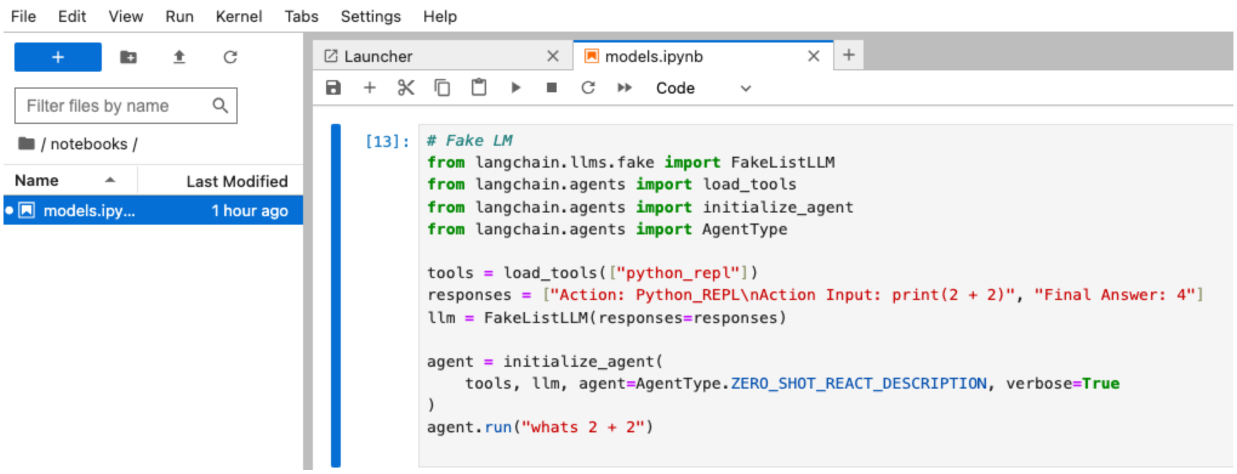


Figure 3.2: Jupyter Lab with a LangChain Agent.

Either one of these two, the `Jupyter notebook` or `JupyterLab`, will give you an **integrated development environment (IDE)** to work on some of the code that we'll be introducing in this book. After installing Python and the notebook or lab, let's quickly explore the differences between dependency management tools (**Docker**, **Conda**, **Pip**, and **Poetry**) and use them to fully set up our environment for our projects with LangChain!

Environment management

Before we explore various methods to set up a Python environment for working with generative models in **LangChain**, it's essential to understand the differences between primary dependency management tools: **Docker**, **Conda**, **Pip**, and **Poetry**. All four are tools widely used in the realm of software development and deployment.

> **Docker** is an open-source platform that provides OS-level virtualization through containerization. It automates the deployment of applications inside lightweight, portable containers, which run consistently on any system with Docker installed.
>
> > **Conda** is a cross-platform package manager and excels at installing and managing packages from multiple channels, not limited to Python. Geared predominantly toward data science and machine learning projects, it can robustly handle intricate dependency trees, catering to complex projects with numerous dependencies.
> >
> > **Pip** is the most commonly used package manager for Python, allowing users to install and manage third-party libraries easily. However, Pip has limitations when handling complex dependencies, increasing the risk of dependency conflicts arising during package installation.
> >
> > **Poetry** is a newer package manager that combines the best features of both Pip and Conda. Boasting a modern and intuitive interface, robust dependency resolution system, and support for virtual environments creation, Poetry offers additional functionalities such as dependency isolation, lock files, and version control.

Poetry and Conda both streamline virtual environment management, whereas working with Pip typically involves utilizing a separate tool like virtualenv. Conda is the installation method recommended here. We'll provide a requirements file for pip as well and instructions for poetry, however, some tweaking might be required in a few cases.We'll go through installation with these different tools in turn. For all instructions, please make sure you have the book's repository downloaded (using the Github user interface) or cloned on your computer, and you've changed into the project's root directory.Here's how you can find the download option on Github:

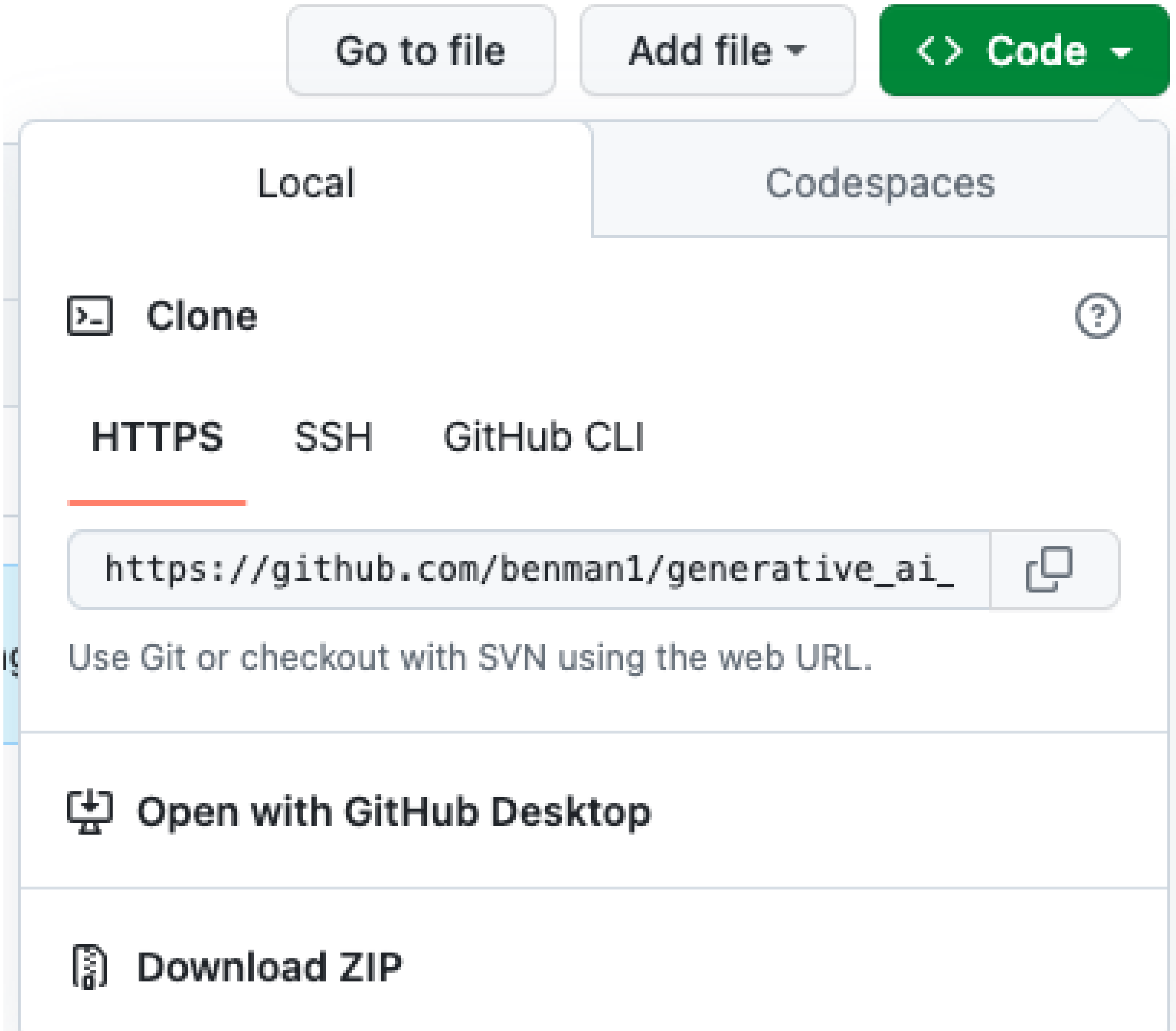


Figure 3.3: Download options in the Github User Interface (UI).

If you are new to git, you can press **Download ZIP**, and then unzip the archive using your favorite tool.Alternatively, to clone the repository using git and change to the project directory, you can type the following commands:

```
git clone https://github.com/benman1/generative_ai_with_langchain.git
cd generative_ai_with_langchain
```

Now that we have the repository on our machine, let's start with Docker!

Docker

Docker is a platform that enables developers to automate deployment, packaging, and management of applications. Docker uses containerization technology, which helps standardize and isolate environments. The advantage of using a container is that it protects your local environment from any - potentially unsafe - code that you run within the container. The downside is that the image might require time to build and might require around 10 Gigabytes in storage capacity.Similar to the other tools for environment management, Docker is useful because you can create a reproducible environment for your project. You can use Docker to create an environment with all the libraries and tools you need for your project, and share that environment with others.To start with Docker, follow these steps:

1. Install Docker on your machine. You can go to the Docker website.in your web browser and follow the installation instructions here: https://docs.docker.com/get-docker/
2. In the terminal, run the following command to build the Docker image (please note: you need to be in the project root for this to work).

```
docker build -t langchain_ai
```

This will pull, the continuumio/miniconda3 image from Docker Hub, and build the image.

1. Start the Docker container interactively using the image created:

```
docker run -it langchain_ai
```

This should start our notebook within the container. We should be able to navigate to the `Jupyter Notebook` from your browser. We can find it at this address: `http://localhost:8080/` Let's look at conda next.

Conda

`Conda` allows users to manage multiple environments for different projects. It works with Python, R, and other languages, and helps with the installation of system libraries as well by maintaining lists of libraries associated with Python libraries.The best way to get started with conda is to install anaconda or miniconda by following the instructions from this link: https://docs.continuum.io/anaconda/install/While the `conda` environment takes up less disk space than Docker, starting from anaconda, the full environment should still take up about 2.5 Gigabytes. The miniconda setup might save you a bit of disk space.There's also a graphical interface to `conda` , Anaconda Navigator, which can be installed on macOS and Windows, and which can install any dependencies as well as the `conda` tool from the terminal.Let's continue with the `conda` tool and install the dependencies of this book.To create a new environment, execute the following command:

```
conda env create --file langchain_ai.yml
```

`Conda` lets us create environments with lots of different libraries, but also different versions of Python. We are using Python 3.10 throughout this book. Activate the environment by running:

```
conda activate langchain_ai
```

This is all, we are done. We can see this should be painless and straightforward. You can now spin up a `jupyter notebook` or `jupyter lab` within the environment, for example:

```
jupyter notebook
```

Let's have a look at pip, an alternative to `conda` .

Pip

`Pip` is the default package manager for Python. It allows you to easily install and manage third-party libraries. We can install individual libraries, but also maintain a full list of Python libraries.If it's not already included in your Python distribution, install pip following instructions on https://pip.pypa.io/To install a library with pip, use the following command. For example, to install the NumPy library, you would use the following command:

```
pip install numpy
```

You can also use `pip` to install a specific version of a library. For example, to install version 1.0 of the NumPy library, you would use the following command:

```
pip install numpy==1.0
```

In order to set up a full environment, we can start with a list of requirements - by convention, this list is in a file called `requirements.txt`. I've included this file in the project's root directory, which lists all essential libraries.You can install all the libraries using this command:

```
pip install -r requirements.txt
```

Please note, however, as mentioned, that Pip doesn't take care of the environments. Virtualenv is a tool that can help to maintain environments, for example different versions of libraries. Let's see this quickly:

```
# create a new environment myenv:
virtualenv myenv
# activate the myenv environment:
source myenv/bin/activate
# install dependencies or run python, for example:
python
# leave the environment again:
deactivate
Please note that in Windows, the activation command is slightly different - you'd run a shell
# activate the myenv environment:
myenv\Scripts\activate.bat
```

Let's do Poetry next.

Poetry

Poetry is a dependency management tool for Python that streamlines library installation and version control. Installation and usage is straightforward as we'll see. Here's a quick run-through of poetry:

1. Install poetry following instructions on https://python-poetry.org/
2. Run `poetry install` in the terminal (from the project root as mentioned before)

The command will automatically create a new environment (if you haven't created one already) and install all dependencies. This concludes the setup for Poetry. We'll get to model providers now.

Model Integrations

Before properly starting with generative AI, we need to set up access to models such as **large language models** (**LLMs**) or text to image models so we can integrate them into our applications. As discussed in *Chapter 1*, *What are Generative Models*, there are various **LLMs** by tech giants, like **GPT-4** by **OpenAI**, **BERT** and **PaLM-2** by **Google**, **LLaMA** by **Meta AI**, and many more.With the help of **LangChain**, we can interact with all of these, for example through **Application Programming Interface** (**APIs**), or we can call open-source models that we have downloaded on our computer. Several of these integrations support text generation and embeddings. We'll focus on text generation in this chapter, and discuss embeddings, vector databases, and neural search in *Chapter 5*, *Building a Chatbot like ChatGPT*.There are many providers for model hosting. For **LLMs**, currently, **OpenAI**, **Hugging Face**, **Cohere**, **Anthropic**, **Azure**, **Google Cloud Platform Vertex AI** (**PaLM-2**), and **Jina AI** are among the many providers supported in **LangChain**, however this list is growing all the time. You can all the supported integrations for **LLMs** at https://integrations.langchain.com/llmsAs for image models, the big developers include **OpenAI** (**DALL-E**), **Midjourney**, Inc. (Midjourney), and Stability AI (**Stable Diffusion**). **LangChain** currently doesn't have out-of-the-box handling of models that are not for text, however, its docs describe how to work with Replicate, which also provides an interface to Stable Diffusion models.For each of these providers, to make calls against their Application Programming Interface (API), you'll first need to create an account and obtain an API key. This is free for all of them. With some of them you don't even have to give them your credit card details.In order to set an API key in an environment, in Python we can do:

```
import os
os.environ["OPENAI_API_KEY"] = "<your token>"
```

Here `OPENAI_API_KEY` is the environment key appropriate for OpenAI. Setting the keys in your environment has the advantage that we don't include them in our code.You can also expose these variables from your terminal like this:

```
export OPENAI_API_KEY=<your token>
```

Let's go through a few prominent model providers in turn. We'll give an example usage for each of them.Let's' start with a Fake LLM that's used for testing so we can show the basic idea!

Fake LLM

The Fake LM is for testing. The LangChain documentation has an example for the tool use with LLMs. You can execute this example in either Python directly or in a notebook.

```
from langchain.llms.fake import FakeListLLM
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
tools = load_tools(["python_repl"])
responses = ["Action: Python_REPL\nAction Input: print(2 + 2)", "Final Answer: 4"]
llm = FakeListLLM(responses=responses)
agent = initialize_agent(
    tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True
)
agent.run("whats 2 + 2")
```

We connect a tool, a Python **Read-Eval-Print Loop** (**REPL**) that will be called depending on the output of the **LLM**. The Fake List **LLM** will give two responses, `responses` , that won't change based on the input. We set up an agent that makes decisions based on the ReAct strategy that we explained in chapter 2, Introduction to LangChain (`ZERO_SHOT_REACT_DESCRIPTION`). We run the agent with a text, the question "what's 2 + 2".We can observe how the Fake LLM output, leads to a call to the Python Interpreter, which returns 4. Please note that the action has to match the `name` attribute of the tool, the `PythonREPLTool` , which is starts like this:

```
class PythonREPLTool(BaseTool):
    """A tool for running python code in a REPL."""
    name = "Python_REPL"
    description = (
        "A Python shell. Use this to execute python commands. "
        "Input should be a valid python command. "
        "If you want to see the output of a value, you should print it out "
        "with `print(...)`."
    )
```

The names and descriptions of the tools are passed to the **LLM**, which then decides based on the provided information.The output of the Python interpreter is passed to the Fake **LLM**, which ignores the observation and returns 4. Obviously, if we change the second response to " `Final Answer: 5` ", the output of the agent wouldn't correspond to the question.In the next sections, we'll make this more meaningful by using an actual **LLM** rather than a fake one. One of the first providers that anyone will think of at the moment is OpenAI.

OpenAI

As explained in *Chapter 1*, *What are Generative Models?*, OpenAI is an American AI research laboratory that is the current market leader in generative AI models, especially LLMs. They offer a spectrum of models with different levels of power suitable for different tasks. We'll see in this chapter how to interact with OpenAI models with **LangChain's** and the OpenAI Python client library. OpenAI also offers an Embedding class for text embedding models.We will mostly use OpenAI for our applications. There are several models to choose from - each model has its own pros, token usage counts, and use cases. The main LLM models are GPT-3.5 and GPT-4 with different token length. You can see the pricing of different models at https://openai.com/pricingWe need to obtain an OpenAI API key first. In order to create an API key, follow these steps:

1. You need to create a login at https://platform.openai.com/
2. Set up your billing information.

3. You can see the **API keys** under *Personal -> View API Keys*.
4. Click **Create new secret key** and give it a **Name**.

Here's how this should look like on the OpenAI platform:

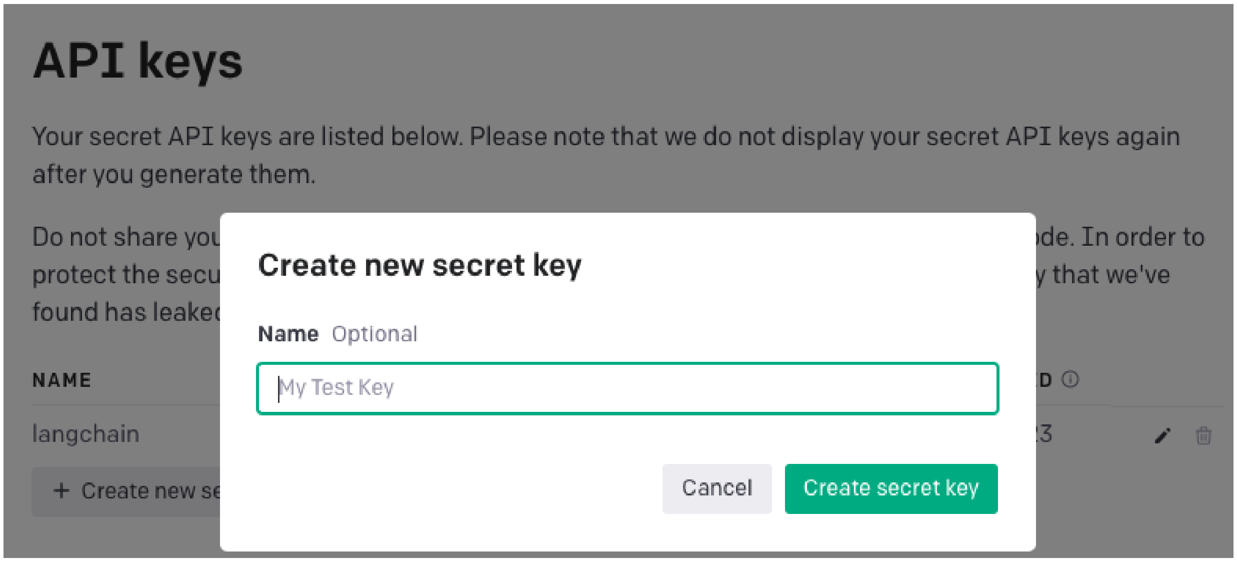


Figure 3.4: OpenAI API platform - Create new secret key.

After pressing "**Create secret key**", you should see the message "API key generated." You need to copy the key to your clipboard and keep it. We can set the key as an environment variable (`OPENAI_API_KEY`) or pass it as a parameter every time you construct a class for OpenAI calls.We can use the `OpenAI` language model class to set up an **LLM** to interact with. Let's create an agent that calculates using this model - I am omitting the imports from the previous example:

```
from langchain.llms import OpenAI
llm = OpenAI(temperature=0., model="text-davinci-003")
agent = initialize_agent(
    tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True
)
agent.run("whats 4 + 4")
```

We should be seeing this output:

```
> Entering new  chain...
 I need to add two numbers
Action: Python_REPL
Action Input: print(4 + 4)
Observation: 8
Thought: I now know the final answer
Final Answer: 4 + 4 = 8
> Finished chain.
'4 + 4 = 8'
```

This looks quite promising, I think. Let's move on to the next provider and more examples!

Hugging Face

Hugging Face is a very prominent player in the NLP space, and has considerable traction in open-source and hosting solutions. The company 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 natural language processing tasks, includes implementations of state-of-the-art and popular models like 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 build ML together.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-transformers models.They 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.In addition to their products, Hugging Face has been involved in initiatives such as the BigScience Research Workshop, where they released an open large language model called BLOOM with 176 billion parameters. They have received significant funding, including a $40 million Series B round and a recent Series C funding round led by Coatue and Sequoia at a $2 billion valuation. 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.In order to use Hugging Face as a provider for your models, you can create an account and API keys at https://huggingface.co/settings/profile 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: https://huggingface.co/google/flan-t5-xxl

```
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)
```

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. We can also get simple recommendations:

软件即服务（SaaS）是一种软件分发模式，
其中云提供商托管应用程序并通过互联网向最终用户提供。
在这种模式下，独立软件供应商（ISV）可以与第三方云提供商签订合同，以托管应用程序。
或者，对于像微软这样的大公司，云提供商可能也是软件供应商。

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 their global data centers for use cases such as writing assistance, summarization, code generation, and semantic search. It provides capabilities such as **software as a service** (**SaaS**), **platform as a service** (**PaaS**), and **infrastructure as a service** (**IaaS**).Authenticating either through Github or Microsoft credentials, we can create an account on Azure under https://azure.microsoft.com/ You can then create new API keys under *Cognitive Services -> Azure OpenAI*. There are a few more steps involved, and personally, I found this process annoying and frustrating, and I gave up. After set up, the models should be accessible through the `AzureOpenAI()` llm class in **LangChain**.

Google Cloud

There are many models and functions available through **Google Cloud Platform** (**GCP**) and Vertex its machine learning platform. Google Cloud provide 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 support for 11 languages with improved accuracy. For models with GCP, you need to have 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 for your project. If you haven't enabled it, you should get a helpful error message pointing you to the right website, where you have to click on "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)
```

能实现纠错

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
```

I've set verbose to True in order to see the reasoning process of the model. It's quite impressive it comes up with the right response even given a misspelling of the name. The step by step prompt instruction is key to the correct answer. There are various models available through Vertex such as these:

Model	Description	Properties
text-bison	Fine-tuned to follow natural language instructions	Max input token: 8,192 Max output tokens: 1,024 Training data: Up to Feb 2023
chat-bison	Fine-tuned for multi-turn conversation	Max input token: 4,096 Max output tokens: 1,024 Training data: Up to Feb 2023 Max turns : 2,500
code-bison	Fine-tuned to generate code based on a natural language description	Max input token: 4,096 Max output tokens: 2,048
codechat-bison	Fine-tuned for chatbot conversations that help with code-related questions	Max input token: 4,096 Max output tokens: 2,048
code-gecko	Fine-tuned to suggest code completion	Max input tokens: 2,048 Max output tokens: 64

Table 3.1: Models available in Vertex Generative AI. You can check out the documentation at https://cloud.google.com/vertex-ai/docs/generative-ai We can also generate code. Let's see if **Code-Bison** model can solve FizzBuzz, a common interview question for entry and mid-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
```

Would you hire code-bison into your team?

Anthropic

Anthropic is an AI startup and public-benefit corporation based in the United States. It was founded in 2021 by former members of OpenAI, including siblings Daniela Amodei and Dario Amodei. The company specializes in developing general AI systems and language models with a focus on responsible AI usage. As of July 2023, Anthropic has raised $1.5 billion in funding. They have also worked on projects such as Claude, an AI chatbot similar to OpenAI's ChatGPT, and have conducted research on the interpretability of machine learning systems, specifically the transformer architecture.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.

Jina AI

Jina AI, founded in February 2020 by Han Xiao and Xuanbin He, 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. Their open-source neural search ecosystem enables businesses and developers to easily build scalable and highly available neural search solutions, allowing for efficient information retrieval. Recently, **Jina AI** launched *Finetuner*, a tool that enables fine-tuning of any deep neural network to specific use cases and requirements.The company has raised a total of $37.5 million in funding through three rounds, with their most recent funding coming from a Series A round in November 2021. Notable investors in **Jina AI** include **GGV Capital** and **Canaan Partners**.You can set up a login under https://cloud.jina.ai/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:

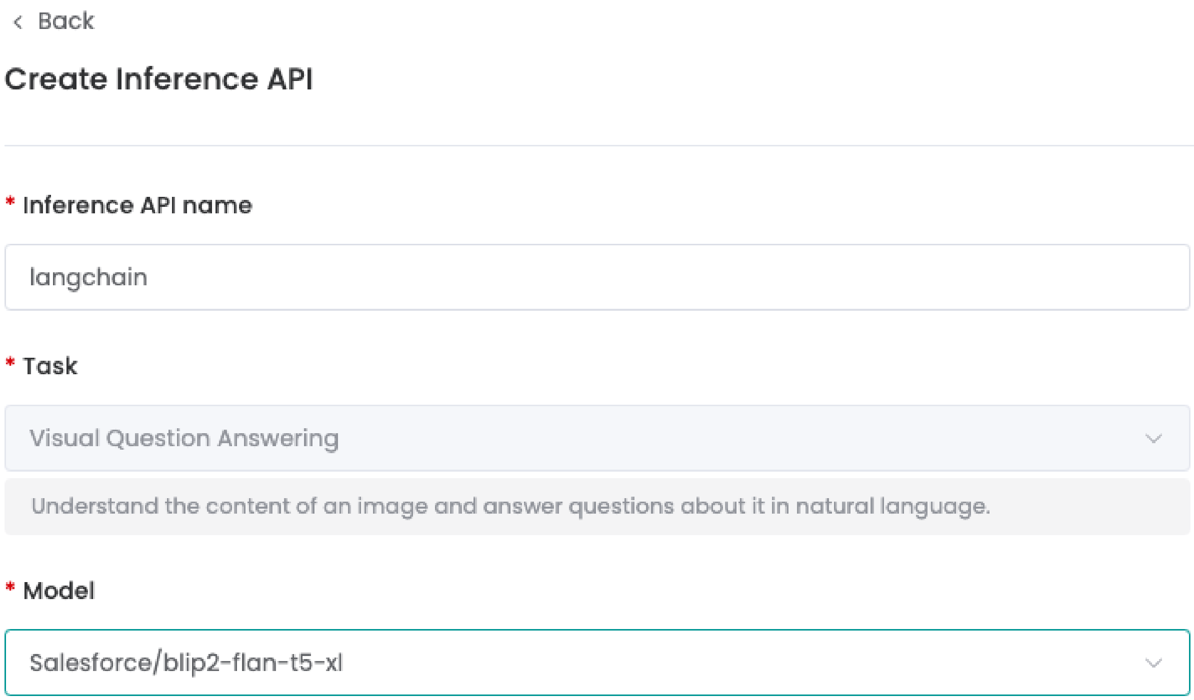


Figure 3.5: Visual Question Answering API in Jina AI.

We get examples for client calls in Python and cURL, and a demo, where we can ask a question. This is cool, 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 `JINA_AUTH_TOKEN` , at https://chat.jina.ai/apiLet'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)
```

```
We should be seeing
```

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

We can set different temperatures, where a low temperature makes the responses more predictable. In this case it makes very little difference. We are starting the conversation with a system message clarifying the purpose of the chatbot.Let's ask for some food recommendations:

```
chat = JinaChat(temperature=0.)
chat(
    [
        SystemMessage(    角色扮演信息
            content="You help a user find a nutritious and tasty food to eat in one word."
        ),
        HumanMessage(    用户query
            content="I like pasta with cheese, but I need to eat more vegetables, what shoulc
        )
```

```
    ]
)
```

I am seeing this response in Jupyter:

```
AIMessage(content='A tasty and nutritious option could be a vegetable pasta dish. Depending c
```

It ignored the one-word instruction, but I actually liked reading the ideas. I think I could try this for my son. With other chatbots, I am getting the suggestion of Ratatouille.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. Chat models are similar to 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()`. This shared interface allows for interchangeability between different types of models in applications as well as between Chat and LLM models.

Replicate

Established in 2019, Replicate Inc. is a San Francisco-based startup 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. The firm, deriving its most recent funding from a Series A funding round of which the invested total was $12.5 million, was spearheaded by Andreessen Horowitz, and involved the participation of Y Combinator, Sequoia, and various independent investors. Ben Firshman, who drove open-source product efforts at Docker, and Andreas Jansson, a former machine learning engineer at Spotify, co-founded Replicate Inc. with the mutual aspiration to eliminate the technical barriers that were hindering the mass acceptance of AI. Consequently, they created Cog, an open-source tool that packs machine learning models into a standard production-ready container that can run on any current operating system and automatically generates an API. These containers can also be deployed on clusters of GPUs through the replicate platform. As a result, developers can concentrate on other essential tasks, thereby enhancing their productivity.You can authenticate with your Github credentials on https://replicate.com/If you then click on your user icon on 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`. In order to run bigger jobs, you need to set up your credit card (under billing).You can find a lot of models available at https://replicate.com/exploreHere's a simple example for creating an image:

```
from langchain.llms import Replicate
text2image = Replicate(
    model="stability-ai/stable-diffusion:db21e45d3f7023abc2a46ee38a23973f6dce16bb082a930b0c49
    input={"image_dimensions": "512x512"},
)
image_url = text2image("a book cover for a book about creating generative ai applications in
```

I got this image:

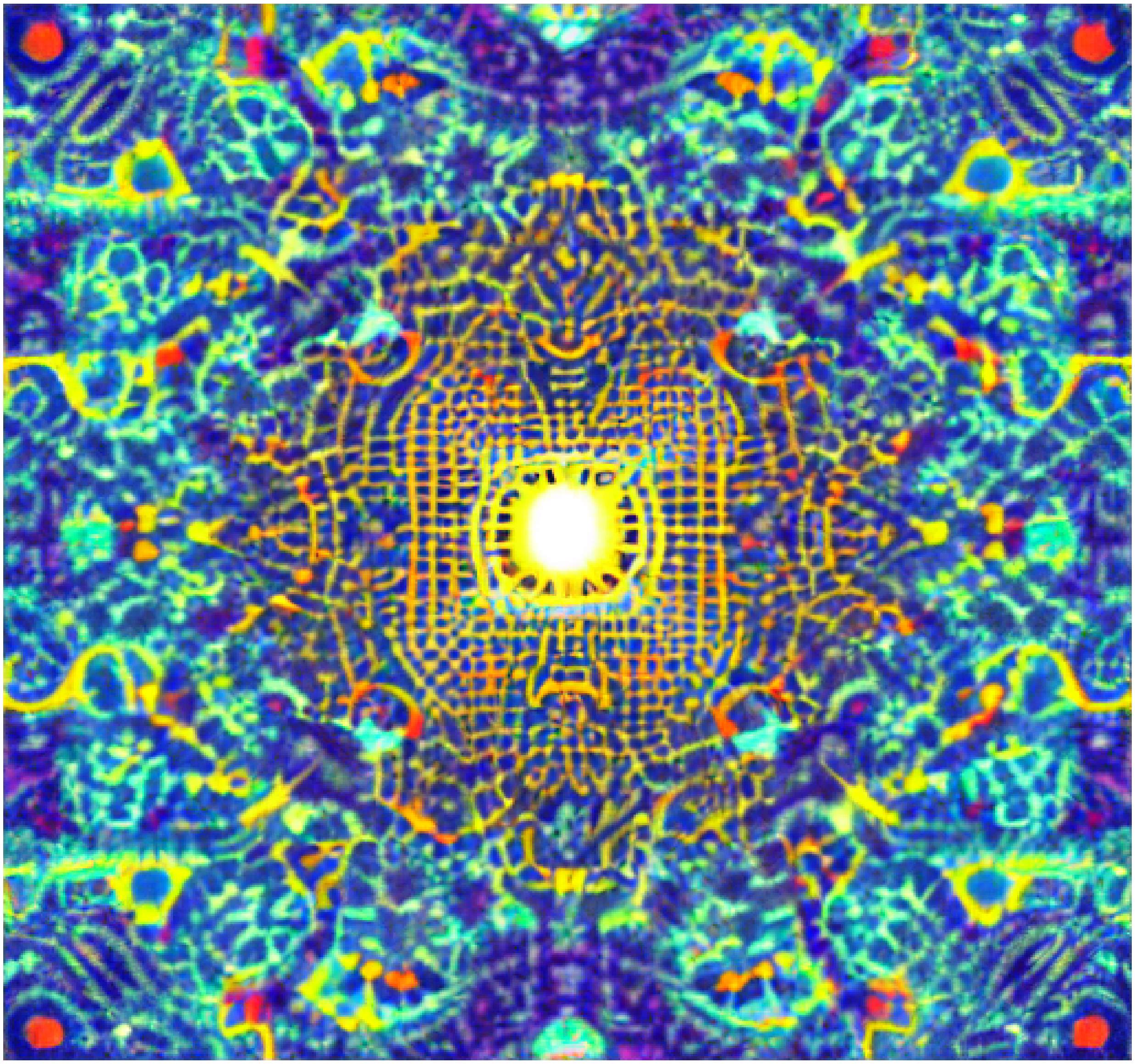


Figure 3.7: A Book Cover for a book about generative AI with Python - Stable Diffusion.

I think it's a nice image - is that an AI chip that creates art?Let's see quickly how to run a model locally in Huggingface transformers or Llama.cpp!

Local Models

We can also run local models from LangChain. Let's preface this with a note of caution: an LLM is big, which means that it'll take up a lot of space. If you have an old computer, you can try hosted services such as google colabs. These will let you run on machines with a lot of memory and different hardware including Tensor Processing Units (TPUs) or GPUs.Since both these use cases can take very long to run or crash the Jupyter notebook, I haven't included this code in the notebook or the dependencies in the setup instructions. I think it's still worth discussing it here. The advantages of running models locally are complete control over the model and not sharing any data over the internet.Let's see this first with the transformers library by Hugging Face.

Hugging Face transformers

I'll quickly show the general recipe of setting up and running a pipeline:

```
from transformers import pipeline
import torch
generate_text = pipeline(
    model="aisquared/dlite-v1-355m",
    torch_dtype=torch.bfloat16,
    trust_remote_code=True,
    device_map="auto",
    framework="pt"
)
generate_text("In this chapter, we'll discuss first steps with generative AI in Python.")
```

This model is quite small (355 million parameters), but relative performant, and instruction tuned for conversations.Please note that we don't need an API token for local models!This will download everything that's needed for the model such as the tokenizer and model weights. We can then run a text completion to give us some content for this chapter.In order to plug in this pipeline into a LangChain agent or chain, we can use it the same way that we've seen before:

```
from langchain import PromptTemplate, LLMChain
template = """Question: {question}
Answer: Let's think step by step."""
prompt = PromptTemplate(template=template, input_variables=["question"])
llm_chain = LLMChain(prompt=prompt, llm=generate_text)
question = "What is electroencephalography?"
print(llm_chain.run(question))
```

In this example, we also see the use of a `PromptTemplate` that gives specific instructions for the task.Let's do Llama.cpp next.

Llama.cpp

Llama.cpp is a C++ program that executes models based on architectures based on Llama, one of the first large open-source models released by Meta AI, which spawned the development of many other models in turn. Please note that you need to have an md5 checksum tool installed. This is included by default in several Linux distributions such as Ubuntu. On MacOs, you can install it with brew like this:

```
brew install md5sha1sum
```

We need to download the llama.cpp repository from Github. You can do this online choosing one of the download options on Github, or you can use a git command from the terminal like this:

```
git clone https://github.com/ggerganov/llama.cpp.git
```

Then we need to install the python requirements, which we can do with the pip package installer - let's also switch to the llama.cpp project root directory for convenience:

```
cd llama.cpp
pip install -r requirements.txt
```

You might want to create a Python environment before you install requirements - but this is up to you. Now we need to compile llama.cpp:

```
make -C . -j4 # runs make in subdir with 4 processes
```

We can parallelize the build with 4 processes. In order to get the Llama model weights, you need to sign up with the T&Cs and wait for a registration email from Meta. There are tools such as the llama model downloader in the pyllama project, but please be advised that they might not conform with the license stipulations by Meta. You can download models from Hugging Face - these models should be compatible with llama.cpp, such as Vicuna or Alpaca. Let's assume you have downloaded the model weights for the 7B Llama model into the models/7B directory.You can download models in much bigger sizes such as 13B, 30B, 65B, however, a note of caution is in order here: these models are fairly big both in terms of memory and disk space.We have to convert the model to llama.cpp format, which is called **ggml**, using the convert script. Then we can optionally quantize the models to save memory when doing inference. Quantization refers to reducing the number of bits that are used to store weights.

```
python3 convert.py models/7B/
./quantize ./models/ggml-model-f16.bin ./models/7B/ggml-model-q4_0.bin q4_0
```

This last file is much smaller than the previous files and will take up much less space in memory as well, which means that you can run it on smaller machines.Once we have chosen a model that we want to run, we can integrate it into an agent or a chain for example as follows:

```
llm = LlamaCpp(
    model_path="./ggml-model-q4_0.bin",
    verbose=True
```

)

This concludes the introduction to model providers. Let's build an application!

Customer Service Helper

In this section, we'll build a text classification app in LangChain for customer service agents. Given a document such as an email, we want to classify it into different categories related to intent, extract the sentiment, and provide a summary.Customer service agents are responsible for answering customer inquiries, resolving issues and addressing complaints. Their work is crucial for maintaining customer satisfaction and loyalty, which directly affects a company's reputation and financial success. Generative AI can assist customer service agents in several ways:

- Sentiment classification: this helps identify customer emotions and allows agents to personalize their response.
- Summarization: this enables agents to understand the key points of lengthy customer messages and save time.
- Intent classification: similar to summarization, this helps predict the customer's purpose and allows for faster problem-solving.
- Answer suggestions: this provides agents with suggested responses to common inquiries, ensuring that accurate and consistent messaging is provided.

These approaches combined can help customer service agents respond more accurately and in a timely manner, ultimately improving customer satisfaction.Here, we will concentrate on the first three points. We'll document lookups, which we can use for answer suggestions in *Chapter 5*, *Building a Chatbot like ChatGPT*.**LangChain** is a very flexible library with many integrations that can enable us to tackle a wide range of text problems. We have a choice between many different integrations to perform these tasks. We could ask any LLM to give us an open-domain (any category) classification or choose between multiple categories. In particular, because of their large training size, LLMs are very powerful models, especially when given few-shot prompts, for sentiment analysis that don't need any additional training. This was analyzed by Zengzhi Wang and others in their April 2023 study "Is ChatGPT a Good Sentiment Analyzer? A Preliminary Study". A prompt, for an LLM for sentiment analysis could be something like this:

主问题？ 包括的选项

prompt

```
Given this text, what is the sentiment conveyed? Is it positive, neutral, or negative?
Text: {sentence}
Sentiment:
```

LLMs can be also very effective at summarization, much better than any previous models. The downside can be that these model calls are slower than more traditional ML models and more expensive.If we want to try out more traditional or smaller models. Cohere and other providers have text classification and sentiment analysis as part of their capabilities. For example, NLP Cloud's model list includes spacy and many others: https://docs.nlpcloud.com/#models-listMany Hugging Face models are supported for these tasks including:

- document-question-answering
- summarization
- text-classification
- text-question-answering

- translation

We can execute these models either locally, by running a `pipeline` in transformer, remotely on the Hugging Face Hub server (`HuggingFaceHub`), or as tool through the `load_huggingface_tool()` loader.Hugging Face contains thousands of models, many fine-tuned for particular domains. For example, `ProsusAI/finbert` is a BERT model that was trained on a dataset called Financial PhraseBank, and can analyze sentiment of financial text. We could also use any local model. For text classification, the models tend to be much smaller, so this would be less of a drag on resources. Finally, text classification could also be a case for embeddings, which we'll discuss in *Chapter 5*, *Building a Chatbot like ChatGPT*.I've decided to try make do as much as I can with smaller models that I can find on Hugging Face for this exercise.We can list the 5 most downloaded models on Hugging Face Hub for text classification through the huggingface API:

```
def list_most_popular(task: str):
    for rank, model in enumerate(
        list_models(filter=task, sort="downloads", direction=-1)
):
        if rank == 5:
            break
        print(f"{model.id}, {model.downloads}\n")
list_most_popular("text-classification")
```

Let's see the list:

Model	**Downloads**
nlptown/bert-base-multilingual-uncased-sentiment	5805259
SamLowe/roberta-base-go_emotions	5292490
cardiffnlp/twitter-roberta-base-irony	4427067
salesken/query_wellformedness_score	4380505
marieke93/MiniLM-evidence-types	4370524

Tablee 3.2: The most popular text classification models on Hugging Face Hub.We can see that these models are about small ranges of categories such as sentiment, emotions, irony, or well-formedness. Let's use the sentiment model.

```
I've asked GPT-3.5 to put together a long rambling customer email complaining about a coffee
from transformers import pipeline
sentiment_model = pipeline(
    task="sentiment-analysis",
    model="nlptown/bert-base-multilingual-uncased-sentiment"
)
print(sentiment_model(customer_email))
```

I am getting this result:

```
[{'label': '2 stars', 'score': 0.28999224305152893}]
```

Not a happy camper. Let's move on!Let's see the 5 most popular models for summarization as well:

Model	**Downloads**
t5-base	2710309
t5-small	1566141
facebook/bart-large-cnn	1150085
sshleifer/distilbart-cnn-12-6	709344
philschmid/bart-large-cnn-samsum	477902

Table 3.3: The most popular summarization models on Hugging Face Hub.

All these models have a relatively small footprint compared to large models. Let's execute the summarization model remotely on a server:

```
from langchain import HuggingFaceHub
summarizer = HuggingFaceHub(
    repo_id="facebook/bart-large-cnn",
    model_kwargs={"temperature":0, "max_length":180}
)
def summarize(llm, text) -> str:
    return llm(f"Summarize this: {text}!")
summarize(summarizer, customer_email)
```

Please note that you need to have your `HUGGINGFACEHUB_API_TOKEN` set for this to work.I am seeing this summary:A customer's coffee machine arrived ominously broken, evoking a profound sense of disbelief and despair. "This heartbreaking display of negligence shattered my dreams of indulging in daily coffee perfection, leaving me emotionally distraught and inconsolable," the customer writes. "I hope this email finds you amidst an aura of understanding, despite the tangled mess of emotions swirling within me as I write to you," he adds.This summary is just passable, but not very convincing. There's still a lot of rambling in the summary. We could try other models or just go for an LLM with a prompt asking to summarize. Let's move on.It could be quite useful to know what kind of issue the customer is writing about. Let's ask **VertexAI**:

```
from langchain.llms import VertexAI
from langchain import PromptTemplate, LLMChain
template = """Given this text, decide what is the issue the customer is concerned about. Vali
* product issues
* delivery problems
* missing or late orders
* wrong product
* cancellation request
* refund or exchange
* bad support experience
* no clear reason to be upset
Text: {email}
Category:
"""
prompt = PromptTemplate(template=template, input_variables=["email"])
llm = VertexAI()
llm_chain = LLMChain(prompt=prompt, llm=llm, verbose=True)
print(llm_chain.run(customer_email))
```

We get "product issues" back, which is correct for the long email example that I am using here.I hope it was exciting to see how quickly we can throw a few models and tools together in LangChain to get something that looks actually useful. We could easily expose this in an interface for customer service agents to see.Let's summarize.

Summary

Key

In this chapter, we've walked through four distinct ways of installing LangChain and other libraries needed in this book as an environment. Then, we've introduced several providers of models for text and images. For each of them, we explained where to get the API token, and demonstrated how to call a model. Finally, we've developed an LLM app for text classification in a use case for customer service. By chaining together various functionalities in LangChain, we can help reduce response times in customer service and make sure answers are accurate and to the point.In the chapters 3 and 4, we'll dive more into use cases such as question answering through augmented retrieval using tools such as web searches and chatbots relying on document search through indexing. Let's see if you remember some of the key takeaways from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. How do you install LangChain?
2. List at least 4 cloud providers of LLMs apart from OpenAI!
3. What are Jina AI and Hugging Face?
4. How do you generate images with LangChain?

5. How do you run a model locally on your own machine rather than through a service?
6. How do you perform text classification in LangChain?
7. How can we help customer service agents in their work through generative AI?

4 Querying with Tools

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

In today's fast-paced business and research landscape, keeping up with the ever-increasing volume of information can be a daunting task. For engineers and researchers in fields like computer science and artificial intelligence staying updated with the latest developments is crucial. However, reading and comprehending numerous papers can be time-consuming and labor-intensive. This is where automation comes into play. In this chapter, we'll describe an approach to automate the summarization of research papers and answering questions, making it easier for researchers to digest and stay informed. By leveraging language models and a series of questions, the summarization we'll develop can summarize the core assertions, implications, and mechanics of a paper in a concise and simplified format. This can not only save time and effort while researching a topic, but it also ensures we can effectively navigate the accelerated pace of scientific progress. We'll also have a play around with functions in OpenAI models, and their application to information extraction. We'll see how they work (or not yet) for an application as parsers of curriculum vitae (CVs). This function syntax is specific to OpenAI's API and has many applications, however, LangChain provides a platform that allows the creation of tools for any large language models (LLMs), which enhance their capabilities. These tools enable LLMs to interact with APIs, access real-time information, and perform various tasks such as retrieval search, database queries, writing emails, or even making phone calls. We'll implement a question-answering app with Retrieval Augmented Generation (RAG). This is a technique to update large language models (LLMs) like GPT by injecting relevant data into the context. Finally, we'll discuss different strategies of decision making in agents. We'll implement two strategies, plan-and-execute (or plan-and-solve) and one-shot-agent, and we'll integrate them into a visual interface as a visual app in the browser (using Streamlit) for question answering.The main sections are:

- What are hallucinations?
- How to summarize long documents?
- Extracting information from documents
- Answer questions with tools
- Reasoning strategies

We'll begin the chapter by discussing the problem of reliability of LLMs.

What are hallucinations?

The rapid development of generative language models, such as GPT-3, Llama, and Claude 2, has brought attention to their limitations and potential risks. One major concern are hallucinations, where the models generate output that is nonsensical, incoherent, or unfaithful to the provided input. Hallucination poses performance and safety risks in real-world applications, such as medical or machine translation. There's another aspect to hallucinations, which is the case, where LLMs generate text that includes sensitive personal information, such as email addresses, phone numbers, and physical addresses. This poses significant privacy concerns as it suggests that language models can memorize and recover such private data from their training corpus, despite not being present in the source input.

> **Hallucination** in the context of LLMs refers to the phenomenon where generated text is unfaithful to the intended content or nonsensical. This term draws a parallel to psychological hallucinations, which involve perceiving something that does not exist. In NLG, hallucinated text may appear fluent and natural grounded in the provided

NLG：Natural Language Generation

context, but lacks specificity or verifiability. **Faithfulness**, where the generated content stays consistent and truthful to the source, is considered an antonym of hallucination.

> **Intrinsic hallucinations** occur when the generated output contradicts the source content, while **extrinsic hallucinations** involve generating information that cannot be verified or supported by the source material. Extrinsic hallucinations can sometimes include factually correct external information, but their unverifiability raises concerns from a factual safety perspective.

Efforts to address hallucination are ongoing, but there is a need for a comprehensive understanding across different tasks to develop effective mitigation methods.Hallucinations in LLMs can be caused by various factors:

1. Imperfect representation learning by the encoder.
2. Erroneous decoding, including attending to the wrong part of the source input and the choice of decoding strategy.
3. Exposure bias, which is the discrepancy between training and inference time.
4. Parametric knowledge bias, where pre-trained models prioritize their own knowledge over the input, leading to the generation of excessive information.

Hallucination mitigation methods can be categorized into two groups: Data-Related Methods and Modeling and Inference Methods (after "Survey of Hallucination in Natural Language Generation", Ziwei Ji and others, 2022):

Data-Related Methods:

Building a Faithful Dataset: Constructing datasets with clean and faithful targets from scratch or rewriting real sentences while ensuring semantic consistency.

Cleaning Data Automatically: Identifying and filtering irrelevant or contradictory information in the existing corpus to reduce semantic noise.

Information Augmentation: Augmenting inputs with external information, such as retrieved knowledge or synthesized data, to improve semantic understanding and address source-target divergence.

Modeling and Inference Methods:

Architecture: Modifying encoder architecture to enhance semantic interpretation, attention mechanisms to prioritize source information, and decoder structures to reduce hallucination and enforce implicit or explicit constraints.

Training: Incorporating planning, reinforcement learning (RL), multi-task learning, and controllable generation techniques to mitigate hallucination by improving alignment, optimizing reward functions, and balancing faithfulness and diversity.

Post-Processing: Correcting hallucinations in the output through generate-then-refine strategies or refining the results specifically for faithfulness using post-processing correction methods.

A result of hallucinations, where automatic fact checking can be applied, is the danger of spreading incorrect information or misuse for political purposes. **Misinformation**, including **disinformation**, deceptive news, and rumors, poses a significant threat to society, especially with the ease of content creation and dissemination through social media. The threats to society include distrust in science, public health narratives, social polarization, and democratic processes. Journalism and archival studies have extensively studied the issue, and fact-checking initiatives have grown in response. Organizations dedicated to fact-checking provide training and resources to independent fact-checkers and journalists, allowing for the scaling of expert fact-checking efforts. Addressing misinformation is crucial to preserving the integrity of information and combating its detrimental impact on society.In the literature, this kind of problem is called textual entailment, where models predict the directional truth relation between a text pair (i.e. "sentence t entails h" if, typically, a human reading t would infer that h is most likely true). In this chapter, we'll focus on **automatic fact-checking** through information augmentation and post-processing.

Facts can be retrieved either from LLMs or using external tools. In the former case, pre-trained language models can take the place of the knowledge base and retrieval modules, leveraging their vast knowledge to answer open-domain questions and using prompts to retrieve specific facts. We can see the general idea in this diagram (source: https://github.com/Cartus/Automated-Fact-Checking-Resources by Zhijiang Guo):

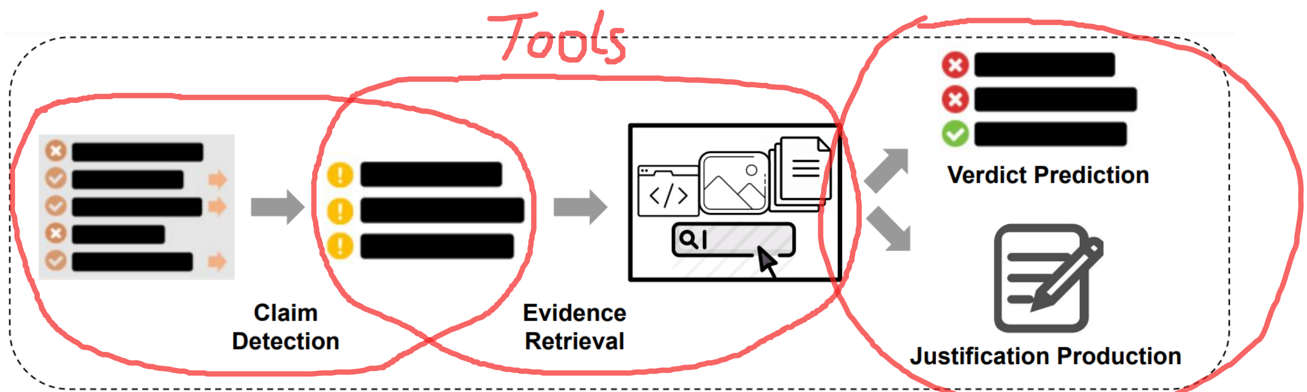


Figure 4.1: Automatic fact-checking pipeline in three stages.

We can distinguish three stages:

1. Claim detection - identify claims that require verification
2. Retrieval - retrieve evidence to find sources supporting or refuting the claim
3. Claim verification - assess the veracity of the claim based on the evidence

Starting with 24-layer BERT-Large in 2018, language models have been pre-trained on large knowledge bases such as Wikipedia, and therefore would be able to answer knowledge questions from Wikipedia or - since their training set increasingly includes other sources - the internet, textbooks, arxiv, and Github. Querying for facts works with simple prompts such as masking prompts. For example, in order to answer the question "Where is Microsoft's headquarter?", the question would be rewritten as "Microsoft's headquarter is in [MASK]" and fed into a language model for the answer. In this approach, the activations in the final Interestingly, if an LLM that has not received the source text (unconditional LLM) yields a smaller loss in generating targets than an LLM that has received the source text (conditioned LLM), this indicates that the generated token is hallucinatory (Fillippova, 2020). The ratio of hallucinated tokens to the total number of target tokens can serve as a measure of the degree of hallucination in the generated output. In LangChain, we have a chain available for fact checking with prompt-chaining, where a model actively questions the assumptions that went into a statement. In this self-checking chain, `LLMCheckerChain`, the model is prompted sequentially times, first to make the assumptions explicit - this looks like this:

```
Here's a statement: {statement}\nMake a bullet point list of the assumptions you made when pr
```

Please note that this is a string template, where the elements in curly brackets will be replaced by variables. Next, these assumptions are fed back to the model in order to check them one by one with a prompt like this:

```
Here is a bullet point list of assertions:
    {assertions}
    For each assertion, determine whether it is true or false. If it is false, explain why.\r
```

Finally, the model is tasked to make a final judgment:

```
In light of the above facts, how would you answer the question '{question}'
```

The `LLMCheckerChain` does this all by itself as this example shows:

```
from langchain.chains import LLMCheckerChain
from langchain.llms import OpenAI
llm = OpenAI(temperature=0.7)
text = "What type of mammal lays the biggest eggs?"
checker_chain = LLMCheckerChain.from_llm(llm, verbose=True)
checker_chain.run(text)
```

The model can return different results to this question, some of which are wrong, and some of which it would correctly identify as false. When I was trying this out, I got results such as the blue whale, the North American beaver, or the extinct Giant Moa. I think this is the right answer:

```
Monotremes, a type of mammal found in Australia and parts of New Guinea, lay the largest eggs
• Monotremes can be found in Australia and New Guinea
• The largest eggs in the mammalian world are laid by monotremes
• The American echidna lays eggs that can grow to 10 cm in length
• Dunnarts lay eggs that can exceed 5 cm in length
• Monotremes can be found in Australia and New Guinea – True
• The largest eggs in the mammalian world are laid by monotremes – True
• The American echidna lays eggs that can grow to 10 cm in length – False, the American echic
• Dunnarts lay eggs that can exceed 5 cm in length – False, dunnarts lay eggs that are typica
The largest eggs in the mammalian world are laid by monotremes, which can be found in Austral
> Finished chain.
```

So, while this doesn't guarantee correct answers, it can put a stop to some incorrect results.As for augmented retrieval (or RAG), we've seen the approach in this chapter, in the section about question answering. Fact-checking approaches involve decomposing claims into smaller checkable queries, which can be formulated as question-answering tasks. Tools designed for searching domain datasets can assist fact-checkers in finding evidence effectively. Off-the-shelf search engines like Google and Bing can also retrieve both topically and evidentially relevant content to capture the veracity of a statement accurately. In the next section, we'll apply this approach to return results based on web searches and other tools. In the next section, we'll implement a chain to summarize documents. We can ask any question to be answered from these documents.

How to summarize long documents?

In this section, we'll discuss automating the process of summarizing long texts and research papers. In today's fast-paced business and research landscape, keeping up with the ever-increasing volume of information can be a daunting task. For engineers and researchers in fields like computer science and artificial intelligence staying updated with the latest developments is crucial. However, reading and comprehending numerous papers can be time-consuming and labor-intensive. This is where automation comes into play. As engineers, we are driven by their desire to build and innovate, avoid repetitive tasks by automating them through the creation of pipelines and processes. This approach, often mistaken for laziness, allows engineers to focus on more complex challenges and utilize their skills more efficiently.Here, we'll build an automation tool that can quickly summarize the content of long texts in a more digestible format. This tool is intended to help researchers keep up with the volume of papers being published daily, particularly in fast-moving fields such as artificial intelligence. By automating the summarization process, researchers can save time and effort, while also ensuring that they stay informed about the latest developments in their field. The tool will be based on LangChain and utilize large language models (LLMs) to summarize the core assertions, implications, and mechanics of a paper in a more concise and simplified manner. It can also answer specific questions about the paper, making it a valuable resource for literature reviews and accelerating scientific research. The author plans to further develop the tool to allow automatic processing of multiple documents and customization for specific research domains. Overall, the approach aims to benefit researchers by providing a more efficient and accessible way to stay updated with the latest research.LangChain supports a map reduce approach for processing documents using LLMs, which allows for efficient processing and analysis of documents. When reading in large texts, and splitting them into documents (chunks) that are suitable for the token context length of the LLM, a chain can be applied to each document individually and then combine the outputs into a single document. The core assertion is that the map reduce process involves two steps:

- Map step - the LLM chain is applied to each document individually, treating the output as a new document, and
- Reduce step - all the new documents are passed to a separate combine documents chain to obtain a single output.

This is illustrated in the figure here:

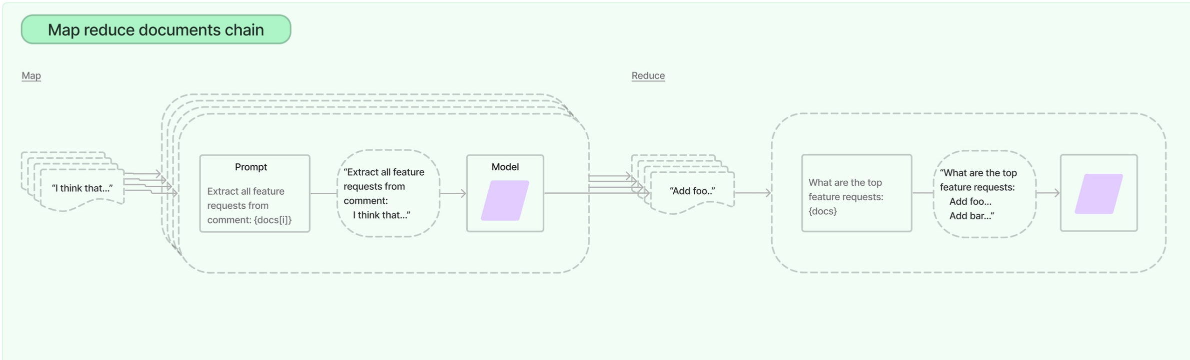


Figure 4.2: Map reduce chain in LangChain (source: LangChain documentation).

The implications of this approach are that it allows for parallel processing of documents and enables the use of LLMs for reasoning, generating, or analyzing the individual documents as well as combining their outputs. The mechanics of the process involve compressing or collapsing the mapped documents to ensure they fit within the combine documents chain, which may also involve utilizing LLMs. The compression step can be performed recursively if needed. Here's a simple example of loading a PDF document and summarizing it:

```
from langchain.chains.summarize import load_summarize_chain
from langchain import OpenAI
from langchain.document_loaders import PyPDFLoader
pdf_loader = PyPDFLoader(pdf_file_path)
docs = pdf_loader.load_and_split()
llm = OpenAI()
chain = load_summarize_chain(llm, chain_type="map_reduce")
chain.run(docs)
```

The variable `pdf_file_path` is a string with the path of a PDF file.The default prompt for both the map and reduce steps is this:

```
Write a concise summary of the following:
{text}
CONCISE SUMMARY:
```

We can specify any prompt for each step. In the text summarization application developed for this chapter on Github, we can see how to pass other prompts. On LangChainHub, we can see the qa-with sources prompt, which takes a reduce/combine prompt like this:

```
Given the following extracted parts of a long document and a question, create a final answer
```

In this prompt, we would formulate a concrete question, but equally we could give the LLM a more abstract instruction to extract assumption and implications. The text would be the summaries from the map steps. An instruction like that would help against hallucinations.Other examples of instructions could be translating the document into a different language or rephrasing in a certain style.Once we start doing a lot of calls, especially here in the map step, we'll see costs increasing. We are doing a lot of calls and using a lot of tokens in total. Time to give this some visibility!

Token usage

When using models, especially in long loops such as with map operations, it's important to track the token usage and understand how much money you are spending. For any serious usage of generative AI, we need to understand the capabilities, pricing options, and use cases for different language models. OpenAI provides different models namely GPT-4, ChatGPT, and InstructGPT that cater to various natural language processing needs. GPT-4 is a powerful language model suitable for solving complex problems with natural language processing. It offers flexible pricing options based on the size and number of tokens used.ChatGPT models, like GPT-3.5-Turbo, specialize in dialogue applications such as chatbots and virtual assistants. They excel in generating responses with accuracy and fluency. The pricing for ChatGPT models is based on the number of tokens used.InstructGPT models are designed for single-turn instruction following and are optimized for quick and accurate response generation. Different models within the InstructGPT family, such as Ada and Davinci, offer varying levels of speed and power. Ada is the fastest model, suitable for applications where speed is crucial, while Davinci is the most powerful model, capable of handling complex instructions. The pricing for InstructGPT models depends on the model's capabilities and ranges from low-cost options like Ada to more expensive options like Davinci.OpenAI's DALL·E, Whisper, and API services for image generation, speech transcription, translation, and access to language models. DALL·E is an AI-powered image generation model that can be seamlessly integrated into apps for generating and editing novel images and art. OpenAI offers three tiers of resolution, allowing users to choose the level of detail they need. Higher resolutions offer more complexity and detail, while lower resolutions provide a more abstract representation. The price per image varies based on the resolution.Whisper is an AI tool that can transcribe speech into text and translate multiple languages into English. It helps capture conversations, facilitates communication, and improves understanding across languages. The cost for using Whisper is based on a per-minute rate.OpenAI's API provides access to powerful language models like GPT-3, enabling developers to create advanced applications. When signing up for the API, users are given an initial token usage limit, representing the number of tokens available to interact with the language model within a specific timeframe. As users' track record and usage increase, OpenAI may increase the token usage limit, granting more access to the model. Users can also request a quota increase if they require more tokens for their applications.We can track the token usage in OpenAI models by hooking into the OpenAI callback:

有论文可以看

```
with get_openai_callback() as cb:
    response = llm_chain.predict(text="Complete this text!")
    print(f"Total Tokens: {cb.total_tokens}")
    print(f"Prompt Tokens: {cb.prompt_tokens}")
    print(f"Completion Tokens: {cb.completion_tokens}")
    print(f"Total Cost (USD): ${cb.total_cost}")
```

In this example, the line with `llm_chain` could be any usage of an OpenAI model. We should see an output with the costs and tokens.There are two other ways of getting the token usage. Alternative to the OpenAI Callback, the `generate()` method of the `llm` class returns a response of type `LLMResult` instead of string. This includes token usages and finish reason, for example (from the LangChain docs):

```
input_list = [
    {"product": "socks"},
    {"product": "computer"},
    {"product": "shoes"}
]
llm_chain.generate(input_list)
```

The result looks like this:

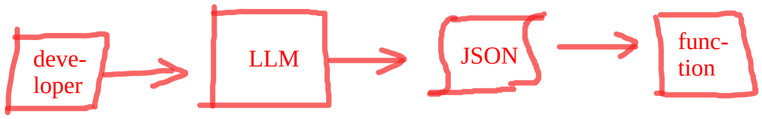
```
    LLMResult(generations=[[Generation(text='\n\nSocktastic!', generation_info={'finish_reasc
```

Finally, the chat completions response format in the OpenAI API includes a usage object with token information, for example it could look like this (excerpt):

```
  {
  "model": "gpt-3.5-turbo-0613",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 17,
    "prompt_tokens": 57,
    "total_tokens": 74
  }
}
```

Next, we'll look at how to extract certain pieces of information from documents using OpenAI functions with LangChain.

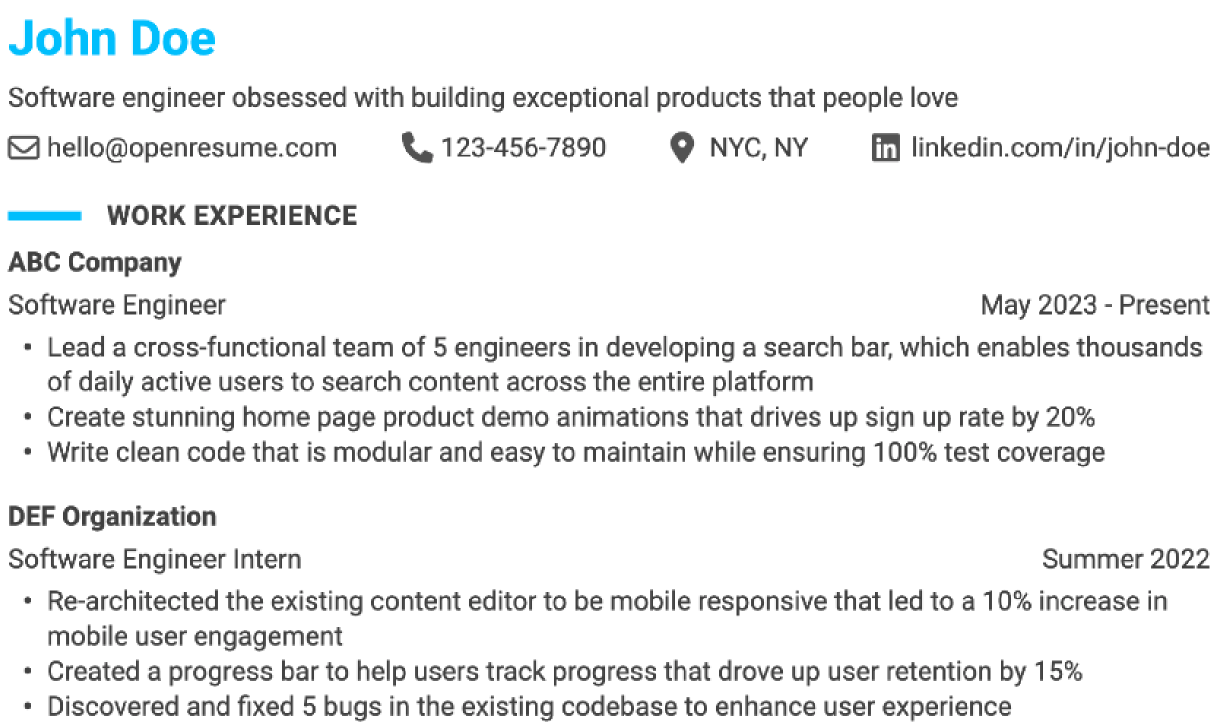
Extracting information from documents



In June 2023, OpenAI announced updates to OpenAI's API, including new capabilities for function calling, which opens up an enhanced functionality. Developers can now describe functions to the gpt-4-0613 and gpt-3.5-turbo-0613 models and have the models intelligently generate a JSON object containing arguments to call those functions. This feature aims to enhance the connection between GPT models and external tools and APIs, providing a reliable way to retrieve structured data from the models.Function calling enables developers to create chatbots that can answer questions using external tools or OpenAI plugins. It also allows for converting natural language queries into API calls or database queries, and extracting structured data from text. The mechanics of the update involve using new API parameters, namely `functions` , in the `/v1/chat/completions` endpoint. The functions parameter is defined through a name, description, parameters, and the function to call itself. Developers can describe functions to the model using JSON Schema and specify the desired function to be called. In LangChain, we can use these function calls in OpenAI for information extraction or for calling plugins. For information extraction, we can specific entities and their properties from a text and their properties from a document in an extraction chain with OpenAI chat models. For example, this can help identifying the people mentioned in the text. By using the OpenAI functions parameter and specifying a schema, it ensures that the model outputs the desired entities and properties with their appropriate types.The implications of this approach are that it allows for precise extraction of entities by defining a schema with the desired properties and their types. It also enables specifying which properties are required and which are optional.The default format for the schema is a dictionary, but we can also define properties and their types in Pydantic providing control and flexibility in the extraction process.Here's an example for a desired schema for information in a Curricum Vitae (CV):

```
from typing import Optional
from pydantic import BaseModel
class Experience(BaseModel):
    start_date: Optional[str]
    end_date: Optional[str]
    description: Optional[str]
class Study(Experience):
    degree: Optional[str]
    university: Optional[str]
    country: Optional[str]
    grade: Optional[str]
class WorkExperience(Experience):
    company: str
    job_title: str
class Resume(BaseModel):
    first_name: str
    last_name: str
    linkedin_url: Optional[str]
    email_address: Optional[str]
    nationality: Optional[str]
    skill: Optional[str]
    study: Optional[Study]
    work_experience: Optional[WorkExperience]
    hobby: Optional[str]
```

We can use this for information extraction from a CV. Here's an example CV from https://github.com/xitanggg/open-resume


John Doe

Software engineer obsessed with building exceptional products that people love

hello@openresume.com 123-456-7890 NYC, NY linkedin.com/in/john-doe

WORK EXPERIENCE

ABC Company

Software Engineer May 2023 - Present

- Lead a cross-functional team of 5 engineers in developing a search bar, which enables thousands of daily active users to search content across the entire platform
- Create stunning home page product demo animations that drives up sign up rate by 20%
- Write clean code that is modular and easy to maintain while ensuring 100% test coverage

DEF Organization

Software Engineer Intern Summer 2022

- Re-architected the existing content editor to be mobile responsive that led to a 10% increase in mobile user engagement
- Created a progress bar to help users track progress that drove up user retention by 15%
- Discovered and fixed 5 bugs in the existing codebase to enhance user experience

Figure 4.3: Extract of an example CV.

We are going to try to parse the information from this resume. Utilizing the `create_extraction_chain_pydantic()` function in LangChain, we can provide our schema as input, and an output will be an instantiated object that adheres to it. In its most simple terms, we can try this code snippet:

```
from langchain.chains import create_extraction_chain_pydantic
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import PyPDFLoader
pdf_loader = PyPDFLoader(pdf_file_path)
docs = pdf_loader.load_and_split()
# please note that function calling is not enabled for all models!
llm = ChatOpenAI(model_name="gpt-3.5-turbo-0613")
chain = create_extraction_chain_pydantic(pydantic_schema=Resume, llm=llm)
return chain.run(docs)
```

We should be getting an output like this:

```
[Resume(first_name='John', last_name='Doe', linkedin_url='linkedin.com/in/john-doe', email_ac
```

It's far from perfect - only one work experience gets parsed out. But it's a good start given the little effort we've put in so far. Please see the example on Github for the full example. We could add more functionality, for example to guess personality or leadership capability.OpenAI injects these function calls into the system message in a certain syntax, which their models have been optimized for. This implies that functions count against the context limit and are correspondingly billed as input tokens. LangChain natively has functionality to inject function calls as prompts. This means we can use model providers other than OpenAI for function calls within LLM apps. We'll look at this now, and we'll build this into an interactive web-app with Streamlit.

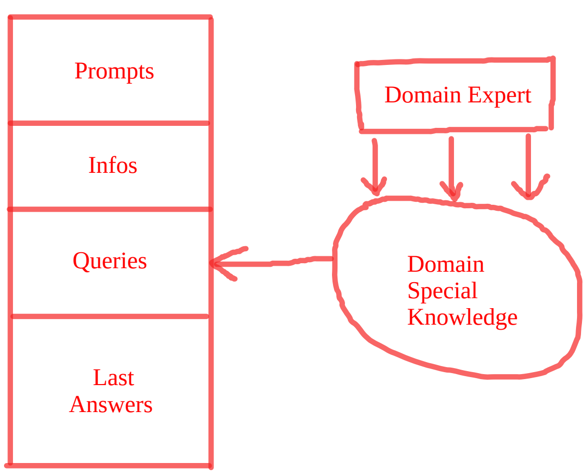
Answer questions with tools

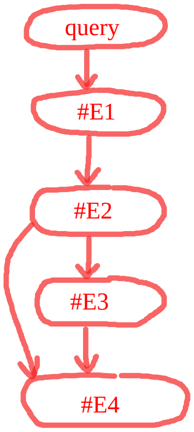
LLMs are trained on general corpus data and may not be as effective for tasks that require domain-specific knowledge. On their own LLMs can't interact with the environment and access external data sources, however, LangChain provides a platform for creating tools that access real-time information, and perform tasks such as weather forecasting, making reservations, suggesting recipes, and managing tasks. Tools within the framework of agents and chains allow for the development of applications powered by LLMs that are data-aware and agentic, and opens up a wide range of approaches to solving problems with LLMs, expanding their use cases and making them more versatile and powerful.One important aspect of tools is their capability to work within specific domains or process specific inputs. For example, an LLM lacks inherent mathematical capabilities. However, a mathematical tool like a calculator can accept mathematical expressions or equations as an input and calculate the outcome. The LLM combined with such a mathematical tool perform calculations and provide accurate answers.Generally, this combination of retrieval methods and LLMs is called **Retrieval Augmented Generation** (**RAG**), and addresses the limitations of LLMs by retrieving relevant data from external sources and injecting it into the context. This retrieved data serves as additional information to augment the prompts given to the LLMs.By grounding LLMs with use-case specific information through RAG, the quality and accuracy of responses are improved. Through retrieval of relevant data, RAG helps in reducing hallucinations responses from LLMs.For example, an LLM used in a healthcare application could retrieve relevant medical information from external sources such as medical literature or databases during inference. This retrieved data can then be incorporated into the context to enhance the generated responses and ensure they are accurate and aligned with domain-specific knowledge. The benefits of implementing RAG in this scenario are twofold. Firstly, it allows for incorporating up-to-date information into responses despite the model's training data cutoff date. This ensures that users receive accurate and relevant information even for recent events or evolving topics.Secondly, RAG enhances ChatGPT's ability to provide more detailed and contextual answers by leveraging external sources of information. By retrieving specific context from sources like news articles or websites related to a particular topic, the responses will be more accurate.

RAG (Retrieval Augmented Generation) works by retrieving information from data sources to supplement the prompt given to the language model, providing the model with the needed context to generate accurate responses. RAG involves several steps:

- **Prompt**: The user provides a prompt to the chatbot, describing their expectations for the output.
- **Research**: A contextual search is performed and retrieves relevant information from various data sources. This could involve querying a database, searching indexed documents based on keywords, or invoking APIs to retrieve data from external sources.
- **Update Resource**: The retrieved context is injected into the original prompt, augmenting it with additional factual information related to the user's query. This enhanced prompt improves accuracy as it provides access to factual data.
- **Narrate**: Based on this augmented input, the LLM generates a response that includes factually correct information and sends it back to the chatbot for delivery to the user.

Therefore, by combining external data sources and injecting relevant context into prompts, RAG enhances LLMs' ability to generate responses that are accurate, up-to-date, and aligned with specific domains or topics of interest.An illustration of augmenting LLMs through tools and reasoning is shown here (source: https://github.com/billxbf/ReWOO, implementation for the paper “Decoupling Reasoning from Observations for Efficient Augmented Language Models Resources” by Binfeng Xu and others, May 2023):





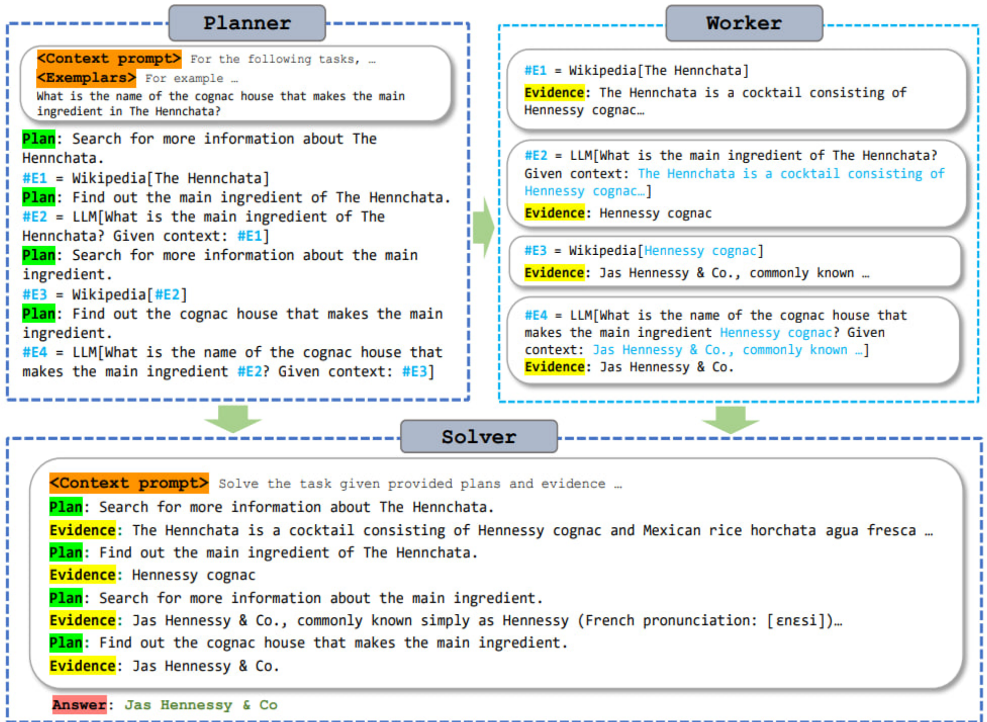


Figure 4.4: Tool-augmented LM paradigm, leveraging foreseeable reasoning ability of language models to improve system parameter and prompt efficiency

Let's see this in action!We have quite a few tools available in LangChain, and - if that's not enough - it's not hard to roll out our own tools. Let's set up an agent with a few tools:

```
from langchain.agents import (
    AgentExecutor, AgentType, initialize_agent, load_tools
)
from langchain.chat_models import ChatOpenAI
def load_agent() -> AgentExecutor:
    llm = ChatOpenAI(temperature=0, streaming=True)
    # DuckDuckGoSearchRun, wolfram alpha, arxiv search, wikipedia
    # TODO: try wolfram-alpha!
    tools = load_tools(
        tool_names=["ddg-search", "wolfram-alpha", "arxiv", "wikipedia"],
        llm=llm
    )
    return initialize_agent(
        tools=tools, llm=llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True
    )
```

工具也会使用llm，可能工具的llm和生成回答的llm不一样

It's an important detail to know that an `AgentExecutor` is a chain, and therefore - if we wanted we could integrate it into a larger chain if we wanted. We could have initialized this chain using a different syntax like this:

```
return MRKLChain.from_chains(llm, chains, verbose=True)
```

In this syntax, we would pass the tools as chain configurations. MRKL stands for Modular Reasoning, Knowledge and Language. The Zero-Shot Agent is the most general-purpose action agent in the MRKL framework.Please notice the parameter `streaming` in the `ChatOpenAI` constructor, which is set to `True`. This makes for a better use experience, since it means that the text response will be updated as it comes in, rather than once all the text has been completed. Currently only the OpenAI, ChatOpenAI, and ChatAnthropic implementations support streaming. All the tools mentioned have their specific purpose that's part of the description, which is passed to the language model. These tools here are plugged into the agent:

- DuckDuckGo - a search engine that focuses on privacy; an added advantage is that it doesn't require developer signup
- Wolfram Alpha - an integration that combines natural language understanding with math capabilities, for questions like "What is 2x+5 = -3x + 7?"
- Arxiv - search in academic pre-print publications; this is useful for research-oriented questions
- Wikipedia - for any question about entities of significant notoriety

Please note that in order to use Wolfram Alpha, you have to set up an account and set the `WOLFRAM_ALPHA_APPID` environment variable with the developer token you create at https://developer.wolframalpha.com/There are lot of other search tools integrated in LangChain apart from DuckDuckGo that let you utilize Google or Bing search engines or work with meta search engines. There's an Open-Meteo - integration for weather information, however, this information is also available through search.Let's make our agent available as a streamlit app.

> **Streamlit** is an open-source app framework for Machine Learning and Data Science teams. It allows users to create beautiful web apps in minutes using Python.

Let's write the code for this using the `load_agent()` function we've just defined:

```
import streamlit as st
from langchain.callbacks import StreamlitCallbackHandler
chain = load_agent()
st_callback = StreamlitCallbackHandler(st.container())
if prompt := st.chat_input():
    st.chat_message("user").write(prompt)
    with st.chat_message("assistant"):
        st_callback = StreamlitCallbackHandler(st.container())
        response = chain.run(prompt, callbacks=[st_callback])
        st.write(response)
```

Please notice that we are using the callback handler in the call to the chain, which means that we'll see responses as they come back from the model. We can start the app locally from the terminal like this:

```
PYTHONPATH=. streamlit run question_answering/app.py
```

> Deployment of Streamlit applications can be local or on a server. Alternatively, you can deploy this on the Streamlit Community Cloud or on Hugging Face Spaces.

- For **Streamlit Community Cloud** do this:
- 1. Create a Github repository
- 2. Go to Streamlit Community Cloud, click on "New app" and select the new repo
- 3. Click "Deploy!"
- As for **Hugging Face Spaces** it works like this:
- 1. Create a Github repo
- 2. Create a Hugging Face account at https://huggingface.co/
- 3. Go to "Spaces" and click "Create new Space". In the form, fill in a name, type of space as "Streamlit", and choose the new repo.

Here's a screenshot from the app looks:

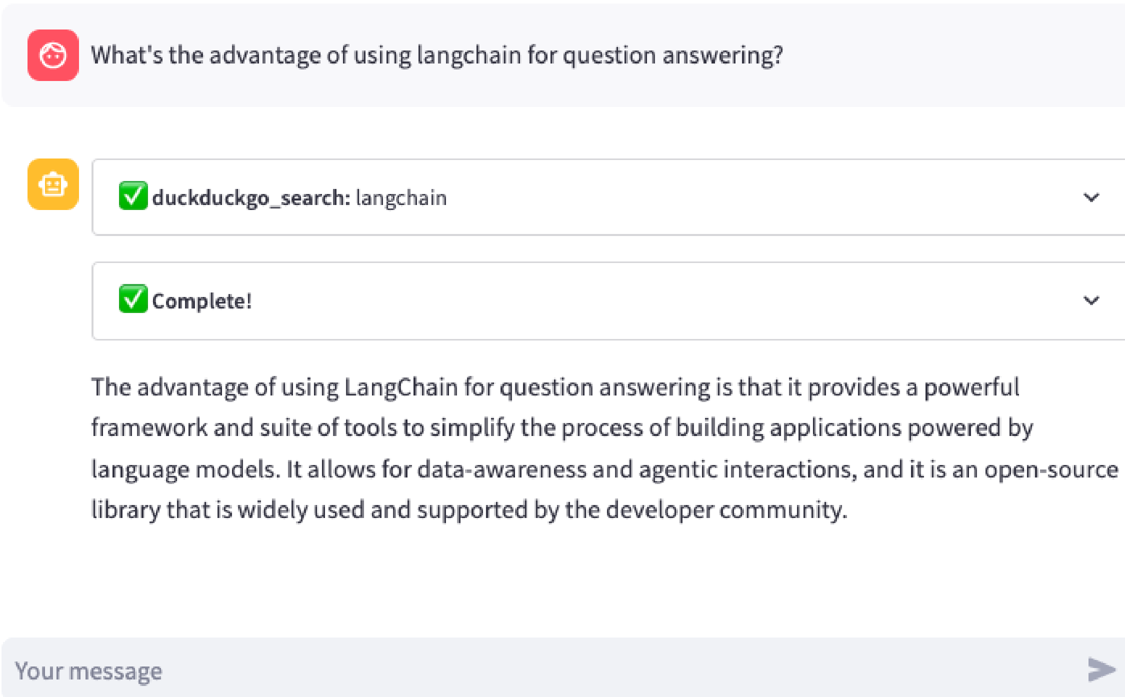


Figure 4.5: Question answering app in Streamlit.

The search works quite well, although depending on the tools used it might still come up with the wrong results. For the question about the mammal with the largest egg, using DuckDuckGo it comes back with a result that discusses eggs in birds and mammals, and sometimes concludes that the ostrich is the mammal with the largest egg, although Platypus also comes back sometimes. Here's the log output (shortened) for the correct reasoning:

```
> Entering new AgentExecutor chain...
I'm not sure, but I think I can find the answer by searching online.
Action: duckduckgo_search
Action Input: "mammal that lays the biggest eggs"
Observation: Posnov / Getty Images. The western long-beaked echidna ...
Final Answer: The platypus is the mammal that lays the biggest eggs.
> Finished chain.
```

You can see that with a powerful framework for automation and problem solving at your behest, you can compress work that can take hundreds of hours into minutes. You can play around with different research questions to see how the tools are used. The actual implementation in the repository for the book allows you to try out different tools, and has an option for self-verification.Retrieval Augmented Generation (RAG) with LLMs can significantly improve the accuracy and quality of responses by injecting relevant data from outside sources into the context. By grounding LLMs with use-case-specific knowledge, we can reduce hallucinations, and make them more useful in real-world scenarios. RAG is more cost-effective and efficient than retraining the models. You can see a very advanced example of augmented information retrieval with LangChain in the BlockAGI project, which is inspired by BabyAGI and AutoGPT, at https://github.com/blockpipe/BlockAGIIn the following sections, we'll compare the main types of agents by their decision making strategies.

Reasoning strategies

The current generation of generative models like LLMs excel at finding patterns in real-world data, such as visual and audio information, and unstructured texts, however, they struggle with symbol manipulation operations required for tasks that involve structured knowledge representation and reasoning. Reasoning problems pose challenges for

LLMs, and there are different reasoning strategies that can complement the pattern completion abilities inherent in neural networks that are generative models. By focusing on enabling symbol-manipulation operations on extracted information, these hybrid systems can enhance the capabilities of language models.**Modular Reasoning, Knowledge, and Language** (**MRKL**) is a framework that combines language models and tools to perform reasoning tasks. In LangChain this consists of three parts:

1. tools,
2. an `LLMChain`, and
3. the agent itself.

The tools are the available resources that the agent can use, such as search engines or databases. The LLMChain is responsible for generating text prompts and parsing the output to determine the next action. The agent class uses the output of the LLMChain to decide which action to take.We've discussed tool use strategies in *Chapter 2*, *Introduction to LangChain*. We can see the reasoning with observation pattern in this diagram:

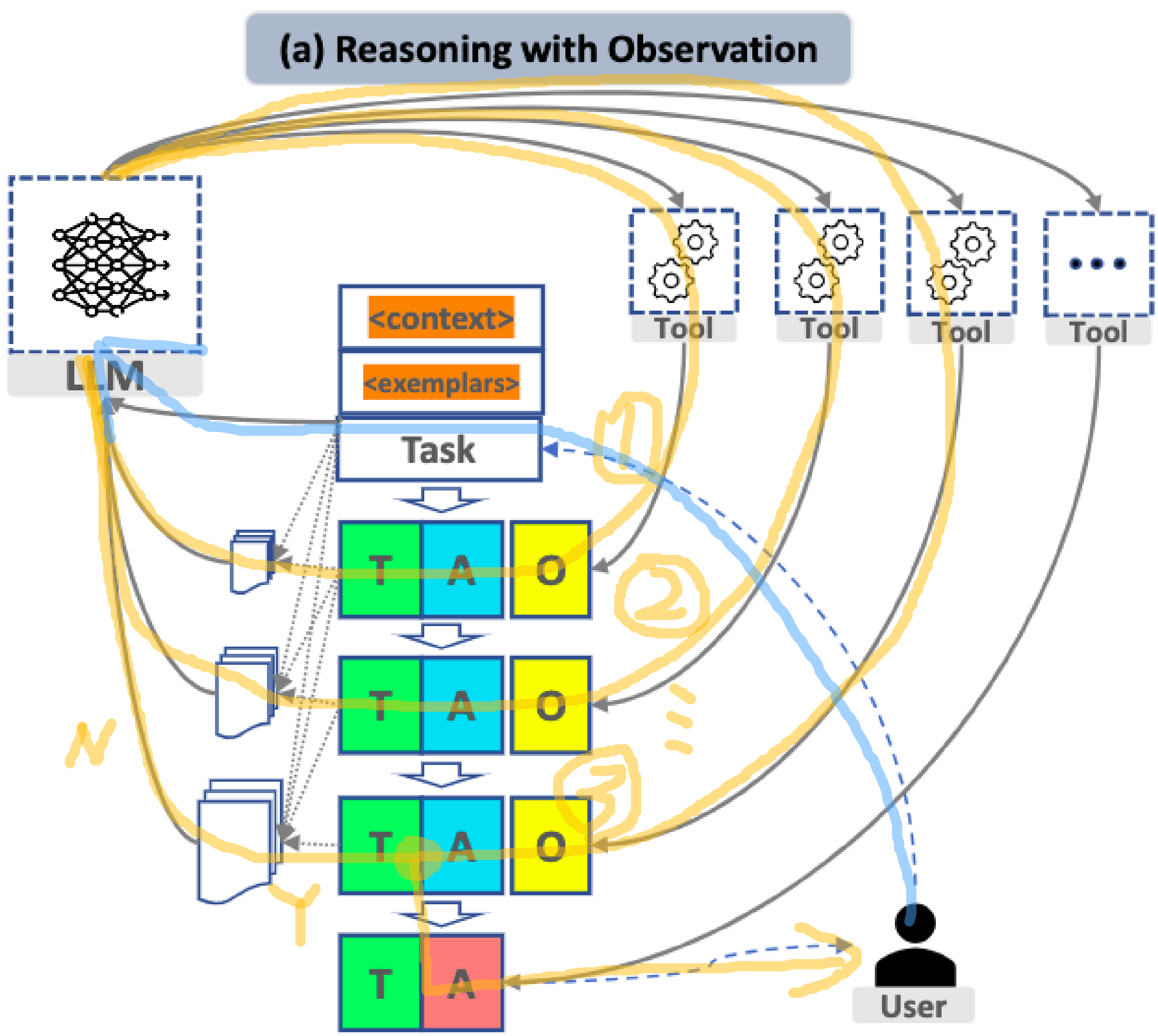


Figure 4.6: Reasoning with observation (source: https://arxiv.org/abs/2305.18323; Binfeng Xu and others, May 2023).

Observation-dependent reasoning involves making judgments, predictions, or choices based on the current state of knowledge or the evidence fetched through observation. In each iteration, the agent is providing a context and examples to a language model (LLM). A user's task is first combined with the context and examples and given to the LLM to initiate reasoning. The LLM generates a thought and an action, and then waits for an observation from tools. The observation is added to the prompt to initiate the next call to the LLM. In LangChain, this is an **action agent** (also: **Zero-Shot Agent**, `ZERO_SHOT_REACT_DESCRIPTION`), which is the default setting when you create an agent. As mentioned, plans can also be made ahead of any actions. This strategy, in LangChain called the **plan-and-execute agent**, is illustrated in the diagram here:

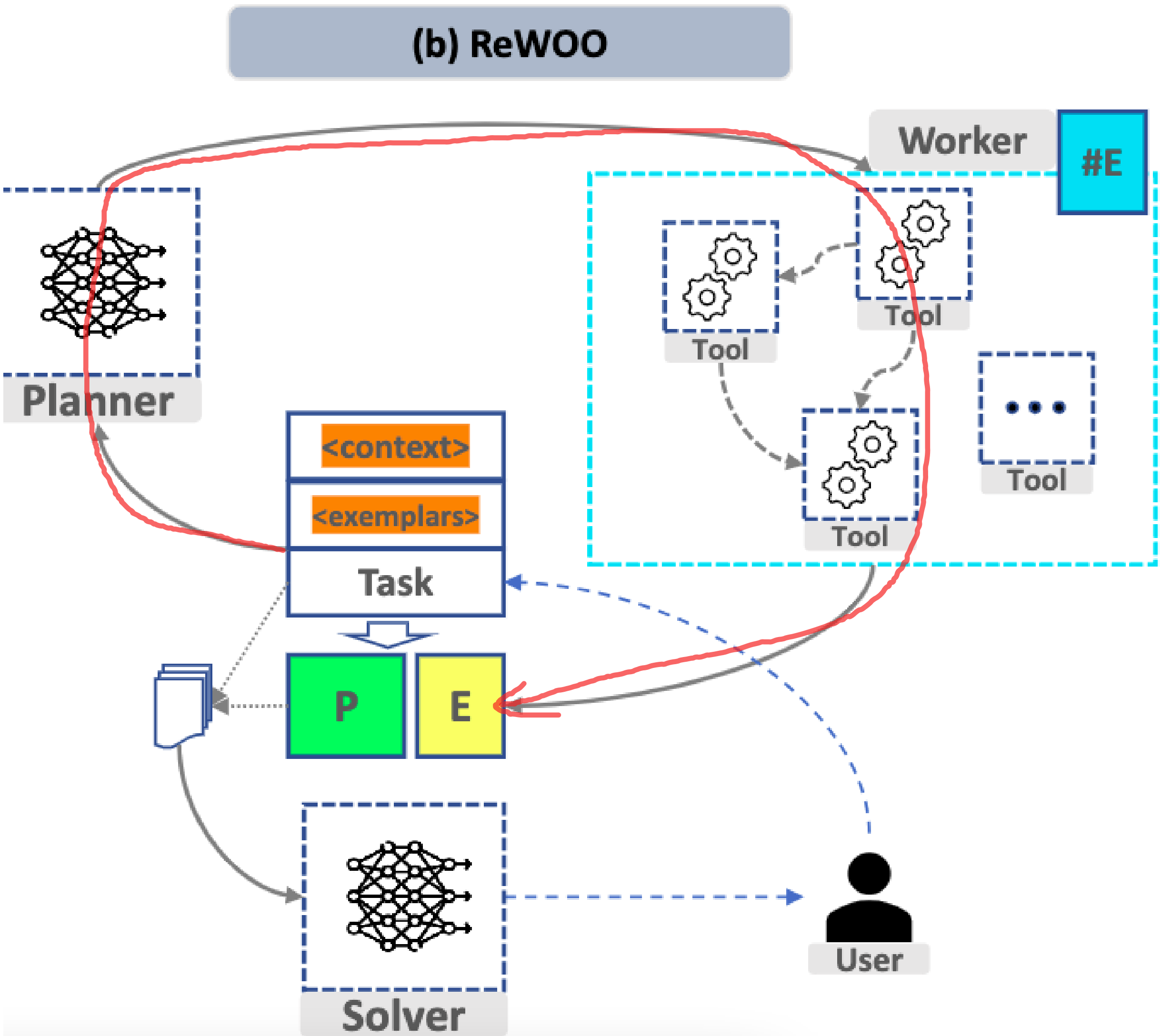


Figure 4.7: Decoupling Reasoning from Observations (source: https://arxiv.org/abs/2305.18323; Binfeng Xu and others, May 2023).

The planner (an LLM), which can be fine-tuned for planning and tool usage, produces a list of plans (P) and calls a worker (in LangChain: the agent) to gather evidence (E) through using tools. P and E are combined with the task, and then fed into the Solver (an LLM) for the final answer. We can write a pseudo algorithm like this:

- Plan out all the steps (Planner)
- For step in steps:

- Determine the proper tools to accomplish the step

The Planner and the Solver can be distinct language models. This opens up the possibility of using smaller, specialized models for Planner and Solver, and using fewer tokens for each of the calls.We can implement plan-and-solve in our research app, let's do it!First, let's add a `strategy` variable to the `load_agent()` function. It can take two values, either "plan-and-solve" or "one-shot-react". For "one-shot-react", the logic stays the same. For "plan-and-solve", we'll define a planner and an executor, which we'll use to create a `PlanAndExecute` agent executor:

```
from typing import Literal
from langchain.experimental import load_chat_planner, load_agent_executor, PlanAndExecute
ReasoningStrategies = Literal["one-shot-react", "plan-and-solve"]
def load_agent(
        tool_names: list[str],
        strategy: ReasoningStrategies = "one-shot-react"
) -> Chain:
    llm = ChatOpenAI(temperature=0, streaming=True)
    tools = load_tools(
        tool_names=tool_names,
        llm=llm
    )
    if strategy == "plan-and-solve":
        planner = load_chat_planner(llm)
        executor = load_agent_executor(llm, tools, verbose=True)
        return PlanAndExecute(planner=planner, executor=executor, verbose=True)
    return initialize_agent(
        tools=tools, llm=llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True
    )
```

For the sake of brevity, I've omitted imports that we had already earlier.Let's define a new variable that's set through a radio button in Streamlit. We'll pass this variable over to the `load_agent()` function:

```
strategy = st.radio(
    "Reasoning strategy",
    ("plan-and-solve", "one-shot-react", ))
```

You might have noticed that the `load_agent()` take a list of strings, `tool_names`. This can be chosen in the user interface (UI) as well:

```
tool_names = st.multiselect(
    'Which tools do you want to use?',
    [
        "google-search", "ddg-search", "wolfram-alpha", "arxiv",
        "wikipedia", "python_repl", "pal-math", "llm-math"
    ],
    ["ddg-search", "wolfram-alpha", "wikipedia"])
```

Finally, still in the app, the agent is loaded like this:

```
agent_chain = load_agent(tool_names=tool_names, strategy=strategy)
```

We can see the UI here:

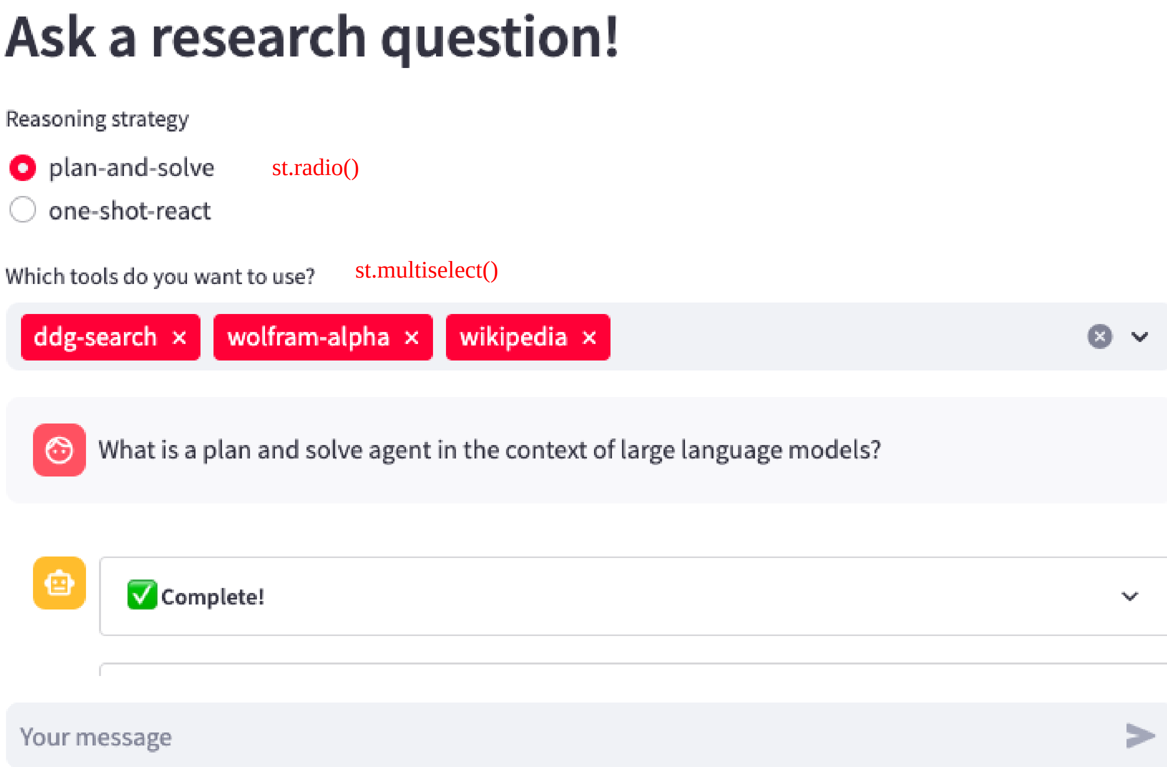


Figure 4.8: Implementing plan-and-execute in our research app.

Please have a look at the app and see the different steps for the question “What is a plan and solve agent in the context of large language models?”. Just briefly, the first step, the plan looks as follows:

1. Define large language models: Large language models are AI models that are trained on vast amounts of text data and can generate human-like text based on the input they receive.
2. Understand the concept of a plan in the context of large language models: In the context of large language models, a plan refers to a structured outline or set of steps that the model generates to solve a problem or answer a question.
3. Understand the concept of a solve agent in the context of large language models: A solve agent is a component of a large language model that is responsible for generating plans to solve problems or answer questions.
4. Recognize the importance of plans and solve agents in large language models: Plans and solve agents help organize the model's thinking process and provide a structured approach to problem-solving or question-answering tasks.
5. Given the above steps taken, please respond to the user's original question: In the context of large language models, a plan is a structured outline or set of steps generated by a solve agent to solve a problem or answer a question. A solve agent is a component of a large language model that is responsible for generating these plans.

Accordingly, the first step is to perform a look up of LLMs:

```
Action:
{
"action": "Wikipedia",
"action_input": "large language models"
}
```

We didn’t discuss another aspect of this, which is the prompting strategy used in these steps. For example, different prompting strategies offer ways to address challenges in complex reasoning problems for LLMs. One approach is **few-shot chain-of-thought** (**CoT**) prompting, where LLMs are guided through step-by-step reasoning demonstrations. For example, in arithmetic reasoning, an LLM can be shown demonstration examples of solving equations to aid its understanding of the process.Another strategy is **zero-shot-CoT** prompting, which eliminates the need for manual demonstrations. Instead, a generic prompt like "Let's think step by step" is appended to the problem statement provided to the LLM. This allows the model to generate reasoning steps without prior explicit examples. In arithmetic reasoning, the problem statement could be augmented with this prompt and fed into an LLM.**Plan-and-Solve (PS) prompting**, involves dividing a complex task into smaller subtasks and executing them step by step according to a plan. For instance, in math reasoning problems like solving equations or word problems involving multiple steps, PS prompting enables an LLM to devise a plan for approaching each sub step, such as extracting relevant variables and calculating intermediate results.To further enhance the quality of reasoning steps and instructions, **PS+** prompting is introduced. It includes more detailed instructions, such as emphasizing the extraction of relevant variables and considering calculation and commonsense. PS+ prompting ensures that the LLMs have a better understanding of the problem and can generate accurate reasoning steps. For example, in arithmetic reasoning, PS+ prompting can guide the LLM to identify key variables, perform calculations correctly, and apply commonsense knowledge during the reasoning process.This concludes our discussion of reasoning strategies. All strategies have their problems with can manifest as calculation errors, missing-step errors, and semantic misunderstandings. However, they help improve the quality of generated reasoning steps, increase accuracy in problem-solving tasks, and enhance LLMs' ability to handle various types of reasoning problems.

Summary

In this chapter, we've talked about the problem of hallucinations, automatic fact-checking, and how to make LLMs more reliable. Of particular emphasis were tools and prompting strategies. We've first looked at and implemented prompting strategies to break down and summarize documents. This can be very helpful for digesting large research articles or analyses. Once we get into making a lot of chained calls to LLMs, this can mean we incur a lot of costs. Therefore, I've dedicated a section to token usage.Tools provide creative solutions to problems and open up new possibilities for LLMs in various domains. For example, a tool could be developed to enable an LLM to perform advanced retrieval search, query a database for specific information, automate email writing, or even handle phone calls.The OpenAI API implements functions, which we can use, among other things, for information extraction in documents. We've implemented a very simple version of a CV parser as an example of this functionality.Tools and function calling is not unique to OpenAI, however. With Streamlit, we can implement different agents that call tools. We've implemented an app that can help answer research questions by relying on external tools such as search engines or Wikipedia.We've then looked at different strategies employed by the agents to make decisions. The main distinction is the point of decision making. We've implemented a plan-and-solve and a one-shot agent into a Streamlit app.I hope this goes to show that in a few lines we can implement apps that can be very impressive in a few cases. It's important to be clear, however, that the apps developed in this chapter have limitations. They can help you significantly increase your efficiency, however, you - as the human - have to apply judgment and improve the writing to make sure it's coherent and makes sense.Let's see if you remember some of the key takeaways from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions from memory. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. What is a hallucination?
2. How does automated fact-checking work?
3. What can we do in LangChain to make sure the output is valid?
4. What is map-reduce in LangChain?
5. How can we count the tokens we are using (and why should we)?
6. What does RAG stand for and what are the advantages of using it?
7. What tools are available in LangChain?
8. Please define plan-and-solve agents
9. Please define one-shot agents
10. How can we implement text input fields and radio buttons in Streamlit?

5 Building a Chatbot like ChatGPT

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

In this chapter, we'll discuss chatbots, what they are, what they can do, and how they can be implemented. We'll start this chapter by discussing the evolution of chatbots and the current state-of-the-art. Understanding and enhancing the capabilities of current chatbots and Large Language Models (LLMs) has practical implications for their safe and effective use in different domains including regulated ones like medicine and law.Proactive communication, important for engaging with customer needs, requires on the technical side, implementations of mechanisms for context and memory. The focus of this chapter is on retrieval mechanisms including vector storage to improve the accuracy of responses and the faithfulness of chatbots to the available information and the current conversation.We'll go through the fundamentals of modern chatbots such as retrieval-augmented language models (RALMs), the technical background of what we need to implement them in LangChain. We'll go into details about methods for loading documents and information including vector storage and embeddings. We'll further discuss more specific methods for memory, which are about maintaining the knowledge and state of the ongoing conversation. Finally, we discuss another important topic from the reputational and legal perspective: moderation. Let's make sure our responses are not abusive, intolerant, or against the spirit of the organization. LangChain allows you to pass any text through a moderation chain to check if it contains harmful content. Throughout the chapter, we'll work on a chatbot implementation with an interface in Streamlit that you can find in the `chat_with_retrieval` directory in the Github repository for the book at https://github.com/benman1/generative_ai_with_langchainThe main sections are:

- What is a chatbot?
- Retrieval and vectors
- Implementing a chatbot
- Don't say anything stupid!

We'll begin the chapter by introducing chatbots and the state-of-the-art of the technology.

What is a chatbot?

A **chatbot** is an Artificial Intelligence program that can chat with users, provide information and support, book things, and perform various other tasks. It is used to reproduce powerful interactions with users and can be utilized in different industries and for different purposes.Chatbots are beneficial because they can automate tasks, provide instant responses, and offer personalized experiences to users. They can be used for customer support, lead generation, sales, information retrieval, and more. Chatbots can save time, improve efficiency, enhance customer experiences, and streamline business processes. Chatbots work by utilizing natural language processing (NLP) and machine learning algorithms. They analyze user input, understand the intent behind it, and generate appropriate responses. They can be designed to work with text-based messaging platforms or voice-based applications.Some use cases for chatbots in customer service include providing 24/7 support, handling frequently asked questions, assisting with product recommendations, processing orders and payments, and resolving simple customer issues. Some more use cases of chatbots include:

- Appointment Scheduling: Chatbots can help users schedule appointments, book reservations, and manage their calendars.

- Information Retrieval: Chatbots can provide users with specific information, such as weather updates, news articles, or stock prices.
- Virtual Assistants: Chatbots can act as personal assistants, helping users with tasks like setting reminders, sending messages, or making phone calls.
- Language Learning: Chatbots can assist in language learning by providing interactive conversations and language practice.
- Mental Health Support: Chatbots can offer emotional support, provide resources, and engage in therapeutic conversations for mental health purposes.
- Education: In educational settings, virtual assistants are being explored as virtual tutors, helping students learn and assess their knowledge, answer questions, and deliver personalized learning experiences.
- HR and Recruitment: Chatbots can assist in the recruitment process by screening candidates, scheduling interviews, and providing information about job openings.
- Entertainment: Chatbots can engage users in interactive games, quizzes, and storytelling experiences.
- Law: Chatbots can be used to provide basic legal information, answer common legal questions, assist with legal research, and help users navigate legal processes. They can also help with document preparation, such as drafting contracts or creating legal forms.
- Medicine: Chatbots can assist with symptom checking, provide basic medical advice, and offer mental health support. They can improve clinical decision-making by providing relevant information and recommendations to healthcare professionals

These are just a few examples, and the use cases of chatbots continue to expand across various industries and domains. Chat technology in any field has the potential to make information more accessible and provide initial support to individuals seeking assistance.

What's the state-of-the-art?

The Turing Test, named after Alan Turing an English computer scientist, cryptanalyst, and mathematician, is a method of inquiry in artificial intelligence (AI) for determining whether or not a computer is capable of thinking like a human being. Despite much debate about the relevance of the Turing Test today and the validity of the competitions that are based around it, the test still stands as a philosophical starting point for discussing and researching AI. As we continue to make advances in AI and better understand and map how the human brain functions, the Turing Test remains foundational for defining intelligence and is a baseline for the debate about what we should expect from technologies for them to be considered thinking machines.Turing proposed that a computer can be said to possess AI if it can mimic human responses under specific conditions. The original Turing Test requires three terminals, each of which is physically separated from the other two. One terminal is operated by a computer, while the other two are operated by humans. During the test, one of the humans works as the questioner, while the second human and the computer function as respondents. The questioner interrogates the respondents within a specific subject area, using a specified format and context. After a preset length of time or number of questions, the questioner is then asked to decide which respondent was human and which was a computer.Since the formation of the test, many AI have been able to pass; one of the first was Joseph Weizenbaum's ELIZA. In 1966, he published an article about his chatbot ELIZA, "ELIZA - a computer program for the study of natural language communication between man and machine." ELIZA was one of the first chatbots ever created and simulated the role of a psychotherapist.Created with a sense of humor to show the limitations of technology, the chatbot employed simplistic rules and vague, open-ended questions as a way of giving an impression of empathic understanding in the conversation, and was an ironic twist often seen as a milestone of artificial intelligence. However, ELIZA had limited knowledge and could only engage in conversations within a specific domain of topics. It also couldn't keep long conversations or learn from the discussion.The Turing Test has been criticized over the years, in particular because historically, the nature of the questioning had to be limited in order for a computer to exhibit human-like intelligence. For many years, a computer might only score high if the questioner formulated the queries, so they had "Yes" or "No" answers or pertained to a narrow field of knowledge. When questions were open-ended and required conversational answers, it was less likely that the computer program could successfully fool the questioner.In addition, a program such as ELIZA could pass the Turing Test by manipulating symbols it does not understand fully. Philosopher John Searle argued that this does not determine intelligence comparable to humans. To many researchers, the question of whether or not a computer can pass a Turing Test has become irrelevant. Instead of focusing on how to convince someone they are conversing with a human and not a computer program, the real focus should be on how to make a human-machine interaction more intuitive and efficient. For example, by using a conversational interface.In 1972, another significant chatbot called PARRY was developed, who acted as a patient with schizophrenia. It had a defined personality, its responses were based on a system of assumptions, and emotional responses were triggered by changes in the user's utterances. In an experiment in 1979, PARRY was tested by five psychiatrists who had to determine whether the patient they were interacting with was a computer program or a real schizophrenic patient. The results varied, with some psychiatrists giving correct diagnoses and others giving incorrect ones.Although several variations of the Turing Test are often more applicable to our current understanding of AI, the original format of the test is still used to this day. For example, the Loebner Prize has been awarded annually since 1990 to the most human-like computer program as voted by a panel of judges. The competition follows the standard rules of the Turing Test. Critics of the award's relevance often downplay it as more about publicity than truly testing if machines can think.IBM Watson is a cognitive computing system developed by IBM that uses natural language processing, machine learning, and other AI technologies to respond to complex questions in natural language. It works by ingesting and processing vast amounts of structured and unstructured data, including text, images, and videos. IBM Watson became famous in 2011 when it competed on the quiz show Jeopardy! and defeated two former champions. Watson has been applied in various fields, including healthcare, finance, customer service, and research. In healthcare, Watson has been used to assist doctors in diagnosing and treating diseases, analyzing medical records, and conducting research. It has also been applied in the culinary field, with the Chef Watson application helping chefs create unique and innovative recipes.In 2018, Google Duplex successfully made an appointment with a hairdresser over the phone in front of a crowd of 7,000. The receptionist was completely unaware that they weren't conversing with a real human. This is considered by some to be a modern-day Turing Test pass, despite not relying on the true format of the test as Alan Turing designed it.Developed by OpenAI, ChatGPT is a language model that uses deep learning techniques to generate human-like responses. It was launched on November 30, 2022, and is built upon OpenAI's proprietary series of foundational GPT models, including GPT-3.5 and GPT-4. ChatGPT allows users to have coherent, natural, and engaging conversations with the AI, refining and steering the conversation towards their desired length, format, style, level of detail, and language used.ChatGPT is considered a game changer because it represents a significant advancement in conversational AI. Because of its ability to generate contextually relevant responses and to understand and respond to a wide range of topics and questions, the chatbot is thought by some to have the best chance of beating the test in its true form of any technology that we have today. But, even with its advanced text-generation abilities, it can be tricked into answering nonsensical questions and therefore would struggle under the conditions of the Turing Test.Overall, ChatGPT's capabilities and user-friendly interface have made it a significant advancement in the field of conversational AI, offering new possibilities for interactive and personalized interactions with AI systems.

Here are some examples of chatbots:

- ELIZA: One of the earliest chatbots, ELIZA was developed in the 1960s and used pattern matching to simulate conversation with users.
- Siri: Siri is a popular voice-based chatbot developed by Apple. It is integrated into Apple devices and can perform tasks, answer questions, and provide information.
- Alexa: Alexa is an intelligent personal assistant developed by Amazon. It can respond to voice commands, play music, provide weather updates, and control smart home devices.
- Google Assistant: Google Assistant is a chatbot developed by Google. It can answer questions, provide recommendations, and perform tasks based on user commands.
- Mitsuku: Mitsuku is a chatbot that has won the Loebner Prize Turing Test multiple times. It is known for its ability to engage in natural and human-like conversations.

These are just a few examples, and there are many more chatbots available in various industries and applications.

One concern of the use of the Turing test and derivatives is that it focuses on imitation and deceit, when it more meaningful tests should emphasize the need for developers to focus on creating useful and interesting capabilities rather than just performing tricks. The use of benchmarks and academic/professional examinations provides more specific evaluations of AI system performance.The current objective of the researchers in the field is to provide a better benchmark for testing the capabilities of artificial intelligence (AI) systems, specifically large language models (LLMs) such as GPT-4. They aim to understand the limits of LLMs and identify areas where they may fail. The advanced AI systems, including GPT-4, excel in tasks related to language processing but struggle with simple visual logic puzzles. LLMs can generate plausible next words based on statistical correlations but may lack reasoning or understanding of abstract concepts. Researchers have different opinions about the capabilities of LLMs, with some attributing their achievements to limited reasoning abilities. The research on testing LLMs and understanding their capabilities has practical implications. It can help in the safe and effective application of LLMs in real-world domains such as medicine and law. By identifying the strengths and weaknesses of LLMs, researchers can determine how to best utilize them.ChatGPT's training has made it better at handling hallucinations compared to its predecessors, which means that it is less likely to generate nonsensical or irrelevant responses. However, it is important to note that ChatGPT can still confidently present inaccurate information, so users should exercise caution and verify the information provided.Context and memory play significant roles in ensuring the chatbot's dialogues deliver accurate information and responses accurately reflective of previous interactions, thus enabling more faithful engagements with users. We'll discuss this in more detail now.

Context and Memory

Context and memory are important aspects of chatbot design. They allow chatbots to maintain conversational context, respond to multiple queries, and store and access long-term memory. They are important factors in conditioning chatbot responses for accuracy and faithfulness. The significance of memory and context in chatbots can be compared to the significance of memory and comprehension in human-human conversations. A conversation without recalling past exchanges or comprehending or knowing about the broader context can be disjointed and cause miscommunication, resulting in an unsatisfactory conversational experience.**Contextual understanding** dramatically impacts the accuracy of chatbot responses. It refers to the ability of the chatbot to comprehend both the entire conversation and what some of the relevant background rather than just the last message from the user. A chatbot that is conscious of the context can maintain a holistic perspective of the conversation, making the chat flow more natural and human-like.**Memory retention** directly influences the faithfulness of the chatbot's performance, which involves consistency in recognizing and remembering facts from previous conversations for future use. This feature enhances the personalized experience for the user.For instance, if a user says, "Show me the cheapest flights," and then follows with, "How about hotels in that area?" without the context of the previous messages, the chatbot wouldn't have a clue what area the user is referring to. In a reversed scenario, a context-aware chatbot would understand that the user is talking about accommodation in the same vicinity as the flight destination.A lack of memory results in inconsistencies throughout conversations (lack of faitfulness). For example, if a user has identified themselves by name in one conversation and the bot forgets this information in the next, it creates an unnatural and impersonal interaction.Both memory and context are vital to making chatbot interactions more productive, accurate, personable, and satisfactory. Without these elements, the bots may come across as deficient, rigid, and unrelatable to their human conversational counterparts. Hence, these characteristics are essential for sophisticated and satisfying interactions between computers and humans.A new aspect of chatbots with LLMs is that they can not only respond to intentions, but more intelligently engage in a dialogue with the user. This is called being proactive.

Intentional vs Proactive

In the context of language models or chatbots, **proactive** refers to the ability of the system to initiate actions or provide information without being explicitly prompted by the user. It involves anticipating the user's needs or preferences based on previous interactions or contextual cues. On the other hand, **intentional** means that the chatbot is designed to understand and fulfill the user's intentions or requests and is programmed to take specific actions or provide relevant responses based on these intentions and the desired outcome.A proactive chatbot is useful because it can connect with customers and improve their experience, creating a better customer journey. This can enhance the user experience by saving time and effort, and it can also improve customer satisfaction by addressing customer inquiries quickly and efficiently potential issues or questions before they arise.Proactive communication is critical for the success of businesses as it improves customer lifetime value (CLV) and reduces operating costs. By actively anticipating customers' needs and providing information proactively, businesses can gain control over communication and frame conversations in a favorable light. This builds trust, customer loyalty, and enhances the organization's reputation. Additionally, proactive communication helps improve organizational productivity by addressing customer inquiries before they ask and reducing incoming support calls.On the technical side, this capability can be achieved through context and memory, and reasoning mechanisms. This is the focus of this chapter. In the next section, we'll discuss the fundamentals of modern chatbots such as retrieval-augmented language models (RALMs) and the technical background of what we need to implement them.

Retrieval and vectors

In chapter 4, we discussed Retrieval-Augmented Generation (RAG), which aims to enhance the generation process by leveraging external knowledge and ensuring that the generated text is accurate and contextually appropriate. In this chapter, we'll further discuss how to combine retrieval and generation techniques to improve the quality and relevance of generated text. Particularly, we'll discuss Retrieval-Augmented Language Models (RALMs), a specific implementation or application of RAG, which refers to language models that are conditioned on relevant documents from a grounding corpus, a collection of written texts, during generation. In the retrieval, semantic filtering and vector storage is utilized to pre-filter relevant information from a large corpus of documents and incorporating that information into the generation process. This retrieval includes vector storage of documents.

> **Retrieval-Augmented Language Models** (**RALMs**) are language models that incorporate a retrieval component to enhance their performance. Traditional language models generate text based on the input prompt, but RALMs go a step further by retrieving relevant information from a large collection of documents and using that information to generate more accurate and contextually relevant responses.

The benefits of RALMs include:

1. Improved performance: By incorporating active retrieval, LMs can access relevant information from external sources, which can enhance their ability to generate accurate and informative responses.
2. Avoiding input length limitations: Retrieval augmented LMs discard previously retrieved documents and only use the retrieved documents from the current step to condition the next generation. This helps prevent reaching the input length limit of LMs, allowing them to handle longer and more complex queries or tasks.

More in detail, the working mechanism of retrieval augmented LMs involves the following steps:

1. Retrieval: RALMs search for relevant documents or passages from a large corpus. The LM retrieves relevant information from external sources based on a vector-based similarity search of the query and the current context.
2. Conditioning: The retrieved information is used to condition the LLM's next generation. This means that the LM incorporates the retrieved information into its language model to generate more accurate and contextually appropriate responses.
3. Iterative process: The retrieval and conditioning steps are performed iteratively, with each step building upon the previous one. This iterative process allows the LLM to gradually improve its understanding and generation capabilities by incorporating relevant information from external sources.

The retrieved information can be used in different ways. It can serve as additional context for the language model, helping it generate more accurate and contextually appropriate responses. It can also be used to provide factual information or answer specific questions within the generated text. There are two main strategies for retrieval augmented generation:

1. Single-time Retrieval-Augmented Generation: This strategy involves using the user input as the query for retrieval and generating the complete answer at once. The retrieved documents are concatenated with the user input and used as input to the language model for generation.
2. Active Retrieval Augmented Generation: This strategy involves actively deciding when and what to retrieve during the generation process. At each step of generation, a retrieval query is formulated based on both the user input and the previously generated output. The retrieved documents are then used as input to the language model for generation. This strategy allows for the interleaving of retrieval and generation, enabling the model to dynamically retrieve information as needed.

Within the active retrieval augmented generation framework, there are two forward-looking methods called FLARE (Forward-Looking Active Retrieval Augmented Generation):

- FLARE with Retrieval Instructions: This method prompts the language model to generate retrieval queries when necessary while generating the answer. It uses retrieval-encouraging instructions, such as "[Search(query)]", to express the need for additional information.

- FLARE Direct: This method directly uses the language model's generation as search queries. It iteratively generates the next sentence to gain insight into the future topic, and if uncertain tokens are present, it retrieves relevant documents to regenerate the next sentence.

Unlike traditional methods where information is retrieved only once and then used for generation, FLARE follows an iterative process. It involves using a prediction of the upcoming sentence as a query to retrieve relevant documents. This allows the system to regenerate the sentence if the confidence in the initial generation is low. RALMs have shown promising results in tasks like question answering, dialogue systems, and information retrieval. They can provide more accurate and informative responses by leveraging external knowledge sources. Additionally, RALMs can be fine-tuned for specific domains or tasks by training them on domain-specific document collections, further enhancing their usefulness in specialized applications.Overall, by incorporating retrieval, RALMs can leverage the vast amount of knowledge present in the document corpus, making them more powerful and useful for various natural language processing tasks. RALMs leverage active retrieval to enhance their performance and overcome limitations in handling complex queries or tasks.LangChain implements a tool chain of different building blocks for building retrieval systems. This includes data loaders, document transformers, embedding models, vector stores, and retrievers. The relationship between them is illustrated in the diagram here (source: LangChain documentation):

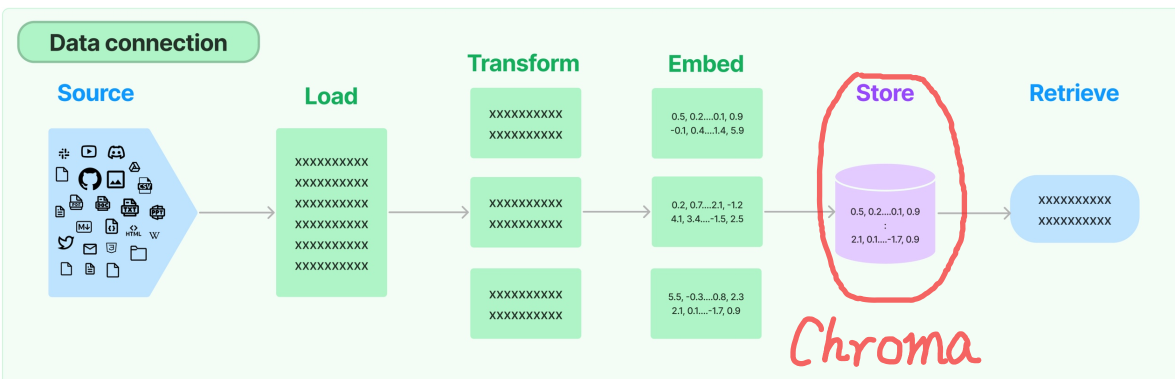


Figure 5.1: Vector stores and data loaders.

In LangChain, we first load documents through data loaders. Then we can transform them, pass these documents to a vector store as embedding. We can then query the vector store or a retriever associated with the vector store. Retrievers in LangChain can wrap the loading and vector storage into a single step. We'll mostly skip transformations in this chapter, however, you'll find explanations with examples of data loaders, embeddings, storage mechanisms, and retrievers.Since we are talking about vector storage, we need to discuss vector search, which is a technique used to search and retrieve vectors (or embeddings) based on their similarity to a query vector. It is commonly used in applications such as recommendation systems, image and text search, and anomaly detection. We'll look into more of the fundamentals behind RALMs, and we'll start with embeddings now. Once you understand embeddings, you'll be able to build everything from search engines to chatbots.

Embeddings

An embedding is a numerical representation of a content in a way that machines can process and understand. The essence of the process is to convert an object such as an image or a text into a vector that encapsulates its semantic content while discarding irrelevant details as much as possible. An embedding takes a piece of content, such as a word, sentence, or image, and maps it into a multi-dimensional vector space. The distance between two embeddings indicates the semantic similarity between the corresponding concepts (the original content).

> **Embeddings** are representations of data objects generated by machine learning models to represent. They can represent words or sentences as numerical vectors (lists of float numbers). As for the OpenAI language embedding models, the embedding is a vector of 1,536 floating point numbers that represent the text. These numbers are derived from a sophisticated language model that captures semantic content.

> As an example – let's say we have the words cat and dog – these could be represented numerically in a space together with all other words in the vocabulary. If the space is 3-dimensional, these could be vectors such as [0.5, 0.2, -0.1] for cat and [0.8, -0.3, 0.6] for dog. These vectors encode information about the relationships of these concepts with other words. Roughly speaking, we would expect the concepts cat and dog to be closer (more similar) to the concept of animal than to the concept of computer or embedding.

Embeddings can be created using different methods. 1 For texts, one simple method is the **bag-of-words** approach, where each word is represented by a count of how many times it appears in a text. This approach, which in the scikit-learn library is implemented as `CountVectorizer`, was popular until **word2vec** came about. 2 Word2vec, which – roughly speaking – learns embeddings by predicting the words in a sentence based on other surrounding words ignoring the word order in a linear model. The general idea of embeddings is illustrated in this figure (source: "Analogies Explained: Towards Understanding Word Embeddings" by Carl Allen and Timothy Hospedales, 2019; https://arxiv.org/abs/1901.09813):

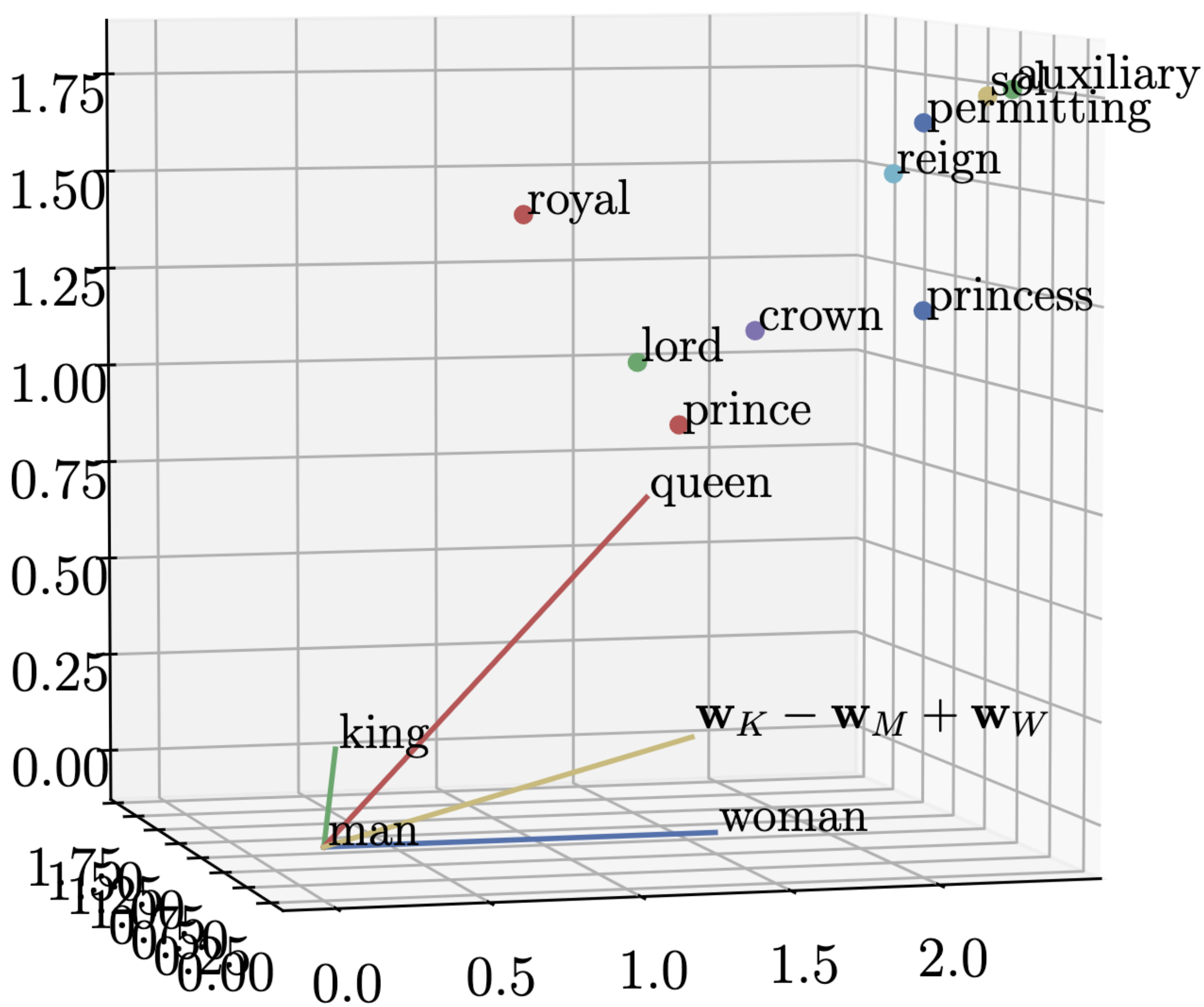


Figure 5.2: Word2Vec word embeddings in a 3D space. We can perform simple vector arithmetic with these vectors, for example the vector for king minus man plus the vector for woman gives us a vector that comes closer to queen.

As for images, embeddings could come from feature extraction stages such as edge detection, texture analysis, and color composition. These features can be extracted over different window sizes to make the representations both scale-invariant and shift-invariant (**scale-space representations**). Nowadays, often, **Convolutional Neural**

Networks (**CNNs**) are pre-trained on large datasets (like ImageNet) to learn a good representation of the image's properties. Since convolutional layers apply a series of filters (or kernels) on the input image to produce a feature map, conceptually this is similar to scale-space. When a pre-trained CNN then runs over a new image, it can output an embedding vector. Today, for most domains including texts and images, embeddings usually come from **transformer-based models**, which consider the context and order of the words in a sentence and the paragraph. Based on the model architecture, most importantly the number of parameters, these models can capture very complex relationships. All these models are trained on large datasets in order to establish the concepts and their relationships.These embeddings can be used in various tasks. By representing data objects as numerical vectors, we can perform mathematical operations on them and measure their similarity or use them as input for other machine learning models. By calculating distances between embeddings, we can perform tasks like search and similarity scoring, or classify objects, for example by topic or category. For example, we could be performing a simple sentiment classifier by checking if embeddings of product reviews are closer to the concept of positive or negative.

Distances metrics between embeddings

There are different distance metrics used in vector similarity calculations such as:

- The **cosine distance** is a similarity measure that calculates the cosine of the angle between two vectors in a vector space. It ranges from -1 to 1, where 1 represents identical vectors, 0 represents orthogonal vectors, and -1 represents vectors that are diametrically opposed.
- **Euclidean distance**: It measures the straight-line distance between two vectors in a vector space. It ranges from 0 to infinity, where 0 represents identical vectors, and larger values represent increasingly dissimilar vectors.
- **Dot product**: It measures the product of the magnitudes of two vectors and the cosine of the angle between them. It ranges from -∞ to ∞, where a positive value represents vectors that point in the same direction, 0 represents orthogonal vectors, and a negative value represents vectors that point in opposite directions.

In LangChain, you can obtain an embedding by using the `embed_query()` method from the `OpenAIEmbeddings` class. Here is an example code snippet:

```
from langchain.embeddings.openai import OpenAIEmbeddings
embeddings = OpenAIEmbeddings()
text = "This is a sample query."
query_result = embeddings.embed_query(text)
print(query_result)
print(len(query_result))
```

This code passes a single string input to the embed_query method and retrieves the corresponding text embedding. The result is stored in the `query_result` variable. The length of the embedding (the number of dimensions) can be obtained using the `len()` function. I am assuming you've set the API key as environment variable as recommended in chapter 3, *Getting started in LangChain*.You can also obtain embeddings for multiple document inputs using the `embed_documents()` method. Here is an example:

```
from langchain.embeddings.openai import OpenAIEmbeddings
words = ["cat", "dog", "computer", "animal"]
embeddings = OpenAIEmbeddings()
doc_vectors = embeddings.embed_documents(words)
```

In this case, the `embed_documents()` method is used to retrieve embeddings for multiple text inputs. The result is stored in the `doc_vectors` variable. We could have retrieved embeddings for long documents – instead, we've retrieved the vectors only for single words each.We can also do arithmetic between these embeddings, for example calculate distances between them:

```
from scipy.spatial.distance import pdist, squareform
import pandas as pd
X = np.array(doc_vectors)
dists = squareform(pdist(X))
```

This gives us the Euclidean distances between our words as a square matrix. Let's plot them:

```
import pandas as pd
df = pd.DataFrame(
```

```
    data=dists,
    index=words,
    columns=words
)
df.style.background_gradient(cmap='coolwarm')
```

The distance plot should look like this:

	cat	dog	computer	animal
cat	0.000000	0.522352	0.575285	0.521214
dog	0.522352	0.000000	0.581203	0.478794
computer	0.575285	0.581203	0.000000	0.591435
animal	0.521214	0.478794	0.591435	0.000000

Figure 5.3: Euclidean distances between embeddings of the words cat, dog, computer, animal.

We can confirm: a cat and a dog are indeed closer to an animal than to a computer. There could be many questions here, for example, if a dog is more an animal than a cat, or why a dog and a cat are only little more distant from a computer than from an animal. Although, these questions can be important in certain applications, let's bear in mind that this is a simple example.In these examples, we've used OpenAI embeddings – in the examples further on, we'll use embeddings from models served by Huggingface. There are a few integrations and tools in LangChain that can help with this process, some of which we'll encounter further on in this chapter. Additionally, LangChain provides a `FakeEmbeddings` class that can be used to test your pipeline without making actual calls to the embedding providers.In the context of this chapter, we'll use them for retrieval of related information (semantic search). However, we still need to talk about the integrations of these embeddings into apps and broader systems, and this is where vector storage comes in.

2023/11/1

How can we store embeddings?

As mentioned, in vector search, each data point is represented as a vector in a high-dimensional space. The vectors capture the features or characteristics of the data points. The goal is to find the vectors that are most similar to a given query vector. In vector search, every data object in a dataset is assigned a vector embedding. These embeddings are arrays of numbers that can be used as coordinates in a high-dimensional space. The distance between vectors can be computed using distance metrics like cosine similarity or Euclidean distance. To perform a vector search, the query vector (representing the search query) is compared to every vector in the collection. The distance between the query vector and each vector in the collection is calculated, and objects with smaller distances are considered more similar to the query.To perform vector search efficiently, vector storage mechanisms are used such as vector databases.

> **Vector search** refers to the process of searching for similar vectors among other stored vectors, for example in a vector database, based on their similarity to a given query vector. Vector search is commonly used in various applications such as recommendation systems, image and text search, and similarity-based retrieval. The goal of vector search is to efficiently and accurately retrieve vectors that are most similar to the query vector, typically using similarity measures such as the dot product or cosine similarity.

A vector storage refers to mechanism used to store vector embeddings, and which is also relevant to how they can be retrieved. Vector storage can be a standalone solution that is specifically designed to store and retrieve vector embeddings efficiently. On the other hand, vector databases are purpose-built to manage vector embeddings and provide several advantages over using standalone vector indices like FAISS. Let's dive into a few of these concepts a bit more. There are three levels to this:

1. Indexing
2. Vector libraries
3. Vector databases

These components work together for the creation, manipulation, storage, and efficient retrieval of vector embeddings. Indexing organizes vectors to optimize retrieval, structuring them so that vectors can be retrieved quickly. There are different algorithms like k-d trees or Annoy for this. Vector libraries provide functions for vector operations like dot product and vector indexing. Finally, vector databases like Milvus or Pinecone are designed to store, manage, and retrieve large sets of vectors. They use indexing mechanisms to facilitate efficient similarity searches on these vectors. Let's look at these in turn. There's a fourth level in LangChain, which is retrievers, and which we'll cover last.

Vector indexing

Indexing in the context of vector embeddings is a method of organizing data to optimize its retrieval and/or the storage. It's similar to the concept in traditional database systems, where indexing allows quicker access to data records. For vector embeddings, indexing aims to structure the vectors – roughly speaking - so that similar vectors are stored next to each other, enabling fast proximity or similarity searches. A typical algorithm applied in this context is K-dimensional trees (k-d trees), but many others like Ball Trees, Annoy, and FAISS are often implemented, especially for high-dimensional vectors which traditional methods can struggle with.

> **K-Nearest Neighbor** (**KNN**) is a simple and intuitive algorithm used for classification and regression tasks. In KNN, the algorithm determines the class or value of a data point by looking at its k nearest neighbors in the training dataset.
>
> Here's how KNN works:

- Choose the value of k: Determine the number of nearest neighbors (k) that will be considered when making predictions.
- Calculate distances: Calculate the distance between the data point you want to classify and all other data points in the training dataset. The most commonly used distance metric is Euclidean distance, but other metrics like Manhattan distance can also be used.
- Find the k nearest neighbors: Select the k data points with the shortest distances to the data point you want to classify.
- Determine the majority class: For classification tasks, count the number of data points in each class among the k nearest neighbors. The class with the highest count becomes the predicted class for the data point. For regression tasks, take the average of the values of the k nearest neighbors.
- Make predictions: Once the majority class or average value is determined, assign it as the predicted class or value for the data point.

> It's important to note that KNN is a lazy learning algorithm, meaning it does not explicitly build a model during the training phase. Instead, it stores the entire training dataset and performs calculations at the time of prediction.

As for alternatives to KNN, there are several other algorithms commonly used for similarity search indexing. Some of them include:

1. Product Quantization (PQ): PQ is a technique that divides the vector space into smaller subspaces and quantizes each subspace separately. This reduces the dimensionality of the vectors and allows for efficient storage and search. PQ is known for its fast search speed but may sacrifice some accuracy.

2. Locality Sensitive Hashing (LSH): This is a hashing-based method that maps similar data points to the same hash buckets. It is efficient for high-dimensional data but may have a higher probability of false positives and false negatives.
3. Hierarchical Navigable Small World (HNSW): HNSW is a graph-based indexing algorithm that constructs a hierarchical graph structure to organize the vectors. It uses a combination of randomization and greedy search to build a navigable network, allowing for efficient nearest neighbor search. HNSW is known for its high search accuracy and scalability.

Examples for PQ are KD-Trees and Ball Trees. In KD-Trees, a binary tree structure is built up that partitions the data points based on their feature values. It is efficient for low-dimensional data but becomes less effective as the dimensionality increases. Ball Tree: A tree structure that partitions the data points into nested hyperspheres. It is suitable for high-dimensional data but can be slower than KD-Tree for low-dimensional data.Apart from HNSW and KNN, there are other graph-based methods like Graph Neural Networks (GNN) and Graph Convolutional Networks (GCN) that leverage graph structures for similarity search.The Annoy (Approximate Nearest Neighbors Oh Yeah) algorithm uses random projection trees to index vectors. It constructs a binary tree structure where each node represents a random hyperplane. Annoy is simple to use and provides fast approximate nearest neighbor search.These indexing algorithms have different trade-offs in terms of search speed, accuracy, and memory usage. The choice of algorithm depends on the specific requirements of the application and the characteristics of the vector data.

Vector libraries

Vector libraries, like Facebook (Meta) Faiss or Spotify Annoy, provide functionality for working with vector data. In the context of vector search, a vector library is specifically designed to store and perform similarity search on vector embeddings. These libraries use the Approximate Nearest Neighbor (ANN) algorithm to efficiently search through vectors and find the most similar ones. They typically offer different implementations of the ANN algorithm, such as clustering or tree-based methods, and allow users to perform vector similarity search for various applications.Here's a quick overview over some open-source libraries for vector storage that shows their popularity in terms of Github stars over time (source: star-history.com):

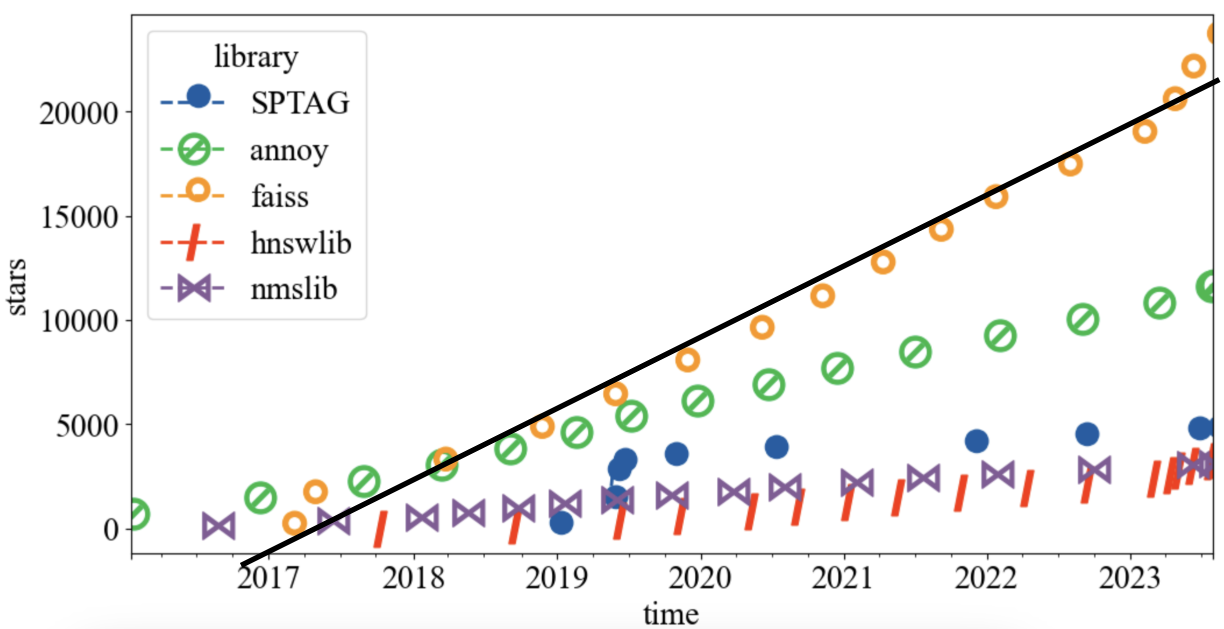


Figure 5.4: Star history for several popular open-source vector libraries.

You can see that faiss has been starred a lot by Github users. Annoy comes second. Others have not found the same popularity yet.Let's quickly go through these:

- FAISS (Facebook AI Similarity Search) is a library developed by Meta (previously Facebook) that provides efficient similarity search and clustering of dense vectors. It offers various indexing algorithms, including PQ, LSH, and HNSW. FAISS is widely used for large-scale vector search tasks and supports both CPU and GPU acceleration.
- Annoy is a C++ library for approximate nearest neighbor search in high-dimensional spaces maintained and developed by Spotify implementing the Annoy algorithm. It is designed to be efficient and scalable, making it suitable for large-scale vector data. It works with a forest of random projection trees.
- hnswlib is a C++ library for approximate nearest neighbor search using the Hierarchical Navigable Small World (HNSW) algorithm. It provides fast and memory-efficient indexing and search capabilities for high-dimensional vector data.
- nmslib (Non-Metric Space Library) is an open-source library that provides efficient similarity search in non-metric spaces. It supports various indexing algorithms like HNSW, SW-graph, and SPTAG.
- SPTAG by Microsoft implements a distributed approximate nearest neighborhood search (ANN). It comes with kd-tree and relative neighborhood graph (SPTAG-KDT) and balanced k-means tree and relative neighborhood graph (SPTAG-BKT).

Both nmslib and hnswlib are maintained by Leo Boytsov, who works as a senior research scientist at Amazon, and Yury Malkov.There are a lot more libraries. You can see an overview at https://github.com/erikbern/ann-benchmarks

Vector databases

A **vector database** is a type of database that is specifically designed to handle vector embeddings making it easier to search and query data objects. It offers additional features such as data management, metadata storage and filtering, and scalability. While a vector storage focuses solely on storing and retrieving vector embeddings, a vector database provides a more comprehensive solution for managing and querying vector data. Vector databases can be particularly useful for applications that involve large amounts of data and require flexible and efficient search capabilities across various types of vectorized data, such as text, images, audio, video, and more.

> **Vector databases** can be used to store and serve machine learning models and their corresponding embeddings. The primary application is **similarity search** (also: **semantic search**), where efficiently search through large volumes of text, images, or videos, identifying objects matching the query based on the vector representation. This is particularly useful in applications such as document search, reverse image search, and recommendation systems.
>
> Other use cases for vector databases are continually expanding as the technology evolves, however, some common use cases for vector databases include:

- Anomaly Detection: Vector databases can be used to detect anomalies in large datasets by comparing the vector embeddings of data points. This can be valuable in fraud detection, network security, or monitoring systems where identifying unusual patterns or behaviors is crucial.
- Personalization: Vector databases can be used to create personalized recommendation systems by finding similar vectors based on user preferences or behavior.
- Natural Language Processing (NLP): Vector databases are widely used in NLP tasks such as sentiment analysis, text classification, and semantic search. By representing text as vector embeddings, it becomes easier to compare and analyze textual data.

These databases are popular because they are optimized for scalability and representing and retrieving data in high-dimensional vector spaces. Traditional databases are not designed to efficiently handle large-dimensional vectors, such as those used to represent images or text embeddings.The characteristics of vector databases include:

1. Efficient retrieval of similar vectors: Vector databases excel at finding close embeddings or similar points in a high-dimensional space. This makes them ideal for tasks like reverse image search or similarity-based recommendations.
2. Specialized for specific tasks: Vector databases are designed to perform a specific task, such as finding close embeddings. They are not general-purpose databases and are tailored to handle large amounts of vector data efficiently.

3. Support for high-dimensional spaces: Vector databases can handle vectors with thousands of dimensions, allowing for complex representations of data. This is crucial for tasks like natural language processing or image recognition.
4. Enable advanced search capabilities: With vector databases, it becomes possible to build powerful search engines that can search for similar vectors or embeddings. This opens up possibilities for applications like content recommendation systems or semantic search.

Overall, vector databases offer a specialized and efficient solution for handling large-dimensional vector data, enabling tasks like similarity search and advanced search capabilities.The market for open-source software and databases is currently thriving due to several factors. Firstly, artificial intelligence (AI) and data management have become crucial for businesses, leading to a high demand for advanced database solutions. In the database market, there is a history of new types of databases emerging and creating new market categories. These market creators often dominate the industry, attracting significant investments from venture capitalists (VCs). For example, MongoDB, Cockroach, Neo4J, and Influx are all examples of successful companies that introduced innovative database technologies and achieved substantial market share. The popular Postgres has an extension for efficient vector search: pg_embedding. Using the Hierarchical Navigable Small Worlds (HNSW) it provides a faster and more efficient alternative to the pgvector extension with IVFFlat indexing.VCs are actively seeking the next groundbreaking type of database, and vector databases, such as Chroma and Marqo, have the potential to be the next big thing. This creates a competitive landscape where companies can raise significant amounts of funding to develop and scale their products.Some examples of vector databases are listed in this table:

Database provider	**Description**	**Business model**	**First released**	**License**	**Indexing**	**Organization**
Chroma	Commercial open-source embedding store	(Partly Open) SaaS model	2022	Apache-2.0	HNSW	Chroma Inc
Qdrant	Managed/Self-hosted vector search engine and database with extended filtering support	(Partly Open) SaaS model	2021	Apache 2.0	HNSW	Qdrant Solutions GmbH
Milvus	Vector database built for scalable similarity search	(Partly Open) SaaS	2019	BSD	IVF, HNSW, PQ, and more	Zilliz
Weaviate	Cloud-native vector database that stores both objects and vectors	Open SaaS	started in 2018 as a traditional graph database, first released in 2019	BSD	custom HNSW algorithm that supports CRUD	SeMI Technologies
Pinecone	Fast and scalable applications using embeddings from AI models	SaaS	first released in 2019	proprietary	built on top of Faiss	Pinecone Systems Inc
Vespa	Commercial Open Source vector database which supports vector search, lexical search, and search	Open SaaS	originally a web search engine (alltheweb), acquired by Yahoo! in 2003 and later developed into and open sourced as Vespa in 2017	Apache 2.0	HNSW, BM25	Yahoo!
Marqo	Cloud-native commercial Open Source search and analytics engine	Open SaaS	2022	Apache 2.0	HNSW	S2Search Australia Pty Ltd

Figure 5.5: Vector databases.

I took the liberty to highlight for each search engine the following perspectives:

- Value proposition. What is the unique feature that makes the whole vector search engine stand out from the crowd?
- Business model. General type of this engine: vector database, big data platform. Managed / Self-hosted.
- Indexing. What algorithm approach to similarity / vector search was taken by this search engine and what unique capabilities it offers?
- License: is it open or close source?

I've omitted other aspects such as Architecture, for example support for sharding or in-memory processing. There are many vector database providers. I've omitted many solutions such as FaissDB or Hasty.ai, and focused on a few ones, which are integrated in LangChain.For the open-source databases, the Github star histories give a good idea of their popularity and traction. Here's the plot over time (source: star-history.com):

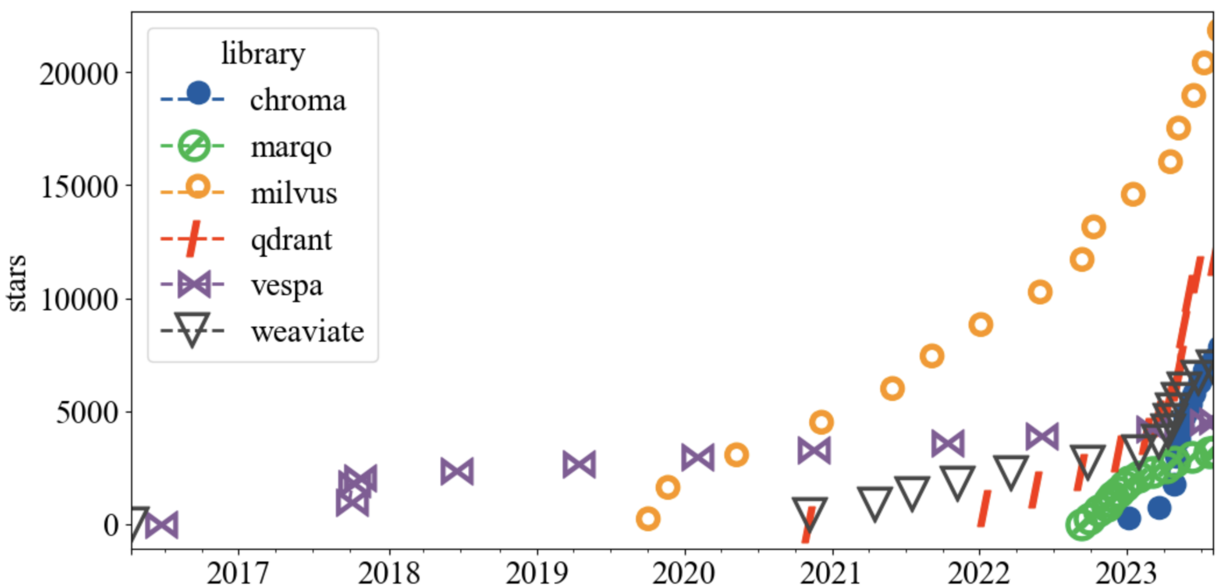


Figure 5.6: Star history of open-source vector databases on Github.

You can see that milvus is very popular, however other libraries such as qdrant, weviate, and chroma have been catching up.In LangChain, a vector storage can be implemented using the `vectorstores` module. This module provides various classes and methods for storing and querying vectors. One example of a vector store implementation in LangChain is the Chroma vector store. Let's see two examples for this!

Chroma

This vector store is optimized for storing and querying vectors using Chroma as a backend. Chroma takes over for encoding and comparing vectors based on their angular similarity. To use Chroma in LangChain, you need to follow these steps:

1. Import the necessary modules:

```
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
```

2. Create an instance of Chroma and provide the documents (splits) and the embedding method:

```
vectorstore = Chroma.from_documents(documents=docs, embedding=OpenAIEmbeddings())
```

The documents (or splits, as seen in chapter 4) will be embedded and stored in the Chroma vector database. We'll discuss document loaders in another section of this chapter. We can use other embedding integrations or we can feed embeddings like this:

```
vector_store = Chroma()
# Add vectors to the vector store:
vector_store.add_vectors(vectors)
```

Here, vectors is a list of numerical vectors (embeddings) that you want to store.We can query the vector store to retrieve similar vectors:

```
similar_vectors = vector_store.query(query_vector, k)
```

Here, `query_vector` is the vector you want to find similar vectors to, and k is the number of similar vectors you want to retrieve.

Pinecone

Here are the steps to integrate Pinecone with LangChain:

1. Start by installing the Pinecone Python client library. You can do this by running the following command in the terminal: `pip install pinecone`.
2. Import Pinecone in your python app: `import pinecone`.
3. Connect to Pinecone: To connect to the Pinecone service, you need to provide your API key. You can obtain an API key by signing up on the Pinecone website. Once you have the API key, pass it to the pinecone wrapper or set it as an environment variable:

```
pinecone.init()
```

1. Create a search index like this:

```
Docsearch = Pinecone.from_texts([“dog”, “cat”], embeddings)
```

The embeddings could be `OpenAIEmbeddings`, for example.

1. Now we can find the most similar documents for a query by similarity:

```
docs = docsearch.similarity_search(“terrier”, include_metadata=True)
```

These documents, we can then query again or use in a question answering chain as we've seen in *Chapter 4*, *Question Answering*.In LangChain, we can load our documents from many sources and in a bunch of formats through the integrated document loaders. You can use the LangChain integration hub to browse and select the appropriate loader for your data source. Once you have selected the loader, you can load the document using the specified loader. Let's briefly look at document loaders in LangChain!

Document loaders

Document loaders are used to load data from a source as **Document** objects, which consist of text and associated metadata. There are different types of integrations available, such as document loaders for loading a simple .txt file (`TextLoader`), loading the text contents of a web page (`WebBaseLoader`), articles from Arxiv (`ArxivLoader`), or loading a transcript of a YouTube video (`YoutubeLoader`). For webpages, the `Diffbot` integration gives a clean extraction of the content. Other integations exist for images such as providing image captions (`ImageCaptionLoader`).Document loaders have a `load()` method that loads data from the configured source and returns it as documents. They may also have a `lazy_load()` method for loading data into memory as and when they are needed.Here is an example of a document loader for loading data from a text file:

```
from langchain.document_loaders import TextLoader
loader = TextLoader(file_path="path/to/file.txt")
documents = loader.load()
```

The `documents` variable will contain the loaded documents, which can be accessed for further processing. Each document consists of the `page_content` (the text content of the document) and `metadata` (associated metadata such as the source URL or title).Similarly, we can load documents from Wikipedia:

```
from langchain.document_loaders import WikipediaLoader
loader = WikipediaLoader("LangChain")
documents = loader.load()
```

It's important to note that the specific implementation of document loaders may vary depending on the programming language or framework being used.In LangChain, vector retrieval in agents or chains is done via retrievers, which access the vector storage. Let's now how this works.

Retrievers in LangChain

Retrievers in LangChain are a type of component that is used to search and retrieve information from a given index. In the context of LangChain, a principal type of retriever is a `vectorstore` retriever. This type of retriever utilizes a vector store as a backend, such as Chroma, to index and search embeddings. Retrievers play a crucial role in question answering over documents, as they are responsible for retrieving relevant information based on the given query.Here are a few examples of retrievers:

- BM25 Retriever: This retriever uses the BM25 algorithm to rank documents based on their relevance to a given query. It is a popular information retrieval algorithm that takes into account term frequency and document length.
- TF-IDF Retriever: This retriever uses the TF-IDF (Term Frequency-Inverse Document Frequency) algorithm to rank documents based on the importance of terms in the document collection. It assigns higher weights to terms that are rare in the collection but frequent in a specific document.
- Dense Retriever: This retriever uses dense embeddings to retrieve documents. It encodes documents and queries into dense vectors and calculates the similarity between them using cosine similarity or other distance metrics.
- kNN retriever: This utilizes the well-known k-nearest neighbors' algorithm to retrieve relevant documents based on their similarity to a given query.

These are just a few examples of retrievers available in LangChain. Each retriever has its own strengths and weaknesses, and the choice of retriever depends on the specific use case and requirements.For example, to use the kNN retriever, you need to create a new instance of the retriever and provide it with a list of texts. Here is an example of how to create a kNN retriever using embeddings from OpenAI:

```
from langchain.retrievers import KNNRetriever
from langchain.embeddings import OpenAIEmbeddings
words = ["cat", "dog", "computer", "animal"]
retriever = KNNRetriever.from_texts(words, OpenAIEmbeddings())
```

Once the retriever is created, you can use it to retrieve relevant documents by calling the `get_relevant_documents()` method and passing a query string. The retriever will return a list of documents that are most relevant to the query.Here is an example of how to use the kNN retriever:

```
result = retriever.get_relevant_documents("dog")
print(result)
```

This will output a list of documents that are relevant to the query. Each document contains the page content and metadata:

```
[Document(page_content='dog', metadata={}),
 Document(page_content='animal', metadata={}),
 Document(page_content='cat', metadata={}),
 Document(page_content='computer', metadata={})]
```

There are a few more specialized retrievers in LangChain such as retrievers from Arxiv, Pubmed, or Wikipedia. For example, the purpose of an **Arxiv retriever** is to retrieve scientific articles from the Arxiv.org archive. It is a tool that allows users to search for and download scholarly articles in various fields such as physics, mathematics, computer science, and more. The functionality of an arxiv retriever includes specifying the maximum number of documents to be downloaded, retrieving relevant documents based on a query, and accessing metadata information of the retrieved documents.A **Wikipedia retriever** allows users to retrieve Wikipedia pages or documents from the Wikipedia website. The purpose of a Wikipedia retriever is to provide easy access to the vast amount of information available on Wikipedia and enable users to extract specific information or knowledge from it.A **PubMed retriever** is a component in LangChain that helps to incorporate biomedical literature retrieval into their language model applications. PubMed contains millions of citations for biomedical literature from various sources.In LangChain, the `PubMedRetriever` class is used to interact with the PubMed database and retrieve relevant documents based on a given query. The `get_relevant_documents()` method of the class takes a query as input and returns a list of relevant documents from PubMed.Here's an example of how to use the PubMed retriever in LangChain:

```
from langchain.retrievers import PubMedRetriever
retriever = PubMedRetriever()
documents = retriever.get_relevant_documents("COVID")
for document in documents:
    print(document.metadata["title"])
```

In this example, the `get_relevant_documents()` method is called with the query "COVID". The method then retrieves relevant documents related to the query from PubMed and returns them as a list. I am getting the following titles as printed output:

```
The COVID-19 pandemic highlights the need for a psychological support in systemic sclerosis p
Host genetic polymorphisms involved in long-term symptoms of COVID-19.
Association Between COVID-19 Vaccination and Mortality after Major Operations.
```

A custom retriever can be implemented in LangChain by creating a class that inherits from the `BaseRetriever` abstract class. The class should implement the `get_relevant_documents()` method, which takes a query string as input and returns a list of relevant documents.Here is an example of how a retriever can be implemented:

```
from langchain.retriever import BaseRetriever
from langchain.schema import Document
class MyRetriever(BaseRetriever):
    def get_relevant_documents(self, query: str) -> List[Document]:
        # Implement your retrieval logic here
        # Retrieve and process documents based on the query
        # Return a list of relevant documents
        relevant_documents = []
        # Your retrieval logic goes here…
        return relevant_documents
```

You can customize this method to perform any retrieval operations you need, such as querying a database or searching through indexed documents.Once you have implemented your retriever class, you can create an instance of it and call the `get_relevant_documents()` method to retrieve relevant documents based on a query.Let's implement a chatbot with a retriever!

Implementing a chatbot!

We'll implement a chatbot now. We start from a similar template as in chapter 4, Question Answering. Same as in the previous chapter, we'll assume you have the environment installed with the necessary libraries and the API keys as per the instructions in chapter 3, *Getting Started with LangChain*. To implement a simple chatbot in LangChain, you can follow this recipe:

1. Load the document
2. Create a vector storage
3. Set up a chatbot with retrieval from the vector storage

We'll generalize this with several formats and make this available through an interface in a web browser through Streamlit. You'll be able to drop in your document and start asking questions. In production, for a corporate deployment for customer engagement, you can imagine that these documents are already loaded in and your vector storage can just be static.Let's start with the document reader. As mentioned, we want to be able to read different formats:

```
from typing import Any
from langchain.document_loaders import (
  PyPDFLoader, TextLoader,
  UnstructuredWordDocumentLoader,
  UnstructuredEPubLoader
)
class EpubReader(UnstructuredEPubLoader):
    def __init__(self, file_path: str | list[str], ** kwargs: Any):
        super().__init__(file_path, **kwargs, mode="elements", strategy="fast")
class DocumentLoaderException(Exception):
    pass
class DocumentLoader(object):
    """Loads in a document with a supported extension."""
    supported_extentions = {
        ".pdf": PyPDFLoader,
        ".txt": TextLoader,
        ".epub": EpubReader,
        ".docx": UnstructuredWordDocumentLoader,
        ".doc": UnstructuredWordDocumentLoader
    }
```

This gives us interfaces to read PDF, text, EPUB, and word documents with different extensions. We'll now implement the loader logic.

```
import logging
import pathlib
from langchain.schema import Document
def load_document(temp_filepath: str) -> list[Document]:
    """Load a file and return it as a list of documents."""
    ext = pathlib.Path(temp_filepath).suffix
    loader = DocumentLoader.supported_extentions.get(ext)
    if not loader:
        raise DocumentLoaderException(
            f"Invalid extension type {ext}, cannot load this type of file"
        )
    loader = loader(temp_filepath)
    docs = loader.load()
    logging.info(docs)
    return docs
```

This doesn't handle a lot of errors at the moment, but this can be extended if needed. Now we can make this loader available from the interface and connect it to vector storage.

```
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import DocArrayInMemorySearch
from langchain.schema import Document, BaseRetriever
def configure_retriever(docs: list[Document]) -> BaseRetriever:
    """Retriever to use."""
    # Split each document documents:
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=200)
    splits = text_splitter.split_documents(docs)
    # Create embeddings and store in vectordb:
    embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
    # Single call to the huggingface model with all texts:
    vectordb = DocArrayInMemorySearch.from_documents(splits, embeddings)
    # Define retriever:
    return vectordb.as_retriever(search_type="mmr", search_kwargs={"k": 2, "fetch_k": 4})
```

DocArray is a Python package that provides a high-level API for representing and manipulating multimodal data. It provides various features like advanced indexing, comprehensive serialization protocols, a unified Pythonic interface, and more. Further, it offers efficient and intuitive handling of multimodal data for tasks such as natural language processing, computer vision, and audio processing.We can initialize the DocArray in-memory vector storage with different distance metrics such as cosine and Euclidean – cosine is the default.For the retriever, we have two main options:

1. Similarity-search: We can retrieve document according to similarity, or
2. Maximum Marginal Relevance (MMR): We can apply diversity-based re-ranking of documents during retrieval to get results that cover different results from the documents retrieved so far.

In the similarity-search, we can set a similarity score threshold. We've opted for MMR, which should give us better generation. We've set the parameter `k` as 2, which means we can get 2 documents back from retrieval.Retrieval can be improved by **contextual compression**, a technique where retrieved documents are compressed and irrelevant information is filtered out. Instead of returning the full documents as-is, contextual compression uses the context of the given query to extract and return only the relevant information. This helps to reduce the cost of processing and improve the quality of responses in retrieval systems.The base compressor is responsible for compressing the contents of individual documents based on the context of the given query. It uses a language model, such as GPT-3, to perform the compression. The compressor can filter out irrelevant information and return only the relevant parts of the document.The base retriever is the document storage system that retrieves the documents based on the query. It can be any retrieval system, such as a search engine or a database. When a query is made to the Contextual Compression Retriever, it first passes the query to the base retriever to retrieve relevant documents. Then, it uses the base compressor to compress the contents of these documents based on the context of the query. Finally, the compressed documents, containing only the relevant information, are returned as the response.We have a few options for contextual compression:

1. `LLMChainExtractor` – this passes over the returned documents and extracts from each only the relevant content.
2. `LLMChainFilter` – this is slightly simpler; it only filters only the relevant documents (rather than the content from the documents).
3. `EmbeddingsFilter` – this applies a similarity filter based on document and the query in terms of embeddings.

The first two compressors require an LLM to call, which means it can be slow and costly. Therefore, the `EmbeddingsFilter` can be a more efficient alternative. We can integrate compression here with a simple switch statement at the end (replacing the return statement):

```
if not use_compression:
    return retriever
embeddings_filter = EmbeddingsFilter(
  embeddings=embeddings, similarity_threshold=0.76
)
return ContextualCompressionRetriever(
  base_compressor=embeddings_filter,
  base_retriever=retriever
)
```

For our chosen compressor, the `EmbeddingsFilter`, we need to include two more additional imports:

```
from langchain.retrievers.document_compressors import EmbeddingsFilter
from langchain.retrievers import ContextualCompressionRetriever
```

We can feed the `use_compression` parameter through the `configure_qa_chain()` to the `configure_retriever()` method (not shown here).Now that we have the mechanism to create the retriever. We can set up the chat chain:

```
from langchain.chains import ConversationalRetrievalChain
from langchain.chains.base import Chain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory
def configure_chain(retriever: BaseRetriever) -> Chain:
    """Configure chain with a retriever."""
    # Setup memory for contextual conversation
    memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
    # Setup LLM and QA chain; set temperature low to keep hallucinations in check
    llm = ChatOpenAI(
        model_name="gpt-3.5-turbo", temperature=0, streaming=True
    )
    # Passing in a max_tokens_limit amount automatically
    # truncates the tokens when prompting your llm!
    return ConversationalRetrievalChain.from_llm(
```

```
        llm, retriever=retriever, memory=memory, verbose=True, max_tokens_limit=4000
    )
```

One final thing for the retrieval logic is taking the documents and passing them to the retriever setup:

```
import os
import tempfile
def configure_qa_chain(uploaded_files):
    """Read documents, configure retriever, and the chain."""
    docs = []
    temp_dir = tempfile.TemporaryDirectory()
    for file in uploaded_files:
        temp_filepath = os.path.join(temp_dir.name, file.name)
        with open(temp_filepath, "wb") as f:
            f.write(file.getvalue())
        docs.extend(load_document(temp_filepath))
    retriever = configure_retriever(docs=docs)
    return configure_chain(retriever=retriever)
```

Now that we have the logic of the chatbot, we need to set up the interface. As mentioned, we'll use streamlit again:

```
import streamlit as st
from langchain.callbacks import StreamlitCallbackHandler
st.set_page_config(page_title="LangChain: Chat with Documents", page_icon="🦜")
st.title("🦜 LangChain: Chat with Documents")
uploaded_files = st.sidebar.file_uploader(
    label="Upload files",
    type=list(DocumentLoader.supported_extentions.keys()),
    accept_multiple_files=True
)
if not uploaded_files:
    st.info("Please upload documents to continue.")
    st.stop()
qa_chain = configure_qa_chain(uploaded_files)
assistant = st.chat_message("assistant")
user_query = st.chat_input(placeholder="Ask me anything!")
if user_query:
    stream_handler = StreamlitCallbackHandler(assistant)
    response = qa_chain.run(user_query, callbacks=[stream_handler])
    st.markdown(response)
```

This gives us a chatbot with retrieval usable via a visual interface with drop-in of custom documents as needed that you can ask questions about.

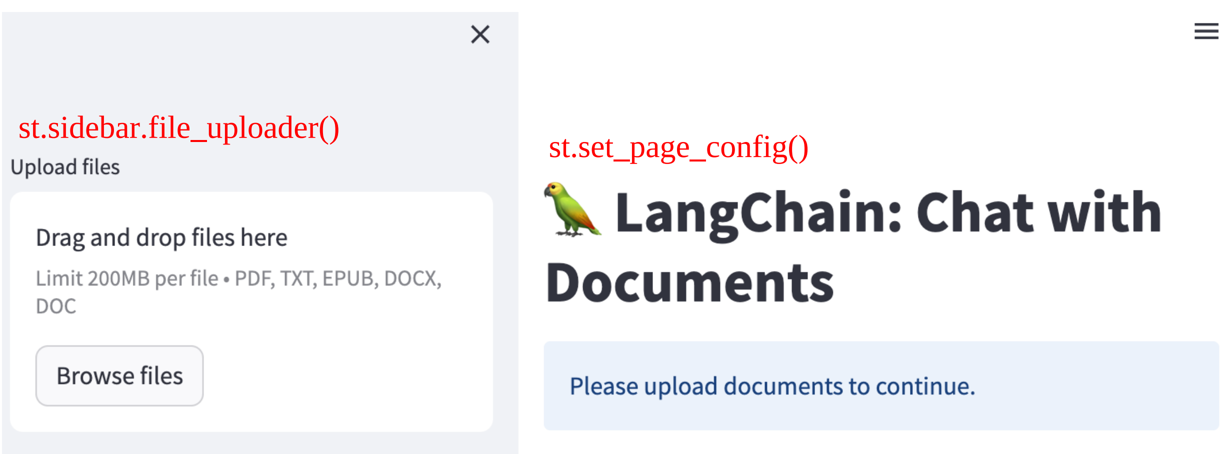


Figure 5.7: Chatbot interface with document loaders in different formats.

You can see the full implementation on Github. You can play around with the chatbot, and see how it works, and when it doesn't.It's important to note that LangChain has limitations on input size and cost. You may need to consider workarounds to handle larger knowledge bases or optimize the cost of API usage. Additionally, fine-tuning models or hosting the LLM in-house can be more complex and less accurate compared to using commercial solutions. We'll look at these use cases in *Chapters 8*, *Conditioning and Fine-Tuning*, and *9*, *LLM applications in Production*.Let's have a look at the available memory mechanisms in LangChain.

Memory mechanisms

A memory is a component in the LangChain framework that allows chatbots and language models to remember previous interactions and information. It is essential in applications like chatbots because it enables the system to maintain context and continuity in conversations.We need memory in chatbots to:

1. Remember previous interactions: Memory allows chatbots to retain information from previous messages exchanged with the user. This helps in understanding user queries and providing relevant responses.
2. Maintain context: By recalling previous interactions, chatbots can maintain context and continuity in conversations. This allows for more natural and coherent conversations with users.
3. Extract knowledge: Memory enables the system to extract knowledge and insights from a sequence of chat messages. This information can then be used to improve the performance and accuracy of the chatbot.

In summary, memory is crucial in chatbots to create a more personalized and human-like conversational experience by remembering and building upon past interactions.Here's a practical example in Python that demonstrates how to use the LangChain memory feature.

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain
# Creating a conversation chain with memory
memory = ConversationBufferMemory()
chain = ConversationChain(memory=memory)
# User inputs a message
user_input = "Hi, how are you?"
# Processing the user input in the conversation chain
response = chain.predict(input=user_input)
# Printing the response
print(response)
# User inputs another message
user_input = "What's the weather like today?"
# Processing the user input in the conversation chain
response = chain.predict(input=user_input)
# Printing the response
print(response)
# Printing the conversation history stored in memory
print(memory.chat_memory.messages)
```

In this example, we create a conversation chain with memory using ConversationBufferMemory, which is a simple wrapper that stores the messages in a variable. The user's inputs are processed using the `predict()` method of the conversation chain. The conversation chain retains the memory of previous interactions, allowing it to provide context-aware responses.Instead of constructing the memory separately from the chain, we could have simplified:

```
conversation = ConversationChain(
    llm=llm,
    verbose=True,
    memory=ConversationBufferMemory()
)
```

We are setting verbose to True in order to see the prompts.After processing the user inputs, we print the response generated by the conversation chain. Additionally, we print the conversation history stored in memory using `memory.chat_memory.messages`. The `save_context()` method is used to store inputs and outputs. You can use the `load_memory_variables()` method to view the stored content. To get the history as a list of messages, a `return_messages` parameter is set to `True`. We'll see examples of this in this section. `ConversationBufferWindowMemory` is a memory type provided by LangChain that keeps track of the interactions in a conversation over time. Unlike `ConversationBufferMemory`, which retains all previous interactions, `ConversationBufferWindowMemory` only keeps the last K interactions, where K is the window size specified.Here's a simple example of how to use `ConversationBufferWindowMemory` in LangChain:

```
from langchain.memory import ConversationBufferWindowMemory
memory = ConversationBufferWindowMemory(k=1)
```

In this example, the window size is set to 1, meaning that only the last interaction will be stored in memory.We can use the `save_context()` method to save the context of each interaction. It takes two arguments: user_input and model_output. These represent the user's input and the corresponding model's output for a given interaction.

```
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.save_context({"input": "not much you"}, {"output": "not much"})
```

We can see the message with `memory.load_memory_variables({})` In LangChain, we can integrate a knowledge graph to enhance the capabilities of language models and enable them to leverage structured knowledge during text generation and inference.

> A **knowledge graph** is a structured knowledge representation model that organizes information in the form of entities, attributes, and relationships. It represents knowledge as a graph, where entities are represented as nodes and relationships between entities are represented as edges.
>
> > Prominent examples of knowledge graphs include Wikidata, which captures structured information from Wikipedia, and Google's Knowledge Graph, which powers search results with rich contextual information.

In a knowledge graph, entities can be any concept, object, or thing in the world, and attributes describe properties or characteristics of these entities. Relationships capture the connections and associations between entities, providing contextual information and enabling semantic reasoning.There's functionality in LangChain for knowledge graphs for retrieval, however, LangChain also provides memory components to automatically create a knowledge graph based on our conversation messages. Instantiate the `ConversationKGMemory` class and pass your language model (LLM) instance as the llm parameter:

```
from langchain.memory import ConversationKGMemory
from langchain.llms import OpenAI
llm = OpenAI(temperature=0)
memory = ConversationKGMemory(llm=llm)
```

As the conversation progresses, we can save relevant information from the knowledge graph into the memory using the `save_context()` function of the `ConversationKGMemory`. We can also customize the conversational memory in LangChain, which involves modifying the prefixes used for the AI and Human messages, as well as updating the prompt template to reflect these changes.To customize the conversational memory, you can follow these steps:

1. Import the necessary classes and modules from LangChain:

```
from langchain.llms import OpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.prompts.prompt import PromptTemplate
llm = OpenAI(temperature=0)
```

1. Define a new prompt template that includes the customized prefixes. You can do this by creating a `PromptTemplate` object with the desired template string.

```
template = """The following is a friendly conversation between a human and an AI. The AI is t
Current conversation:
{history}
Human: {input}
AI Assistant:"""
PROMPT = PromptTemplate(input_variables=["history", "input"], template=template)
conversation = ConversationChain(
    prompt=PROMPT,
    llm=llm,
    verbose=True,
    memory=ConversationBufferMemory(ai_prefix="AI Assistant"),
)
```

In this example, the AI prefix is set to AI Assistant instead of the default AI.The `ConversationSummaryMemory` is a type of memory in LangChain that generates a summary of the conversation as it progresses. Instead of storing all messages verbatim, it condenses the information, providing a summarized version of the conversation. This is particularly useful for long conversation chains where including all previous messages might exceed token limits.To use `ConversationSummaryMemory`, first create an instance of it, passing the language model (llm) as an argument. Then, use the `save_context()` method to save the interaction context, which includes the user input and AI output. To retrieve the summarized conversation history, use the `load_memory_variables()` method.Example:

```
from langchain.memory import ConversationSummaryMemory
from langchain.llms import OpenAI
# Initialize the summary memory and the language model
memory = ConversationSummaryMemory(llm=OpenAI(temperature=0))
# Save the context of an interaction
memory.save_context({"input": "hi"}, {"output": "whats up"})
# Load the summarized memory
memory.load_memory_variables({})
```

LangChain also allows combining multiple memory strategies using the CombinedMemory class. This is useful when you want to maintain different aspects of the conversation history. For instance, one memory could be used to store the complete conversation log and

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory, CombinedMemory, ConversationSummaryMem
# Initialize language model (with desired temperature parameter)
llm = OpenAI(temperature=0)
# Define Conversation Buffer Memory (for retaining all past messages)
conv_memory = ConversationBufferMemory(memory_key="chat_history_lines", input_key="input")
# Define Conversation Summary Memory (for summarizing conversation)
summary_memory = ConversationSummaryMemory(llm=llm, input_key="input")
# Combine both memory types
memory = CombinedMemory(memories=[conv_memory, summary_memory])
# Define Prompt Template
_DEFAULT_TEMPLATE = """The following is a friendly conversation between a human and an AI. Th
Summary of conversation:
{history}
Current conversation:
{chat_history_lines}
Human: {input}
AI:"""
PROMPT = PromptTemplate(input_variables=["history", "input", "chat_history_lines"], template=
# Initialize the Conversation Chain
conversation = ConversationChain(llm=llm, verbose=True, memory=memory, prompt=PROMPT)
# Start the conversation
conversation.run("Hi!")
```

In this example, we first instantiate the language model and the different types of memories we're using - `ConversationBufferMemory` for retaining the full conversation history and `ConversationSummaryMemory` for creating a summary of the conversation. We then combine these memories using CombinedMemory. We also define a prompt template that accommodates our memory usage and finally, we create and run the `ConversationChain` by providing our language model, memory, and prompt to it.The `ConversationSummaryBufferMemory` is used to keep a buffer of recent interactions in memory, and compiles old interactions into a summary instead of completely flushing them out. The threshold for flushing interactions is determined by token length and not by the number of interactions. To use this, the memory buffer needs to be instantiated with the LLM model and a `max_token_limit`. `ConversationSummaryBufferMemory` offers a method called `predict_new_summary()` which can be used directly to generate a conversation summary.Zep is a memory store and search engine that is designed to store, summarize, embed, index, and enrich chatbot or AI app histories. It provides developers with a simple and low-latency API to access and manipulate the stored data.A practical example of using Zep is to integrate it as the long-term memory for a chatbot or AI app. By using the `ZepMemory` class, developers can initialize a `ZepMemory` instance with the Zep server URL, API key, and a unique session identifier for the user. This allows the chatbot or AI app to store and retrieve chat history or other relevant information.For example, in Python, you can initialize a ZepMemory instance as follows:

```
from langchain.memory import ZepMemory
# Set this to your Zep server URL
ZEP_API_URL = "http://localhost:8000"
ZEP_API_KEY = "<your JWT token>"  # optional
session_id = str(uuid4())  # This is a unique identifier for the user
# Set up ZepMemory instance
memory = ZepMemory(
    session_id=session_id,
    url=ZEP_API_URL,
    api_key=ZEP_API_KEY,
    memory_key="chat_history",
)
```

Once the memory is set up, you can use it in your chatbot's chain or with your AI agent to store and retrieve chat history or other relevant information.Overall, Zep simplifies the process of persisting, searching, and enriching chatbot or AI app histories, allowing developers to focus on developing their AI applications rather than building memory infrastructure.

Don't say anything stupid!

The role of moderation in chatbots is to ensure that the bot's responses and conversations are appropriate, ethical, and respectful. It involves implementing mechanisms to filter out offensive or inappropriate content, as well as discouraging abusive behavior from users.In the context of moderation, a constitution refers to a set of guidelines or rules that govern the behavior and responses of the chatbot. It outlines the standards and principles that the chatbot should adhere to, such as avoiding offensive language, promoting respectful interactions, and maintaining ethical standards. The constitution serves as a framework for ensuring that the chatbot operates within the desired boundaries and provides a positive user experience.Moderation and having a constitution in chatbots are crucial for creating a safe, respectful, and inclusive environment for users, protecting brand reputation, and complying with legal obligations.Moderation and having a constitution are important in chatbots for several reasons:

1. Ensuring ethical behavior: Chatbots have the potential to interact with a wide range of users, including vulnerable individuals. Moderation helps ensure that the bot's responses are ethical, respectful, and do not promote harmful or offensive content.
2. Protecting users from inappropriate content: Moderation helps prevent the dissemination of inappropriate or offensive language, hate speech, or any content that may be harmful or offensive to users. It creates a safe and inclusive environment for users to interact with the chatbot.
3. Maintaining brand reputation: Chatbots often represent a brand or organization. By implementing moderation, the developer can ensure that the bot's responses align with the brand's values and maintain a positive reputation.
4. Preventing abusive behavior: Moderation can discourage users from engaging in abusive or improper behavior. By implementing rules and consequences, such as the "two strikes" rule mentioned in the example, the developer can discourage users from using provocative language or engaging in abusive behavior.
5. Legal compliance: Depending on the jurisdiction, there may be legal requirements for moderating content and ensuring that it complies with laws and regulations. Having a constitution or set of guidelines helps the developer adhere to these legal requirements.

You can add a moderation chain to an LLMChain to ensure that the generated output from the language model is not harmful.If the content passed into the moderation chain is deemed harmful, there are different ways to handle it. You can choose to throw an error in the chain and handle it in your application, or you can return a message to the user explaining that the text was harmful. The specific handling method depends on your application's requirements.In LangChain, first, you would create an instance of the `OpenAIModerationChain` class, which is a pre-built moderation chain provided by LangChain. This chain is specifically designed to detect and filter out harmful content.

```
from langchain.chains import OpenAIModerationChain
moderation_chain = OpenAIModerationChain()
```

Next, you would create an instance of the LLMChain class, which represents your language model chain. This is where you define your prompt and interact with the language model.

```
from langchain.chains import LLMChain
llm_chain = LLMChain(model_name="gpt-3.5-turbo")
```

To append the moderation chain to the language model chain, you can use the `SequentialChain` class. This class allows you to chain multiple chains together in a sequential manner.

```
from langchain.chains import SequentialChain
chain = SequentialChain([llm_chain, moderation_chain])
```

Now, when you want to generate text using the language model, you would pass your input text through the moderation chain first, and then through the language model chain.

```
input_text = "Can you generate a story for me?"
output = chain.generate(input_text)
```

The moderation chain will evaluate the input text and filter out any harmful content. If the input text is deemed harmful, the moderation chain can either throw an error or return a message indicating that the text is not allowed. I've added an example for moderation to the chatbot app on Github.Further, Guardrails can be used to define the behavior of the language model on specific topics, prevent it from engaging in discussions on unwanted topics, guide the conversation along a predefined path, enforce a particular language style, extract structured data, and more.

> In the context of large language models, **guardrails** (**rails** for short) refer to specific ways of controlling the output of the model. They provide a means to add programmable constraints and guidelines to ensure the output of the language model aligns with desired criteria.

Here are a few ways guardrails can be used:

- Controlling Topics: Guardrails allow you to define the behavior of your language model or chatbot on specific topics. You can prevent it from engaging in discussions on unwanted or sensitive topics like politics.
- Predefined Dialog Paths: Guardrails enable you to define a predefined path for the conversation. This ensures that the language model or chatbot follows a specific flow and provides consistent responses.
- Language Style: Guardrails allow you to specify the language style that the language model or chatbot should use. This ensures that the output is in line with your desired tone, formality, or specific language requirements.
- Structured Data Extraction: Guardrails can be used to extract structured data from the conversation. This can be useful for capturing specific information or performing actions based on user inputs.

Overall, guardrails provide a way to add programmable rules and constraints to large language models and chatbots, making them more trustworthy, safe, and secure in their interactions with users. By appending the moderation chain to your language model chain, you can ensure that the generated text is moderated and safe for use in your application.

Summary

In chapter 4, we discussed Retrieval-Augmented Generation (RAG), which involves the utilization of external tools or knowledge resources such as document corpora. In that chapter, we focused on the process. In this chapter, the focus is on methods relevant to building chatbots based on RALMs, and, more particularly, the use of external tools to retrieve relevant information that can be incorporated into the content generation. The main sections of the chapter include an introduction to chatbots, retrieval and vector mechanisms, implementing a chatbot, memory mechanisms, and the importance of appropriate responses.The chapter started with an overview over chatbots. We discussed the evolution and current state of chatbots and language processing models (LLMs) highlighting the practical implications and enhancements of the capabilities of the current technology. We then discussed the importance of proactive communication and the technical implementations required for context, memory, and reasoning. We explored retrieval mechanisms, including vector storage, with the goal to improve the accuracy of chatbot responses. We went into details with methods for loading documents and information, including vector storage and embedding. Additionally, we discussed memory mechanisms for maintaining knowledge and the state of ongoing conversations are examined. The chapter concludes with a discussion on moderation, emphasizing the importance of ensuring responses are respectful and aligned with organizational values. We've implemented a chatbot in this chapter, which explores a few features discussed in this chapter, and can serve as a starting point to investigate issues like memory and context, moderation of speech, but can also be interesting for issues like hallucinations or others.In chapter 9, we'll discuss how you can train LLMs on your documents as another way of conditioning models on your data!Let's see if you remember some of the key takeaways from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions from memory. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. Please name 5 different chatbots!
2. What are some important aspects in developing a chatbot?
3. What does RALMs stand for?
4. What is an embedding?
5. What is vector search?
6. What is a vector database?
7. Please name 5 different vector databases!
8. What is a retriever in LangChain?
9. What is memory and what are the memory options in LangChain?
10. What is moderation, what's a constitution, and how do they work?

6 Developing Software

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

While this book is about integrating generative AI and, in particular, large language models (LLMs) into software applications, in this chapter, we'll talk about how we can leverage LLMs to help in software development. This is a big topic and software development was highlighted in reports by several consultancies such as KPMG and McKinsey as one of the domains impacted most by generative AI.We'll first discuss how LLMs could help in coding tasks, and we'll go through a lot of literature as an overview to see how far we have come in automating software engineers. We'll also discuss a lot of the recent progress and new models. Then, we'll play around with a few models evaluating the generated code qualitatively. Next, we'll implement a fully-automated agent for software development tasks. We go through the design choices and show a bit of the results that we got in an agent implementation of only a few lines of Python with LangChain. There are a lot of possible extensions to this approach, which we'll also go through.Throughout the chapter, we'll work on different approaches to software development, which you can find in the `software_development` directory in the Github repository for the book at https://github.com/benman1/generative_ai_with_langchainThe main sections are:

- Software development and AI
- Writing code with LLMs
- Automated software development

We'll begin the chapter by giving a broad overview over the state-of-the-art of using AI for software development.

Software development and AI

The emergence of powerful AI systems like ChatGPT has sparked great interest in using AI as a tool to assist software developers. A June 2023 report by KPMG estimated that about 25% of software development tasks could be automated away. A McKinsey report from the same month highlighted software development as a function, where generative AI can have a significant impact in terms of cost reduction and efficiency gain. The idea of utilizing artificial intelligence to aid programming is not new, but has rapidly evolved alongside advances in computing and AI. The two areas are intertwined as we'll see.Early efforts in language and compiler design the 1950s and 60s sought to make it easier to write software. Data processing languages like **FLOW-MATIC** (also known as: **Business Language version 0**), designed under Grace Hopper at Remington Rand in 1955, generated code from English-like statements. Similarly, programming languages such as **BASIC (Beginners' All-purpose Symbolic Instruction Code)**, created at Dartmouth College in 1963, aimed to make it easier to write software in an interpreted environment.Other efforts further simplified and standardized the programming syntax and interfaces. The **flow-based programming (FBP)** paradigm, invented by J. Paul Morrison in the early 1970s, allows to define applications as connected black box processes, which exchange data by message passing. Visual low-code or no-code platforms followed in the same mold with popular proponents such as LabVIEW, extensively used for system design in Electronical Engineering, and the KNIME extract, transform, load tool for data science.Some of the earliest efforts to automate coding itself through AI were **expert systems**, which emerged in the 1980s. As a form of narrow AI, they focused on encoding domain knowledge and rules to provide guidance. These would be formulated in a very specific syntax and executed in rule engines. These encoded best practices for programming tasks like debugging, though their usefulness was constrained by the need for meticulous rule-based programming.For software development, from command line editors such as ed (1969), to vim and emacs (1970s), to today's integrated development environment (IDEs) such as Visual Studio (first released in 1997) and PyCharm (since 2010), these tools have helped developers write code, navigate in complex projects, refactor, get highlighting and setup and run tests. IDE's also integrated and provide feedback from code validation tools, some of which have been around since the 1970s. Prominently, Lint, written by Stephen Curtis Johnson in 1978 at Bell Labs can flag bugs, stylistic errors and suspicious constructs. Many tools apply formal methods; however, machine learning has been applied including genetic programming and neural network based approaches for at least 20 years. In this chapter, we'll how far we've come with analyzing code using deep neural networks, especially transformers.This brings us to the present day, where models have been trained to produce full or partial programs based on natural language descriptions (in coding assistants or chatbots) or some code inputs (completion).

Present day

Researchers at DeepMind published two papers in the journals Nature and Science, respectively, that represent important milestones in using AI to transform foundational computing, in particular using reinforcement learning to discover optimized algorithms. In October 2022, they released algorithms discovered by their model **AlphaTensor** for matrix multiplication problems, which can speed up this essential computation required by deep learning models, but also in many other applications. **AlphaDev** uncovered novel sorting algorithms that were integrated into widely used C++ libraries, improving performance for millions of developers. It also generalized its capabilities, discovering a 30% faster hashing algorithm now used billions of times daily. These discoveries demonstrate AlphaDev's ability to surpass human-refined algorithms and unlock optimizations difficult at higher programming levels.Their model **AlphaCode**, published as a paper in February 2022, showcases an AI-powered coding engine that creates computer programs at a rate comparable to that of an average programmer. They report results on different datasets including HumanEval and others, which we'll come to in the next section. The DeepMind researchers highlight the large-scale sampling of candidate pool of algorithms and a filtering step to select from it. The model was celebrated as a breakthrough achievement; however, the practicality and scalability of their approach is unclear.Today, new code LLMs such as ChatGPT and Microsoft's Copilot are highly popular generative AI models, with millions of users and significant productivity-boosting capabilities. There are different tasks related to programming that LLMs can tackle such as these:

1. Code completion: This task involves predicting the next code element based on the surrounding code. It is commonly used in integrated development environments (IDEs) to assist developers in writing code.
2. Code summarization/documentation: This task aims to generate a natural language summary or documentation for a given block of source code. This summary helps developers understand the purpose and function of the code without having to read the actual code.
3. Code search: The objective of code search is to find the most relevant code snippets based on a given natural language query. This task involves learning the joint embeddings of the query and code snippets to return the expected ranking order of code snippets. Neural code search is specifically focused on in the experiment mentioned in the text.
4. Bug finding/fixing: AI systems can reduce manual debugging efforts and enhance software reliability and security. Many bugs and vulnerabilities are hard to find for programmers, although there are typical patterns for which code validation tools exist. As an alternative, LLMs can spot problems with a code and (when prompted) correct them. Thus, these systems can reduce manual debugging efforts and help improve software reliability and security.
5. Test generation: Similar to code completion, LLMs can generate unit tests (compare Bei Chen and others, 2022) and other types of tests enhancing the maintainability of a code base.

AI programming assistants combine the interactivity of earlier systems with cutting-edge natural language processing. Developers can query bugs in plain English or describe desired functions, receiving generated code or debugging tips. However, risks remain around code quality, security, and excessive dependence. Striking the right balance of computer augmentation while maintaining human oversight is an ongoing challenge.Let's look at the current performance of AI systems for coding, particularly code LLMs.

Code LLMs

Quite a few AI models have emerged, each with their own strengths and weaknesses, which are continuously competing with each other to improve and deliver better results. This comparison should give an overview over some of the largest and most popular models:

Model	**Reads files**	**Runs code**	**Tokens**
ChatGPT; GPT 3.5/4	No	No	up to 32k
ChatGPT: Code interpreter	Yes	Yes	up to 32k
Claude 2	Yes	No	100k
Bard	No	Yes	1k
Bing	Yes	No	32k

Figure 6.1: Public chat interfaces for software development.

While this competition benefits users by providing a wider range of options, it also means that relying solely on ChatGPT may no longer be the optimal choice. Users now face the decision of selecting the most suitable model for each specific task.The latest wave leverages machine learning and neural networks for more flexible intelligence. Powerful pre-trained models like GPT-3 enable context-aware, conversational support. Deep learning approaches also empower bug detection, repair recommendations, automated testing tools, and code search.Microsoft's GitHub Copilot, which is based on OpenAI's Codex, draws on open source code to suggest full code blocks in real-time. According to a Github report in June 2023, developers accepted the AI assistant's suggestions about 30 percent of the time, which suggests that the tool can provide useful suggestions, with less experienced developers profiting the most.

> **Codex** is a model, developed by OpenAI. It is capable of parsing natural language and generating code and powers GitHub Copilot. A descendant of the GPT-3 model, it has been fine-tuned on publicly available code from GitHub, 159 gigabytes of Python code from 54 million GitHub repositories, for programming applications.

To illustrate the progress made in creating software, let's look at quantitative results in a benchmark: the **HumanEval dataset**, introduced in the Codex paper ("*Evaluating Large Language Models Trained on Code*", 2021) is designed to test the ability of large language models to complete functions based on their signature and docstring. It evaluates the functional correctness of synthesizing programs from docstrings. The dataset includes 164 programming problems that cover various aspects such as language comprehension, algorithms, and simple mathematics. Some of the problems are comparable to simple software interview questions. A common metric on HumanEval is pass@k (pass@1) – this refers to the fraction of correct samples when generating k code samples per problem.This table summarizes the progress of AI models on the HumanEval task (source: Suriya Gunasekar and others, "*Textbooks Are All You Need*", 2023; https://arxiv.org/pdf/2306.11644.pdf):

Date	Model	Model size	Dataset size (Tokens)	HumanEval (Pass@1)	MBPP (Pass@1)
2021 Jul	Codex-300M [CTJ+21]	300M	100B	13.2%	-
2021 Jul	Codex-12B [CTJ+21]	12B	100B	28.8%	-
2022 Mar	CodeGen-Mono-350M [NPH+23]	350M	577B	12.8%	-
2022 Mar	CodeGen-Mono-16.1B [NPH+23]	16.1B	577B	29.3%	35.3%
2022 Apr	PaLM-Coder [CND+22]	540B	780B	35.9%	47.0%
2022 Sep	CodeGeeX [ZXZ+23]	13B	850B	22.9%	24.4%
2022 Nov	GPT-3.5 [Ope23]	175B	N.A.	47%	-
2022 Dec	SantaCoder [ALK+23]	1.1B	236B	14.0%	35.0%
2023 Mar	GPT-4 [Ope23]	N.A.	N.A.	67%	-
2023 Apr	Replit [Rep23]	2.7B	525B	21.9%	-
2023 Apr	Replit-Finetuned [Rep23]	2.7B	525B	30.5%	-
2023 May	CodeGen2-1B [NHX+23]	1B	N.A.	10.3%	-
2023 May	CodeGen2-7B [NHX+23]	7B	N.A.	19.1%	-
2023 May	StarCoder [LAZ+23]	15.5B	1T	33.6%	52.7%
2023 May	StarCoder-Prompted [LAZ+23]	15.5B	1T	40.8%	49.5%
2023 May	PaLM 2-S [ADF+23]	N.A.	N.A.	37.6%	50.0%
2023 May	CodeT5+ [WLG+23]	2B	52B	24.2%	-
2023 May	CodeT5+ [WLG+23]	16B	52B	30.9%	-
2023 May	InstructCodeT5+ [WLG+23]	16B	52B	35.0%	-
2023 Jun	WizardCoder [LXZ+23]	16B	1T	57.3%	51.8%
2023 Jun	**phi-1**	1.3B	7B	50.6%	55.5%

Figure 6.2: Model comparison on coding task benchmarks (HumanEval and MBPP). The performance metrics are self-reported. This table only includes models as opposed to other approaches, for example reasoning strategies. Llama2's self-reported performance on HumanEval is 29.9%.

Please note that the data used in training most LLM models includes some amount of source code. For example, The Pile dataset, which was curated by EleutherAI's GPT-Neo for training open-source alternatives of the GPT models, GPT-Neo, includes at least about 11% of code from Github (102.18GB). The Pile was used in training of Meta's Llama, Yandex's YaLM 100B, and many others.Although, HumanEval has been broadly used as a benchmark for code LLMs, there are a multitude of benchmarks for programming. Here's an example question and the response from an advanced computer science test given to Codex:

Write a recursive function, called biggest_last(), that takes a list of integers as input, where the list elements can be in any order. If the input list is empty or only contains one integer, then it returns the input. Otherwise the function proceeds by starting with the first two elements and checking whether the larger one is the second one. If it is not then the two values should be exchanged, i.e., the smaller value put into the first location and the larger value into the subsequent location. Once this check is performed then the function moves on by one location and repeats this until it has processed the entire list. The function outputs that list with the largest integer in the last location of the list. The output order of the other integers may be different than their input order. Remember a recursive function is one than calls itself to assist in performing its task.

```
def biggest_last(values):
    if len(values) < 2:
        return values
    else:
        if values[0] > values[1]:
            values[0], values[1] = values[1],values[0]
        return [values[0]] + biggest_last(values[1:])
```

Figure 6.3: A question given in a CS2 exam (left) and the Codex response (source "My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises" James Finnie-Ansley and others, 2023).

There are many interesting studies that shed a light on AI's capability to help software developers or that expand on that capability as summarized in this table:

Authors	Publication Date	Conclusions	Task	Model/Strategy Analyzed
Abdullah Al Ishtiaq and others	April 2021	Pre-trained language models like BERT can enhance code search through improved semantic understanding.	Code search	BERT
Mark Chen et al. (OpenAI)	July 2021	Evaluates Codex on code generation, shows potential to advance program synthesis	Code generation	Codex
Ankita Sontakke and others	March 2022	Even state-of-the-art models produce poor quality code summaries, indicating they may not understand code.	Code summarization	Transformer models
Bei Chen et al. (Microsoft)	July 2022	CODE-T leverages LLMs to auto-generate test cases, reducing human effort and improving code evaluation. It achieves 65.8% HumanEval pass@1.	Code generation, testing	CODET
Eric Zelikman et al. (Stanford)	December 2022	Parsel framework enables LLMs to decompose problems and leverage strengths, improving performance on hierarchical reasoning	Program synthesis, planning	Codex
James Finnie-Ansley and others	January 2023	Codex outperforms most students on advanced CS2 programming exams.	CS2 programming	Codex
Yue Liu and others	February 2023	Existing automated code generation has limitations in robustness and reliability.	Code generation	5 NMT models
Mingyang Geng and others	February 2023	A two-stage approach significantly increased effectiveness of code summarization.	Code summarization	LLM + reinforcement learning
Noah Shinn et al.	March 2023	Reflexion enables trial-and-error learning via verbal reflection, achieving 91% HumanEval pass@1	Coding, reasoning	Reflexion
Haoye Tian and others	April 2023	ChatGPT shows promise for programming assistance but has limitations in robustness, generalization, and attention span.	Code generation, program repair, code summarization	ChatGPT
Chuqin Geng and others	April 2023	ChatGPT demonstrates impressive capabilities for intro programming education but would only get a B-grade as a student.	Intro functional programming course	ChatGPT
Xinyun Chen and others	April 2023	Self-debugging technique enables language models to identify and correct mistakes in generated code.	Code generation	Self-Debugging
Masum Hasan and others	April 2023	Transforming text to an intermediate formal language enabled more efficient app code generation from descriptions.	App code generation	Seq2seq networks
Anis Koubaa and others	May 2023	ChatGPT struggles with complex programming problems and is not yet suitable for fully automated programming. It performs much worse than human programmers.	Programming problem solving	ChatGPT
Wei Ma and others	May 2023	ChatGPT understands code syntax but is limited in analyzing dynamic code behavior.	Complex code analysis	ChatGPT
Raymond Li et al. (BigCode)	May 2023	Introduces 15.5B parameter StarCoder trained on 1 trillion GitHub tokens, achieves 40% HumanEval pass@1	Code generation, multiple languages	StarCoder
Amos Azaria and others	June 2023	ChatGPT has errors and limitations, so outputs should be independently verified. It is best used by experts well-versed in the domain.	General capabilities and limitations	ChatGPT

Adam Hörnemalm	June 2023	ChatGPT increased efficiency for coding and planning but struggled with communication. Developers wanted more integrated tooling.	Software development	ChatGPT
Suriya Gunasekar et al. (Microsoft)	June 2023	High-quality data enables smaller models to match larger models, altering scaling laws	Code generation	Phi-1

Figure 6.2: Literature review of AI for programming tasks. The publication dates refer mostly to the preprint releases.

This is just a small subset of studies, but hopefully this helps to shed a light on some of the developments in the field. Recent research explored how ChatGPT can support programmers' daily work activities like coding, communication, and planning. Other research describes new models (such as Codex, StarCoder, or Phi-1) or approaches for planning or reasoning to execute these models.Most recently, the paper "*Textbooks Are All You Need*" by Suriya Gunasekar and others at Microsoft Research (2023) introduced phi-1, a 1.3B parameter Transformer-based language model for code. The paper demonstrates how high-quality data can enable smaller models to match larger models for code tasks. The authors start with a 3 TB corpus of code from The Stack and StackOverflow. A large language model (LLM) filters this to select 6B high-quality tokens. Separately, GPT-3.5 generates 1B tokens mimicking textbook style. A small 1.3B parameter model phi-1 is trained on this filtered data. Phi-1 is then fine-tuned on exercises synthesized by GPT-3.5. Results show phi-1 matches or exceeds the performance of models over 10x its size on benchmarks like HumanEval and MBPP.The core conclusion is that high-quality data significantly impacts model performance, potentially altering scaling laws. Instead of brute force scaling, data quality should take precedence. The authors reduce costs by using a smaller LLM to select data, rather than expensive full evaluation. Recursively filtering and retraining on selected data could enable further improvements. It's important to appreciate that there's a massive step change in difficulty between short code snippets, where task specifications are translated directly into code and the right API calls have to be issued in a sequence specific to the task, and generating complete programs, which relies on a much deeper understanding and reasoning about the task, the concepts behind, and planning how to accomplish it. However, reasoning strategies can make a big difference for short snippets as well as the paper "*Reflexion: Language Agents with Verbal Reinforcement Learning*" by Noah Shinn and others (2023) shows. The authors propose a framework called Reflexion that enables LLM agents (implemented in LangChain) to learn quickly and efficiently from trial-and-error using verbal reinforcement. The agents verbally reflect on task feedback signals and store their reflective text in an episodic memory buffer, which helps the agents make better decisions in subsequent trials. The authors demonstrate the effectiveness of Reflexion in improving decision-making in diverse tasks such as sequential decision-making, coding, and language reasoning. Reflexion has the potential to outperform previous state-of-the-art models, such as GPT-4, in specific tasks, as shown by its 91% pass@1 accuracy on the HumanEval coding benchmark, which beats any approach previously published including GPT-4's 67% (as reported by OpenAI).

Outlook

Looking forward, advances in multimodal AI may further evolve programming tools. Systems capable of processing code, documentation, images, and more could enable a more natural workflow. The future of AI as a programming partner is bright, but requires thoughtful coordination of human creativity and computer-enabled productivity.While promising, effectively leveraging AI programming assistants requires establishing standards through workshops to create useful prompts and pre-prompts for tasks. Focused training ensures proper validation of generated code. Integrating AI into existing environments rather than stand-alone browsers improves developer experience. As research continues, AI programming assistants present opportunities to increase productivity if thoughtfully implemented with an understanding of limitations. With careful oversight, AI stands to automate tedious tasks, freeing developers to focus on complex programming problems.Legal and ethical concerns arise during the pre-training phase, specifically regarding the rights of content creators whose data is used to train the models. Copyright laws and fair use exemptions are debated in relation to the use of copyrighted data by machine learning models.For example, the Free Software Foundation has raised concerns about potential copyright violations associated with code snippets generated by Copilot and Codex. They question whether training on public repositories falls within fair use, how developers can identify infringing code, the nature of machine learning models as modifiable source code or compilations of training data, and the copyrightability of machine learning models. Further, an internal GitHub study found that a small percentage of generated code contained direct copies from the training data, including incorrect copyright notices. OpenAI recognizes the legal uncertainty surrounding these copyright implications and calls for authoritative resolution. The situation has been compared to the Authors Guild, Inc. v. Google, Inc. court case regarding fair use of text snippets in Google Books. Ideally, we want to be able to do this without relying on a cloud-based service that charges us for a request and that may force us to give up the ownership of our data. However, it's very convenient to outsource the AI so that all we have to implement are the prompts and the strategies of how to issue calls with our client. Many of the open-source models have made impressive progress on coding tasks, and they have the advantage of full transparency and openness about their development process. Most of them have been trained on code that's been released under permissive licenses, therefore they are not coming with the same legal concerns as other commercial products.There is a broader impact of these systems beyond coding itself on education and the ecosystem around software development. For example, the emergence of ChatGPT resulted in a massive traffic decline for the popular Stack Overflow question-and-answer forum for programmers. After initially blocking any contributions generated using large language models (LLMs), Stack Overflow launched Overflow AI to bring enhanced search, knowledge ingestion, and other AI features to Stack products. New semantic search is to provide intelligent, conversational results using Stack's knowledge base.Large language models like Codex and ChatGPT excel in code generation for common problems, but struggle with new ones and long prompts. Most importantly, ChatGPT understands syntax well but has limitations in analyzing dynamic code behavior. In programming education, AI models surpass many students but have a lot of room for improvement, however, they haven't yet reached the level of being able to replace programmers and human intelligence. Scrutiny is necessary as mistakes can occur, making expert supervision crucial. The potential of AI tools in coding is encouraging but challenges remain in robustness, generalization, attention span, and true semantic understanding. Further development is needed to ensure reliable and transparent AI programming tools that can augment developers, allowing them to write code faster with fewer bugs.In the next section, we'll see how we can generate software code with LLMs and how we can execute this from within LangChain.

Writing code with LLMs

Let's start off by applying a model to write code for us. We can use one of the publicly available models for generating code. I've listed a few examples before such as ChatGPT or Bard. From LangChain, we can call OpenAI's LLMs, PaLM's code-bison, or a variety of open-source models for example through Replicate, HuggingFace Hub, or – for local models – Llama.cpp, GPT4All, or HuggingFace Pipeline integrations.Let's have a look at StarCoder This screenshot shows the model in a playground on HuggingFace Spaces:

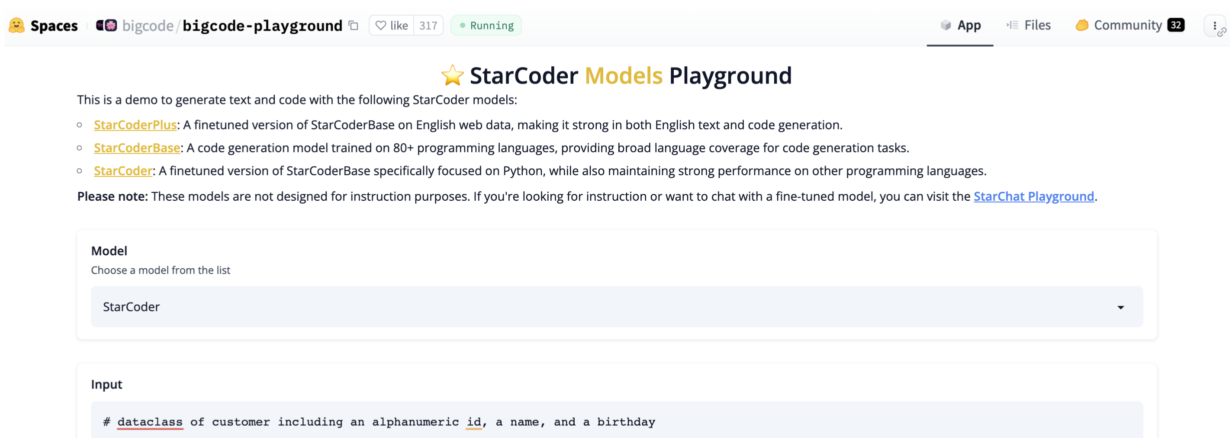


Figure 6.3: StarCoder Models Playground. We can choose between different models: StarCoder, StarCoderPlus, StarCoderBase.This is available at https://huggingface.co/spaces/bigcode/bigcode-playground

Please note that, as the description says, the StarCoder model is not instruction-tuned, which means that we can't give it tasks to do. We can't say "write a class that…" but we can ask it to complete a text as shown in the screenshot, where we prompt the model with "`# dataclass of customer including an alphanumeric id, a name, and a birthday`" – let's try this!We can toggle settings for temperature, max new tokens, top-n, and a repetition penalty. For anything non-trivial, we need to get the max new tokens setting up.I am getting this code, which gives us a useful data model for our hypothetical customer:

```
@dataclass(frozen=True)  # frozen means that the class cannot be modified after creation. Thi
class Customer:
    cust_id : str = field()   # this defines how to create instances from the class; it's cal
    firstname : str = ""      # these are default values if no value was provided when instar
    lastname : str = ""       # note that there can only ever exist one instance variable per
    def __post_init__(self):
        """This method runs automatically once upon instantiation"""
        self._validate_cust_id()    # validate input before saving anything into memory
        self.__set_fullname__()     # set full name based on given inputs
    @property        # getter function used by Python interpreter whenever trying to access p
    def fullname(self)->str:
        return f"{self.lastname}, {self.firstname}"
    ## private methods below here
    ### validation functions go here
    def _validate_cust_id(self):
        assert len(self.cust_id)==8, \
            ValueError("Customer ID must have exactly eight characters")
        try:
            int(self.cust_id[0]) == True
        except Exception as err:
            raise TypeError('First character of CustID should be numeric') from None
    #### setter functions go here
    def __set_fullname__(self):
        '''sets the customers' full name'''
        names = [x.strip().title() for x in re.split("[\W]+", self.firstname)] + [''] * max((
        print(names)
        self.firstname=' '.join(filter(None,[n[:3]for n in reversed(names)])+[''])
        self.lastname =''.join([' '*(4-len(l))+' '+ l[-2:]if not l=='I' else '' for l in filt
```

This is missing the imports, which would usually come before the comment prompt, so I can't fault the model for it, we need to add these two lines to the top:

```
import re
from dataclasses import dataclass, field
```

This doesn't throw an error when we run it as is, which is impressive, however, there are a few things not ideal or even wrong with it when we look a bit more in detail.On the first glance, I personally don't like the commenting style and the lack of docstrings. There are some `print()` statements left in the code; often this is done for debugging purposes. This shouldn't be in production code. It's probably not ideal to have empty strings as defaults for `firstname` and `lastname`. I would expect the customer id to be assigned based on a hashing function – I can't see why the id should have exactly 8 characters as enforced in the validation function for the property. There are more problems with this code that affect its correctness and readability, but there are a few more crucial problems in the code including attempting to write read-only attributes. `firstname` and `lastname` in the dataclass – `frozen=True` in a dataclass means that attributes can't be changed after creation. The logic about creating a variable name from `firstname` and `lastname` using regular expressions in a non-standard syntax is strange to say the least – and incorrect. The last name gets dropped in the process. The filters on the reversed name is also highly suspect. I leave it here. This is giving us some good ideas and a structure to start with, but it's not production code. It doesn't even work. You can see this code as `customer.py` in the book's Github repo.Let's try this again. Perhaps we started off on a bad foot. We started a code snippet in bad syntax expected for beginners and expected code that works. That's not realistic. Let's try again, and start with a prompt that is more up to standard:

```
"""Customer data model.
Here we implement our data model for customers, a dataclass with
fields firstname, lastname, customer_id (a hash).
"""
```

We get the following result:

```
@dataclass(frozen=True)  # frozen means that the object cannot be modified after creation!
class Customer:
    """A class representing one single customer."""
    firstname: str = ""
```

```
    lastname: str = ""
    id_: int = field(default=-1024, init=False)

    def __post_init__(self):
        self.__hash__()

    @property
    def fullname(self)->str:
        return f"{self.firstname} {self.lastname}"

    def __eq__(self, other:"Customer")->bool:
        if isinstance(other, type(self)):
            return all([
                getattr(self, attr)==getattr(other,attr)
                for attr in ["firstname", "lastname"]])
        else:
            raise TypeError("Cannot compare different types")

    def __lt__(self, other:"Customer"):
        try:
            assert len({type(x) for x in [self, other]})==1

            if not hasattr(self,"fullname"):
                setattr(self, 'fullname',f'{self.firstname}{self.lastname}')

            if not hasattr(other,'fullname'):
                setattr(other, 'fullname',f'{other.firstname}{other.lastname}')
            return self.fullname < other.fullname

        except AssertionError as e:
            print('Error:',e )

    def __repr__(self):
        return f"Customer('{self.firstname}', '{self.lastname}','{hex(abs(self._customer_id))

    def __hash__(self):
        hsh = abs(hash((self.firstname+self.lastname))) % ((sys.maxsize + 1)*2)+ sys.maxsize*
        self.id_=hsh

        return hsh
```

It's good to see that the customer id is created using a hash as expected. There's some boilerplate code for comparing two different customer objects. However, again there are problems, similar ones to before. First, it's missing the imports, which I don't understand after our prompt which would be a module docstring to be found at the start of a file, and the imports would come right after. Second, it's again attempting to set an attribute after initialization of the class that's supposed to be frozen. After fixing these two problems, we get our first `Customer()`, then there's a problem, where the customer id is referenced with the wrong name. After fixing this, we can initialize our customer, look at the attributes, and compare one customer to another. I can see how this approach is starting to become useful for writing boilerplate code. You can see this code as `customer2.py` in the book's Github repo. Let's try an instruction-tuned model so we can give it tasks! StarChat, which is based on StarCoder, is available on HuggingFace under https://huggingface.co/spaces/HuggingFaceH4/starchat-playground This screenshot shows an example with StarChat:

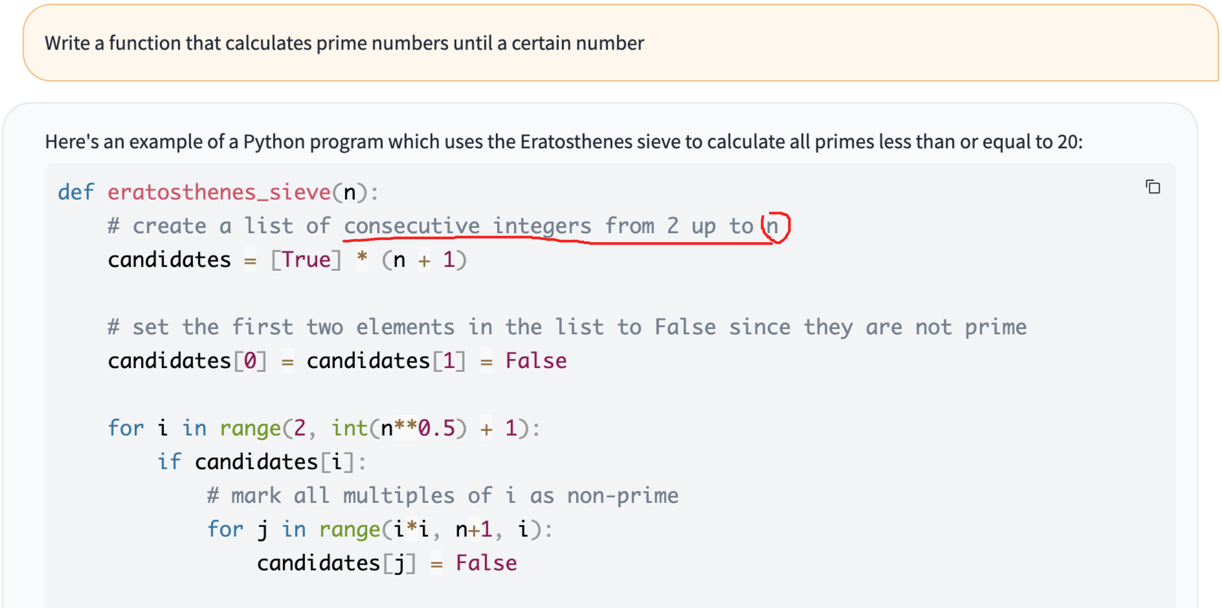
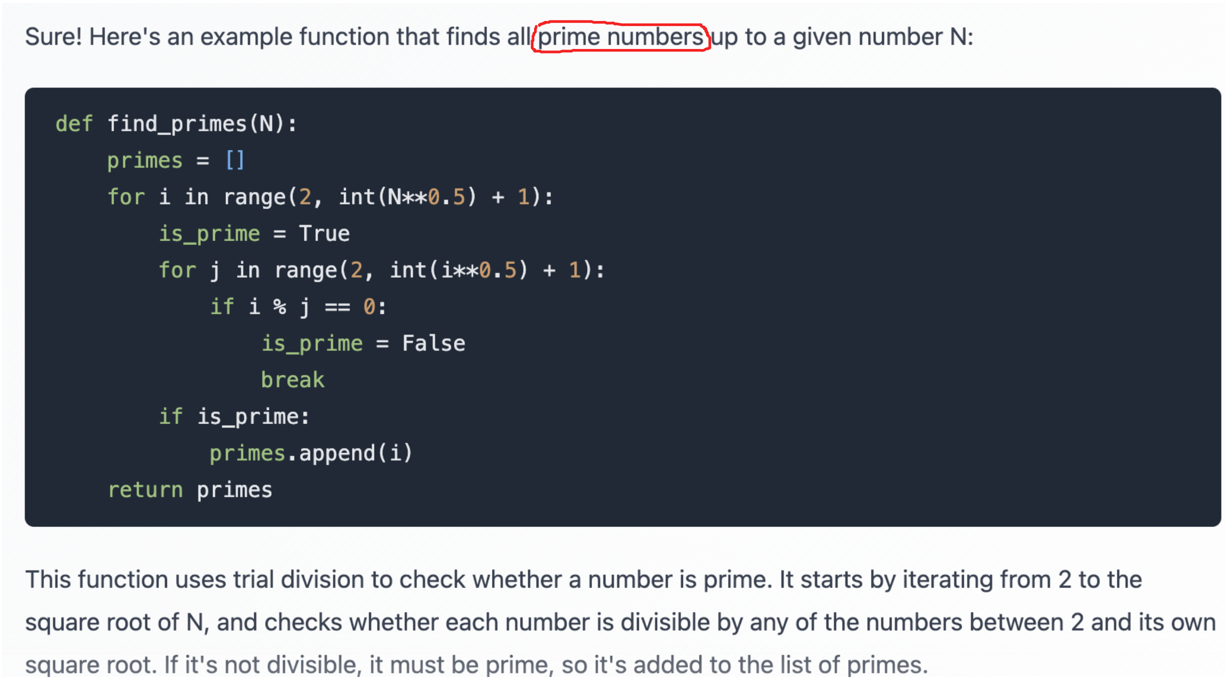


Figure 6.4: StarChat implementing a function in Python for calculating prime numbers. Please note that not all the code is visible in the screenshot.

You can find the complete code listing on Github.For this very example that should be well-known in first year Computer Science courses, no imports are needed. The algorithm's implementation is straightforward. It executes right away and gives the expected result. Within LangChain, we can use the `HuggingFaceHub` integration like this:

```
from langchain import HuggingFaceHub
llm = HuggingFaceHub(
    task="text-generation",
    repo_id="HuggingFaceH4/starchat-alpha",
    model_kwargs={
        "temperature": 0.5,
        "max_length": 1000
    }
)
print(llm(text))
```

As of August 2023, this LangChain integration has some issues with timeouts – hopefully, this is going to get fixed soon. We are not going to use it here.Llama2 is not one of the best models for coding with a pass@1 of about 29 as mentioned earlier, however, we can try it out on HuggingFace chat:



Sure! Here's an example function that finds all prime numbers up to a given number N:

```
def find_primes(N):
    primes = []
    for i in range(2, int(N**0.5) + 1):
        is_prime = True
        for j in range(2, int(i**0.5) + 1):
            if i % j == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(i)
    return primes
```

This function uses trial division to check whether a number is prime. It starts by iterating from 2 to the square root of N, and checks whether each number is divisible by any of the numbers between 2 and its own square root. If it's not divisible, it must be prime, so it's added to the list of primes.

Figure 6.5: HuggingFace chat with Llama2 at https://huggingface.co/chat/

Please note that this is only the beginning of the output. Llama2 finds a good implementation and the explanations are spot on. Well done, StarCoder and Llama2! – Or perhaps, this was just too easy? There so many ways to get code completion or generation. We can run even try a small local model:

```
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
checkpoint = "Salesforce/codegen-350M-mono"
model = AutoModelForCausalLM.from_pretrained(checkpoint)
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
pipe = pipeline(
    task="text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=500
)
text = """
def calculate_primes(n):
    \"\"\"Create a list of consecutive integers from 2 up to N.
    For example:
    >>> calculate_primes(20)
    Output: [2, 3, 5, 7, 11, 13, 17, 19]
    \"\"\"
"""
```

CodeGen is a model by Salesforce AI Research. CodeGen 350 Mono performed 12.76% pass@1 in HumanEval. As of July 2023, new versions of CodeGen was released with only 6B parameters that are very competitive, which clocks in at a performance of 26.13%. This last model was trained on the BigQuery dataset containing C, C++, Go, Java, Javascript, and Python, as well as the BigPython dataset, which consists of 5.5TB of Python code. Another interesting, small model is Microsoft's CodeBERT (2020), a model for program synthesis that has been trained and tested on Ruby, Javascript, Go, Python, Java, and PHP.. Since this model was released before the HumanEval benchmark, the performance statistics for the benchmark were not part of the initial publication.We can now get the output from the pipeline directly like this:

```
completion = pipe(text)
print(completion[0]["generated_text"])
```

Alternatively, we can wrap this pipeline via the LangChain integration:

```
llm = HuggingFacePipeline(pipeline=pipe)
llm(text)
```

This is a bit verbose. There's also the more convenient constructor method `HuggingFacePipeline.from_model_id()`.I am getting something similar to the StarCoder output. I had to add an `import math`, but the function works. This pipeline we could use in a LangChain agent, however, please note that this model is not instruction-tuned, so you cannot give it tasks, only completion tasks. You can also use all of these models for code embeddings. Other models that have been instruction-tuned and are available for chat, can act as your techie assistant to help with advice, document and explain existing code, or translate code into other programming languages – for the last task they need to have been trained on enough samples in these languages.Please note that the approach taken here is a bit naïve. For example, we could be taking more samples and choose between them such as in the a few of the papers we've discussed.Let's now try to implement a feedback cycle for code development, where we validate and run the code and change it based on feedback.

Automated software development

We'll now going to write a fully-automated agent that will write code for us and fix any problems responding to feedback.In LangChain, we have several integrations for code execution like these the `LLMMathChain`, which executes Python code to solve math questions, and the `BashChain` that executes Bash terminal commands, which can help with system administration tasks. However, these are for problem solving with code rather than creating a software.This can, however, work quite well.

```
from langchain.llms.openai import OpenAI
from langchain.agents import load_tools, initialize_agent, AgentType
llm = OpenAI()
tools = load_tools(["python_repl"])
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=Tru
result = agent("What are the prime numbers until 20?")
print(result)
```

We can see how the prime number calculations get processed quite well under the hood between OpenAI's LLM and the Python interpreter:

```
Entering new AgentExecutor chain...
I need to find a way to check if a number is prime
Action: Python REPL
Action Input:
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
Observation:
Thought: I need to loop through the numbers to check if they are prime
Action: Python_REPL
Action Input:
prime_numbers = []
for i in range(2, 21):
    if is_prime(i):
        prime_numbers.append(i)
Observation:
Thought: I now know the prime numbers until 20
Final Answer: 2, 3, 5, 7, 11, 13, 17, 19
Finished chain.
{'input': 'What are the prime numbers until 20?', 'output': '2, 3, 5, 7, 11, 13, 17, 19'}
```

We get to the right answer about the prime numbers, however, it's not entirely clear how this approach would scale for building software products, where it is about modules, abstractions, separation of concerns, and maintainable code. There are a few interesting implementations for this around. The MetaGPT library approaches this with an agent simulation, where different agents represent job roles in a company or IT department:

```
from metagpt.software_company import SoftwareCompany
from metagpt.roles import ProjectManager, ProductManager, Architect, Engineer
async def startup(idea: str, investment: float = 3.0, n_round: int = 5):
```

```
    """Run a startup. Be a boss."""
    company = SoftwareCompany()
    company.hire([ProductManager(), Architect(), ProjectManager(), Engineer()])
    company.invest(investment)
    company.start_project(idea)
    await company.run(n_round=n_round)
```

This is a really inspiring use case of an agent simulation. The llm-strategy library by Andreas Kirsch generates code for dataclasses using decorator patterns. Other examples for automatic software development include AutoGPT and BabyGPT although these often tend to get stuck in loops or stop because of failures. A simple planning and feedback loop like this can be implemented in LangChain with a ZeroShot Agent and a planner. The Code-It project by Paolo Rechia and Gpt-Engineer by AntonOsika both follows such as pattern as illustrated in this graph for Code-It:

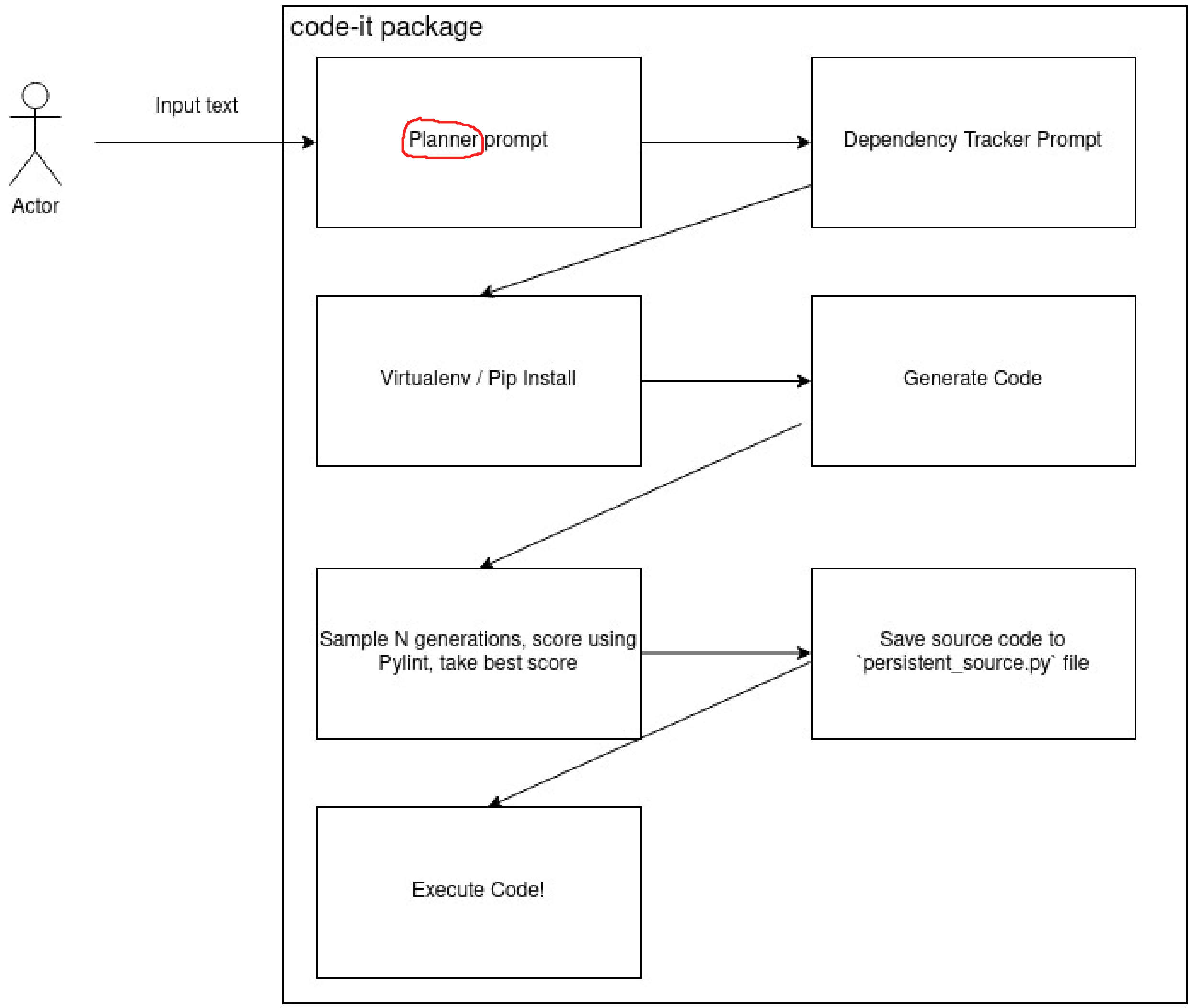


Figure 6.6: Code-It control flow (source: https://github.com/ChuloAI/code-it).

Many of these steps consist of specific prompts that are sent to LLMs with instructions to break down the project or to set up the environment. It's quite impressive to implement the full feedback loop with all the tools. We can implement a relatively simple feedback loop in different ways in LangChain, for example using `PlanAndExecute` chain, a `ZeroShotAgent`, or `BabyAGI`. Let's go with `PlanAndExecute`! The main idea is to set up a chain and execute it with the objective of writing a software, like this:

```
llm = OpenAI()
planner = load_chat_planner(llm)
```

```
executor = load_agent_executor(
    llm,
    tools,
    verbose=True,
)
agent_executor = PlanAndExecute(
    planner=planner,
    executor=executor,
    verbose=True,
    handle_parsing_errors="Check your output and make sure it conforms!",
    return_intermediate_steps=True
)
agent_executor.run("Write a tetris game in python!")
```

I am omitting the imports here, but you can find the full implementation in the Github repo of the book. The other options can be found there as well. There are a few more pieces to this, but this could already write some code, depending on the instructions that we give. One thing we need is clear instructions for a language model to write Python code in a certain form:

```
DEV_PROMPT = (
    "You are a software engineer who writes Python code given tasks or objectives. "
    "Come up with a python code for this task: {task}"
    "Please use PEP8 syntax and comments!"
)
software_prompt = PromptTemplate.from_template(DEV_PROMPT)
software_llm = LLMChain(
    llm=OpenAI(
        temperature=0,
        max_tokens=1000
    ),
    prompt=software_prompt
)
```

We need to make sure, we take a model that is able to come up with code. We've discussed already the models we can choose between for this. I've chosen a longer context so we don't get cut off in the middle of a function, and a low temperature, so it doesn't get too wild.However, on its own this model wouldn't be able to store it to file, do anything meaningful with it, and act on the feedback from the execution. We need to come up with code and then test it, and see if it works. Let's see how we can implement this – that's in the `tools` argument to the agent executor, let's see how this is defined!

```
software_dev = PythonDeveloper(llm_chain=software_llm)
code_tool = Tool.from_function(
    func=software_dev.run,
    name="PythonREPL",
    description=(
        "You are a software engineer who writes Python code given a function description or t
    ),
    args_schema=PythonExecutorInput
)
```

The `PythonDeveloper` class has all the logic about taking tasks given in any form and translating them into code. I won't go into all the detail here, however, the main idea is here:

```
class PythonDeveloper():
    """Execution environment for Python code."""
    def __init__(
            self,
            llm_chain: Chain,
    ):
        self. llm_chain = llm_chain
    def write_code(self, task: str) -> str:
        return self.llm_chain.run(task)
    def run(
            self,
            task: str,
    ) -> str:
        """Generate and Execute Python code."""
        code = self.write_code(task)
```

```
        try:
            return self.execute_code(code, "main.py")
        except Exception as ex:
            return str(ex)
    def execute_code(self, code: str, filename: str) -> str:
        """Execute a python code."""
        try:
            with set_directory(Path(self.path)):
                ns = dict(__file__=filename, __name__="__main__")
                function = compile(code, "<>", "exec")
                with redirect_stdout(io.StringIO()) as f:
                    exec(function, ns)
                    return f.getvalue()
```

I am again leaving out a few pieces. The error handling if very simplistic here. In the implementation on Github, we can distinguish different kinds of errors we are getting, such as these:

- `ModuleNotFoundError` : this means that the code tries to work with packages that we don't have installed. I've implemented logic to install these packages.
- `NameError` : using variable names that don't exist.
- `SyntaxError` : the code often doesn't close parentheses or is not even code
- `FileNotFoundError` : the code relies on files that don't exist. I've found a few times that the code tried showing images that were made up.
- `SystemExit` : if something more dramatic happens and Python crashes.

I've implemented logic to install packages for `ModuleNotFoundError` , and clearer messages for some of these problems. In the case of missing images, we could add a generative image model to create these. Returning all this as enriched feedback to the code generation, results in increasingly specific output such as this:

```
Write a basic tetris game in Python with no syntax errors, properly closed strings, brackets,
```

The Python code itself gets compiled and executed in a subdirectory and we take redirect the output of the Python execution in order to capture it – both of this is implemented as Python contexts. Please be cautious with executing code on your system, because some of these approaches are quite sensitive to security, because they lack a sandboxed environment, although tools and frameworks exists such as codebox-api, RestrictedPython, pychroot, or setuptools' DirectorySandbox to just name a few of these for Python.So let's set tools:

```
ddg_search = DuckDuckGoSearchResults()
tools = [
    codetool,
    Tool(
        name="DDGSearch",
        func=ddg_search.run,
        description=(
            "Useful for research and understanding background of objectives. "
            "Input: an objective. "
            "Output: background information about the objective. "
        )
    )
]
```

An internet search is definitely worth adding to make sure we are implementing something that has to do with our objective. I've seen a few implementations of Rock, Paper, Scissors instead of Tetris.We can define additional tools such as a planner that breaks down the tasks into functions. You can see this in the repo.Running our agent executor with the objective to implement tetris, every time the results are a bit different. I see several searches for requirements and game mechanics, and several times a code is produced and run. The pygame library is installed. The final code snippet is not the final product, but it brings up a window:

```
# This code is written in PEP8 syntax and includes comments to explain the code
# Import the necessary modules
import pygame
import sys
# Initialize pygame
pygame.init()
```

```
# Set the window size
window_width = 800
window_height = 600
# Create the window
window = pygame.display.set_mode((window_width, window_height))
# Set the window title
pygame.display.set_caption('My Game')
# Set the background color
background_color = (255, 255, 255)
# Main game loop
while True:
    # Check for events
    for event in pygame.event.get():
        # Quit if the user closes the window
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
    # Fill the background with the background color
    window.fill(background_color)
    # Update the display
    pygame.display.update()
```

The code is not too bad in terms of syntax – I guess the prompt must have helped. However, in terms of functionality it's very far from Tetris. This implementation of a fully-automated agent for software development is still quite experimental. It's also very simple and basic, consisting only of about 340 lines of Python including the imports, which you can find on Github.I think a better approach could be to break down all the functionality into functions and maintain a list of functions to call, which can be used in all subsequent generations of code. We could also try a test-driven development approach or have a human give feedback rather than a fully automated process.An advantage to our approach is however that it's easy to debug, since all steps including searches and generated code are written to a log file in the implementation.Let's summarize!

Summary

In this chapter, we've discussed LLMs for source code, and how they can help in developing software. There are quite a few areas, where LLMs can benefit software development, mostly as coding assistants.We've applied a few models for code generation using naïve approaches and we've evaluated them qualitatively. We've seen that the suggested solutions seem superficially correct but don't actually perform the task or are full of bugs. This could particularly affect beginners, and could have significant implications regarding safety and reliability.In previous chapters, we've seen LLMs used as goal-driven agents to interact with external environments. In coding, compiler errors, results of code execution can be used to provide feedback as we've seen. Alternatively, we could have used human feedback or implemented tests. Let's see if you remember some of the key takeaways from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions from memory. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. What can LLMs do to help in software development?
2. How do you measure a code LLM's performance on coding tasks?
3. Which code LLM models are available, both open- and closed-source?
4. How does the Reflexion strategy work?
5. What options do we have available to establish a feedback loop for writing code?
6. What do you think is the impact of generative AI on software development?

7 LLMs for Data Science

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

This chapter is about how generative AI can automate data science. Generative AI, in particular large language models (LLMs) have the potential to accelerate scientific progress across various domains, especially by providing efficient analysis of research data and aiding in literature review processes. A lot of current approaches that fall within the domain of AutoML can help data scientists increase their productivity and help make the data science more repeatable. I'll first give an overview over automation in data science and then we'll discuss how data science is affected by generative AI.Next, we'll discuss how we can use code generation and tools in different ways to answer questions related to data science. This can come in the form of doing a simulation or of enriching out dataset with additional information. Finally, we put the focus on exploratory analysis of structured datasets. We can set up agents to run SQL or tabular data in Pandas. We'll see how we can ask questions about the dataset, statistical questions about the data, or ask for visualizations.Throughout the chapter, we'll work on different approaches to doing data science with LLMs, which you can find in the `data_science` directory in the Github repository for the book at https://github.com/benman1/generative_ai_with_langchainThe main sections are:

- Automated data science
- Agents can answer data science questions
- Data exploration with LLMs

Let's start by discussing how data science can be automated and which parts of it, and how generative AI will impact data science.

Automated data science

Data science is a field that combines computer science, statistics, and business analytics to extract knowledge and insights from data. Data scientists use a variety of tools and techniques to collect, clean, analyze, and visualize data. They then use this information to help businesses make better decisions.The work of a data scientist can vary depending on the specific role and industry. However, some common tasks that data scientists might perform include:

- Collecting data: Data scientists need to collect data from a variety of sources, such as databases, social media, and sensors.
- Cleaning data: Data scientists need to clean data to remove errors and inconsistencies.
- Analyzing data: Data scientists use a variety of statistical and machine learning techniques to analyze data.
- Visualizing data: Data scientists use data visualizations to communicate insights to stakeholders.
- Building models: Data scientists build models to predict future outcomes or make recommendations.

Data analysis is a subset of data science that focuses on extracting insights from data. Data analysts use a variety of tools and techniques to analyze data, but they typically do not build models.The overlap between data science and data analysis is that both fields involve working with data to extract insights. However, data scientists typically have a more technical skillset than data analysts. Data scientists are also more likely to build models and sometimes deploy models into production. Data scientists sometimes deploy models into production so that they can be used to make decisions in real time, however, we'll avoid automatic deployment of models in this discussion.Here is a table that summarizes the key differences between data science and data analysis:

Feature	Data Science	Data Analysis
Technical skillset	More technical	Less technical
Machine learning	Yes	No
Model deployment	Sometimes	No
Focus	Extracting insights and building models	Extracting insights

Figure 7.1: Comparison of Data Science and Data Analysis.

The common denominator between the two is collecting data, cleaning data, analyzing data, visualizing data, all of which fall into the category of extracting insights. Data science, additionally is about training machine learning models and usually it has a stronger focus on statistics. In some cases, depending on the setup in the company and industry practices, deploying models and writing software can be added to the list for data science. Automatic data analysis and data science aims to automate many of the tedious, repetitive tasks involved in working with data. This includes data cleaning, feature engineering, model training, tuning, and deployment. The goal is to make data scientists and analysts more productive by enabling faster iterations and less manual coding for common workflows.A lot of these tasks can be automated to some degree. Some of the tasks for data science are similar to those of a software developer that we talked about in chapter 6, *Developing Software*, namely writing and deploying software although with a narrower focus, on models.Data science platforms such as Weka, H2O, KNIME, RapidMiner, and Alteryx are unified machine learning and analytics engines that can be used for a variety of tasks, including preprocessing of large volumes of data and feature extraction. All of these come with a graphical user interface (GUI), have the capability to integrate 3rd party data source, and write custom plug-ins. KNIME is mostly open-source, however also offers a commercial product called KNIME Server. Apache Spark is a versatile tool that can be used for a variety of tasks involved in data science. It can be used to to clean, transform, extract features, and prepare high-volume data for analysis and also to train and deploy machine learning models, both in streaming scenarios, when it's about real-time decisions or monitoring events.Further, at its most fundamental, libraries for scientific computing such as NumPy can be serve for all tasks involved in automated data science. Deep learning and machine learning libraries such as TensorFlow, Pytorch, and Scikit-Learn can be used for a variety of tasks beyond creating complex machine learning models, including data preprocessing and feature extraction. Orchestration tools such as Airflow, Kedro, or others can help in all these tasks, and include a lot of integrations with specific tools related to all steps in data science.Several data science tools have generative AI support. In *Chapter 6*, *Developing Software*, we've already mentioned GitHub Copilot, but there are others such as the PyCharm AI Assistant, and even more to the point, Jupyter AI, which is a subproject of Project Jupyter that brings generative artificial intelligence to Jupyter notebooks. Jupyter AI allows users to generate code, fix errors, summarize content, and even create entire notebooks using natural language prompts. The tool connects Jupyter with LLMs from various providers, allowing users to choose their preferred model and embedding.Jupyter AI prioritizes responsible AI and data privacy. The underlying prompts, chains, and components are open source, ensuring transparency. It saves metadata about model-generated content, making it easy to track AI-generated code within the workflow. Jupyter AI respects user data privacy and only contacts LLMs when explicitly requested, which is done through LangChain integrations.To use Jupyter AI, users can install the appropriate version for JupyterLab and access it through a chat UI or the magic command interface. The chat interface features Jupyternaut, an AI assistant that can answer questions, explain code, modify code, and identify errors. Users can also generate entire notebooks from text prompts.The software allows users to teach Jupyternaut about local files and interact with LLMs using magic commands in notebook environments. It supports multiple providers and offers customization options for the output format. This screenshot from the documentation shows the chat feature, the Jupyternaut chat:

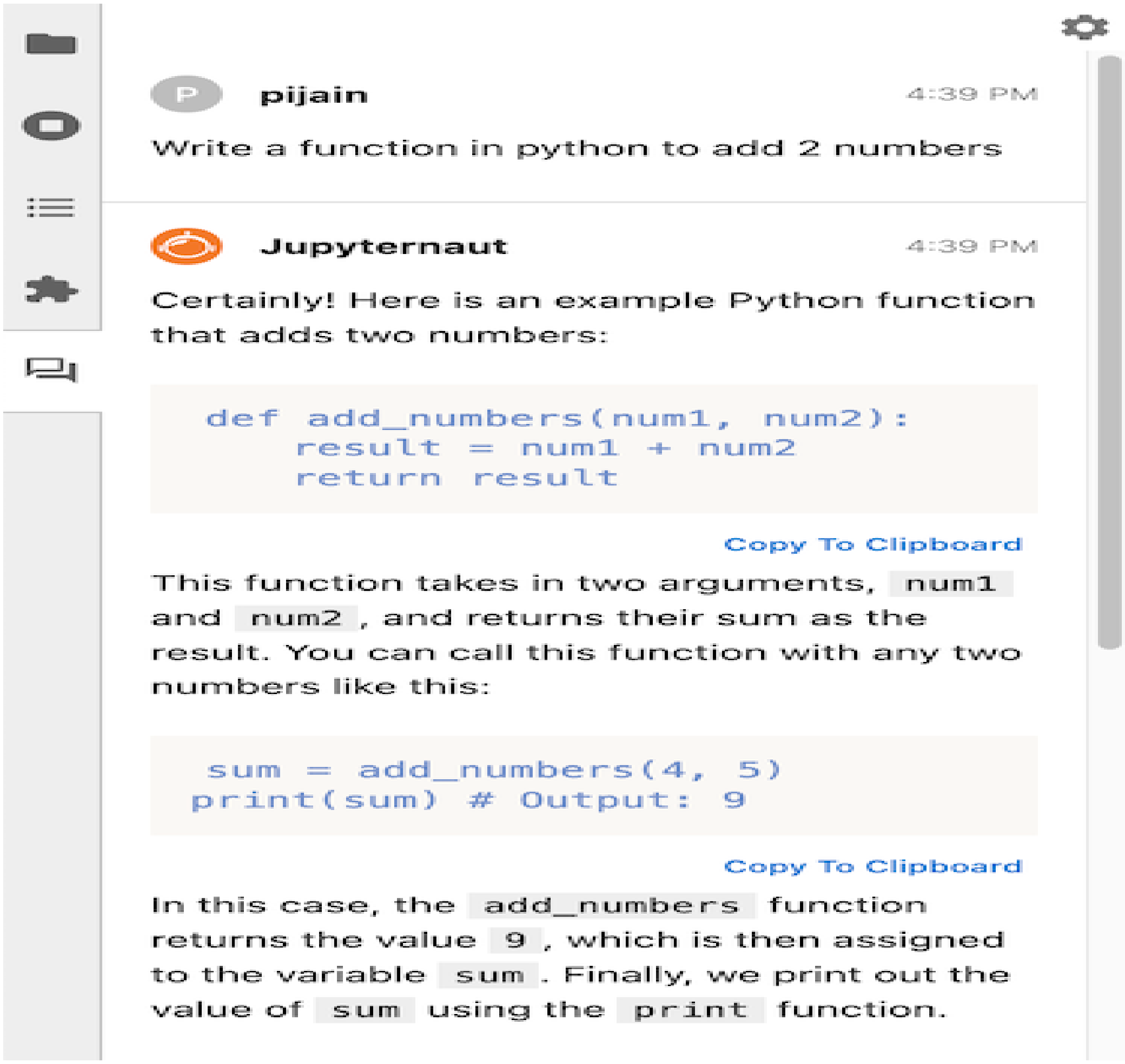


Figure 7.2: Jupyter AI – Jupyternaut chat.

It should be plain to see that having a chat like that at your fingertips to ask questions, create simple functions, or change existing functions can be a boon to data scientists. The benefits of using these tools include improved efficiency, reduced manual effort in tasks like model building or feature selection, enhanced interpretability of models, identification and fixing of data quality issues, integration with other scikit-learn pipelines (pandas_dq), and overall improvement in the reliability of results.Overall, automated data science can greatly accelerate analytics and ML application development. It allows data scientists to focus on higher value and creative aspects of the process. Democratizing data science for business analysts is also a key motivation behind automating these workflows. In the following sections, we'll look into these steps in turn, and we'll discuss automating them, and we'll highlight how generative AI can make a contribution to improving the workflow and create efficiency gains.

Data collection

Automated data collection is the process of collecting data without human intervention. Automatic data collection can be a valuable tool for businesses. It can help businesses to collect data more quickly and efficiently, and it can free up human resources to focus on other tasks. Generally, in the context of data science or analytics we refer to

ETL (Extract, Transform, and Load) as the process that not only takes data from one or more sources (the data collection), but also prepares it for specific use cases. The ETL process typically follows these steps:

1. Extract: The data is extracted from the source systems. This can be done using a variety of methods, such as web scraping, API integration, or database queries.
2. Transform: The data is transformed into a format that can be used by the data warehouse or data lake. This may involve cleaning the data, removing duplicates, and standardizing the data format.
3. Load: The data is loaded into the data warehouse or data lake. This can be done using a variety of methods, such as bulk loading or incremental loading.

ETL and data collection can be done using a variety of tools and techniques, such as:

- Web scraping: Web scraping is the process of extracting data from websites. This can be done using a variety of tools, such as Beautiful Soup, Scrapy, Octoparse, .
- APIs (Application Programming Interfaces): These are a way for software applications to talk to each other. Businesses can use APIs to collect data from other companies without having to build their own systems.
- Query languages: Any database can serve as data source including of the SQL (Structured Query Language) or the no-SQL variety.
- Machine learning: Machine learning can be used to automate the process of data collection. For example, businesses can use machine learning to identify patterns in data and then collect data based on those patterns.

Once the data has been collected, it can be processed to prepare it for use in a data warehouse or data lake. The ETL process will typically clean the data, remove duplicates, and standardize the data format. The data will then be loaded into the data warehouse or data lake, where it can be used by data analysts or data scientists to gain insights into the business.There are many ETL tools including commercial ones such as AWS glue, Google Dataflow, Amazon Simple Workflow Service (SWF), dbt, Fivetran, Microsoft SSIS, IBM InfoSphere DataStage, Talend Open Studio or open-source tools such as Airflow, Kafka, and Spark. In Python are many more tools, too many to list all, such as Pandas for data extraction and processing, and even celery and joblib, which can serve as ETL orchestration tools. In LangChain, there's an integration with Zapier, which is an automation tool that can be used to connect different applications and services. This can be used to automate the process of data collection from a variety of sources.Here are some of the benefits of using automated ETL tools:

- Increased accuracy: Automated ETL tools can help to improve the accuracy of the data extraction, transformation, and loading process. This is because the tools can be programmed to follow a set of rules and procedures, which can help to reduce human error.
- Reduced time to market: Automated ETL tools can help to reduce the time it takes to get data into a data warehouse or data lake. This is because the tools can automate the repetitive tasks involved in the ETL process, such as data extraction and loading.
- Improved scalability: Automated ETL tools can help to improve the scalability of the ETL process. This is because the tools can be used to process large volumes of data, and they can be easily scaled up or down to meet the needs of the business.
- Improved compliance: Automated ETL tools can help to improve compliance with regulations such as GDPR and CCPA. This is because the tools can be programmed to follow a set of rules and procedures, which can help to ensure that data is processed in a compliant manner.

The best tool for automatic data collection will depend on the specific needs of the business. Businesses should consider the type of data they need to collect, the volume of data they need to collect, and the budget they have available.

Visualization and EDA

Automated EDA (Exploratory Data Analysis) and visualization refer to the process of using software tools and algorithms to automatically analyze and visualize data, without significant manual intervention. Traditional EDA involves manually exploring and summarizing data to understand its various aspects before performing machine learning or deep learning tasks. It helps in identifying patterns, detecting inconsistencies, testing assumptions, and gaining insights. However, with the advent of large datasets and the need for efficient analysis, automated EDA has become important.Automated EDA and visualization tools provide several benefits. They can speed up the data analysis process, reducing the time spent on tasks like data cleaning, handling missing values, outlier detection, and feature engineering. These tools also enable a more efficient exploration of complex datasets by generating interactive visualizations that provide a comprehensive overview of the data.Several tools are available for automated EDA and visualization, including:

- D-Tale: A library that facilitates easy visualization of pandas data frames. It supports interactive plots, 3D plots, heatmaps, correlation analysis, custom column creation.
- ydata-profiling (previously pandas profiling): An open-source library that generates interactive HTML reports (`ProfileReport`) summarizing different aspects of the dataset such as missing values statistics, variable types distribution profiles, correlations between variables. It works with Pandas as well as Spark DataFrames.
- Sweetviz: A Python library that provides visualization capabilities for exploratory data analysis with minimal code required. It allows for comparisons between variables or datasets.
- Autoviz: This library automatically generates visualizations for datasets regardless of their size with just a few lines of code.
- DataPrep: With just a few lines you can collect data from common data sources do EDA and data cleaning such as standardization of column names or entries.
- Lux: Displays a set of visualizations with interesting trends and patterns in the dataset displayed via an interactive widget that users can quickly browse in order to gain insights.

The use of generative AI in data visualization adds another dimension to automated EDA by allowing algorithms to generate new visualizations based on existing ones or specific user prompts. Generative AI has the potential to enhance creativity by automating part of the design process while maintaining human control over the final output.Overall, automated EDA and visualization tools offer significant advantages in terms of time efficiency, comprehensive analysis, and the generation of meaningful visual representations of data. Generative AI has the potential to revolutionize data visualization in a number of ways. For example, it can be used to create more realistic and engaging visualizations, which can help in business communication and to communicate data more effectively to stakeholders to provide each user with the information they need to gain insights and make informed decisions.Generative AI can enhance and extend the creation that traditional tools are capable of by making personalized visualizations tailored to the individual needs of each user. Further, Generative AI can be used to create interactive visualizations that allow users to explore data in new and innovative ways.

Pre-processing and feature extraction

Automated data preprocessing is the process of automating the tasks involved in data preprocessing. This can include tasks such as data cleaning, data integration, data transformation, and feature extraction. It is related to the transform step in ETL, so there’s a lot of overlap in tools and techniques.Data preprocessing is important because it ensures that data is in a format that can be used by data analysts and machine learning models. This includes removing errors and inconsistencies from the data, as well as converting it into a format that is compatible with the analytical tools that will be used.Manually engineering features can be tedious and time consuming, so automating this process is valuable. Recently, several open source Python libraries have emerged to help auto-generate useful features from raw data as we’ll see.Featuretools offers a general-purpose framework that can synthesize many new features from transactional and relational data. It integrates across multiple ML frameworks making it flexible. Feature Engine provides a simpler set of transformers focused on common data transformations like handling missing data. For optimizing feature engineering specifically for tree-based models, ta from Microsoft shows strong performance through techniques like automatic crossing.AutoGluon Features applies neural network style automatic feature generation and selection to boost model accuracy. It is tightly integrated with the AutoGluon autoML capabilities. Finally, TensorFlow Transform operates directly on Tensorflow pipelines to prepare data for models during training. It has progressed rapidly with diverse open source options now available. Featuretools provides the most automation and flexibility while integrating across ML frameworks. For tabular data, ta and Feature Engine offer easy-to-use transformers optimized for different models. Tf.transform is ideal for TensorFlow users, while AutoGluon specializes in the Apache MXNet deep learning software framework.As for time series data, Tsfel is a library that extracts features from time series data. It allows users to specify the window size for feature extraction and can analyze the temporal complexity of the features. It computes statistical, spectral, and temporal features.On the other hand, tsflex is a time series feature extraction toolkit that is flexible and efficient for sequence data. It makes few assumptions about the data structure and can handle missing data and unequal lengths. It also computes rolling features.Both libraries offer more modern options for automated time series feature engineering compared to tsfresh. Tsfel is more full-featured, while tsflex emphasizes flexibility on complex sequence data.There are a few tools that focus on data quality for machine learning and data science that come with data profiling and automatic data transformations. For example, the pandas-dq library, which can be integrated with scikit-learn pipelines, offers a range of useful features for data profiling, train-test comparison, data cleaning, data imputation (filling missing values), and data transformation (e.g., skewness correction). It helps improve the quality of data analysis by addressing potential issues before modeling.More focused on improved reliability through early identification of potential issues or errors are tools like Great Expectations and Deequ. Great Expectations is a tool for validating, documenting, and profiling data to maintain quality and improve communication between teams. It allows users to assert expectations on the data, catch issues quickly through unit tests for data, create documentation and reports based on expectations. Deequ is built on top of Apache Spark for defining unit tests for data quality in large datasets. It lets users explicitly state assumptions about the dataset and verifies them through checks or constraints on attributes. By ensuring adherence to these assumptions, it prevents crashes or wrong outputs in downstream applications.All these libraries allow data scientists to shorten feature preparation and expand the feature space to improve model quality. Automated feature engineering is becoming essential to leveraging the full power of ML algorithms on complex real-world data.

AutoML

Automated Machine Learning (AutoML) frameworks are tools that automate the process of machine learning model development. They can be used to automate tasks such as data cleaning, feature selection, model training, and hyperparameter tuning. This can save data scientists a lot of time and effort, and it can also help to improve the quality of machine learning models.The basic idea of AutoML is illustrated in this diagram from the Github repo of the mljar autoML library (source: https://github.com/mljar/mljar-supervised):

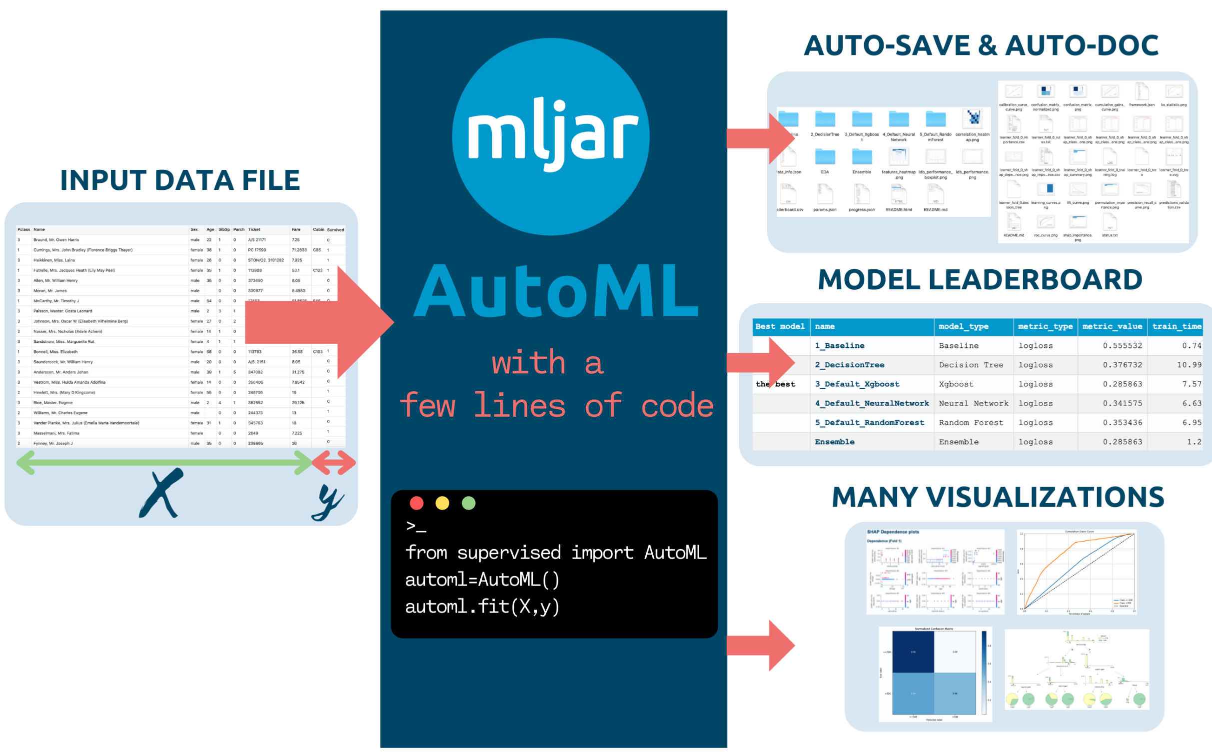


Figure 7.3: How AutoML works.

Load some data, try different combinations of preprocessing methods, ML algorithms, training and model parameters, create explanations, compare results in a leaderboard together with visualizations. The main value proposition of an AutoML framework is the ease-of-use of and an increased developer productivity in finding a machine learning model, understanding it, and getting it to production. AutoML tools have been around for a long time. One of the first broader frameworks was AutoWeka, written in Java, and it was designed to automate the process of machine learning model development for tabular data in the Weka (Waikato Environment for Knowledge Analysis) machine learning suite, which is developed at the University of Waikato.In the years since AutoWeka was released, there have been many other AutoML frameworks developed. Some of the most popular AutoML frameworks today include auto-sklearn, autokeras, NASLib, Auto-Pytorch, tpot, optuna, autogluon, and ray (tune). These frameworks are written in a variety of programming languages, and they support a variety of machine learning tasks.Recent advances in autoML and neural architecture search have allowed tools to automate large parts of the machine learning pipeline. Leading solutions like Google AutoML, Azure AutoML, and H2O AutoML/Driverless AI can automatically handle data prep, feature engineering, model selection, hyperparameter tuning, and deployment based on the dataset and problem type. These make machine learning more accessible to non-experts as well.Current autoML solutions can handle structured data like tables and time series data very effectively. They can automatically generate relevant features, select algorithms like tree ensembles, neural networks or SVMs, and tune hyperparameters. Performance is often on par or better than manual process due to massive hyperparameter search. AutoML for unstructured data like images, video and audio is also advancing rapidly with neural architecture search techniques.Open source libraries like AutoKeras, AutoGluon, and AutoSklearn provide accessible autoML capabilities as well. However, most autoML tools still require some coding and data science expertise. Fully automating data science remains challenging and autoML does have limitations in flexibility and controllability. But rapid progress is being made with more user-friendly and performant solutions coming to market.Here's a tabular summary of frameworks:

Framework	**Language**	**ML Frameworks**	**First Release**	**Key Features**	**Data Types**	**Maintainer**	**Github stars**
Auto-Keras	Python	Keras	2017	Neural architecture search, easy to use	Images, text, tabular	Keras Team (DATA Lab, Texas A&M)	8896
Auto-PyTorch	Python	PyTorch	2019	Neural architecture search, hyperparameter tuning	Tabular, text, image, time series	AutoML Group, Univ. of Freiburg	2105
Auto-Sklearn	Python	Scikit-learn	2015	Automated scikit-learn workflows	Tabular	AutoML Group, Univ. of Freiburg	7077
Auto-WEKA	Java*	WEKA	2012	Bayesian optimization	Tabular	University of British Columbia	315
AutoGluon	Python	MXNet, PyTorch	2019	Optimized for deep learning	Text, image, tabular	Amazon	6032
AWS SageMaker Autopilot	Python	XGBoost, sklearn	2020	Cloud-based, simple	Tabular	Amazon	-
Azure AutoML	Python	Scikit-learn, PyTorch	2018	Explainable models	Tabular	Microsoft	-
DataRobot	Python, R	Multiple	2012	Monitoring, explainability	Text, image, tabular	DataRobot	-
Google AutoML	Python	TensorFlow	2018	Easy to use, cloud-based	Text, image, video, tabular	Google	-
H2O AutoML	Python, R	XGBoost, GBMs	2017	Automatic workflow, ensembling	Tabular, time	h2o.ai	6430

					series, images		
hyperopt-sklearn	Python	Scikit-learn	2014	Hyperparameter tuning	Tabular	Hyperopt team	1451
Ludwig	Python	Transformers/Pytorch	2019	Low-code framework for building and tuning custom LLMs and deep neural networks	Multiple	Linux Foundation	9083
MLJar	Python	Multiple	2019	Explainable, customizable	Tabular	MLJar	2714
NASLib	Python	PyTorch, TensorFlow/Keras	2020	Neural architecture search	Images, text	AutoML Group, Univ. of Freiburg	421
Optuna	Python	Agnostic	2019	Hyperparameter tuning	Agnostic	Preferred Networks Inc	8456
Ray (Tune)	Python	Agnostic	2018	Distributed hyperparameter tuning; Accelerating ML workloads	Agnostic	University of California, Berkeley	26906
TPOT	Python	Scikit-learn, XGBoost	2016	Genetic programming, pipelines	Tabular	Epistasis Lab, Penn State	9180
TransmogrifAI	Scala	Spark ML	2018	AutoML on Spark	Text, tabular	Salesforce	2203

Figure 7.4: Comparison of open-source AutoML frameworks. Weka can be accessed from Python as pyautoweka. Stars for Ray Tune and H2O concern the whole project rather than only the automl part. The H2O commercial product related to AutoML is Driverless AI. Most projects are maintained by a community of contributors not affiliated with just one company

I have only included the biggest frameworks, libraries or products – omitting a few. Although the focus is on open-source frameworks in Python, I've included a few big commercial products. Github stars aim to show the popularity of the framework – they are not relevant to proprietary products. Pycaret is another big project (7562 stars) that gives the option to train several models simultaneously and compare them with relatively low amounts of code. Projects like Nixtla's Statsforecast and MLForecast, or Darts have similar functionality specific to time series data.Libraries like Auto-ViML and deep-autoviml handle various types of variables and are built on scikit-learn and keras, respectively. They aim to make it easy for both novices and experts to experiment with different kinds of models and deep learning. However, users are advised to exercise their own judgement for accurate and explainable results.Important features of AutoML frameworks include the following:

- Deployment: Some solutions, especially those in the cloud, can be directly deployed to production. Others export to tensorflow or other formats.
- Types of data: Most solutions focus on tabular datasets; deep learning automl frameworks often work with different types of data. For example, autogluon facilitates rapid comparison and prototyping of ml solutions for images, text, time series in addition to tabular data. A few that focus on hyperparameter optimization such as optuna and ray tune, are totally agnostic to the format.
- Explainability: This can be very important depending on the industry, related to regulation (for example, healthcare or insurance) or reliability (finance). For a few solutions, this is a unique selling point.
- Monitoring: After deployment, the model performance can deteriorate (drift). A few providers provide monitoring of performance.
- Accessibility: Some providers require coding or at least basic data science understanding, others are turnkey solutions that require very little to no code. Typically, low and no-code solutions are less customizable.
- Open Source: The advantage of open-source platforms is that they are fully transparent about the implementation and the availability of methods and their parameters, and that they are fully extensible.
- Transfer Learning: This capability means being able to extend or customize existing foundation models.

There is a lot more to cover here that would go beyond the scope of this chapter such as the number of available methods. Less-well supported are features such as self-supervised learning, reinforcement learning, or generative image and audio models. For deep learning, a few libraries focus on the backend being specialized in Tensorflow, Pytorch, or MXNet. Auto-Keras, NASLib, and Ludwig have broader support, especially because they work with Keras. Starting with version 3.0, which is scheduled for release in fall 2023, Keras supports the three major backends TensorFlow, JAX, and PyTorch. Sklearn has its own hyperparameter optimization tools such as grid search, random search, successive halving. More specialized libraries such as auto-sklearn and hyperopt-sklearn go beyond this by offering methods for Bayesian Optimization. Optuna can integrate with a broad variety of ML frameworks such as AllenNLP, Catalyst, Catboost, Chainer, FastAI, Keras, LightGBM, MXNet, PyTorch, PyTorch Ignite, PyTorch, Lightning, TensorFlow, and XGBoost. Ray Tune comes with its own integrations among which is optuna. Both of them come with cutting edge parameter optimization algorithms and mechanisms for scaling (distributed training).In addition to the features listed above, some of these frameworks can automatically perform feature engineering tasks, such as data cleaning and feature selection, for example removing highly correlated features, and generating performance results graphically. Each of the listed tools has their own implementations for each step of the process such as feature selection and feature transformations – what differs is the extent to which this is automated. More specifically, the advantages of using AutoML frameworks include:

- Time savings: AutoML frameworks can save data scientists a lot of time by automating the process of machine learning model development.
- Improved accuracy: AutoML frameworks can help to improve the accuracy of machine learning models by automating the process of hyperparameter tuning.
- Increased accessibility: AutoML frameworks make machine learning more accessible to people who do not have a lot of experience with machine learning.

However, there are also some disadvantages to using AutoML frameworks:

- Black box: AutoML frameworks can be "black boxes," meaning that it can be difficult to understand how they work. This can make it difficult to debug problems with AutoML models.
- Limited flexibility: AutoML frameworks can be limited in terms of the types of machine learning tasks that they can automate.

A lot of the above tools have at least some kind of automatic feature engineering or preprocessing functionality, however, there are a few more specialized tools for this.

The impact of generative models

Generative AI and LLMs like GPT-3 have brought about significant changes to the field of data science and analysis. These models, particularly LLMs have the potential to revolutionize all steps involved in data science in a number of ways offering exciting opportunities for researchers and analysts. Generative AI models, such as ChatGPT, have the ability to understand and generate human-like responses, making them valuable tools for enhancing research productivity.Generative AI can play a crucial role in analyzing and interpreting research data. These models can assist in data exploration, uncover hidden patterns or correlations, and provide insights that may not be apparent through traditional methods. By automating certain aspects of data analysis, generative AI saves time and resources, allowing researchers to focus on higher-level tasks.Another area where generative AI can benefit researchers is in performing literature reviews and identifying research gaps. ChatGPT and similar models can summarize vast amounts of information from academic papers or articles, providing a concise overview of existing knowledge. This helps researchers identify gaps in the literature and guide their own investigations more efficiently. We've looked at this aspect of using generative AI models in *Chapter 4*, *Question Answering*.Other use cases for generative AI can be:

- Automatically generate synthetic data: Generative AI can be used to automatically generate synthetic data that can be used to train machine learning models. This can be helpful for businesses that do not have access to large amounts of real-world data.
- Identify patterns in data: Generative AI can be used to identify patterns in data that would not be visible to human analysts. This can be helpful for businesses that are looking to gain new insights from their data.

- Create new features from existing data: Generative AI can be used to create new features from existing data. This can be helpful for businesses that are looking to improve the accuracy of their machine learning models.

According to recent reports by the likes of McKinsey and KPMG, the consequences of AI relate to what data scientists will work on, how they will work, and who can work on data science tasks. The main areas of key impact include:

- Democratization of AI: Generative models allow many more people to leverage AI by generating text, code, and data from simple prompts. This expands the use of AI beyond data scientists.
- Increased productivity: By auto-generating code, data, and text, generative AI can accelerate development and analysis workflows. This allows data scientists and analysts to focus on higher-value tasks.
- Innovation in data science: Generative AI is bringing about is the ability to explore data in new and more creative ways, and generate new hypotheses and insights that would not have been possible with traditional methods
- Disruption of industries: New applications of generative AI could disrupt industries by automating tasks or enhancing products and services. Data teams will need to identify high-impact use cases.
- Limitations remain: Current models still have accuracy limitations, bias issues, and lack of controllability. Data experts are needed to oversee responsible development.
- Importance of governance: Rigorous governance over development and ethical use of generative AI models will be critical to maintaining stakeholder trust.
- Need for partnerships - Companies will need to build ecosystems with partners, communities and platform providers to effectively leverage generative AI capabilities.
- Changes to data science skills - Demand may shift from coding expertise to abilities in data governance, ethics, translating business problems, and overseeing AI systems.

Regarding democratization and innovation of data science, more specifically, generative AI is also having an impact on the way that data is visualized. In the past, data visualizations were often static and two-dimensional. However, generative AI can be used to create interactive and three-dimensional visualizations that can help to make data more accessible and understandable. This is making it easier for people to understand and interpret data, which can lead to better decision-making.Again, one of the biggest changes that generative AI is bringing about is the democratization of data science. In the past, data science was a very specialized field that required a deep understanding of statistics and machine learning. However, generative AI is making it possible for people with less technical expertise to create and use data models. This is opening up the field of data science to a much wider range of people.LLMs and generative AI can play a crucial role in automated data science by offering several benefits:

- Natural Language Interaction: LLMs allow for natural language interaction, enabling users to communicate with the model using plain English or other languages. This makes it easier for non-technical users to interact with and explore the data using everyday language, without requiring expertise in coding or data analysis.
- Code Generation: Generative AI can automatically generate code snippets to perform specific analysis tasks during EDA. For example, it can generate code to retrieve data (for example, SQL) clean data, handle missing values, or create visualizations (for example in Python). This feature saves time and reduces the need for manual coding.
- Automated Report Generation: LLMs can generate automated reports summarizing the key findings of EDA. These reports provide insights into various aspects of the dataset such as statistical summary, correlation analysis, feature importance, etc., making it easier for users to understand and present their findings.
- Data Exploration and Visualization: Generative AI algorithms can explore large datasets comprehensively and generate visualizations that reveal underlying patterns, relationships between variables, outliers or anomalies in the data automatically. This helps users gain a holistic understanding of the dataset without manually creating each visualization.

Further, we could think that generative AI algorithms should be able to learn from user interactions and adapt their recommendations based on individual preferences or past behaviors. They improve over time through continuous adaptive learning and user feedback, providing more personalized and useful insights during automated EDA.Finally, generative AI models can identify errors or anomalies in the data during EDA by learning patterns from existing datasets (Intelligent Error Identification). They can detect inconsistencies and highlight potential issues quickly and accurately.Overall, LLMs and generative AI can enhance automated EDA by simplifying user interaction, generating code snippets, identifying errors/ anomalies efficiently, automating report generation, facilitating comprehensive data exploration, visualization creation, and adapting to user preferences for more effective analysis of large and complex datasets.However, while these models offer immense potential to enhance research and aiding in literature review processes, they should not be treated as infallible sources. As we've seen earlier, LLMs work by analogy and struggle with reasoning and math. Their strength is creativity, not accuracy, and therefore, researchers must exercise critical thinking and ensure that the outputs generated by these models are accurate, unbiased, and aligned with rigorous scientific standards.One notable example is Microsoft's Fabric, which incorporates a chat interface powered by generative AI. This allows users to ask data-related questions using natural language and receive instant answers without having to wait in a data request queue. By leveraging LLMs like OpenAI models, Fabric enables real-time access to valuable insights.Fabric stands out among other analytics products due to its comprehensive approach. It addresses various aspects of an organization's analytics needs and provides role-specific experiences for different teams involved in the analytics process, such as data engineers, warehousing professionals, scientists, analysts, and business users.With the integration of Azure OpenAI Service at every layer, Fabric harnesses generative AI's power to unlock the full potential of data. Features like Copilot in Microsoft Fabric provide conversational language experiences, allowing users to create dataflows, generate code or entire functions, build machine learning models, visualize results, and even develop custom conversational language experiences.Anecdotally, ChatGPT is (and Fabric in extension) often produces incorrect SQL queries. This is fine when used by analysts who can check the validity of the output, but a total disaster as a self-service analytics tool for non-technical business users. Therefore, organizations must ensure that they have reliable data pipelines in place and employ data quality management practices while using Fabric for analysis.While the possibilities of generative AI in data analytics are promising, caution must be exercised. The reliability and accuracy of LLMs should be verified using first-principled reasoning and rigorous analysis. While these models have shown their potential in ad-hoc analysis, idea generation during research, and summarizing complex analyses, they may not always be suitable for self-service analytical tools for non-technical users due to the need for validation by domain experts.Let's start to use agents to run code or call other tools to answer questions!

Agents can answer data science questions

As we've seen with Jupyter AI (Jupyternaut chat) – and in chapter 6, *Developing Software* – there's a lot of potential to increase efficiency creating and creating software with generative AI (code LLMs). This is a good starting point for the practical part of this chapter as we look into the use of generative AI in data science.We've already seen different agents with tools before. For example, the LLMMathChain can execute Python to answer math questions as illustrated here:

```
from langchain import OpenAI, LLMMathChain
llm = OpenAI(temperature=0)
llm_math = LLMMathChain.from_llm(llm, verbose=True)
llm_math.run("What is 2 raised to the 10th power?")
```

While this is useful to extract information and feed it back, it's less obvious to see how to plug this into a traditional EDA process. Similarly, the CPAL (`CPALChain`) and PAL (`PALChain`) chains can answer more complex reasoning questions while keeping hallucinations in check, but it's hard to come up with real-life use cases for them.With the `PythonREPLTool` we can create simple visualizations of toy data or train with synthetic data, which can be nice for illustration or bootstrapping of a project. This is an example from the LangChain documentation:

```
from langchain.agents.agent_toolkits import create_python_agent
from langchain.tools.python.tool import PythonREPLTool
from langchain.llms.openai import OpenAI
from langchain.agents.agent_types import AgentType
agent_executor = create_python_agent(
    llm=OpenAI(temperature=0, max_tokens=1000),
    tool=PythonREPLTool(),
    verbose=True,
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
)
agent_executor.run(
    """Understand, write a single neuron neural network in PyTorch.
Take synthetic data for y=2x. Train for 1000 epochs and print every 100 epochs.
Return prediction for x = 5"""
)
```

Please note that this should be executed with caution since the Python code is executed directly on the machine without any safeguards in place. This actually works, and creates a dataset, trains a model, and we get a prediction back:

```
Entering new AgentExecutor chain...
I need to write a neural network in PyTorch and train it on the given data
Action: Python_REPL
Action Input:
import torch
model = torch.nn.Sequential(
    torch.nn.Linear(1, 1)
)
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
# Define the data
x_data = torch.tensor([[1.0], [2.0], [3.0], [4.0]])
y_data = torch.tensor([[2.0], [4.0], [6.0], [8.0]])
for epoch in range(1000):  # Train the model
    y_pred = model(x_data)
    loss = loss_fn(y_pred, y_data)
    if (epoch+1) % 100 == 0:
        print(f'Epoch {epoch+1}: {loss.item():.4f}')
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
# Make a prediction
x_pred = torch.tensor([[5.0]])
y_pred = model(x_pred)
Observation: Epoch 100: 0.0043
Epoch 200: 0.0023
Epoch 300: 0.0013
Epoch 400: 0.0007
Epoch 500: 0.0004
Epoch 600: 0.0002
Epoch 700: 0.0001
Epoch 800: 0.0001
Epoch 900: 0.0000
Epoch 1000: 0.0000
Thought: I now know the final answer
Final Answer: The prediction for x = 5 is y = 10.00.
```

Again, this is very cool, but it's hard to see how that would scale up without more serious engineering similar to what we did in chapter 6, *Developing Software*.LLMs and tools can be useful if we want to enrich our data with category or geographic information. For example, if our company offers flights from Tokyo, and we want to know the distances of our customers from Tokyo, we can use Wolfram Alpha as a tool. Here's a simplistic example:

```
from langchain.agents import load_tools, initialize_agent
from langchain.llms import OpenAI
from langchain.chains.conversation.memory import ConversationBufferMemory
llm = OpenAI(temperature=0)
tools = load_tools(['wolfram-alpha'])
memory = ConversationBufferMemory(memory_key="chat_history")
agent = initialize_agent(tools, llm, agent="conversational-react-description", memory=memory,
agent.run(
    """How far are these cities to Tokyo?
* New York City
* Madrid, Spain
* Berlin
""")
```

Please make sure you've set the OPENAI_API_KEY and WOLFRAM_ALPHA_APPID environment variables as discussed in chapter 3, *Getting Started with LangChain*. Here's the output:

```
> Entering new AgentExecutor chain...
AI: The distance from New York City to Tokyo is 6760 miles. The distance from Madrid, Spain t
> Finished chain.
'
The distance from New York City to Tokyo is 6760 miles. The distance from Madrid, Spain to To
```

Now, a lot of these questions are very simple. However, we can give agents datasets to work with and here's where it can get very powerful when we connect more tools. Let's start with asking and answering questions about structured datasets!

Data exploration with LLMs

Data exploration is a crucial and foundational step in data analysis, allowing researchers to gain a comprehensive understanding of their datasets and uncover significant insights. With the emergence of LLMs like ChatGPT, researchers can harness the power of natural language processing to facilitate data exploration.As we've mentioned earlier Generative AI models, such as ChatGPT, have the ability to understand and generate human-like responses, making them valuable tools for enhancing research productivity. Asking our questions in natural language and getting responses in digestible pieces and shape can be a great boost to analysis.LLMs can assist in exploring not only textual data but also other forms of data such as numerical datasets or multimedia content. Researchers can leverage ChatGPT's capabilities to ask questions about statistical trends in numerical datasets or even query visualizations for image classification tasks.Let's load up a dataset and work with that. We can quickly get a dataset from scikit-learn:

```
from sklearn.datasets import load_iris
df = load_iris(as_frame=True)["data"]
```

The Iris dataset is well-known – it's a toy dataset, but it will help us illustrate the capabilities of using generative AI for data exploration. We'll use the DataFrame in the following. We can create a Pandas dataframe agent now and we'll see how easy it is to get simple stuff done!

```
from langchain.agents import create_pandas_dataframe_agent
from langchain import PromptTemplate
from langchain.llms.openai import OpenAI
PROMPT = (
    "If you do not know the answer, say you don't know.\n"
    "Think step by step.\n"
    "\n"
    "Below is the query.\n"
    "Query: {query}\n"
)
prompt = PromptTemplate(template=PROMPT, input_variables=["query"])
llm = OpenAI()
agent = create_pandas_dataframe_agent(llm, df, verbose=True)
```

I've put the instruction for the model to say it doesn't know when in doubt and thinking step by step, both to reduce hallucinations. Now we can query our agent against the DataFrame:

```
agent.run(prompt.format(query="What's this dataset about?"))
```

We get the answer `'This dataset is about the measurements of some type of flower.'` which is correct. Let's show how to get a visualization:

```
agent.run(prompt.format(query="Plot each column as a barplot!"))
```

It's not perfect, but we are getting a nice-looking plot:

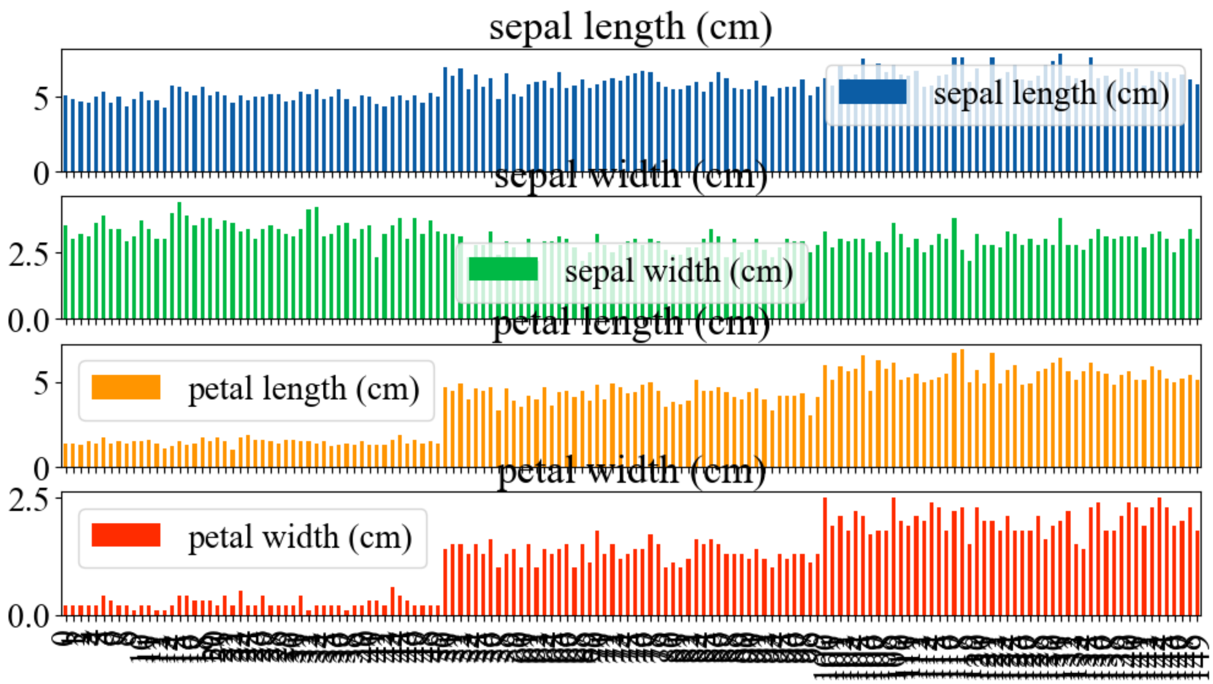


Figure 7.5: Iris dataset barplots.

We can also ask to see the distributions of the columns visually, which will give us this neat plot:

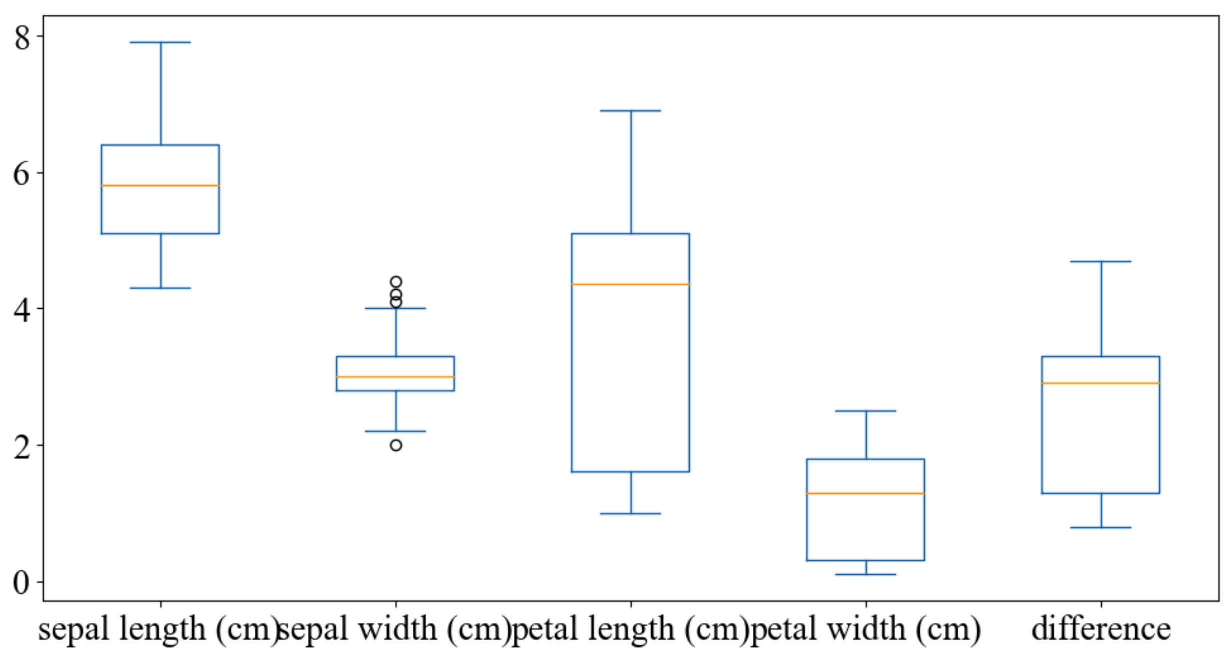


Figure 7.6: Iris dataset boxplots.

We can request the plot to use other plotting backends such as seaborn, however, please note that these have to be installed. We can also ask more questions about the dataset like which row has the biggest difference between petal length and petal width. We get the answer with the intermediate steps as follows (shortened):

```
df['difference'] = df['petal length (cm)'] - df['petal width (cm)']
df.loc[df['difference'].idxmax()]
Observation: sepal length (cm)    7.7
sepal width (cm)     2.8
petal length (cm)    6.7
petal width (cm)     2.0
difference           4.7
Name: 122, dtype: float64
Thought: I now know the final answer
Final Answer: Row 122 has the biggest difference between petal length and petal width.
```

I think that's worth a pat on the back, LLM!Next steps could be adding more instructions to the prompt about plotting such about the sizes of plots. It's a bit harder to implement the same plotting logic in a streamlit app, because we need to use the plotting functionality in corresponding streamlit functions, for example, `st.bar_chart()` , however, this can be done as well. You can find explanations for this on the Streamlit blog ("Building a Streamlit and scikit-learn app with ChatGPT").What about statistical tests?

```
agent.run(prompt.format(query="Validate the following hypothesis statistically: petal width a
```

We get this response:

```
Thought: I should use a statistical test to answer this question.
Action: python_repl_ast
Action Input: from scipy.stats import ks_2samp
Observation:
Thought: I now have the necessary tools to answer this question.
Action: python_repl_ast
Action Input: ks_2samp(df['petal width (cm)'], df['petal length (cm)'])
Observation: KstestResult(statistic=0.6666666666666666, pvalue=6.639808432803654e-32, statist
Thought: I now know the final answer
Final Answer: The p-value of 6.639808432803654e-32 indicates that the two variables come from
```

'The p-value of 6.639808432803654e-32 indicates that the two variables come from different distributions.'That's check for statistical test! That's cool. We can ask fairly complex questions about the dataset with simple prompts in plain English. There's also the pandas-ai library, which uses LangChain under the hood and provides similar functionality. Here's an example from the documentation with an example dataset:

```
import pandas as pd
from pandasai import PandasAI
df = pd.DataFrame({
    "country": ["United States", "United Kingdom", "France", "Germany", "Italy", "Spain", "Ca
    "gdp": [19294482071552, 2891615567872, 2411255037952, 3435817336832, 1745433788416, 11812
    "happiness_index": [6.94, 7.16, 6.66, 7.07, 6.38, 6.4, 7.23, 7.22, 5.87, 5.12]
})
from pandasai.llm.openai import OpenAI
llm = OpenAI(api_token="YOUR_API_TOKEN")
pandas_ai = PandasAI(llm)
pandas_ai(df, prompt='Which are the 5 happiest countries?')
```

This will give us the requested result similarly to before when we were using LangChain directly. Please note that pandas-ai is not part of the setup for the book, so you'll have to install it separately if you want to use it.For data in SQL-databases, we can connect with a `SQLDatabaseChain` . The LangChain documentation shows this example:

```
from langchain.llms import OpenAI
from langchain.utilities import SQLDatabase
from langchain_experimental.sql import SQLDatabaseChain
db = SQLDatabase.from_uri("sqlite:///../../../../notebooks/Chinook.db")
llm = OpenAI(temperature=0, verbose=True)
db_chain = SQLDatabaseChain.from_llm(llm, db, verbose=True)
db_chain.run("How many employees are there?")
```

We are connecting to a database first. Then we can ask questions about the data in natural language. This can also be quite powerful. An LLM will create the queries for us. I would expect this to be particularly useful when we don't know about the schema of the database.The `SQLDatabaseChain` can also check queries and autocorrect them if the `use_query_checker` option is set.Let's summarize!

Summary

In this chapter, we've explored the state-of-the-art in automated data analysis and data science. There are quite a few areas, where LLMs can benefit data science, mostly as coding assistants or in data exploration.We've started off with an overview over frameworks that cover each step in the data science process such as AutoML methods, and we've discussed how LLMs can help us further increasing productivity and making data science and data analysis more accessible, both to stakeholders and to developers or users. We've then investigated how code generation and tools, similar to code LLMs *Chapter 6*, *Developing Software*, can help in data science tasks by creating functions or models that we can query, or how we can enrich data using LLMs or third-party tools like Wolfram Alpha.We then had a look at using LLMs in data exploration. In *Chapter 4*, *Question Answering*, we looked at ingesting large amounts of textual data for analysis. In this chapter, we focused on exploratory analysis of structured datasets in SQL or tabular form. In conclusion, AI technology has the potential to revolutionize the way we can analyze data, and ChatGPT plugins or Microsoft Fabric are examples of this. However, at the current state-of-affairs, AI can't replace data scientists, only help enable them.Let's see if you remember some of the key takeaways from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions from memory. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. What's the difference between data science and data analysis?
2. What steps are involved in data science?
3. Why would we want to automate data science/analysis?
4. What frameworks exist for automating data science tasks and what can they do?
5. How can generative AI help data scientists?
6. What kind of agents and tools can we use to answer simple questions?
7. How can we get an LLM to work with data?

2023/11/02

8 Customizing LLMs and their output

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

This chapter is about techniques and best practices to improve the reliability and performance of large language models (LLMs) in certain scenarios such as on complex reasoning and problem-solving tasks. Generally, this process of adapting a model for a certain task or making sure that our model output corresponds to what we expect is called conditioning. In this chapter, we'll discuss fine-tuning and prompting as methods for conditioning.Fine-tuning involves training the pre-trained base model on specific tasks or datasets relevant to the desired application. This process allows the model to adapt and become more accurate and contextually aligned for the intended use case. Similarly, by providing additional input or context at inference time, large language models (LLMs) can generate text tailored to a particular tasks or style.Prompt design is highly significant for unlocking LLM reasoning capabilities, the potential for future advancements in models and prompting techniques, and these principles and techniques form a valuable toolkit for researchers and practitioners working with large language models. Understanding how LLMs generate text token-by-token helps create better reasoning prompts.Prompting is still an empirical art - trying variations to see what works is often needed. But some prompt engineering insights can transfer across models and tasks. We'll discuss the tools in LangChain to enable advanced prompt engineering strategies like few-shot learning, dynamic example selection, and chained reasoning.Throughout the chapter, we'll work on fine-tuning and prompting with LLMs, which you can find in the `notebooks` directory in the Github repository for the book at https://github.com/benman1/generative_ai_with_langchainThe main sections are:

- Conditioning and Alignment
- Fine-Tuning
- Prompt Engineering

Let's start by discussing conditioning and alignment, why it's important, and how we can achieve it.

Conditioning and alignment

Alignment, in the context of generative AI models, refers to ensuring that the outputs of these models are consistent with human values, intentions, or desired outcomes. It involves guiding the model's behavior to be in line with what is considered ethical, appropriate, or relevant within a given context. The concept of alignment is crucial to avoid generating outputs that may be biased, harmful, or deviate from the intended purpose. Addressing alignment involves careful attention to the biases present in training data, iterative feedback loops involving human reviewers, refining objective functions during training/fine-tuning stages, leveraging user feedback, and continuous monitoring during deployment to ensure ongoing alignment.There are several reasons one may want to condition a large language model. The first is to control the content and style of the outputs. For example, conditioning on certain keywords or attributes like formality level can produce more relevant and high-quality text. Conditioning also encompasses safety measures to prevent the generation of malicious or harmful content. For example, avoiding generating misleading information, inappropriate suggestions, or potentially dangerous instructions, or – more generally aligning the model with certain values.The potential benefits of conditioning large language models are numerous. By providing more specific and relevant input, we can achieve outputs that are tailored to our needs. For instance, in a customer support chatbot, conditioning the model with user queries allows it to generate responses that address their concerns accurately. Conditioning also helps control biased or inappropriate outputs by constraining the model's creativity within specific boundaries.Furthermore, by conditioning large language models, we can make them more controllable and adaptable. We can fine-tune and shape their behavior according to our requirements and create AI systems that are reliable in specific domains such as legal advice or technical writing.However, there are potential downsides to consider as well. Conditioning models too heavily might result in overfitting, where they become excessively reliant on specific inputs and struggle with generating creative or diverse outputs in different contexts. Moreover, conditioning should be utilized responsibly since large language models have the tendency to amplify biases present in the training data. Care must be taken not to exacerbate issues related to bias or controversial topics when conditioning these models.

Benefits of alignment include:

- Enhanced User Experience: Aligned models generate outputs that are relevant to user queries or prompts.
- Trust-building: Ensuring ethical behavior helps build trust among users/customers.
- Brand Reputation: By aligning with business goals regarding branding consistency and desired tone/style guidelines.

- Mitigating Harmful Effects: Alignment with safety, security, and privacy considerations helps prevent the generation of harmful or malicious content.

Potential downsides include:

- Challenging Balance: Striking a balance between extreme alignment (overly conservative) and creative freedom (overly permissive) can be difficult.
- Limitations of Automated Metrics: Quantitative evaluation metrics might not capture alignment nuances fully.
- Subjectivity: Alignment judgments are often subjective, requiring careful consideration and consensus building on desired values and guidelines.

Pre-training a large model on diverse data to learn patterns and language understanding results in a base model that has a broad understanding of various topics but lacks specificity or alignment to any particular context. While base models such as GPT-4 are capable of generating impressive text on a wide range of topics, conditioning them can enhance their capabilities in terms of task relevance, specificity, and coherence, and make their outputs more relevant and on-topic. Without conditioning, these models tend to generate text that may not always align perfectly with the desired context. By conditioning them, we can guide the language models to produce outputs that are more closely related to the given input or instructions. The major advantage of conditioning is that it allows guiding the model without extensive retraining. It also enables interactive control and switching between modes. Conditioning can happen at different stages of the model development cycle—from fine-tuning to output generation in various contexts. There are several options for achieving alignment of large language models. One approach is to condition during fine-tuning, by training the model on a dataset reflective of the desired outputs. This allows the model to specialize, but requires access to relevant training data. Another option is to dynamically condition the model at inference time by providing conditional input along with the main prompt. This is more flexible but introduces some complexity during deployment.In the next section, I will summarize key methods for alignment such as fine-tuning and prompt engineering, discuss the rationale, and examine their relative pros and cons.

Methods for alignment

With the advent of large pre-trained language models like GPT-3, there has been growing interest in techniques to adapt these models for downstream tasks. This process is known as fine-tuning. Fine-tuning allows pre-trained models to be customized for specific applications while leveraging the vast linguistic knowledge acquired during pre-training. The idea of adapting pre-trained neural networks originated in computer vision research in the early 2010s. In NLP, Howard and Ruder (2018) demonstrated the effectiveness of fine-tuning pre-trained contextual representations like ELMo and ULMFit on downstream tasks. The seminal BERT model (Devlin and others., 2019) established fine-tuning of pre-trained transformers as the de facto approach in NLP.The need for fine-tuning arises because pre-trained LMs are designed to model general linguistic knowledge, not specific downstream tasks. Their capabilities manifest only when adapted to particular applications. Fine-tuning allows pre-trained weights to be updated for target datasets and objectives. This enables knowledge transfer from the general model while customizing it for specialized tasks. Several approaches have been proposed for alignment, with trade-offs in efficacy and efficiency and it's worth to delve a bit more into the details of each of these alignment methods. **Full**

Fine-Tuning involves updating all the parameters of the pre-trained language model during fine-tuning. The model is trained end-to-end on the downstream tasks, allowing the weights to be updated globally to maximize performance on the target objectives. FFT consistently achieves strong results across tasks but requires extensive computational resources and large datasets to avoid overfitting or forgetting.In **Adapter Tuning** additional trainable adapter layers are inserted, usually bottleneck layers, into the pre-trained model while keeping the original weights frozen. Only the newly added adapter layers are trained on the downstream tasks. This makes tuning parameter-efficient as only a small fraction of weights are updated. However, as the pre-trained weights remain fixed, adapter tuning has a risk of underfitting to the tasks. The insertion points and capacity of the adapters impact overall efficacy.**Prefix Tuning**: This prepends trainable vectors to each layer of the LM, which are optimized during fine-tuning while the base weights remain frozen. The prefixes allow injection of inductive biases into the model. Prefix tuning has a smaller memory footprint compared to adapters but has not been found to be as effective. The length and initialization of prefixes impact efficacy.In **Prompt Tuning**, the input text is appended with trainable prompt tokens which provide a soft prompt to induce the desired behavior from the LM. For example, a task description can be provided as a prompt to steer the model. Only the added prompt tokens are updated during training while the pre-trained weights are frozen. Performance is heavily influenced by prompt engineering. Automated prompting methods are being explored.**Low-Rank Adaptation (LoRA)** adds pairs of low-rank trainable weight matrices to the frozen LM weights. For example, to each weight W, low-rank matrices B and A are added such that the forward pass uses W + BA. Only B and A are trained, keeping the base W frozen. LoRA achieves reasonable efficacy with greater parameter efficiency than full tuning. The choice of rank r impacts tradeoffs. LoRA enables tuning giant LMs on limited hardware.Another way to ensure proper alignment of outputs is through **human oversight** methods like human-in-the-loop systems. These systems involve human reviewers who provide feedback and make corrections if necessary. Human involvement helps align generated outputs with desired values or guidelines set by humans.Here is a table summarizing the different techniques for steering generative AI outputs:

Stage	**Technique**	**Examples**
Training	Pre-training	Training on diverse data
	Objective Function	Careful design of training objective
	Architecture and Training Process	Optimizing model structure and training
Fine-Tuning	Specialization	Training on specific datasets/tasks
Inference-Time Conditioning	Dynamic Inputs	Prefixes, control codes, context examples
Human Oversight	Human-in-the-Loop	Human review and feedback

Figure 8.1: Steering generative AI outputs.

Combining these techniques provides developers with more control over the behavior and outputs of generative AI systems. The ultimate goal is to ensure that human values are incorporated at all stages, from training to deployment, in order to create responsible and aligned AI systems.Furthermore, careful design choices in the pre-training objective function also impact what behaviors and patterns the language model learns initially. By incorporating ethical considerations into these objective functions, developers can influence the initial learning process of large language models.We can distinguish a few more approaches in fine-tuning such as online and offline. InstructGPT was considered a game-changer because it demonstrated the potential to significantly improve language models, such as GPT-3, by incorporating reinforcement learning from human feedback (RLHF). Let's talk about the reasons why InstructGPT had such a transformative impact.

Reinforcement learning with human feedback

In their March 2022 paper, Ouyang and others from OpenAI demonstrated using reinforcement learning from human feedback (RLHF) with proximal policy optimization (PPO) to align large language models like GPT-3 with human preferences. Reinforcement learning from human feedback (RLHF) is an online approach that fine-tunes LMs using human preferences. It has three main steps:

1. Supervised pre-training: The LM is first trained via standard supervised learning on human demonstrations.
2. Reward model training: A reward model is trained on human ratings of LM outputs to estimate reward.
3. RL fine-tuning: The LM is fine-tuned via reinforcement learning to maximize expected reward from the reward model using an algorithm like PPO.

The main change, RLHF, allows incorporating nuanced human judgments into language model training through a learned reward model. As a result, human feedback can steer and improve language model capabilities beyond standard supervised fine-tuning. This new model can be used to follow instructions that are given in natural language, and it can answer questions in a way that's more accurate and relevant than GPT-3. InstructGPT outperformed GPT-3 on user preference, truthfulness, and harm reduction, despite having 100x fewer parameters.Starting in March 2022, OpenAI started releasing the GPT-3.5 series models, upgraded versions of GPT-3, which include fine-tuning with RLHF.There are three advantages of fine-tuning that were immediately obvious to users of these models:

1. Steerability: the capability of models to follow instructions (instruction-tuning)
2. Reliable output-formatting: this became important, for example, for API calls/function calling)
3. Custom tone: this makes it possible to adapt the output style as appropriate to task and audience.

InstructGPT opened up new avenues for improving language models by incorporating reinforcement learning from human feedback methods beyond traditional fine-tuning approaches. RL training can be unstable and computationally expensive, notwithstanding, its success inspired further research into refining RLHF techniques, reducing data requirements for alignment, and developing more powerful and accessible models for a wide range of applications.

Offline Approaches

Offline methods circumvent the complexity of online RL by directly utilizing human feedback. We can distinguish between ranking-based on language-based approaches:

- Ranking-based: Human rankings of LM outputs are used to define optimization objectives for fine-tuning, avoiding RL entirely. This includes methods like Preference Ranking Optimization (PRO; Song et al., 2023) and Direct Preference Optimization (DPO; Rafailov et al., 2023).
- Language-based: Human feedback is provided in natural language and utilized via standard supervised learning. For example, Chain of Hindsight (CoH; Liu et al., 2023) converts all types of feedback into sentences and uses them to fine-tune the model, taking advantage of the language comprehension capabilities of language models.

Direct Preference Optimization (DPO) is a simple and effective method for training language models to adhere to human preferences without needing to explicitly learn a reward model or use reinforcement learning. While it optimizes the same objective as existing RLHF methods, it is much simpler to implement, more stable, and achieves strong empirical performance.Researchers from Meta, in the paper "*LIMA: Less Is More for Alignment*", simplified alignment by minimizing a supervised loss on only 1,000 carefully curated prompts in fine-training the LLaMa model. Based on the favorable human preferences when comparing their outputs to DaVinci003 (GPT-3.5), they conclude that fine-training has only minimal importance. This they refer to as the superficial alignment hypothesis.Offline approaches offer more stable and efficient tuning. However, they are limited by static human feedback. Recent methods try to combine offline and online learning.While both DPO and RLHF with PPO aim to align LLMs with human preferences, they differ in terms of complexity, data requirements, and implementation details. DPO offers simplicity but achieves strong performance by directly optimizing probability ratios. On the other hand, RLHF with PPO in InstructGPT introduces more complexity but allows for nuanced alignment through reward modeling and reinforcement learning optimization.

Low-Rank Adaptation ?

LLMs have achieved impressive results in Natural Language Processing and are now being used in other domains such as Computer Vision and Audio. However, as these models become larger, it becomes difficult to train them on consumer hardware and deploying them for each specific task becomes expensive. There are a few methods that reduce computational, memory, and storage costs, while improving performance in low-data and out-of-domain scenarios.Low-Rank Adaptation (LoRA) freezes the pre-trained model weights and introduces trainable rank decomposition matrices into each layer of the Transformer architecture to reduce the number of trainable parameters. LoRA achieves comparable or better model quality compared to fine-tuning on various language models (RoBERTa, DeBERTa, GPT-2, and GPT-3) while having fewer trainable parameters and higher training throughput.The QLORA method is an extension of LoRA, which enables efficient fine-tuning of large models by backpropagating gradients through a frozen 4-bit quantized model into learnable low-rank adapters. This allows fine-tuning a 65B parameter model on a single GPU. QLORA models achieve 99% of ChatGPT performance on Vicuna using innovations like new data types and optimizers. In particular, QLORA reduces the memory requirements for fine-tuning a 65B parameter model from >780GB to <48GB without affecting runtime or predictive performance.

Quantization refers to techniques for reducing the numerical precision of weights and activations in neural networks like large language models (LLMs). The main purpose of quantization is to reduce the memory footprint and computational requirements of large models.

Some key points about quantization of LLMs:

- It involves representing weights and activations using fewer bits than standard single-precision floating point (FP32). For example, weights could be quantized to 8-bit integers.
- This allows shrinking model size by up to 4x and improving throughput on specialized hardware.
- Quantization typically has a minor impact on model accuracy, especially with re-training.
- Common quantization methods include scalar, vector, and product quantization which quantize weights separately or in groups.
- Activations can also be quantized by estimating their distribution and binning appropriately.
- Quantization-aware training adjusts weights during training to minimize quantization loss.
- LLMs like BERT and GPT-3 have been shown to work well with 4-8 bit quantization via fine-tuning.

Parameter-Efficient Fine-tuning (PEFT) methods enable the use of small checkpoints for each task, making the models more portable. These small trained weights can be added on top of the LLM, allowing the same model to be used for multiple tasks without replacing the entire model.In the next section, we'll discuss methods for conditioning large language models (LLMs) at inference time.

Inference-Time conditioning

One commonly used approach is **conditioning at inference time** (output generation phase) where specific inputs or conditions are provided dynamically to guide the output generation process. LLM fine-tuning may not always be feasible or beneficial in certain scenarios:

1. Limited Fine-Tuning Services: Some models are only accessible through APIs that lack or have restricted fine-tuning capabilities.
2. Insufficient Data: In cases where there is a lack of data for fine-tuning, either for the specific downstream task or relevant application domain.
3. Dynamic Data: Applications with frequently changing data, such as news-related platforms, may struggle to fine-tune models frequently, leading to potential drawbacks.
4. Context-Sensitive Applications: Dynamic and context-specific applications like personalized chatbots cannot perform fine-tuning based on individual user data.

For conditioning at inference time, most commonly, we provide a textual prompt or instruction at the beginning of the text generation process. This prompt can be a few sentences or even a single word, acting as an explicit indication of the desired output. Some common techniques for dynamic inference-time conditioning include:

- Prompt tuning: Providing natural language guidance for intended behavior. Sensitive to prompt design.
- Prefix tuning: Prepending trainable vectors to LLM layers.
- Constraining tokens: Forcing inclusion/exclusion of certain words
- Metadata: Providing high-level info like genre, target audience, etc.

Prompts can facilitate generating text that adheres to specific themes, styles, or even mimics a particular author's writing style. These techniques involve providing contextual information during inference time such as for in-context learning or retrieval augmentation.An example of prompt tuning is prefixing prompts, where instructions like "Write a child-friendly story about..." are prepended to the prompt. For example, in chatbot applications, conditioning the model with user messages helps it generate responses that are personalized and pertinent to the ongoing conversation. Further examples include prepending relevant documents to prompts to assist LLMs with writing tasks (e.g., news reports, Wikipedia pages, company documents), or retrieving and prepending user-specific data (financial records, health data, emails) before prompting an LLM to ensure personalized answers. By conditioning LLM outputs on contextual information at runtime, these methods can guide models without relying on traditional fine-tuning processes.Often demonstrations are part of the instructions for reasoning tasks, where few-shot examples are provided to induce desired behavior. Powerful LLMs, such as GPT-3, can solve tasks without further training through prompting techniques. In this approach, the problem to be solved is presented to the model as a text prompt, possibly with some text examples of similar problems and their solutions. The model must provide a completion of the prompt via inference. **Zero-shot prompting** involves no solved examples, while few-shot prompting includes a small number of examples of similar (problem, solution) pairs. It has shown that prompting provides easy control over large frozen models like GPT-3 and allows steering model behavior without extensive fine-tuning. Prompting enables conditioning models on new knowledge with low overhead, but careful prompt engineering is needed for best results. This is what we'll discuss as part of this chapter.In prefix tuning, continuous task-specific vectors are trained and supplied to models at inference time. continuous task-specific vectors. Similar ideas have been proposed for adapter-approaches such as parameter efficient transfer learning (PELT) or Ladder Side-Tuning (LST).Conditioning at inference time can also happen during sampling such as grammar-based sampling, where the output can be constrained to be compatible with certain well-defined patterns, such as a programming language syntax.

Conclusions

Full fine-tuning consistently achieves strong results but often requires extensive resources, and trade-offs exist between efficacy and efficiency. Methods like adapters, prompts, and LoRA reduce this burden via sparsity or freezing, but can be less effective. The optimal approach depends on constraints and objectives. Future work on improved techniques tailor-made for large LMs could push the boundaries of both efficacy and efficiency. Recent work blends offline and online learning for improved stability. Integrating world knowledge and controllable generation remain open challenges.Prompt-based techniques allow flexible conditioning of LLMs to induce desired behaviors without intensive training. Careful prompt design, optimization, and evaluation is key to effectively controlling LLMs. Prompt-based techniques allow conditioning LLMs on specific behaviors or knowledge in a flexible, low-resource manner.

Evaluations

Alignment is evaluated on alignment benchmarks like HUMAN and generalization tests like FLAN. There are a few core benchmarks with high differentiability to accurately assess model strengths and weaknesses such as these:

- English knowledge: MMLU
- Chinese knowledge: C-Eval
- Reasoning: GSM8k / BBH (Algorithmic)
- Coding: HumanEval / MBPP

After balancing these directions, additional benchmarks like MATH (high-difficulty reasoning) and Dialog could be pursued.A particularly interesting evaluation is in math or reasoning, where generalization abilities would be expected to be very strong. The MATH benchmark demonstrates high-level difficulty, and GPT-4 achieves varying scores based on prompting methods. Results range from naive prompting via few-shot evaluations to PPO + process-based reward modeling. If fine-tuning involves dialog data only, it might negatively affect existing capabilities such as MMLU or BBH. Prompt engineering is essential, as biases and query difficulty impact evaluations.There are quantitative metrics like perplexity (measuring how well a model predicts data) or BLEU score (capturing similarity between generated text and reference text). These metrics provide rough estimates but may not fully capture semantic meaning or alignment with higher-level goalsOther metrics include user preferences ratings through human evaluation, pairwise preference, utilizing pre-trained reward model for online small/medium models or automated LLM-based assessments (for example GPT-4). Human evaluations can sometimes be problematic since humans can be swayed by subjective criteria such as an authoritative tone in the response rather than the actual accuracy. Conducting evaluations where users assess the quality, relevance, appropriateness of generated text against specific criteria set beforehand provides more nuanced insights into alignment. Fine-tuning is not intended to solely improve user preferences on a given set of prompts. Its primary purpose is to address AI safety concerns by reducing the occurrence of undesirable outputs such as illegal, harmful, abusive, false, or deceptive content. This focus on mitigating risky behavior is crucial in ensuring the safety and reliability of AI systems. Evaluating and comparing models based purely on user preferences without considering the potential harm they may cause can be misleading and prioritize suboptimal models over safer alternatives. In summary, evaluating LLM alignment requires careful benchmark selection, consideration of differentiability, and a mix of automatic evaluation methods and human judgments. Attention to prompt engineering and specific evaluation aspects is necessary to ensure accurate assessment of model performance.In the next section, we'll fine-tune a small open-source LLM (OpenLLaMa) for a question answering with PEFT and quantization, and we'll deploy it on HuggingFace.

Fine-Tuning

As we've discussed in the first section of this chapter, the goal of model fine-tuning for LLMs is to optimize a model to generate outputs that are more specific to a task and context than the original foundation model. Amongst the multitude of tasks and scenarios, where we might want to apply this approach are these:

- Software Development
- Document classification
- Question-Answering
- Information Retrieval
- Customer Support

In this section, we'll fine-tune a model for question answering. This recipe is not specific to LangChain, but we'll point out a few customizations, where LangChain could come in. For performance reasons, we'll run this on Google Colab instead of the usual local environment.

> **Google Colab** is a computation environment that provides different means for hardware acceleration of computation tasks such as Tensor Processing Units (TPUs) and Graphical Processing Units (GPUs). These are available both in free and professional tiers. For the purpose of the task in this section, the free tier is completely sufficient. You can sign into a Colab environment at this url: https://colab.research.google.com/

Please make sure you set your google colab machine settings in the top menu to TPU or GPU in order to make sure you have sufficient resources to run this and that the training doesn't take too long. We'll install all required libraries in the Google Colab environment – I am adding the versions of these libraries that I've used in order to make our fine-tuning repeatable:

- peft: Parameter-Efficient Fine-Tuning (PEFT; version 0.5.0)
- trl: Proximal Policy Optimization (0.6.0)
- bitsandbytes: k-bit optimizers and matrix multiplication routines, needed for quantization (0.41.1)
- accelerate: train and use PyTorch models with multi-GPU, TPU, mixed-precision (0.22.0)
- transformers: HuggingFace transformers library with backends in JAX, PyTorch and TensorFlow (4.32.0)
- datasets: community-driven open-source library of datasets (2.14.4)
- sentencepiece: Python wrapper for fast tokenization (0.1.99)
- wandb: for monitoring the training progress on Weights and Biases (0.15.8)
- langchain for loading the model back as a langchain llm after training (0.0.273)

We can install these libraries from the Colab notebook as follows:

```
!pip install -U accelerate bitsandbytes datasets transformers peft trl sentencepiece wandb la
```

In order to download and train models from HuggingFace, we need to authenticate with the platform. Please note that if you want to push your model to HuggingFace later, you need to generate a new API token with write permissions on HuggingFace: https://huggingface.co/settings/tokens

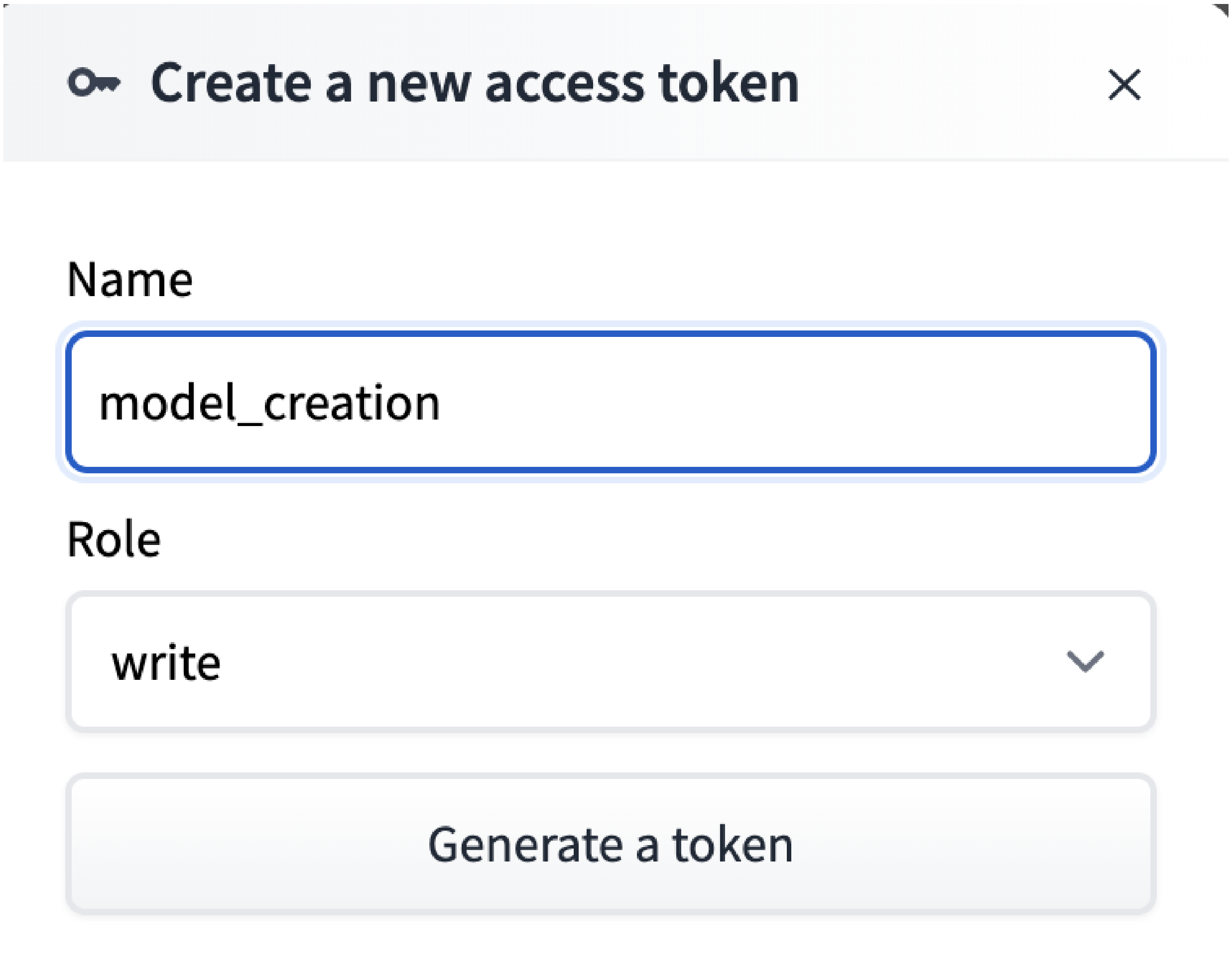


Figure 8.3: Creating a new API token on HuggingFace write permissions.

We can authenticate from the notebook like this:

```
import notebook_login
notebook_login()
```

When prompted, paste your HuggingFace access token.

> A note of caution before, we start: when executing the code, you need to log into different services so make sure you pay attention when running the notebook!

Weights and Biases (W&B) is an MLOps platform that can help developers to monitor and document ML training workflows from end to end. As mentioned earlier, we will use W&B to get an idea of how well the training is working, in particular if the model is improving over time. For W&B, we need to name the project; alternatively, we can use wandb's `init()` method:

```
import os
os.environ["WANDB_PROJECT"] = "finetuning"
```

In order to authenticate with W&B, you need to create a free account with them for this at https://www.wandb.ai You can find your API key on the Authorize page: https://wandb.ai/authorizeAgain, we need to paste in our API token. If the previous training run is still active – this could be from a previous execution of the notebook if you are running a second time –, let's make sure we start a new one! This will ensure that we get new reports and dashboard on W&B:

wandb.login(key="xxxxxxxxxxxxxxxx")

```
if wandb.run is not None:
    wandb.finish()
```

Next, we'll need to choose a dataset against which we want to optimize. We can use lots of different datasets here that are appropriate for coding, storytelling, tool use, SQL generation, grade-school math questions (GSM8k), or many other tasks. HuggingFace provides a wealth of datasets, which can be viewed at this url: https://huggingface.co/datasetsThese cover a lot of different, even the most niche tasks. We can also customize our own dataset. For example, we can use langchain to set up training data. There are quite a few methods available for filtering that could help reduce redundancy in the dataset. It would have been appealing to show data collection as a practical recipe in this chapter. However, because of the complexity I am leaving it out of scope for the book.It might be harder to filter for quality from web data, but there are a lot of possibilities. For code models, we could apply code validation techniques to score segments as a quality filter. If the code comes from Github, we can filter by stars or by stars by repo owner. For texts in natural language, quality filtering is not trivial. Search engine placement could serve as a popularity filter since it's often based on user engagement with the content. Further, knowledge distillation techniques could be tweaked as a filter by fact density and accuracy.In this recipe, we are fine-tuning for question answering performance with the Squad V2 dataset. You can see a detailed dataset description on HuggingFace: https://huggingface.co/spaces/evaluate-metric/squad_v2

```
from datasets import load_dataset
dataset_name = "squad_v2"
dataset = load_dataset(dataset_name, split="train")
eval_dataset = load_dataset(dataset_name, split="validation")
```

We are taking both training and validation splits. The Squad V2 dataset bas a part that's supposed to be used in training and another one in validation as we can see in the output of `load_dataset(dataset_name)` :

```
DatasetDict({
    train: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers'],
        num_rows: 130319
    })
    validation: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers'],
        num_rows: 11873
    })
})
```

We'll use the validation splits for early stopping. Early stopping will allow us to stop training when the validation error begins to degrade.The Squad V2 dataset is composed of various features, which we can see here:

```
{'id': Value(dtype='string', id=None),
 'title': Value(dtype='string', id=None),
 'context': Value(dtype='string', id=None),
 'question': Value(dtype='string', id=None),
 'answers': Sequence(feature={'text': Value(dtype='string', id=None),
 'answer_start': Value(dtype='int32', id=None)}, length=-1, id=None)}
```

The basic idea in training is prompting the model with a question and comparing the answer to the dataset.We want a small model that we can run locally at a decent token rate. LLaMa-2 models require signing a license agreement with your email address and to get confirmed (which, to be fair, can be very fast) as it comes with restrictions to commercial use. LLaMa derivates such as OpenLLaMa have been performing quite well as can be evidenced on the HF leaderboard: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboardOpenLLaMa version 1 cannot be used for coding tasks, because of the tokenizer. Therefore, let's use v2! We'll use a 3 billion parameter model, which we'll be able to use even on older hardware:

```
model_id = "openlm-research/open_llama_3b_v2"
new_model_name = f"openllama-3b-peft-{dataset_name}"
```

We can use even smaller models such as `EleutherAI/gpt-neo-125m` which can also give a very good compromise between resource use and performance.Let's load the model:

```
import torch
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
```

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
)
device_map="auto"
base_model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True,
)
base_model.config.use_cache = False
```

The Bits and Bytes configuration makes it possible to quantize our model in 8, 4, 3 or even 2 bits with a much-accelerated inference and lower memory footprint without a incurring a big cost in terms of performance. We are going to store model checkpoints on Google Drive; you need to confirm your login to your google account:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

We'll need to authenticate with Google for this to work. We can set our output directory for model checkpoints and logs to our Google Drive:

```
output_dir = "/content/gdrive/My Drive/results"
```

If you don't want to use google drive, just set this to a directory on your computer.For training, we need to set up a tokenizer:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(model_id, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

Now we'll define our training configuration. We'll set up LORA and other training arguments:

```
from transformers import TrainingArguments, EarlyStoppingCallback
from peft import LoraConfig
# More info: https://github.com/huggingface/transformers/pull/24906
base_model.config.pretraining_tp = 1
peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=64,
    bias="none",
    task_type="CAUSAL_LM",
)
training_args = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-4,
    logging_steps=10,
    max_steps=2000,
    num_train_epochs=100,
    evaluation_strategy="steps",
    eval_steps=5,
    save_total_limit=5
    push_to_hub=False,
    load_best_model_at_end=True,
    report_to="wandb"
)
```

A few comments to explain some of these parameters are in order. The `push_to_hub` argument means that we can push the model checkpoints to the HuggingSpace Hub regularly during training. For this to work you need to set up the HuggingSpace authentication (with write permissions as mentioned). If we opt for this, as `output_dir` we can use `new_model_name`. This will be the repository name under which the model will be available here on

HuggingFace: https://huggingface.co/modelsAlternatively, as I've done here, we can save your model locally or to the cloud, for example google drive to a directory.I've set `max_steps` and `num_train_epochs` very high, because I've noticed that training can still improve after many steps. We are using early stepping together with a high number of maximum training steps to get the model to converge to higher performance. For early stopping, we need to set the `evaluation_strategy` as `"steps"` and `load_best_model_at_end=True`. `eval_steps` is the number of update steps between two evaluations. `save_total_limit=5` means that only last 5 models are saved. Finally, `report_to="wandb"` means that we'll send training stats, some model metadata, and hardware information to W&B, where we can look at graphs and dashboards for each run. The training can then use our configuration:

```
from trl import SFTTrainer
trainer = SFTTrainer(
    model=base_model,
    train_dataset=dataset,
    eval_dataset=eval_dataset,
    peft_config=peft_config,
    dataset_text_field="question",  # this depends on the dataset!
    max_seq_length=512,
    tokenizer=tokenizer,
    args=training_args,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=200)]
)
trainer.train()
```

The training can take quite a while, even running on TPU device. The evaluating and early stopping slows the training down by a lot. If you disable the early stopping, you can make this much faster.We should see some statistics as the training progresses, but it's nicer to show the graph of performance as we can see it on W&B:

Figure 8.4: Fine-tuning training loss over time (steps).

After training is done, we can save the final checkpoint on disk for re-loading:

```
trainer.model.save_pretrained(
    os.path.join(output_dir, "final_checkpoint"),
)
```

We can now share our final model with friends in order to brag about the performance we've achieved by manually pushing to HuggingFace:

```
trainer.model.push_to_hub(
    repo_id=new_model_name
)
```

We can now load the model back using the combination of our HuggingFace username and the repository name (new model name).Let's quickly show how to use this model in LangChain. usually, the peft model is stored as an adapter, not as a full model, therefore the loading is a bit different:

```
from peft import PeftModel, PeftConfig
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
from langchain.llms import HuggingFacePipeline
model_id = 'openlm-research/open_llama_3b_v2'
config = PeftConfig.from_pretrained("benji1a/openllama-3b-peft-squad_v2")
model = AutoModelForCausalLM.from_pretrained(model_id)
model = PeftModel.from_pretrained(model, "benji1a/openllama-3b-peft-squad_v2")
tokenizer = AutoTokenizer.from_pretrained(model_id, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_length=256
)
llm = HuggingFacePipeline(pipeline=pipe)
```

We've done everything so far on Google Colab, but we can equally execute this locally, just note that you need to have the huggingface peft library installed!So far, we've shown how to fine-tune and deploy an open-source LLM. Some commercial models can be fine-tuned on custom data as well. For example, both OpenAI's GPT-3.5 and Google's PaLM model offer this capability. This has been integrated with a few Python libraries. With the Scikit-LLM library, this is only a few lines of code in either case:Fine-tuning a PaLM model for text classification can be done like this:

```
from skllm.models.palm import PaLMClassifier
clf = PaLMClassifier(n_update_steps=100)
clf.fit(X_train, y_train) # y_train is a list of labels
labels = clf.predict(X_test)
```

Similarly, you can fine-tune the GPT-3.5 model for text classification like this:

```
from skllm.models.gpt import GPTClassifier
clf = GPTClassifier(
        base_model = "gpt-3.5-turbo-0613",
        n_epochs = None, # int or None. When None, will be determined automatically by OpenAI
        default_label = "Random", # optional
)
clf.fit(X_train, y_train) # y_train is a list of labels
labels = clf.predict(X_test)
```

Interestingly, in the fine-tuning available on OpenAI, all inputs are passed through a moderation system to make sure that the inputs are compatible with safety standards.This concludes fine-tuning. On the extreme end, LLMs can be deployed and queried without any task-specific tuning. By prompting, we can accomplish few-shot learning or even zero-shot learning as we'll discuss in the next section.

Prompt Engineering

Prompts are important for steering the behavior of large language models (LLMs) because they allow aligning the model outputs to human intentions without expensive retraining. Carefully engineered prompts can make LLMs suitable for a wide variety of tasks beyond what they were originally trained for. Prompts act as instructions that demonstrate to the LLM what is the desired input-output mapping. The picture below shows a few examples for prompting different language models (source: "Pre-train, Prompt, and Predict - A Systematic Survey of Prompting Methods in Natural Language Processing" by Liu and colleagues, 2021):

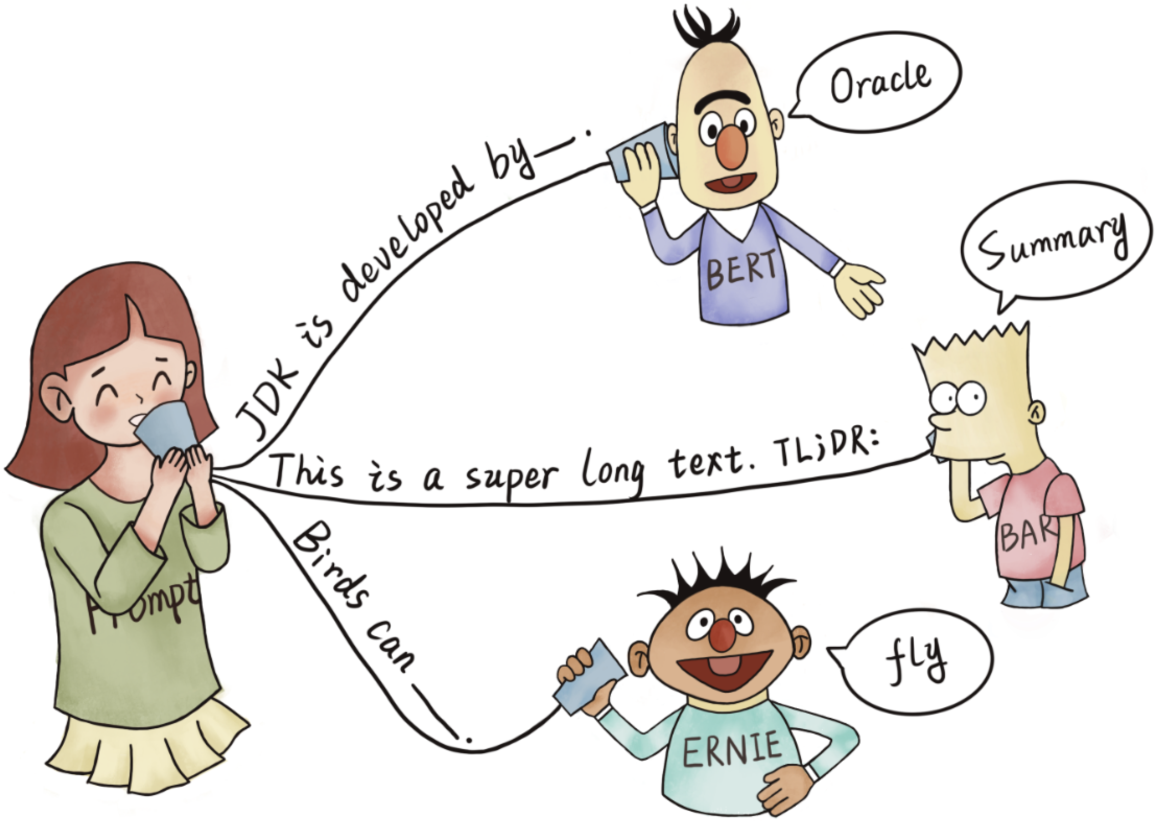


Figure 8.5: Prompt examples, particularly knowledge probing in cloze form, and summarization.

Prompt engineering, also known as in-context learning, refers to techniques for steering large language model (LLM) behavior through carefully designed prompts, without changing the model weights. The goal is to align the model outputs with human intentions for a given task. By designing good prompt templates, models can achieve strong results, sometimes comparable to fine-tuning. But how do good prompts look like?

Structure of Prompts

Prompts consist of three main components:

- Instructions that describe the task requirements, goals and format of inputs/outputs
- Examples that demonstrate the desired input-output pairs
- The input that the model must act on to generate the output

Instructions explain the task to the model unambiguously. Examples provide diverse demonstrations of how different inputs should map to outputs. The input is what the model must generalize to.Basic prompting methods include zero-shot prompting with just the input text, and few-shot prompting with a few demonstration examples showing desired input-output pairs. Researchers have identified biases like majority label bias and recency bias that contribute to variability in few-shot performance. Careful prompt design through example selection, ordering, and formatting can help mitigate these issues.More advanced prompting techniques include instruction prompting, where the task requirements are described explicitly rather than just demonstrated. Self-consistency sampling generates multiple outputs and selects the one that aligns best with the examples. Chain-of-thought (CoT) prompting generates explicit reasoning steps leading to the final output. This is especially beneficial for complex reasoning tasks. CoT prompts can be manually written or generated automatically via methods like augment-prune-select.This table gives a brief overview of a few methods of prompting compared to fine-tuning:

Technique	Method	Key Idea	Results
Fine-tuning	Fine-tune on explanation dataset generated via prompting	Improves model's reasoning abilities	73% accuracy on commonsense QA dataset
Zero-shot prompting	Simply feeding the task text to the model and asking for results.	Text: "i'll bet the video game is a lot more fun than the film." Sentiment:	
Chain-of-Thought (CoT)	Prefix responses with "Let's think step by step"	Gives model space to reason before answering	Quadrupled accuracy on math dataset
Few-shot prompting	Provide few demos consisting of input and desired output to help the model understand	Shows desired reasoning format	Tripled accuracy on grade school math
Least-to-most prompting	Prompt model for simpler subtasks to solve incrementally. "To solve {question}, we need to first solve: "	Decomposes problems into smaller pieces	Boosted accuracy from 16% to 99.7% on some tasks
Selection-inference prompting	Alternate selection and inference prompts	Guides model through reasoning steps	Lifts performance on long reasoning tasks
Self-consistency	Pick most frequent answer from multiple samples	Increases redundancy	Gained 1-24 percentage points across benchmarks
Verifiers	Train separate model to evaluate responses	Filters out incorrect responses	Lifted grade school math accuracy ~20 percentage points

Figure 8.6: Pronpting techniques for LLMs.

Some prompting techniques incorporate external information retrieval to provide missing context to the LLM before generating the output. For open-domain QA, relevant paragraphs can be retrieved via search engines and incorporated into the prompt. For closed-book QA, few-shot examples with evidence-question-answer format work better than question-answer format.There are different techniques to improve the reliability of large language models (LLMs) in complex reasoning tasks:

1. Prompting and Explanation: prompting the model to explain its reasoning step-by-step before answering using prompts like "Let's think step by step" (as in CoT) significantly improves accuracy in reasoning tasks.
2. Providing few-shot examples of reasoning chains helps demonstrate the desired format and guides LLMs in generating coherent explanations.
3. Alternate Selection and Inference Prompts: Utilizing a combination of specialized selection prompts (narrow down the answer space) and inference prompts (generate the final response) leads to better results compared to generic reasoning prompts alone.
4. Problem Decomposition: Breaking down complex problems into smaller subtasks or components using a least-to-most prompting approach helps improve reliability, as it allows for a more structured and manageable problem-solving process.
5. Sampling Multiple Responses: Sampling multiple responses from LLMs during generation and picking the most common answer increases consistency, reducing reliance on a single output. In particular, training separate verifier models that evaluate candidate responses generated by LLMs helps filter out incorrect or unreliable answers, improving overall reliability.

Finally, fine-tuning LLMs on explanation datasets generated through prompting enhances their performance and reliability in reasoning tasks

> **Few-shot learning** presents the LLM with just a few input-output examples relevant to the task, without explicit instructions. This allows the model to infer the intentions and goals purely from demonstrations. Carefully selected, ordered and formatted examples can greatly improve the model's inference abilities. However, few shot learning can be prone to biases and variability across trials. Adding explicit instructions can make the

intentions more transparent to the model and improve robustness. Overall, prompts combine the strengths of instructions and examples to maximize steering of the LLM for the task at hand.

Instead of hand-engineering prompts, methods like automatic prompt tuning learn optimal prompts by directly optimizing prefix tokens on the embedding space. The goal is to increase the likelihood of desired outputs given inputs. Overall, prompt engineering is an active area of research for aligning large pre-trained LLMs with human intentions for a wide variety of tasks. Careful prompt design can steer models without expensive retraining.In this section, we'll go through a few (but not all) of the techniques mentioned beforehand. Let's discuss the tools that LangChain provides tools to create prompt templates in Python!

Templating

Prompts are the instructions and examples we provide to language models to steer their behavior. Prompt templating refers to creating reusable templates for prompts that can be configured with different parameters. LangChain provides tools to create prompt templates in Python. Templates allow prompts to be dynamically generated with variable input. We can create a basic prompt template like this:

```
from langchain import PromptTemplate
prompt = PromptTemplate("Tell me a {adjective} joke about {topic}")
```

This template has two input variables - {adjective} and {topic}. We can format these with values:

```
prompt.format(adjective="funny", topic="chickens")
# Output: "Tell me a funny joke about chickens"
```

The template format defaults to Python f-strings, but Jinja2 is also supported.Prompt templates can be composed into pipelines, where the output of one template is passed as input to the next. This allows modular reuse.

Chat Prompt Templates

For conversational agents, we need chat prompt templates:

```
from langchain.prompts import ChatPromptTemplate
template = ChatPromptTemplate.from_messages([
  ("human", "Hello, how are you?"),
  ("ai", "I am doing great, thanks!"),
  ("human", "{user_input}"),
])
template.format_messages(user_input="What is your name?")
```

This formats a list of chat messages instead of a string. This can be useful for taking the history of a conversation into account. We've looked at different memory methods in Chapter 5. These are similarly relevant in this context to make sure model outputs are relevant and on point.Prompt templating enables reusable, configurable prompts. LangChain provides a Python API for conveniently creating templates and formatting them dynamically. Templates can be composed into pipelines for modularity. Advanced prompt engineering can further optimize prompting.

Advanced Prompt Engineering

LangChain provides tools to enable advanced prompt engineering strategies like few-shot learning, dynamic example selection, and chained reasoning.

Few-Shot Learning

The `FewShotPromptTemplate` allows showing the model just a few demonstration examples of the task to prime it, without explicit instructions. For instance:

```
from langchain.prompts import FewShotPromptTemplate, PromptTemplate
example_prompt = PromptTemplate("{input} -> {output}")
examples = [
```

```
  {"input": "2+2", "output": "4"},
  {"input": "3+3", "output": "6"}
]
prompt = FewShotPromptTemplate(
  examples=examples,
  example_prompt=example_prompt
)
```

The model must infer what to do from the examples alone.

Dynamic Example Selection

To choose examples tailored to each input, `FewShotPromptTemplate` can accept an `ExampleSelector` rather than hardcoded examples:

```
from langchain.prompts import SemanticSimilarityExampleSelector
selector = SemanticSimilarityExampleSelector(...)
prompt = FewShotPromptTemplate(
   example_selector=selector,
   example_prompt=example_prompt
)
```

`ExampleSelector` implementations like `SemanticSimilarityExampleSelector` automatically find the most relevant examples for each input.

Chained Reasoning

When asking an LLM to reason through a problem, it is often more effective to have it explain its reasoning before stating the final answer. For example:

```
from langchain.prompts import PromptTemplate
reasoning_prompt = "Explain your reasoning step-by-step. Finally, state the answer: {questior
prompt = PromptTemplate(
  reasoning_prompt=reasoning_prompt,
  input_variables=["questions"]
)
```

This encourages the LLM to logically think through the problem first, rather than just guessing the answer and trying to justify it after. This is called **Zero-Shot Chain of Thought**. Asking an LLM to explain its thought process aligns well with its core capabilities.**Few-Shot Chain of Thought** prompting is a few-shot prompt, where the reasoning is explained as part of the example solutions, with the idea to encourage an LLM to explain its reasoning before making a decision. It has been shown that this kind of prompting can lead to more accurate results, however, this performance boost was found to be proportional to the size of the model, and the improvements seemed to be negligible or even negative in smaller models.In **Tree of Thoughts (ToT)** prompting, we are generating multiple problem-solving steps or approaches for a given prompt and then using the AI model to critique these steps. The critique will be based on the model's judgment of the solution's suitability to the problem. Let's walk through a more detailed example of implementing ToT using LangChain.First, we'll define our 4 chain components with `PromptTemplates`. We need a solution template, an evaluation template, a reasoning template, and a ranking template. Let's first generate solutions:

1

```
solutions_template = """
Generate {num_solutions} distinct solutions for {problem}. Consider factors like {factors}.
Solutions:
"""
solutions_prompt = PromptTemplate(
   template=solutions_template,
   input_variables=["problem", "factors", "num_solutions"]
)
```

Let's ask the LLM to evaluate these solutions:

2

```
evaluation_template = """
Evaluate each solution in {solutions} by analyzing pros, cons, feasibility, and probability c
Evaluations:
"""
evaluation_prompt = PromptTemplate(
  template=evaluation_template,
  input_variables=["solutions"]
)
```

Now we'll reason a bit more about them:

3

```
reasoning_template = """
For the most promising solutions in {evaluations}, explain scenarios, implementation strategi
Enhanced Reasoning:
"""
reasoning_prompt = PromptTemplate(
  template=reasoning_template,
  input_variables=["evaluations"]
)
```

Finally, we can rank these solutions given our reasoning so far:

4

```
ranking_template = """
Based on the evaluations and reasoning, rank the solutions in {enhanced_reasoning} from most
Ranked Solutions:
"""
ranking_prompt = PromptTemplate(
  template=ranking_template,
  input_variables=["enhanced_reasoning"]
)
```

Next, we create chains from these templates before we'll put it all together:

```
chain1 = LLMChain(
   llm=SomeLLM(),
   prompt=solutions_prompt,
   output_key="solutions"
)
chain2 = LLMChain(
   llm=SomeLLM(),
   prompt=evaluation_prompt,
   output_key="evaluations"
)
```

Finally. we connect these chains into a `SequentialChain`:

```
tot_chain = SequentialChain(
   chains=[chain1, chain2, chain3, chain4],
   input_variables=["problem", "factors", "num_solutions"],
   output_variables=["ranked_solutions"]
)
tot_chain.run(
   problem="Prompt engineering",
   factors="Requirements for high task performance, low token use, and few calls to the LLM",
   num_solutions=3
)
```

This allows us to leverage the LLM at each stage of the reasoning process. The ToT approach helps avoid dead-ends by fostering exploration.These techniques collectively enhance the accuracy, consistency, and reliability of large language models' reasoning capabilities on complex tasks by providing clearer instructions, fine-tuning with targeted data, employing problem breakdown strategies, incorporating diverse sampling approaches, integrating verification mechanisms, and adopting probabilistic modeling frameworks.Prompt design is highly significant for unlocking LLM reasoning capabilities, the potential for future advancements in models and prompting techniques, and these principles and techniques form a valuable toolkit for researchers and practitioners working with large language models.Let's summarize!

Summary

In Chapter 1, we discussed the basic principles of generative models, particularly LLMs, and their training. We focused mostly on the pre-training step, which is – generally speaking – adjusting the models to the correlations within words and wider segments of texts. Alignment it the assessment of model outputs against expectations and conditioning is the process of making sure the output is according to expectations. Conditioning allows steering generative AI to improve safety and quality, but it is not a complete solution. In this chapter, the focus is on conditioning, in particular through fine-tuning and prompting. In fine-tuning the language model is trained on many examples of tasks formulated as natural language instructions, along with appropriate responses. Often this is done through reinforcement learning with human feedback (RLHF), which involves training on a dataset of human-generated (prompt, response) pairs, followed by reinforcement learning from human feedback, however, other techniques have been developed that have been shown to produce competitive results with lower resource footprints. In the first recipe of this chapter, we've implemented a fine-tuning of a small open-source model for question answering.There are many techniques to improve the reliability of LLMs in complex reasoning tasks including step-by-step prompting, alternate selection, and inference prompts, problem decomposition, sampling multiple responses, and employing separate verifier models. These methods have shown to enhance accuracy and consistency in reasoning tasks. We've discussed and compared several techniques. LangChain provides building blocks to unlock advanced prompting strategies like few-shot learning, dynamic example selection, and chained reasoning decomposition as we've shown in the examples.Careful prompt engineering is key to aligning language models with complex objectives. Reliability in reasoning can be improved by breaking down problems and adding redundancy. The principles and techniques that we've discussed in this chapter provide a toolkit for experts working with LLMs. We can expect future advancements in both model training and prompting techniques. As these methods and LLMs continue to develop, they will likely become even more effective and useful for a broader range of applications.Let's see if you remember some more key points from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions from memory. I'd recommend you go back to the corresponding sections of this chapter, if you are unsure about any of them:

1. What's alignment in the context of LLMs?
2. What are different methods of conditioning and how can we distinguish them?
3. How's moderation related to conditioning?
4. What is instruction tuning and what's its importance?
5. What is quantization?
6. What are a few methods for fine-tuning?
7. What is few-shot learning?
8. What is Chain of Thought prompting?
9. Explain Tree of Thought prompting!

9 Generative AI in Production

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

In this book so far, we've talked about models, agents, and LLM apps as well as different use cases, but there are many issues that become important when performance and regulatory requirements needs to be ensured, models and applications need to be deployed at scale, and finally monitoring has to be in place. In this chapter, we'll discuss evaluation and observability, summarizing a broad range of topics that encompass the governance and lifecycle management of operationalized AI and decision models, including generative AI models. While offline evaluation provides a preliminary understanding of a model's abilities in a controlled setting, observability in production offers continuing insights into its performance in live environments. Both are crucial at different stages of a model's life cycle and complement each other to ensure optimal operation and results from large language models. We'll discuss a few tools for either case and we'll give examples.We'll also discuss deploying of models and applications built around LLMs giving an overview over available tools and examples for deployment with Fast API and Ray Serve.Throughout the chapter, we'll work on … with LLMs, which you can find in the GitHub repository for the book at https://github.com/benman1/generative_ai_with_langchainThe main sections of this chapter are:

- Introduction
- How to evaluate your LLM app?
- How to deploy your LLM app?
- How to observe your LLM app?

Let's start by introducing MLOps for LLMs and other generative models, what it means and includes.

Introduction

As we've discussed in this book, LLMs have gained significant attention in recent years due to their ability to generate human-like text. From creative writing to conversational chatbots, these generative AI models have diverse applications across industries. However, taking these complex neural network systems from research to real-world deployment comes with significant challenges. This chapter explores the practical considerations and best practices for productionizing generative AI responsibly. We discuss the computational requirements for inference and serving, techniques for optimization, and critical issues around data quality, bias, and transparency. Architectural and infrastructure decisions can make or break a generative AI solution when scaled to thousands of users. At the same time, maintaining rigorous testing, auditing, and ethical safeguards is essential for trustworthy deployment.Deploying applications consisting of models and agents with their tools in production comes with several key challenges that need to be addressed to ensure their effective and safe use:

- **Data Quality and Bias**: Training data can introduce biases that get reflected in model outputs. Careful data curation and monitoring model outputs is crucial.
- **Ethical/Compliance Considerations**: LLMs can generate harmful, biased or misleading content. Review processes and safety guidelines must be established to prevent misuse. Adhering to regulations like HIPAA in specialized industries such as healthcare.
- **Resource Requirements**: LLMs require massive compute resources for training and serving. Efficient infrastructure is critical for cost-effective deployment at scale.

- **Drift or Performance Degradation**: Models need continuous monitoring to detect issues like data drift or performance degradation over time.
- **Lack of Interpretability**: LLMs are often black boxes, making their behaviors and decisions opaque. Interpretability tools are important for transparency.

Taking a trained LLM from research into real-world production involves navigating many complex challenges around aspects like scalability, monitoring, and unintended behaviors. Responsibly deploying capable yet unreliable models involves diligent planning around scalability, interpretability, testing, and monitoring. Techniques like fine-tuning, safety interventions, and defensive design enable developing applications that are helpful, harmless, and honest. With care and preparation, generative AI holds immense potential to benefit industries from medicine to education.Several key patterns can help address the challenges highlighted above:

- **Evaluations**: Solid benchmark datasets and metrics are essential to measure model capabilities, regressions, and alignment with goals. Metrics should be carefully selected based on the task.
- **Retrieval Augmentation**: Retrieving external knowledge provides useful context to reduce hallucinations and add recent information beyond pre-training data.
- **Fine-tuning**: Further tuning LLMs on task-specific data improves performance on target use cases. Techniques like adapter modules reduce overhead.
- **Caching**: Storing model outputs can significantly reduce latency and costs for repeated queries. But cache validity needs careful consideration.
- **Guardrails**: Validating model outputs syntactically and semantically ensures reliability. Guidance techniques directly shape output structure.
- **Defensive UX**: Design anticipating inaccuracies, such as disclaimers on limitations, attributions, and collecting rich user feedback.
- **Monitoring**: Continuously tracking metrics, model behaviors, and user satisfaction provides insight into model issues and business impact.

In Chapter 5, we've already covered safety-aligned techniques like Constitutional AI for mitigating risks like generating harmful outputs. Further, LLMs have the potential to generate harmful or misleading content. It is essential to establish ethical guidelines and review processes to prevent the dissemination of misinformation, hate speech, or any other harmful outputs. Human reviewers can play a crucial role in evaluating and filtering the generated content to ensure compliance with ethical standards.Not only for legal, ethical, and reputational reasons, but also in order to maintain performance, we need to continuously evaluate model performance and outputs in order to detect issues like data drift or loss of capabilities. We'll be discussing techniques to interpret model behaviors and decisions. Improving transparency in high-stakes domains.LLMs or generative AI models require significant computational resources for deployment due to their size and complexity. This includes high-performance hardware, such as GPUs or TPUs, to handle the massive amount of computations involved. Scaling large language models or generative AI models can be challenging due to their resource-intensive nature. As the size of the model increases, the computational requirements for training and inference also increase exponentially. Distributed techniques, such as data parallelism or model parallelism, are often used to distribute the workload across multiple machines or GPUs. This allows for faster training and inference times. Scaling also involves managing the storage and retrieval of large amounts of data associated with these models. Efficient data storage and retrieval systems are required to handle the massive model sizes.Deployment also involves considerations for optimizing inference speed and latency. Techniques like model compression, quantization, or hardware-specific optimizations may be employed to ensure efficient deployment. We've discussed some of this in *Chapter 8*. LLMs or generative AI models are often considered black boxes, meaning it can be difficult to understand how they arrive at their decisions or generate their outputs. Interpretability techniques aim to provide insights into the inner workings of these models. This can involve methods like attention visualization, feature importance analysis, or generating explanations for model outputs. Interpretability is crucial in domains where transparency and accountability are important, such as healthcare, finance, or legal systems.As we discussed in *Chapter 8*, Large language models can be fine-tuned on specific tasks or domains to improve their performance on specific use cases. Transfer learning allows models to leverage pre-trained knowledge and adapt it to new tasks. Transfer learning and fine-tuning on domain-specific data unlocks new use cases while requiring additional diligence.With insightful planning and preparation, generative AI promises to transform industries from creative writing to customer service. But thoughtfully navigating the complexities of these systems remains critical as they continue permeating diverse domains. This chapter aims to provide a practical guide for teams of the pieces that we've left out so far aiming to build impactful and responsible generative AI applications. We mention strategies for data curation, model development, infrastructure, monitoring, and transparency. Before we continue our discussion, a few words on terminology is in place.

Terminology

MLOps is a paradigm that focuses on deploying and maintaining machine learning models in production reliably and efficiently. It combines the practices of DevOps with machine learning to transition algorithms from experimental systems to production systems. MLOps aims to increase automation, improve the quality of production models, and address business and regulatory requirements. **LLMOps** is a specialized sub-category of MLOps. It refers to the operational capabilities and infrastructure necessary for fine-tuning and operationalizing large language models as part of a product. While it may not be drastically different from the concept of MLOps, the distinction lies in the specific requirements connected to handling, refining, and deploying massive language models like GPT-3, which houses 175 billion parameters.The term **LMOps** is more inclusive than LLMOps as it encompasses various types of language models, including both large language models and smaller generative models. This term acknowledges the expanding landscape of language models and their relevance in operational contexts.**FOMO (Foundational Model Orchestration)** specifically addresses the challenges faced when working with foundational models. It highlights the need for managing multi-step processes, integrating with external resources, and coordinating workflows involving these models.The term **ModelOps** focuses on the governance and lifecycle management of AI and decision models as they are deployed. Even more broadly, **AgentOps** involves the operational management of LLMs and other AI agents, ensuring their appropriate behavior, managing their environment and resource access, and facilitating interactions between agents while addressing concerns related to unintended outcomes and incompatible objectives.While FOMO emphasizes the unique challenges of working specifically with foundational models, LMOps provides a more inclusive and comprehensive coverage of a wider range of language models beyond just the foundational ones. LMOps acknowledges the versatility and increasing importance of language models in various operational use cases, while still falling under the broader umbrella of MLOps. Finally, AgentOps explicitly highlights the interactive nature of agents consisting of generative models operating with certain heuristics and includes tools. The emergence of all very specialized terms underscores the rapid evolution of the field; however, their long-term prevalence is unclear. MLOps is an established term widely used in the industry, with significant recognition and adoption. Therefore, we'll stick to MLOps for the remainder of this chapter.Before productionizing any agent or model, we should first evaluate its output, so we should start with this. We will focus on the evaluation methods provided by LangChain.

How to evaluate your LLM apps?

Evaluating LLMs either as standalone entities or in conjunction with an agent chain is crucial to ensure they function correctly and produce reliable results, and is an integral part of the machine learning lifecycle. The evaluation process determines the performance of the models in terms of effectiveness, reliability, and efficiency. The goal of evaluating large language models is to understand their strengths and weaknesses, enhancing accuracy and efficiency while reducing errors, thereby maximizing their usefulness in solving real-world problems. This evaluation process typically occurs offline during the development phase. Offline evaluations provide initial insights into model performance under controlled test conditions and include aspects like hyper-parameter tuning, benchmarking against peer models or established standards. They offer a necessary first step towards refining a model before deployment.Evaluations provide insights into how well an LLM can generate outputs that are relevant, accurate, and helpful. In LangChain, there are various ways to evaluate outputs of LLMs, including comparing chain outputs, pairwise string comparisons, string distances, and embedding distances. The evaluation results can be used to determine the preferred model based on the comparison of outputs. Confidence intervals and p-values can also be calculated to assess the reliability of the evaluation results. LangChain provides several tools for evaluating the outputs of large language models. A common approach is to compare the outputs of different models or prompts using the `PairwiseStringEvaluator` . This prompts an evaluator model to choose between two model outputs for the same input and aggregates the results to determine an overall preferred model.Other evaluators allow assessing model outputs based on specific criteria like correctness, relevance, and conciseness. The `CriteriaEvalChain` can score outputs on custom or predefined principles without needing reference labels. Configuring the evaluation model is also possible by specifying a different chat model like ChatGPT as the evaluator.Let's compare outputs of different prompts or LLMs with the `PairwiseStringEvaluator` , which prompts an LLM to select the preferred output given a specific input.

Comparing two outputs

This evaluation requires an evaluator, a dataset of inputs, and two or more LLMs, chains, or agents to compare. The evaluation aggregates the results to determine the preferred model.The evaluation process involves several steps:

1. Create the Evaluator: Load the evaluator using the `load_evaluator()` function, specifying the type of evaluator (in this case, `pairwise_string`).
2. Select Dataset: Load a dataset of inputs using the `load_dataset()` function.
3. Define Models to Compare: Initialize the LLMs, Chains, or Agents to compare using the necessary configurations. This involves initializing the language model and any additional tools or agents required.
4. Generate Responses: Generate outputs for each of the models before evaluating them. This is typically done in batches to improve efficiency.
5. Evaluate Pairs: Evaluate the results by comparing the outputs of different models for each input. This is often done using a random selection order to reduce positional bias.

Here's an example from the documentation for pairwise string comparisons:

```
from langchain.evaluation import load_evaluator
evaluator = load_evaluator("labeled_pairwise_string")
evaluator.evaluate_string_pairs(
    prediction="there are three dogs",
    prediction_b="4",
    input="how many dogs are in the park?",
    reference="four",
)
```

The output from the evaluator should look as follows:

```
    {'reasoning': 'Both responses are relevant to the question asked, as they both provide a
     'value': 'B',
     'score': 0}
```

The evaluation result includes a score between 0 and 1, indicating the effectiveness of the agent, sometimes along with reasoning that outlines the evaluation process and justifies the score.In this illustration of against the reference, both results are factually incorrect based on the input. We could remove the reference and let an LLM judge the outputs instead, however, this is potentially dangerous since the specified can also be incorrect.

Comparing against criteria

LangChain provides several predefined evaluators for different evaluation criteria. These evaluators can be used to assess outputs based on specific rubrics or criteria sets. Some common criteria include conciseness, relevance, correctness, coherence, helpfulness, and controversiality.The `CriteriaEvalChain` allows you to evaluate model outputs against custom or predefined criteria. It provides a way to verify if an LLM or Chain's output complies with a defined set of criteria. You can use this evaluator to assess correctness, relevance, conciseness, and other aspects of the generated outputs.The `CriteriaEvalChain` can be configured to work with or without reference labels. Without reference labels, the evaluator relies on the LLM's predicted answer and scores it based on the specified criteria. With reference labels, the evaluator compares the predicted answer to the reference label and determines its compliance with the criteria.The evaluation LLM used in LangChain, by default, is GPT-4. However, you can configure the evaluation LLM by specifying other chat models, such as ChatAnthropic or ChatOpenAI, with the desired settings (for example, temperature). The evaluators can be loaded with a custom LLM by passing the LLM object as a parameter to the `load_evaluator()` function.LangChain supports both custom criteria and predefined principles for evaluation. Custom criteria can be defined using a dictionary of `criterion_name: criterion_description pairs`. These criteria can be used to assess outputs based on specific requirements or rubrics.Here's is a simple example:

```
custom_criteria = {
    "simplicity": "Is the language straightforward and unpretentious?",
    "clarity": "Are the sentences clear and easy to understand?",
    "precision": "Is the writing precise, with no unnecessary words or details?",
    "truthfulness": "Does the writing feel honest and sincere?",
```

```
    "subtext": "Does the writing suggest deeper meanings or themes?",
}
evaluator = load_evaluator("pairwise_string", criteria=custom_criteria)
evaluator.evaluate_string_pairs(
    prediction="Every cheerful household shares a similar rhythm of joy; but sorrow, in each
    prediction_b="Where one finds a symphony of joy, every domicile of happiness resounds in
    " identical notes; yet, every abode of despair conducts a dissonant orchestra, each"
    " playing an elegy of grief that is peculiar and profound to its own existence.",
    input="Write some prose about families.",
)
```

We can get a very nuanced comparison of the two outputs as this result shows:

```
{'reasoning': 'Response A is simple, clear, and precise. It uses straightforward language to
```

Alternatively, you can use the predefined principles available in LangChain, such as those from Constitutional AI. These principles are designed to evaluate the ethical, harmful, and sensitive aspects of the outputs. The use of principles in evaluation allows for a more focused assessment of the generated text.

String and semantic comparisons

LangChain supports string comparison and distance metrics for evaluating LLM outputs. String distance metrics like Levenshtein and Jaro provide a quantitative measure of similarity between predicted and reference strings. Embedding distances using models like SentenceTransformers calculate semantic similarity between generated and expected texts.Embedding distance evaluators can use embedding models, such as those based on GPT-4 or Hugging Face embeddings, to compute vector distances between predicted and reference strings. This measures the semantic similarity between the two strings and can provide insights into the quality of the generated text. Here's a quick example from the documentation:

```
from langchain.evaluation import load_evaluator
evaluator = load_evaluator("embedding_distance")
evaluator.evaluate_strings(prediction="I shall go", reference="I shan't go")
```

The evaluator returns the score 0.0966466944859925. You can change the embeddings used with the `embeddings` parameter in the `load_evaluator()` call. This often gives better results than older string distance metrics, but these are also available and allows for simple unit testing and assessment of accuracy. String comparison evaluators compare predicted strings against reference strings or inputs.String distance evaluators use distance metrics, such as Levenshtein or Jaro distance, to measure the similarity or dissimilarity between predicted and reference strings. This provides a quantitative measure of how similar the predicted string is to the reference string.Finally, there's an agent trajectory evaluator, where the `evaluate_agent_trajectory()` method is used to evaluate the input, prediction, and agent trajectory.We can also use LangSmith to compare our performance against a dataset. We'll talk about this companion project for LangChain – LangSmith – more in the section on observability.

Benchmark dataset

With LangSmith, we can evaluate the model performance against a dataset. Let's step through an example.First of all, please make sure you create an account on LangSmith here: https://smith.langchain.com/ You can obtain an API key and set it as `LANGCHAIN_API_KEY` in your environment. We can also set environment variables for project id and tracing:

```
import os
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = "My Project"
```

This configures LangChain to log traces. If we don't tell LangChain the project id, it will log against the `default` project. After this setup, when we run our LangChain agent or chain, we'll be able to see the traces on LangSmith. Let's log a run!

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI()
llm.predict("Hello, world!")
```

We'll see this on LangSmith like this:LangSmith allows us to list all runs so far on the LangSmith project page: https://smith.langchain.com/projects

```
from langsmith import Client
client = Client()
runs = client.list_runs()
print(runs)
```

We can list runs from a specific project or with by `run_type`, for example "chain". Each run comes with inputs and outputs as we can see here:

```
print(f"inputs: {runs[0].inputs}")
print(f"outputs: {runs[0]. outputs}")
```

We can create a dataset from existing agent runs with the `create_example_from_run()` function – or from anything else. Here's how to create a dataset with a set of questions:

```
questions = [
    "A ship's parts are replaced over time until no original parts remain. Is it still the sa
    "If someone lived their whole life chained in a cave seeing only shadows, how would they
    "Is something good because it is natural, or bad because it is unnatural? Why can this be
    "If a coin is flipped 8 times and lands on heads each time, what are the odds it will be
    "Present two choices as the only options when others exist. Is the statement \"You're eit
    "Do people tend to develop a preference for things simply because they are familiar with
    "Is it surprising that the universe is suitable for intelligent life since if it weren't,
    "If Theseus' ship is restored by replacing each plank, is it still the same ship? What is
    "Does doing one thing really mean that a chain of increasingly negative events will follo
    "Is a claim true because it hasn't been proven false? Why could this impede reasoning?",
]
shared_dataset_name = "Reasoning and Bias"
ds = client.create_dataset(
    dataset_name=shared_dataset_name, description="A few reasoning and cognitive bias questic
)
for q in questions:
    client.create_example(inputs={"input": q}, dataset_id=ds.id)
```

We can then run an LLM agent or chain on the dataset like this:

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
llm = ChatOpenAI(model="gpt-4", temperature=0.0)
def construct_chain():
    return LLMChain.from_string(
        llm,
        template="Help out as best you can.\nQuestion: {input}\nResponse: ",
    )
```

We use a constructor function to initialize for each input. In order to evaluate the model performance against this dataset, we need to define an evaluator as we've seen in the previous section.

```
from langchain.evaluation import EvaluatorType
from langchain.smith import RunEvalConfig
evaluation_config = RunEvalConfig(
    evaluators=[
        # Arbitrary criterion as a key: value pair in the criteria dict:
        RunEvalConfig.Criteria({"helpfulness": "Is the response helpful?"}),
        RunEvalConfig.Criteria({"insightful": "Is the response carefully thought out?"})
    ]
)
```

We'll pass a dataset and evaluators to `run_on_dataset()` to generate metrics and feedback:

```
from langchain.smith import run_on_dataset
results = run_on_dataset(
  client=client,
  dataset=dataset,
  llm_factory=lambda: my_agent,
```

```
  evaluation=evaluation_config
)
```

Similarly, we could pass a dataset and evaluators to arun_on_dataset `()` to generate metrics and feedback asynchronously. We can view the evaluator feedback in the LangSmith UI to identify areas for improvement:

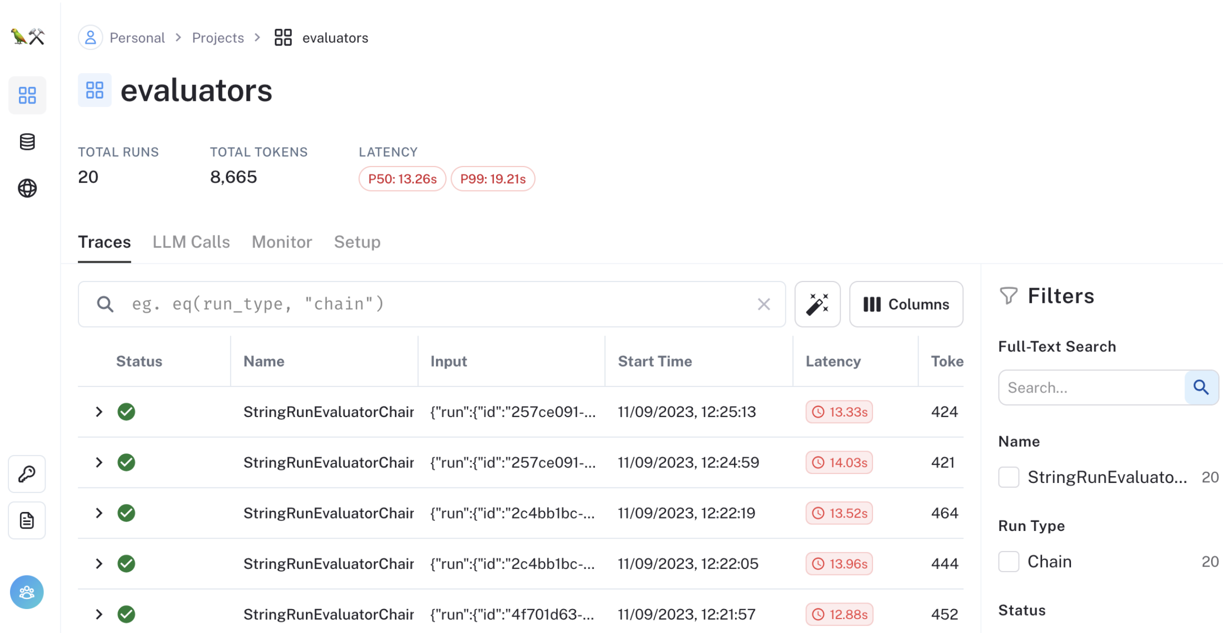


Figure 9.1: Evaluators in LangSmith.

We can click on any of these evaluations to see some detail, for example, for the careful thinking evaluator, we get this prompt that includes the original answer by the LLM:

```
You are assessing a submitted answer on a given task or input based on a set of criteria. Her
[BEGIN DATA]
***
[Input]: Is something good because it is natural, or bad because it is unnatural? Why can thi
***
[Submission]: The argument that something is good because it is natural, or bad because it is
***
[Criteria]: insightful: Is the response carefully thought out?
***
[END DATA]
Does the submission meet the Criteria? First, write out in a step by step manner your reasoni
```

We get this evaluation:

```
The criterion is whether the response is insightful and carefully thought out.
The submission provides a clear and concise explanation of the "appeal to nature" fallacy, de
Therefore, the submission does meet the criterion of being insightful and carefully thought c
Y
Y
```

A way to improve performance for a few types of problems is to do few-shot prompting. LangSmith can help us with this as well. You can find more examples for this in the LangSmith documentation.This concludes evaluation. Now that we've evaluated our agents, let's say we are happy with the performance and we deploy it!

How to deploy your LLM apps?

Given the increasing use of LLMs in various sectors, it's imperative to understand how to effectively deploy models and apps into production. Deployment Services and Frameworks can help to scale the technical hurdles. There are lots of different ways to productionize LLM-apps or applications with generative AI. Deployment for production requires research into and knowledge of the generative AI ecosystem, which encompasses different aspects including:

- Models and LLM-as-a-Service: LLMs and other models either run directly or offered as an API on vendor-provided infrastructure.
- Reasoning Heuristics: Retrieval Augmented Generation (RAG), Tree-of-Thought, and others.
- Vector Databases: Aid in retrieving contextually relevant information for prompts.
- Prompt Engineering Tools: These facilitate in-context learning without requiring expensive fine-tuning or sensitive data.
- Pre-training and fine-tuning: For models specialized for specific tasks or domains.
- Prompt Logging, Testing, and Analytics: An emerging sector inspired by the desire to understand and improve the performance of Large Language Models.
- Custom LLM Stack: A set of tools for shaping and deploying solutions built on open-source models.

We've discussed models in *Chapter 1* and *Chapter 3*, reasoning heuristics in chapters 4-7, vector databases in Chapter 5, and prompts and fine-tuning in *Chapter 8*. In this chapter, we'll focus on logging, monitoring, and custom tools for deployment.LLMs are typically utilized using external LLM providers or self-hosted models. With external providers, computational burdens are shouldered by companies like OpenAI or Anthropic, while LangChain facilitates business logic implementation. However, self-hosting open-source LLMs can significantly decrease costs, latency, and privacy concerns.Some tools with infrastructure offer the full package. For example, you can deploy LangChain agents with Chainlit creating ChatGPT-like UIs with Chainlit. Some of the key features include intermediary steps visualisation, element management & display (images, text, carousel, and others) as well as cloud deployment. BentoML is a framework that enables the containerization of machine learning applications to use them as microservices running and scaling independently with automatic generation of OpenAPI and gRPC endpoints.You can also deploy LangChain to different cloud service endpoints, for example, an Azure Machine Learning Online Endpoint. With Steamship, LangChain developers can rapidly deploy their apps, which includes: production-ready endpoints, horizontal scaling across dependencies, persistent storage of app state, multi-tenancy support, and more.Here is a table summarizing services and frameworks for deploying large language model applications:

Name	**Description**	**Type**
Streamlit	Open-source Python framework for building and deploying web apps	Framework
Gradio	Lets you wrap models in an interface and host on Hugging Face	Framework
Chainlit	Build and deploy conversational ChatGPT-like apps	Framework
Apache Beam	Tool for defining and orchestrating data processing workflows	Framework
Vercel	Platform for deploying and scaling web apps	Cloud Service
FastAPI	Python web framework for building APIs	Framework
Fly.io	App hosting platform with autoscaling and global CDN	Cloud Service
DigitalOcean App Platform	Platform to build, deploy and scale apps	Cloud Service
Google Cloud	Services like Cloud Run to host and scale containerized apps	Cloud Service
Steamship	ML infrastructure platform for deploying and scaling models	Cloud Service
Langchain-serve	Tool to serve LangChain agents as web APIs	Framework
BentoML	Framework for model serving, packaging and deployment	Framework
OpenLLM	Provides open APIs to commercial LLMs	Cloud Service
Databutton	No-code platform to build and deploy model workflows	Framework
Azure ML	Managed ML ops service on Azure for models	Cloud Service

Figure 9.2: Services and frameworks for deploying large language model applications.

All of these are well-documented with different use cases, often directly referencing LLMs. We've already shown examples with Streamlit and Gradio, and we've discussed how to deploy them to HuggingFace Hub as an example.There are a few main requirements for running LLM applications:

- Scalable infrastructure to handle computationally intensive models and potential spikes in traffic
- Low latency for real-time serving of model outputs
- Persistent storage for managing long conversations and app state
- APIs for integration into end-user applications
- Monitoring and logging to track metrics and model behavior

Maintaining cost efficiency can be challenging with large volumes of user interactions and high costs associated with LLM services. Strategies to manage efficiency include self-hosting models, auto-scaling resource allocations based on traffic, using spot instances, independent scaling, and batching requests to better utilize GPU resources.The choice of the tools and the infrastructure determines trade-offs between these requirements. Flexibility and ease is very important, because we want to be able to iterate rapidly, which is vital due to the dynamic nature of ML and LLM landscapes. It's crucial to avoid getting tied to one solution. A flexible, scalable serving layer that accommodates various models is key. Model composition and cloud providers' selection forms part of this flexibility equation.For most flexibility, Infrastructure as Code (IaC) tools like Terraform, CloudFormation, or Kubernetes YAML files can recreate your infrastructure reliably and quickly. Moreover, continuous integration and continuous delivery (CI/CD) pipelines can automate testing and deployment processes to reduce errors and facilitate quicker feedback and iteration.Designing a robust LLM application service can be a complex task requiring an understanding the trade-offs and critical considerations when evaluating serving frameworks. Leveraging one of these solutions for deployment allows developers to focus on developing impactful AI applications rather than infrastructure. As mentioned LangChain plays nicely with several open-source projects and frameworks like Ray Serve, BentoML, OpenLLM, Modal, and Jina. In the next section, we'll deploy a chat service webserver based on FastAPI.

Fast API webserver

FastAPI is a very popular choice for deployment of webservers. Designed to be fast, easy to use, and efficient, it is a modern, high-performance web framework for building APIs with Python. Lanarky is a small, open-source library for deploying LLM applications that provides convenient wrappers around Flask API as well as Gradio for deployment of LLM applications. This means you can get a REST API endpoint as well as the in-browser visualization at once and you only need a few lines of code.

> A **REST API** (Representational State Transfer Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other over the internet. It follows the principles of REST, which is an architectural style for designing networked applications. A REST API uses HTTP methods (such as GET, POST, PUT, DELETE) to perform operations on resources, and it typically sends and receives data in a standardized format, such as JSON or XML.

In the library documentation, there are several examples including a Retrieval QA with Sources Chain, a Conversational Retrieval app, and a Zero Shot Agent. Following another example, we'll implement a chatbot webserver with Lanarky. We'll set up a web server using Lanarky that integrates with Gradio, creates a `ConversationChain` instance with an LLM model and settings, and defines routes for handling HTTP requests.First, we'll import the necessary dependencies, including FastAPI for creating the web server, `mount_gradio_app` for integrating with Gradio, `ConversationChain` and `ChatOpenAI` from Langchain for handling LLM conversations, and other required modules:

```
from fastapi import FastAPI
from lanarky.testing import mount_gradio_app
from langchain import ConversationChain
from langchain.chat_models import ChatOpenAI
from lanarky import LangchainRouter
from starlette.requests import Request
from starlette.templating import Jinja2Templates
```

Please note that you need to set your environment variables as explained in chapter 3. A `create_chain()` function is defined to create an instance of `ConversationChain`, specifying the LLM model and its settings:

```
def create_chain():
    return ConversationChain(
        llm=ChatOpenAI(
```

```
            temperature=0,
            streaming=True,
        ),
        verbose=True,
    )
```

We set the chain as a `ConversationChain`.

```
chain = create_chain()
```

The app variable is assigned to `mount_gradio_app`, which creates a `FastAPI` instance titled *ConversationChainDemo* and integrates it with Gradio:

```
app = mount_gradio_app(FastAPI(title="ConversationChainDemo"))
```

The templates variable gets set to a `Jinja2Templates` class, specifying the directory where templates are located for rendering. This specifies how the webpage will be shown allowing all kind of customization:

```
templates = Jinja2Templates(directory="webserver/templates")
```

An endpoint for handling HTTP GET requests at the root path (`/`) is defined using the FastAPI decorator `@app.get`. The function associated with this endpoint returns a template response for rendering the index.xhtml template:

```
@app.get("/")
async def get(request: Request):
    return templates.TemplateResponse("index.xhtml", {"request": request})
```

The router object is created as a `LangchainRouter` class. This object is responsible for defining and managing the routes associated with the `ConversationChain` instance. We can add additional routes to the router for handling JSON-based chat that even work with WebSocket requests:

```
langchain_router = LangchainRouter(
    langchain_url="/chat", langchain_object=chain, streaming_mode=1
)
langchain_router.add_langchain_api_route(
    "/chat_json", langchain_object=chain, streaming_mode=2
)
langchain_router.add_langchain_api_websocket_route("/ws", langchain_object=chain)
app.include_router(langchain_router)
```

Now our application knows how to handle requests made to the specified routes defined within the router, directing them to the appropriate functions or handlers for processing.We will use Uvicorn to run our application. Uvicorn excels in supporting high-performance, asynchronous frameworks like FastAPI and Starlette. It is known for its ability to handle a large number of concurrent connections and perform well under heavy loads due to its asynchronous nature.We can run the webserver from the terminal like this:

```
uvicorn webserver.chat:app -reload
```

This command starts a webserver, which you can view in your browser, at this local address: http://127.0.0.1:8000The reload switch (`--reload`) is particularly handy, because it means the server will be automatically restarted once you've made any changes.Here's a snapshot of the chatbot application we've just deployed:

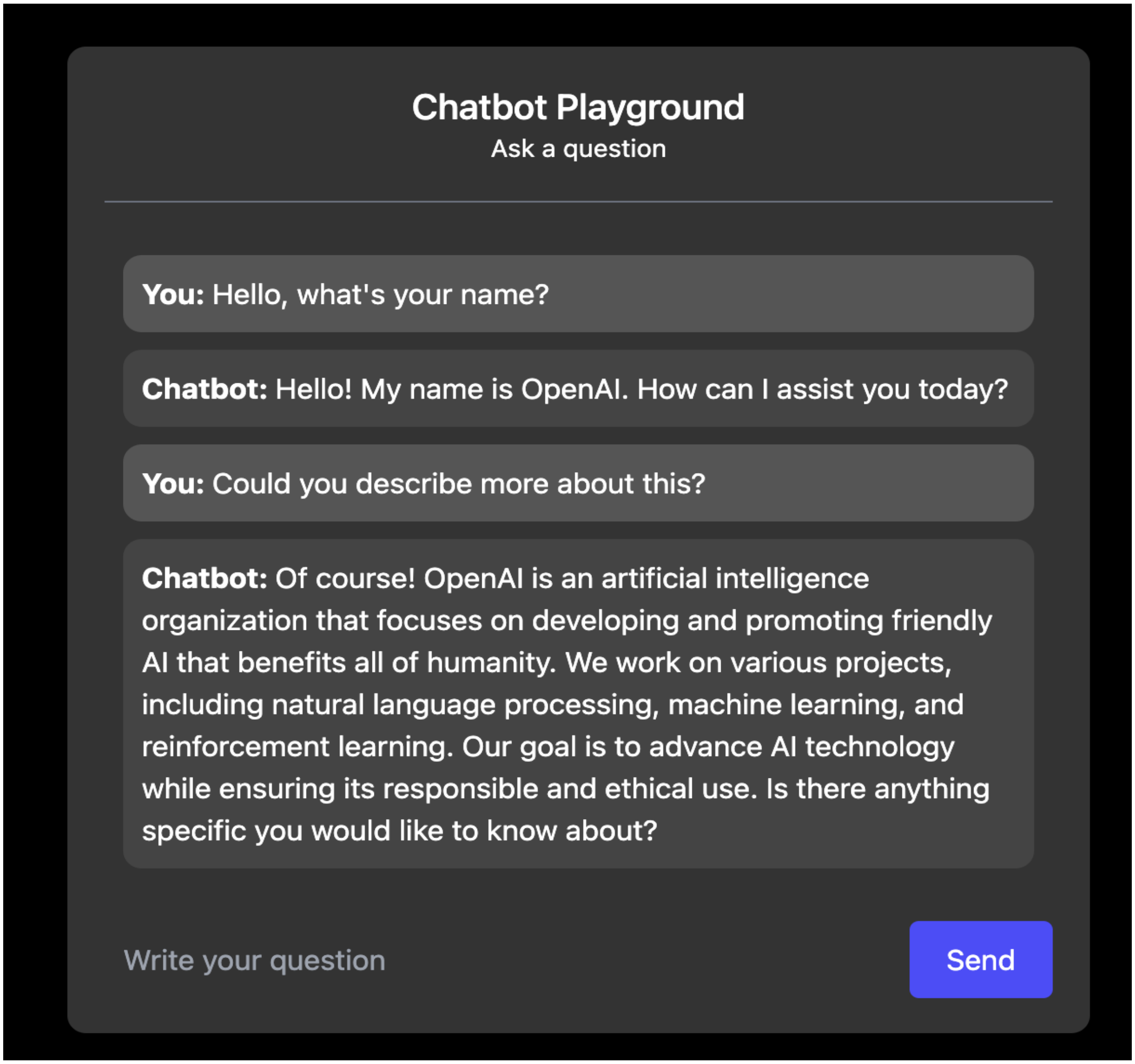


Figure 9.3: Chatbot in Flask/Lanarky

I think this looks quite nice for little work we've put in. It also comes with a few nice features such as REST API, a web UI, and a websocket interface. While Uvicorn itself does not provide built-in load balancing functionality, it can work together with other tools or technologies such as Nginx or HAProxy to achieve load balancing in a deployment setup, which distribute incoming client requests across multiple worker processes or instances. The use of Uvicorn with load balancers enables horizontal scaling to handle large traffic volumes, improves response times for clients, enhances fault tolerance.In the next section, we'll see how to build robust and cost-effective generative AI applications with Ray. We'll built a simple search engine using LangChain for text processing and Ray for scaling indexing and serving.

Ray

Ray provides a flexible framework to meet infrastructure challenges of complex neural networks in production by scaling out generative AI workloads across clusters. Ray helps with common deployment needs like low-latency serving, distributed training, and large-scale batch inference. Ray also makes it easy to spin up on-demand fine-tuning or scale existing workloads from one machine to many. Some capability includes:

- Schedule distributed training jobs across GPU clusters using Ray Train
- Deploy pre-trained models at scale for low-latency serving with Ray Serve
- Run large batch inference in parallel across CPUs and GPUs with Ray Data
- Orchestrate end-to-end generative AI workflows combining training, deployment, and batch processing

We'll use LangChain and Ray to build a simple search engine for the Ray documentation following an example implemented by Waleed Kadous for the anyscale Blog and on the langchain-ray repository on Github. You can see this as an extension of the recipe in *Channel 5*. You can see the full code for this recipe under semantic search here: https://github.com/benman1/generative_ai_with_langchain You'll also see how to run this as a FastAPI server.First, we'll ingest and index the Ray docs so we can quickly find relevant passages for a search query:

```
# Load the Ray docs using the LangChain loader
loader = RecursiveUrlLoader("docs.ray.io/en/master/")
docs = loader.load()
# Split docs into sentences using LangChain splitter
chunks = text_splitter.create_documents(
    [doc.page_content for doc in docs],
    metadatas=[doc.metadata for doc in docs])
# Embed sentences into vectors using transformers
embeddings = LocalHuggingFaceEmbeddings('multi-qa-mpnet-base-dot-v1')
# Index vectors using FAISS via LangChain
db = FAISS.from_documents(chunks, embeddings)
```

This builds our search index by ingesting the docs, splitting into sentences, embedding the sentences, and indexing the vectors. Alternatively, we can accelerate the indexing by parallelizing the embedding step:

```
# Define shard processing task
@ray.remote(num_gpus=1)
def process_shard(shard):
  embeddings = LocalHuggingFaceEmbeddings('multi-qa-mpnet-base-dot-v1')
  return FAISS.from_documents(shard, embeddings)
# Split chunks into 8 shards
shards = np.array_split(chunks, 8)
# Process shards in parallel
futures = [process_shard.remote(shard) for shard in shards]
results = ray.get(futures)
# Merge index shards
db = results[0]
for result in results[1:]:
  db.merge_from(result)
```

By running embedding on each shard in parallel, we can significantly reduce indexing time. We save the database index to disk:

```
db.save_local(FAISS_INDEX_PATH)
```

`FAISS_INDEX_PATH` is an arbitrary file name. I've set it to `faiss_index.db`.Next, we'll see how we can serve search queries with Ray Serve.

```
# Load index and embedding
db = FAISS.load_local(FAISS_INDEX_PATH)
embedding = LocalHuggingFaceEmbeddings('multi-qa-mpnet-base-dot-v1')
@serve.deployment
class SearchDeployment:
  def __init__(self):
    self.db = db
    self.embedding = embedding

  def __call__(self, request):
    query_embed = self.embedding(request.query_params["query"])
    results = self.db.max_marginal_relevance_search(query_embed)
    return format_results(results)
deployment = SearchDeployment.bind()
# Start service
serve.run(deployment)
```

This lets us serve search queries as a web endpoint! Running this gives me this output:

```
Started a local Ray instance.
View the dashboard at 127.0.0.1:8265
```

We can now query it from Python:

```
import requests
query = "What are the different components of Ray"
         " and how can they help with large language models (LLMs)?”
response = requests.post("http://localhost:8000/", params={"query": query})
print(response.text)
```

For me, the server fetches the Ray use cases page at: http://https://docs.ray.io/en/latest/ray-overview/use-cases.xhtmlWhat I really liked was the monitoring with the Ray Dashboard, which looks like this:

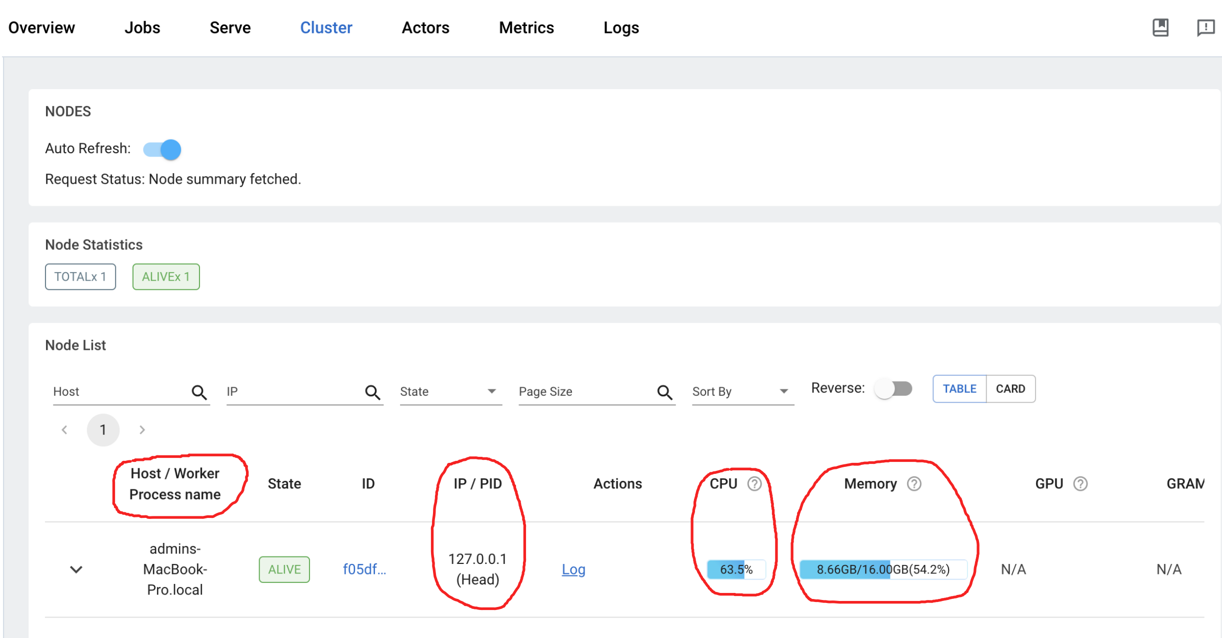


Figure 9.4: Ray Dashboard.

This dashboard very powerful as it can give you a whole bunch of metrics and other information. Collecting metrics is really easy, since all you have to do is setting and updating variables of type `Counter`, `Gauge`, `Histogram` or other types within the deployment object or actor. For time series charts you should have either Prometheus or Grafana server installed. As you can see in the full implementation on Github, we can also spin this up as a FastAPI server. This concludes our simple semantic search engine with LangChain and Ray. As models and LLM apps grow more sophisticated and highly interwoven into the fabric of business applications, observability and monitoring during production become necessary to ensure their accuracy, efficiency, and reliability ongoing. The next section focuses on the significance of monitoring LLMs and highlights key metrics to track for a comprehensive monitoring strategy.

2023/11/03

How to observe LLM apps?

The dynamic nature of real-world operations means that the conditions assessed during offline evaluations hardly cover all potential scenarios that LLMs may encounter in production systems. Thus comes the need for observability in production – a more on-going, real-time observation to capture anomalies that offline tests could not anticipate.Observability allows monitoring behaviors and outcomes as the model interacts with actual input data and users in production. It includes logging, tracking, tracing and alerting mechanisms to ensure healthy system functioning, performance optimization and catching issues like model drift early.As discussed, LLMs have become increasingly important components of many applications in sectors like health, e-commerce, and education.

> Tracking, tracing, and monitoring are three important concepts in the field of software operation and management. While all related to understanding and improving a system's performance, they each have distinct roles. While tracking and tracing are about keeping detailed historical records for analysis and debugging, monitoring is aimed at real-time observation and immediate awareness of issues to ensure optimal system functionality at all times. All three of these concepts fall within the category of observability.
>
>> **Monitoring** is the ongoing process of overseeing the performance of a system or application. This might involve continuously collecting and analyzing metrics related to system health such as memory usage, CPU utilization, network latency, and the overall application/service performance (like response time). Effective monitoring includes setting up alert systems for anomalies or unexpected behaviors – sending notifications when certain thresholds are exceeded. While tracking and tracing are about keeping detailed historical records for analysis and debugging, monitoring is aimed at real-time observation & immediate awareness of issues to ensure optimal system functionality at all times.

The chief aim for monitoring and observability is to provide insights into model performance and behavior through real-time data. This helps in:

- **Preventing model drift**: Models can degrade over time due to changes in the characteristics of input data or user behavior. Regular monitoring can identify such situations early and apply corrective measures.
- **Performance optimization**: By tracking metrics like inference times, resource usage, and throughput, you can make adjustments to improve the efficiency and effectiveness of LLMs in production.
- **A/B Testing**: It helps compare how slight differences in models may result in different outcomes which aids in decision-making towards model improvements.
- **Debugging Issues**: Monitoring helps identify unforeseen problems that can occur during runtime, enabling rapid resolution.

It's important to consider the monitoring strategy that consists of a few considerations:

- **Metrics to monitor**: Define key metrics of interest such as prediction accuracy, latency, throughput etc. based on desired model performance.
- **Monitoring Frequency**: Frequency should be determined based on how critical the model is to operations - a highly critical model may require near real-time monitoring.
- **Logging**: Logs should provide comprehensive details regarding every relevant action performed by the LLM so analysts can track back any anomalies.
- **Alerting Mechanism**: The system should raise alerts if it detects anomalous behavior or drastic performance drops.

Monitoring LLMs serves multiple purposes, including assessing model performance, detecting abnormalities or issues, optimizing resource utilization, and ensuring consistent and high-quality outputs. By continuously evaluating the behavior and performance of LLMs via validation, shadow launches, and interpretation along with dependable offline evaluation, organizations can identify and mitigate potential risks, maintain user trust, and provide an optimal experience.Here's a list of relevant metrics:

- **Inference Latency**: Measure the time it takes for the LLM to process a request and generate a response. Lower latency ensures a faster and more responsive user experience.
- **Query per Second (QPS)**: Calculate the number of queries or requests that the LLM can handle within a given time frame. Monitoring QPS helps assess scalability and capacity planning.
- **Token Per Second (TPS)**: Track the rate at which the LLM generates tokens. TPS metrics are useful for estimating computational resource requirements and understanding model efficiency.
- **Token Usage**: The number of tokens correlates with the resource usage such as hardware utilization, latency, and costs.
- **Error Rate**: Monitor the occurrence of errors or failures in LLM responses, ensuring error rates are kept within acceptable limits to maintain the quality of outputs.

- **Resource Utilization**: Measure the consumption of computational resources, such as CPU, memory, and GPU, to optimize resource allocation and avoid bottlenecks.
- **Model Drift**: Detect changes in LLM behavior over time by comparing its outputs to a baseline or ground truth, ensuring the model remains accurate and aligned with expected outcomes.
- **Out-of-Distribution Inputs**: Identify inputs or queries falling outside the intended distribution of the LLM's training data, which can cause unexpected or unreliable responses.
- **User Feedback Metrics**: Monitor user feedback channels to gather insights on user satisfaction, identify areas for improvement, and validate the effectiveness of the LLM.

Data scientists and machine learning engineers should check for staleness, incorrect learning, and bias using model interpretation tools like LIME and SHAP. The most predictive features changing suddenly could indicate a data leak. Offline metrics like AUC do not always correlate with online impacts on conversion rate, so it is important to find dependable offline metrics that translate to online gains relevant to the business ideally direct metrics like clicks and purchases that the system impacts directly.Effective monitoring enables the successful deployment and utilization of LLMs, boosting confidence in their capabilities and fostering user trust. It should be cautioned, however, that you should study the privacy and data protection policy when relying on cloud service platforms.In the next section, we'll look at monitoring the trajectory of an agent.

Tracking and Tracing

> **Tracking** generally refers to the process of recording and managing information about a particular operation or series of operations within an application or system. For example, in machine learning applications or projects, tracking can involve keeping a record of parameters, hyperparameters, metrics, outcomes across different experiments or runs. It provides a way to document the progress and changes over time.

> **Tracing** is a more specialized form of tracking. It involves recording the execution flow through software/systems. Particularly in distributed systems where a single transaction might span multiple services, tracing helps in maintaining an audit or breadcrumb trail, a detailed information about that request path through the system. This granular view enables developers to understand the interaction between various microservices and troubleshoot issues like latency or failures by identifying exactly where they occurred in the transaction path.

Tracking the trajectory of agents can be challenging due to their broad range of actions and generative capabilities. LangChain comes with functionality for trajectory tracking and evaluation. Seeing the traces of an agent is actually really easy! You just have to set the return_ `intermediate_steps` parameter to `True` when initializing an agent or an LLM. Let's have a quick look at this. I'll skip the imports and setting up the environment. You can find the full listing on github under monitoring at this address: https://github.com/benman1/generative_ai_with_langchain/We'll define a tool. It's very convenient to use the `@tool` decorator, which will use the function docstring as description of the tool. The first tool sends a ping to a website address and returns information about packages transmitted and latency or – in the case of an error – the error message:

```
@tool
def ping(url: HttpUrl, return_error: bool) -> str:
    """Ping the fully specified url. Must include https:// in the url."""
    hostname = urlparse(str(url)).netloc
    completed_process = subprocess.run(
        ["ping", "-c", "1", hostname], capture_output=True, text=True
    )
    output = completed_process.stdout
    if return_error and completed_process.returncode != 0:
        return completed_process.stderr
    return output]
```

Now we set up an agent that uses this tool with an LLM to make the calls given a prompt:

```
llm = ChatOpenAI(model="gpt-3.5-turbo-0613", temperature=0)
agent = initialize_agent(
    llm=llm,
    tools=[ping],
    agent=AgentType.OPENAI_MULTI_FUNCTIONS,
```

```
    return_intermediate_steps=True,  # IMPORTANT!
)
result = agent("What's the latency like for https://langchain.com?")
```

The agent reports this:

```
The latency for https://langchain.com is 13.773 ms
```

In `results["intermediate_steps"]` we can see all lot of information about the agent's actions:

```
[(_FunctionsAgentAction(tool='ping', tool_input={'url': 'https://langchain.com', 'return_erro
```

By providing visibility into the system and aiding in problem identification and optimization efforts, this kind of tracking and evaluation can be very helpful. The LangChain documentation demonstrates how to use a trajectory evaluator to examine the full sequence of actions and responses they generate, and grade an OpenAI functions agent. Let's have a look beyond LangChain and see what's out there for observability!

Observability tools

There are quite a few tools available as integrations in LangChain or through callbacks:

- **Argilla**: Argilla is an open-source data curation platform that can integrate user feedback (human-in-the-loop workflows) with prompts and responses to curate datasets for fine-tuning.
- **Portkey**: Portkey adds essential MLOps capabilities like monitoring detailed metrics, tracing chains, caching, and reliability through automatic retries to LangChain.
- **Comet.ml**: Comet offers robust MLOps capabilities for tracking experiments, comparing models and optimizing AI projects.
- **LLMonitor**: Tracks lots of metrics including cost and usage analytics (user tracking), tracing, and evaluation tools (open-source).
- **DeepEval**: Logs default metrics like relevance, bias, and toxicity. Can also help in testing and in monitoring model drift or degradation.
- **Aim**: An open-source visualization and debugging platform for ML models. It logs inputs, outputs, and the serialized state of components, enabling visual inspection of individual LangChain executions and comparing multiple executions side-by-side.
- **Argilla**: An open-source platform for tracking training data, validation accuracy, parameters, and more across machine learning experiments.
- **Splunk**: Splunk's Machine Learning Toolkit can provide observability into your machine learning models in production.
- **ClearML**: An open-source tool for automating training pipelines, seamlessly moving from research to production.
- **IBM Watson OpenScale**: A platform providing insights into AI health with fast problem identification and resolution to help mitigate risks.
- **DataRobot MLOps**: Monitors and manages models to detect issues before they impact performance.
- **Datadog APM Integration**: This integration allows you to capture LangChain requests, parameters, prompt-completions, and visualize LangChain operations. You can also capture metrics such as request latency, errors, and token/cost usage.
- **Weights and Biases (W&B) Tracing**: We've already shown an example of using (W&B) for monitoring of fine-training convergence, but it can also fulfill the role of tracking other metrics and of logging and comparing prompts.
- **Langfuse**: With this open-source tool, we can conveniently monitor detailed information along the traces regarding latency, cost, scores of our LangChain agents and tools.

Most of these integrations are very easy to integrate into LLM pipelines. For example, For W&B, you can enable tracing by setting the `LANGCHAIN_WANDB_TRACING` environment variable to `True`. Alternatively, you can use a context manager with `wandb_tracing_enabled()` to trace a specific block of code. With Langfuse, we can hand over a `langfuse.callback.CallbackHandler()` as an argument to the `chain.run()` call.Some of these tools are open-source, and what's great about these platforms is that it allows full customization and on-premise deployment for use cases, where privacy is important. For example, Langfuse is open-source and provides an option of self-hosting. Choose the option that best suits your needs and follow the instructions provided in the LangChain documentation to enable tracing for your agents.Having been released only recently, I am sure there's much more to come for the platform, but it's already great to see traces of how agents execute, detecting loops and latency issues. It enables sharing traces and stats with collaborators to discuss improvements.

LangSmith

LangSmith is a framework for debugging, testing, evaluating, and monitoring LLM applications developed and maintained by LangChain AI, the organization behind LangChain. LangSmith serves as an effective tool for MLOps, specifically for LLMs, by providing features that cover multiple aspects of the MLOps process. It can help developers take their LLM applications from prototype to production by providing features for debugging, monitoring, and optimizing. LangSmith aims to reduce the barrier to entry for those without a software background by providing a simple and intuitive user interface. LangSmith is a platform for debugging, testing, and monitoring large language models (LLMs) built with LangChain. It allows you to:

- Log traces of runs from your LangChain agents, chains, and other components
- Create datasets to benchmark model performance
- Configure AI-assisted evaluators to grade your models
- View metrics, visualizations, and feedback to iterate and improve your LLMs

LangSmith fulfils the requirements for MLOps for agents by providing features and capabilities that enable developers to debug, test, evaluate, monitor, and optimize language model applications. Its integration within the LangChain framework enhances the overall development experience and facilitates the full potential of language model applications. By using both two platforms, developers can take their LLM applications from prototype to production stage and optimize latency, hardware efficiency, and cost.We can get a large set of graphs for a bunch of important statistics as we can see here:

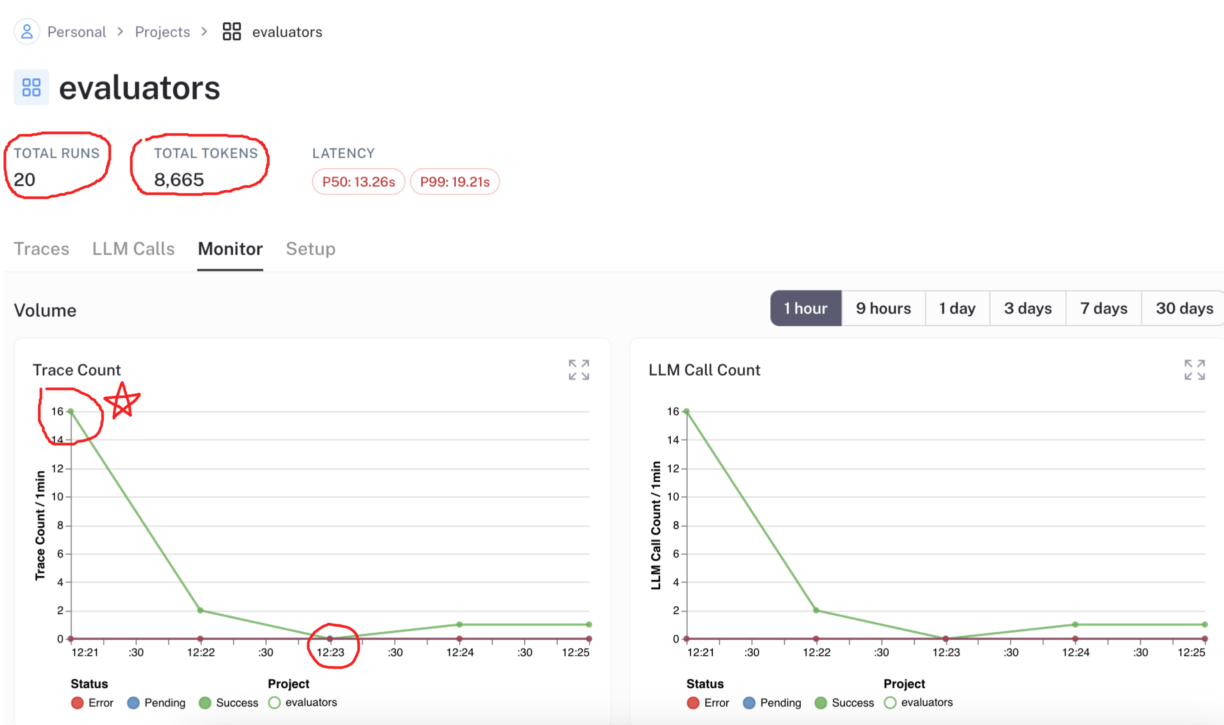


Figure 9.5: Evaluator metrics in LangSmith.

The monitoring dashboard includes the following graphs that can be broken down into different time intervals:

Statistics	Category
Trace Count, LLM Call Count, Trace Success Rates, LLM Call Success Rates	Volume
Trace Latency (s), LLM Latency (s), LLM Calls per Trace, Tokens / sec	Latency
Total Tokens, Tokens per Trace, Tokens per LLM Call	Tokens
% Traces w/ Streaming, % LLM Calls w/ Streaming, Trace Time-to-First-Token (ms), LLM Time-to-First-Token (ms)	Streaming

Figure 9.6: Statistisc in LangSmith.

Here's a tracing example in LangSmith for the benchmark dataset run that we've seen in the section on evaluation:

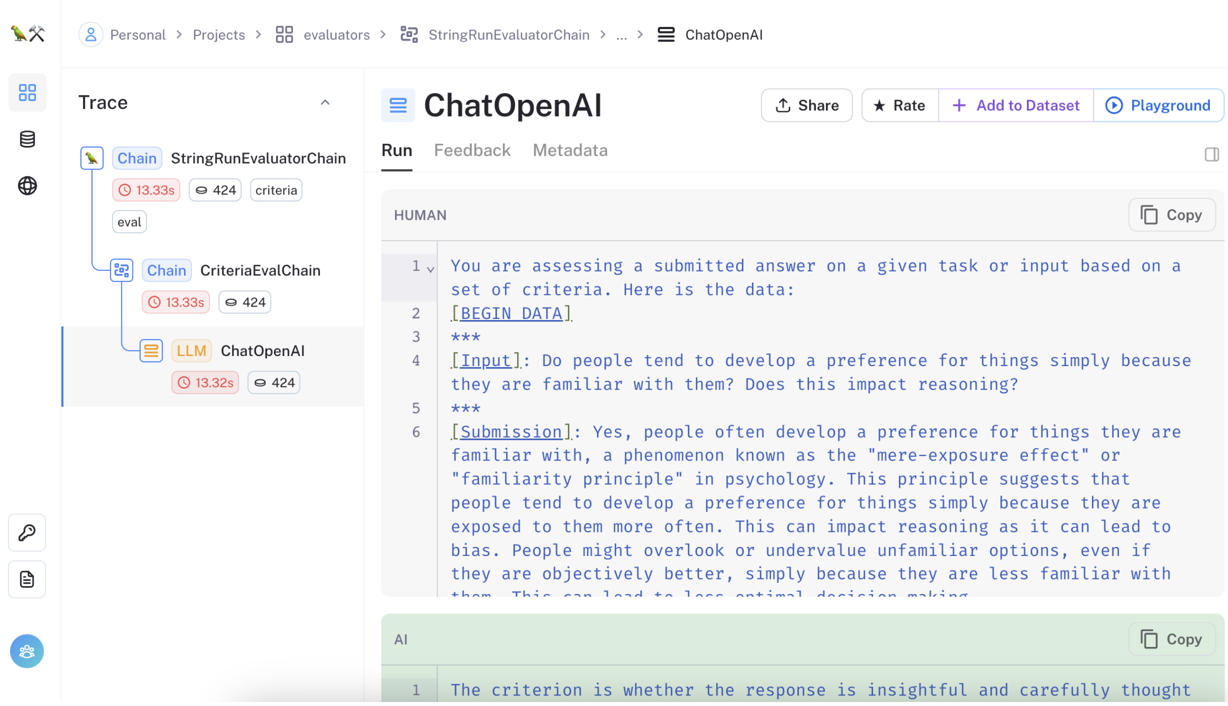


Figure 9.7: Tracing in LangSmith.

The platform itself is not open source, however, LangChain AI, the company behind LangSmith and LangChain, provide some support for self-hosting for organizations with privacy concerns. There are however a few alternatives to LangSmith such as Langfuse, Weights and Biases, Datadog APM, Portkey, and PromptWatch, with some overlap in features. We'll focus about LangSmith here, because it has a large set of features for evaluation and monitoring, and because it integrates with LangChain.In the next section, we'll demonstrate the utilization of PromptWatch.io for prompt tracking of LLMs in production environments.

PromptWatch

PromptWatch records information about the prompt and the generated output during this interaction.Let's get the inputs out of the way.

```
from langchain import LLMChain, OpenAI, PromptTemplate
from promptwatch import PromptWatch
from config import set_environment
set_environment()
```

As mentioned in Chapter 3, I've set all API keys in the environment in the set_environment() function.

```
prompt_template = PromptTemplate.from_template("Finish this sentence {input}")
my_chain = LLMChain(llm=OpenAI(), prompt=prompt_template)
```

Using the `PromptTemplate` class, the prompt template is set to with one variable, `input`, indicating where the user input should be placed within the prompt.Inside the `PromptWatch` block, the `LLMChain` is invoked with an input prompt as an example of the model generating a response based on the provided prompt.

```
with PromptWatch() as pw:
    my_chain("The quick brown fox jumped over")
```

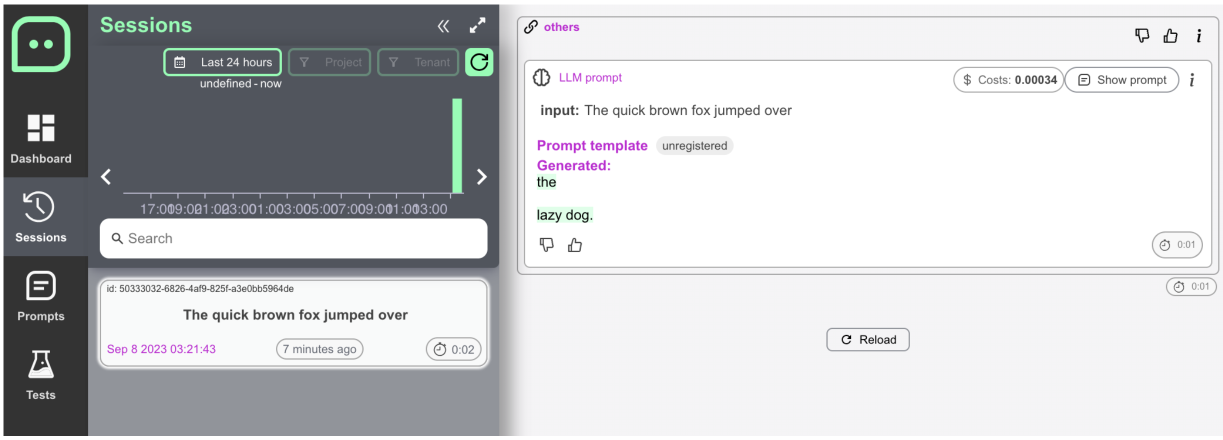


Figure 9.8: Prompt tracking at PromptWatch.io.

This seems quite useful. By leveraging PromptWatch.io, developers and data scientists can effectively monitor and analyze LLMs' prompts, outputs, and costs in real-world scenarios.PromptWatch.io offers comprehensive chain execution tracking and monitoring capabilities for LLMs. With PromptWatch.io, you can track all aspects of LLM chains, actions, retrieved documents, inputs, outputs, execution time, tool details, and more for complete visibility into your system. The platform allows for in-depth analysis and troubleshooting by providing a user-friendly, visual interface that enables users to identify the root causes of issues and optimize prompt templates. PromptWatch.io can also help with unit testing and for versioning prompt templates.Let's summarize this chapter!

Summary

Successfully deploying LLMs and other generative AI models in a production setting is a complex but manageable task that requires careful consideration of numerous factors. It requires addressing challenges related to data quality, bias, ethics, regulatory compliance, interpretability, resource requirements, and ongoing monitoring and maintenance, among others.The evaluation of LLMs is an important step in assessing their performance and quality. LangChain supports comparative evaluation between models, checking outputs against criteria, simple string matching, and semantic similarity metrics. These provide different insights into model quality, accuracy, and appropriate generation. Systematic evaluation is key to ensuring large language models produce useful, relevant, and sensible outputs. Monitoring LLMs is a vital aspect of deploying and maintaining these complex systems. With the increasing adoption of LLMs in various applications, ensuring their performance, effectiveness, and reliability is of utmost importance. We've discussed the significance of monitoring LLMs, highlighted key metrics to track for a comprehensive monitoring strategy, and have given examples of how to track metrics in practice.LangSmith provides powerful capabilities to track, benchmark, and optimize large language models built with LangChain. Its automated evaluators, metrics, and visualizations help accelerate LLM development and validation.Let's see if you remember the key points from this chapter!

Questions

Please have a look to see if you can come up with the answers to these questions from. If you are unsure about any of them, you might want to refer to the corresponding section in the chapter:

1. In your opinion what is the best term for describing the operationalization of language models, LLM apps, or apps that rely on generative models in general?

2. How can we evaluate LLMs apps?
3. Which tools can help for evaluating LLM apps?
4. What are considerations for production deployment of agents?
5. Name a few tools for deployment?
6. What are important metrics for monitoring LLMs in production?
7. How can we monitor these models?
8. What’s LangSmith?

10 The Future of Generative Models

In this book, so far, we’ve discussed generative models for building applications. We’ve explored LLMs and image models for content creation, tool use, agent strategies, semantic search with retrieval augmented generation, and conditioning of models with prompts and fine-tuning. Further, we’ve implemented a few simple applications, for example, for developers and data scientists. In this chapter, we’ll discuss where this leaves us and where the future leads us.The pace of progress in AI has accelerated dramatically in the past year, with breakthroughs like DALL-E, Midjourney, and ChatGPT producing astounding results. These generative AI models can create photorealistic images, write essays and code, and have conversational abilities surpassing most humans. Venture funding for generative AI startups skyrocketed in 2022, almost matched the total investments from the previous five years combined. Recently, major players like Salesforce and Accenture have made big commitments to generative AI with multibillion dollar investments. Unique customization of foundation models for specific use cases is seen as the real value creation opportunity. But it remains uncertain which entities - big tech firms, startups, or foundation model developers - will capture most upside.On a technical level, generative models like ChatGPT often function as black boxes, with limited transparency into their decision-making processes. A lack of model interpretability makes it difficult to fully understand model behavior or to control outputs. There are also concerns around potential biases that could emerge from imperfect training data. On a practical level, generative models require extensive computational resources for training and deployment. For many organizations, acquiring the infrastructure to effectively utilize these AI systems remains a barrier.On the positive side, AI can democratize skills, allowing amateurs to produce professional quality output in design, writing, etc. Businesses can benefit from faster, cheaper, on-demand work. However, there are major concerns around job losses, especially for specialized middle-class roles like graphic designers, lawyers and doctors. Their work is being automated while low skilled workers learn to leverage AI as a superpower. More ominously, AI could be weaponized by militaries, terrorists, criminals and governments for propaganda and influence. Deepfakes produced in real-time will proliferate scams and erode trust. The path forward balances enthusiasm with practicality, prioritizing human dignity. By acknowledging risks, fostering open discussion, and enacting thoughtful policies, we can build an equitable future enabled by AI's enlivening possibilities.The main sections of this chapter are:

- Current State of Generative AI
- Possible Future Capabilities
- Societal Implications
- Practical Implementation
- The Road Ahead

Let's start from the current state of models and their capabilities.

Current State of Generative AI

As discussed in this book, in recent years, generative AI models have attained new milestones in producing human-like content across modalities including text, images, audio and video. Leading models like OpenAI's GPT-4 and DALL-E 2, Google's Imagen and Parti, and Anthropic's Claude display impressive fluency in language generation along with creative visual artistry.Between 2022 and 2023, models have progressed in strides. If generative models were previously capable to produce barely coherent text or grainy images, now we see high-quality 3D models, videos, and generate coherent and contextually relevant prose and dialogue, rivaling or even surpassing the fluency levels of humans. These AI models leverage gargantuan datasets and computational scale, enabling them to capture intricate linguistic patterns, display a nuanced understanding of knowledge about the world, translate texts, summarize content, answer natural language questions, create appealing visual art, and acquire the capability to describe images. Seemingly by magic, the AI generated outputs mimic human ingenuity — painting original art, writing poetry, producing human-level prose, and even engaging in sophisticated aggregation and synthesis of information from diverse sources. But let's be a bit more nuanced. Generative Models come with weaknesses as well as strengths. Deficiencies still persist compared to human cognition, including the frequent generation of plausible yet incorrect or nonsensical statements. Hallucinations show a lack of grounding in reality, given that they are based on patterns in data rather than an understanding of the real world. Further, models exhibit difficulties performing mathematical, logical, or causal reasoning. They are easily confused by complex inferential questions, which could limit their applicability in certain fields of work. The black box problem of lack of explainability for predictions as well as the models themselves hampers troubleshooting efforts, and controlling model behaviors within desired parameters remains challenging. AI models may exhibit harmful unintended biases that pose significant ethical concerns—a problem greatly due to the biases present in the training data itself. This issue of bias not only skews output but can propagate and amplify societal disparities.Here is a table visualizing the key strengths and deficiencies of current LLMs compared to human cognition:

Strengths of LLMs	**Deficiencies of LLMs**
Language Fluency - Ability to generate grammatically coherent, contextual prose and dialogue. GPT-4 produces human-level prose.	Factual Accuracy - LLMs frequently generate plausible but incorrect or nonsensical statements. Lack of grounding in reality.
Knowledge Synthesis - Sophisticated aggregation and presentation of information from diverse sources.	Logical Reasoning - Inability to perform mathematical, logical or causal reasoning. Easily confused by complex inferential questions.

Strengths of LLMs	Deficiencies of LLMs
Creative Output - Imaginative and original text, art, music reflecting human ingenuity. Claude writes poetry, DALL-E 2 paints original art.	Controllability - Difficulty constraining model behaviors within desired parameters. Can exhibit harmful unintended biases.
	Bias - Potential to propagate and amplify societal biases present in training data. Raises ethical concerns.
	Transparency - Lack of explainability for model predictions. The "black box" problem limits troubleshooting.

Figure 10.1: Strengths and Deficiencies of LLMs.

While generative AI capabilities have come a long way, their problematic areas need addressing for these technologies to effectively function in the future. Nonetheless, their profound potential indicates an exciting future if developed and regulated responsibly.The weaknesses of generative models define some of the technical challenges, as we'll see now.

Technical Challenges

While rapid progress has been made, significant technical obstacles remain to realize the full potential of generative AI safely and responsibly. As mentioned, generative AI models, despite their considerable advances, are grappling with significant technical challenges that need to be overcome to allow their full potential to be harnessed safely and responsibly. We've discussed some of these issues and potential solutions in previous chapters.This table shows a summary for a few of these challenges together with the technical approaches to tackle them:

Challenge	Description	Potential Solutions

Challenge	Description	Potential Solutions
Realistic and Diverse Content Generation	Existing models struggle with logical consistency and factual plausibility. Generates repetitive, bland samples lacking human nuance.	RLHF Reinforcement learning from human feedback Data augmentation and synthesis techniques Modular domain knowledge
Output Quality Control	Lack of mechanisms to reliably constrain properties of generated content. Models sporadically produce harmful, biased or nonsensical results.	Constrained optimization objectives Moderation systems Interruption and correction techniques
Avoiding Bias	Models inadvertently amplify societal biases present in training data. Developing techniques to curtail prejudice remains difficult.	Balanced and representative training data Bias mitigation algorithms Ongoing testing and audits

Challenge	Description	Potential Solutions
Factual Accuracy	Inability to reason about objective truths limits reliability for real-world applications. Grounding models in common sense and physics is an open problem.	Incorporating knowledge bases Knowledge Graph Hybrid neuro-symbolic architectures Retrieval augmented generation
Explainability	The opaque behavior of large neural networks poses hurdles for troubleshooting failures or bias, necessitating explainable AI techniques.	Model introspection techniques Concept attribution methods Simplified model architectures
Data Privacy	Collecting and processing massive datasets raises challenges around consent, anonymization, access control and misuse of data.	Differential privacy and secure multi-party computation Synthetic data generation Federated learning

Challenge	Description	Potential Solutions
Latency and Compute	Deploying huge models requires substantial computing resources, delaying real-time interactivity needed for many applications.	Model distillation into smaller form factors Optimized inference engines Dedicated AI hardware accelerators
Data Licenses	Organizations may need to obtain a commercial license to use existing datasets or to build bespoke datasets to train generative models. This can be a complex and time-consuming process.	Open-source and synthetic data

Figure 10.2: Technical Challenges and Potential Solutions.

Primarily, the content generated by these models is often hindered by a lack of realism and diversity. While they have displayed impressive abilities to mimic human-like language and creativity, they still falter when it comes to producing content that is logically consistent and factually plausible. Their outputs often lack human nuance turning out to be quite repetitive and bland.

A

Potential solutions include reinforcement learning from human feedback to improve coherence and nuance, controlled data augmentation and synthesis techniques, and architectures incorporating modular domain knowledge. Another critical hurdle is the control of output quality. Despite rigorous training and development, existing AI mechanisms fall short in reliably constraining properties of the generated content. This results in sporadic production of content that can be harmful, biased or outright nonsensical, posing a risk to their wider acceptance and application.

B

Promising approaches involve 1 constrained optimization objectives, 2 human-in-the-loop moderation systems, and 3 techniques to interrupt and correct model output during generation. Bias is indeed a major issue with these AI models as they frequently and inadvertently amplify societal prejudices present in their training data. Developing corrective techniques to curtail 减少 such biases remains a complicated issue. Strategies like balanced and representative training data, bias mitigation algorithms, and ongoing testing and audits for fairness seek to address this problem.The inability of these AI models to reason about objective truths noticeably limits their reliability for real-world applications. Grounding these models in common sense and physics represents an open problem that the AI community is still grappling with. Hybrid neuro-symbolic architectures, incorporation of knowledge bases, and retrieval augmented generation offer promising directions.The black box nature of AI presents another complex challenge: explainability. The opaque behavior of large neural networks poses hurdles for troubleshooting failures or bias, which emphasizes the need for more transparent AI techniques. Model introspection, concept attribution methods, and simplified model architectures could provide solutions.Furthermore, the issue of data privacy rises to prominence due to the collection and processing of extensive datasets. This aspect introduces challenges around consent, anonymization, access control and misuse of data. Techniques like differential privacy, secure multi-party computation, synthetic data generation, and federated learning may help address privacy risks.Last but not least, deploying these enormous models demands substantial computing resources, leading to significant latency and compute issues. This could delay the real-time interactivity required for many applications, indicating that efficiency improvements are key. Solutions involve model distillation into smaller form factors, optimized inference engines, and dedicated AI hardware accelerators.Looking ahead, generative AI systems are poised to become more powerful and multifaceted. Let's see how!

Possible Future Capabilities

The current doubling time in training compute of very large models is about 8 months, outstripping scaling laws such as Moore's Law (transistor density at cost increases at a rate of currently about 18 months) or Rock's Law (costs of hardware like GPUs and TPUs halve every 4 years).This graph illustrates this trend in training compute of large models (Source: Epoch, *Parameter, Compute and Data Trends in Machine Learning*. Published online at epochai.org. Retrieved from: https://epochai.org/mlinputs/visualization):

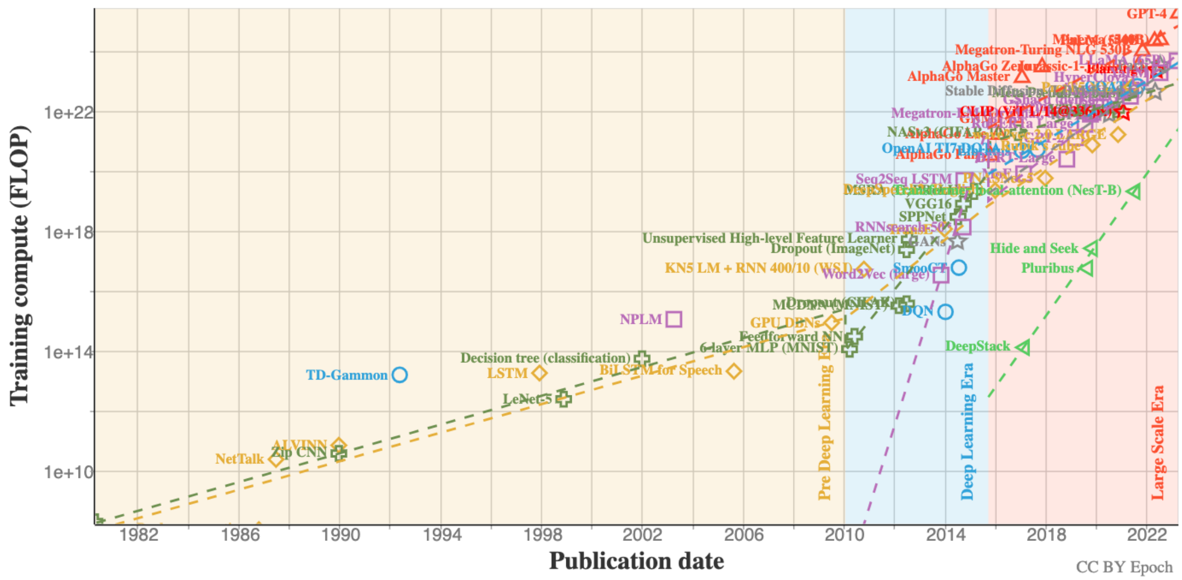


Figure 10.3: Training FLOPs of notable AI systems.

As discussed in chapter 1, parameter sizes for large systems have been increasing at a similar rate as the training compute, which means that we could be seeing much larger and more expensive systems if this growth continues. Empirically derived scaling laws predict the performance of LLMs based on the training budget, dataset size, and the number of parameters. This could mean that highly powerful systems would be concentrated in the hands of Big Tech.

> The **KM scaling** law, proposed by Kaplan and colleagues, derived through empirical analysis and fitting of model performance with varied data sizes, model sizes, and training compute, presents power-law relationships, indicate a strong dependence, between model performance and factors such as model size, dataset size, and training compute.
>
> > The **Chinchilla scaling law**, developed by the Google DeepMind team, involved experiments with a wider range of model sizes and data sizes, and suggests an optimal allocation of compute budget to model size and data size, which can be determined by optimizing a specific loss function under a constraint.

However, future progress may depend more on data efficiency and model quality than sheer size. Though massive models grab headlines, computing power and energy constraints put a limit on unrestrained model growth. The future will see co-existence of massive, general models with smaller and accessible specialized niche models that provide faster and cheaper training, maintenance, and inference. It has already been shown that smaller specialized models can prove highly performant. We've recently seen models such as phi-1 (*Textbooks Are All You Need*, 2023, Gunasekar and colleagues), on the order of 1 billion parameters, that – despite its smaller scale – achieve high accuracy on evaluation benchmarks. The authors suggest that improving data quality can dramatically change the shape of scaling laws.More work has shown that models can be substantially smaller with only a modest drop in accuracy (*One Wide Feedforward is All You Need*, Pessoa Pires and others, 2023), which supports the argument for a democratization of model training and access. Further, techniques such as transfer learning, distillation and prompting techniques can enable smaller models to leverage capabilities of large foundations without replicating their costs. In order to compensate for limitations, tools like search engines and calculators have been incorporated into agents and multi-step reasoning strategies, plugins, and extensions may be increasingly used to expand capabilities.AI training costs are dropping because of different factors – according to ARK Investment Management LLC, about 70% per year. A recently released AI training tools by Mosaic ML can train language models to GPT-3 level performance for roughly one-tenth the estimated $4.6 million just two years ago. This will enable experimentation, but advances will increasingly emerge from training regimes, data quality, and novel architectures rather than model size alone. After an arms race dominated by resource-rich big tech firms, responsible, economical innovation may become the priority.In a timeframe of 3-5 years (2025-2027), constraints around computing and talent availability could ease considerably, eroding the centralized moat. Specifically, if cloud computing costs decline as projected, and AI skills become more widespread through education and automated tools, self-training customized LLMs may become feasible for many companies. This could better serve needs for personalization and data privacy.Some abilities, however, such as in-context learning, are not predictable according to the scaling laws and only emerge in large models. It has been further speculated that enormous models trained on even more data may exhibit more behaviors and skills, where extreme scaling could eventually produce **artificial general intelligence** (**AGI**) – reasoning on par or beyond human intellect. However, from a neuroscience perspective, the threat of AGI taking over the world seems highly exaggerated at our present stage of technology (compare Jaan Aru and others, *The feasibility of artificial consciousness through the lens of neuroscience*; 2023):

- **Lack of embodied, embedded information**: LLMs today are trained purely on textual data rather than the rich multimodal inputs that allow humans to develop common sense reasoning about the physical world. This absence of grounded learning poses a major obstacle to developing human-level intelligence.
- **Different architecture from biological brains**: The relatively simple stacked transformer architecture used in models like GPT-4 lacks the complex recurrent and hierarchical structures of the thalamocortical system thought to enable consciousness and general reasoning in humans.
- **Narrow capabilities**: Existing models remain specialized for particular domains like text and fall short in flexibility, causal reasoning, planning, social skills, and general problem-solving intelligence. This could change either with increasing tool use or with fundamental changes to the models.
- **Minimal social abilities or intent**: Current AI systems have no innate motivations, social intelligence, or intent beyond their training objectives. Fears of malicious goals or desire for domination seem unfounded.
- **Limited real-world knowledge**: Despite ingesting huge datasets, the factual knowledge and common sense of large models remains very restricted compared to humans. This impedes applicability in the physical world.
- **Data-driven limitations**: Reliance on pattern recognition from training data rather than structured knowledge makes reliable generalization to novel situations difficult.

Given these arguments, the risk of today's AI precipitously evolving into malicious superintelligence seems highly improbable. That said, thoughtfully addressing longer-term safety research and ethics remains prudent as capabilities continue advancing. But fears of imminent world takeover are not substantiated by evidence from neuroscience or current model capabilities. Therefore, claims of inevitable, imminent AGI lack rigorous support.However, the sheer pace of advancement creates unease surrounding human obsolescence and job displacement, which could further divide economic classes. Unlike physical automation of the past, generative AI threatens cognitive job categories previously considered safe from automation. Managing this workforce transition ethically and equitably will require foresight and planning. There are also philosophical debates around whether AI should be creating art, literature or music that has historically reflected the human condition. Let’s think a bit more broadly about the societal impact!

Societal Implications

The advent of highly capable generative AI will likely transform many aspects of society in coming years. As generative models continue to develop and add value to businesses and creative projects, generative AI will shape the future of technology and human interaction across domains. While their widespread adoption brings forth numerous benefits and opportunities for businesses and individuals, it is crucial to address the ethical and societal concerns that arise from increasing reliance on AI models in various fields.Generative AI offers immense potential benefits across personal, societal, and industrial realms if deployed thoughtfully. At a personal level, these models can enhance creativity and productivity, and increase accessibility to services like healthcare, education and finance. Democratizing access to knowledge resources, they can help students learn or aid professionals make decisions by synthesizing expertise. As virtual assistants, they provide instant, customized information to facilitate routine tasks. By automating rote tasks, they may free up human time for higher-value work, boosting economic output. Economically, the gains in productivity will most likely result in massive disruptions of certain job categories. New industries and jobs may emerge to support AI systems. Thoughtfully considering and addressing these changes is crucial. As models become better and running them becomes cheaper, this could trigger a massive expansion of generative AI and LLM applications to new areas. On top of the reduction in hardware costs, with every cumulative doubling of AI systems produced, costs may fall by 10-30% according to Wright's Law. This cost curve reflects efficiencies like reuse of code, tools and techniques. A virtuous cycle arises as lower costs expand adoption, which drives further cost reductions. This will lead to a feedback cycle of more efficiency driving more use driving more efficiency.

cost ↓ 20%
prods ↑ 100%
0: x, y
If: n given
n: $x(1-20\%)^n$
$2^n y$

Wright's Law, also known as the **experience curve effect**, is an observation in economics and business that states that for many products, costs decline by a fixed percentage each time cumulative production doubles. Specifically, it states that costs tend to decrease by a fixed percentage (typically ranging from 10-30%) for each cumulative doubling of production.

The law is named after Theodore Paul Wright, an American aircraft engineer who first observed this phenomenon in 1936 when analyzing trends in aircraft production costs. Wright noticed that every time the cumulative production of airframes doubled, the labor required to produce them decreased by 10-15%.

This relationship can be expressed mathematically as:

$$C_x = C_1 x^{\log_2 b},$$

Where C_1 represents the cost to produce the first unit, C_x the cost to produce the xth unit, and b is the progress ratio, which has been estimated across numerous industries between 0.75 to 0.9.

The logic behind Wright's Law is that as production increases, workers become more efficient in manufacturing a product through practice, standardized workflows, and developing better tools and processes. Companies also identify ways to optimize supply chains, logistics and resource utilization to reduce costs.

Industrially, the models bring extensive opportunities to augment human capabilities and remake workflows. In content production, generative AI can draft initial versions for marketing campaigns or journalistic pieces faster than humans, enabling greater creativity and customization. For developers, autogenerated code and rapid iterations accelerate software building. Researchers can quickly synthesize discoveries from papers to advance science.Generative AI also facilitates new levels of personalization at scale for consumers. Recommendations can be tailored down to the individual. Marketing across segments and geographies can be customized. Overall, these models can enhance productivity across sectors from industrial design to supply chains.As for the spread of the technology, two primary scenarios exist. In the first scenario, each company or individual trains their own tailored model using their proprietary data. However, this requires considerable AI/ML expertise to properly develop, train and deploy such systems - talent that remains scarce and expensive currently. The computational costs are also extremely high, with specialized hardware like clusters of GPUs costing substantial sums only feasible for large entities. There are further risks around data privacy compliance when models are trained on sensitive information. If these barriers around expertise, computing requirements and data privacy can be overcome, personalized LLMs fine-tuned to an organization's particular goals and data could significantly enhance their productivity and efficiency by automating routine tasks and offering insights customized to the specific business. However, a downside is that models trained on small, private datasets may lack the generalization capability of those trained on much larger diverse public corpuses.Both centralized and self-service models can co-exist serving different use cases. In the near term, large tech firms have strengths in providing industry-specific fine-tuning services given their resources. But over time, more in-house training may emerge driven by customization and privacy needs.The pace of progress on reducing costs, spreading expertise, and addressing robustness challenges will determine how long any centralized advantage persists. Rapid innovations in these areas favor erosion of the moat, but platform effects around dominant frameworks, datasets and models could enable continued concentration among current leaders.If robust tools emerge to simplify and automate AI development, custom generative models may even be viable for local governments, community groups, and individuals to address hyper-local challenges. While centralized Big Tech firms benefit currently from economies of scale, distributed innovation from smaller entities could unlock generative AI's full potential across all sectors of society.Finally, the emergence of generative AI intersects with broader shifts in how we produce and consume creative works. The internet has already fostered a remix culture where derivative works and collaborative content creation are commonplace. As AI models generate new artifacts by recombining existing materials, they align with remix culture principles of iterative, collective production.However, the scale at which generative models can synthesize and repurpose copyrighted content raises challenging legal questions. With models trained on vast datasets of books, articles, images and more, attributing rights and royalties could become incredibly complex. Current detection mechanisms are unable to find content authored by generative AI at above chance level. This speaks to wider debates surrounding authorship and copyright law.Let's look at the different aspects, where generative models will have profound near-term impacts, starting with creative endeavors.

Creative industries and advertising

The gaming and entertainment industries are leveraging generative AI to craft uniquely immersive user experiences. Major efficiency gains from automating creative tasks could increase leisure time spent online. Generative AI can enable machines to generate new and original content, such as art, music, and literature, by learning from patterns and examples. This has implications for creative industries, as it can enhance the creative process and potentially create new revenue streams. It also unlocks new scales of personalized, dynamic content creation for media, film, and advertising. However, generative content requires extensive quality control around accuracy and eliminating biases before full deployment.For media, film, and advertising, AI unlocks new scales of personalized, dynamic content creation. In journalism, automated article generation using massive datasets can free up reporters to focus on more complex investigative stories. **AI-generated content (AIGC)** is playing a growing role in transforming media production and delivery by enhancing efficiency and diversity. In journalism, text generation tools automate writing tasks traditionally done by human reporters, significantly boosting productivity while maintaining timeliness. Media outlets like Associated Press generate thousands of stories per year using AIGC. Robot reporters like Los Angeles Times' Quakebot can swiftly produce articles on breaking news. Other applications include Bloomberg News' Butletin service where chatbots create personalized one-sentence news summaries. AIGC also enables AI news anchors that co-present broadcasts with real anchors by mimicking human appearance and speech from text input. Chinese news agency Xinhua's virtual presenter Xin Xiaowei is an example, presenting broadcasts from different angles for an immersive effect.AIGC is transforming movie creation from screenwriting to post-production. AI screenwriting tools analyze data to generate optimized scripts. Visual effects teams blend AI-enhanced digital environments and de-aging with live footage for immersive visuals. Deep fake technology recreates or revives characters convincingly.AI also powers automated subtitle generation, even predicting dialogue in silent films by training models on extensive audio samples. This expands accessibility via subtitles and recreates voiceovers synchronized to scenes. In post-production, AI color grading and editing tools like Colourlab.Ai and Descript simplify processes like color correction using algorithms.In advertising, AIGC unlocks new potential for efficient, customized advertising creativity and personalization. AI-generated content allows advertisers to create personalized, engaging ads tailored to individual consumers at scale. Platforms like Creative Advertising System (CAS) and Personalized Advertising Copy Intelligent Generation System (SGS-PAC) leverage data to automatically generate ads with messaging targeted to specific user needs and interests.AI also assists in advertising creativity and design – Tools like Vinci produce customized attractive posters from product images and slogans, while companies like Brandmark.io generate logo variations based on user preferences. GAN technologies automate product listing generation with keywords for effective peer-to-peer marketing. Synthetic ad production is also on the rise, enabling highly personalized, scalable campaigns that save time.In music, tools like Google's Magenta, IBM's Watson Beat, or Sony CSL's Flow Machine can generate original melodies and compositions. AIVA similarly creates unique compositions from parameters tuned by users. LANDR's AI mastering uses machine learning to process and improve digital audio quality for musicians.In visual arts, MidJourney uses neural networks to generate inspirational images that can kickstart painting projects. Artists have used its outputs to create prize-winning works. DeepDream's algorithm imposes patterns on images, creating psychedelic art. GANs can generate abstract paintings converging on a desired style. AI painting conservation analyzes artwork to digitally repair damage and restore pieces.Animation tools like Adobe's Character Animator or Anthropic's Claude can help with the generation of customized characters, scenes and motion sequences opening animation potential for non-professionals. ControlNet adds constraints to steer diffusion models, increasing output variability. For all these applications, advanced AI expands creative possibilities through both generative content and data-driven insights. It is important to note however that generative content requires extensive quality control around accuracy and eliminating biases before full deployment. For advertising, ethical use of consumer data and human oversight remain important. In all cases, properly attributing contributions of human artists, developers and training data remains an ongoing challenge as adoption spreads.

Economic

The deployment of generative AI and other technologies could help accelerate productivity growth, partially compensating for declining employment growth and enabling overall economic growth. Assuming energy and computing can scale sustainably, the huge productivity gains from integrating generative AI into business processes seem likely to usher automation of many tasks over the next decade. However, this transition may disrupt labor markets, requiring adjustments. Research by McKinsey and Company estimates 30-50% of current work activities could be automated by 2030-2060. Generative AI could boost global productivity by $6-8 trillion annually by 2030, adding 15-40% onto previous estimates for AI's economic impact. According to Tyna Eloundou and colleagues (*GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models*, 2023) around 80% of US workers have at least 10% of work tasks affected by LLMs, while 19% may have over 50% of tasks impacted. Effects span all wage levels, with higher-wage jobs facing more exposure. Just 15% of all US worker tasks could be done significantly faster with LLMs alone. But with LLM-powered software, this increases to 47-56% of all tasks, showing the big impact of complementary technologies.From a geographic perspective, external private investment in generative AI, mostly from tech giants and venture capital firms, is largely concentrated in North America, reflecting the continent's current domination of the overall AI investment landscape. Generative AI–related companies based in the United States raised about $8 billion from 2020 to 2022, accounting for 75 percent of total investments in such companies during that period.Automation adoption is likely to be faster in developed economies, where higher wages will make it economically feasible sooner. The scale of work automation does not directly equate to job losses. Like other technologies, generative AI typically enables individual activities within occupations to be automated, not entire occupations. However, organizations may decide to realize the benefits of increased productivity by reducing employment in some job categories.Productivity growth, the main engine of GDP growth over the past 30 years, slowed down in the past decade. The deployment of generative AI and other technologies could help accelerate productivity growth, partially compensating for declining employment growth and enabling overall economic growth. Based on estimates by analysts at McKinsey and Company, the automation of individual work activities could provide the global economy with an annual productivity boost of 0.2 to 3.3 percent from 2023 to 2040 depending on the rate of automation adoption, however, only if individuals affected by the technology were to shift to other work activities that at least match their 2022 productivity levels.

Education

One potential near-future scenario is that the rise of personalized AI tutors and mentors could democratize access to education for high-demand skills aligned with an AI-driven economy. In the education sector, generative AI is already transforming how we teach and learn. Tools like ChatGPT can be used to automatically generate personalized lessons and customized content for individual students. This reduces instructor workloads substantially by automating repetitive teaching tasks. AI tutors provide real-time feedback on student writing assignments, freeing up teachers to focus on more complex skills. Virtual simulations powered by generative AI can also create engaging, tailored learning experiences adapted to different learners' needs and interests.However, risks around perpetuating biases and spreading misinformation need to be studied further as these technologies evolve. The accelerating pace of knowledge and the obsolescence of scientific findings means that training children's curiosity-driven learning should focus on developing the cognitive mechanisms involved in initiating and sustaining curiosity, such as awareness of knowledge gaps and the use of appropriate strategies to resolve them.While AI tutors tailored to each student could enhance outcomes and engagement, poorer schools may be left behind, worsening inequality. Governments should promote equal access to prevent generative AI from becoming a privilege of the affluent. Democratizing opportunity for all students remains vital.If implemented thoughtfully, personalized AI-powered education could make crucial skills acquisition accessible to anyone motivated to learn. Interactive AI assistants that adapt courses to students' strengths, needs and interests could make learning efficient, engaging and equitable. But challenges around access, biases, and socialization need addressing.

Jobs

Assuming energy and computing can scale sustainably, the huge productivity gains from integrating generative AI into business processes seem likely to usher automation of many tasks over the next decade. This transition may disrupt labor markets, requiring adjustments.Automation enabled by AI will likely displace many administrative, customer service, writing, legal, and creative jobs in the near term, while professions in agriculture and construction might be virtually unaffected. However, past industrial revolutions ultimately led to new types of jobs and industries, albeit with difficult workforce transitions. The same dynamic is likely to play out with AI automation over the long run. This will impact almost all occupations to some degree, though some will be affected more heavily. Generative AI's ability to analyze and generate natural language content could significantly increase the automation potential for activities like communicating, collaborating and reporting across many white-collar occupations. However, the extent to which entire jobs are eliminated remains uncertain. Past technological innovations ultimately created new types of work, even if the transitions were difficult.According to research at McKinsey and company, about 75 percent of the value that generative AI use cases could deliver falls across four areas: Customer operations, marketing and sales, software engineering, and R&D. Prominent examples include generative AI's ability to support interactions with customers, generate creative content for marketing and sales, and draft computer code based on natural-language prompts, among many other tasks.Language models and Generative AI carry the potential to disrupt and automate tasks within various industries that were traditionally performed by humans. As for different roles, these predictions seem credible:

- Junior software engineers may be augmented or replaced by AI coding assistants.
- Analysts and advisors utilizing data insights from AI Customer service agents replaced by conversational AI.
- Technical writers and journalists aided by AI content generation.
- Teachers leveraging AI for course prep and personalized tutoring.
- Paralegals utilizing AI for summarization and document review.
- Graphic designers will be empowered by AI image generation, however, making image creation and manipulation available to many more people could also have an impact on wages.
- However, demand will remain strong for senior software engineers to develop specialized AI solutions and systems.
- Data scientists may pivot from building predictive models to focusing more on validating, debugging and maximizing value from AI systems.
- Programmers will increasingly code tools to assist AI development.
- New roles like prompt engineering have started to emerge.

AI can perform certain tasks that extend to tasks involving natural language processing, content creation, and even intricate creative work, efficiently and with fewer errors than humans. Less skilled individuals may be able to perform more highly skilled work, while highly skilled individuals may be left with fewer job opportunities. For example, paralegals use boilerplate documents and fill in the necessary information to cater to clients' needs. AI, equipped with vast knowledge of legal documents, legislation, university courses, journals, news articles, and court cases, can perform this task even better than paralegals. The result is a potential decrease in the need for junior lawyers for drafting purposes, with paralegals using AI-powered software to communicate clients' specific requirements.Software developers and data scientists alike can benefit from the potential of LLMs but must carefully consider its capabilities and limitations for optimal use. For junior developers and data scientists, LLMs can automate routine tasks, provide basic solutions, and reduce errors, accelerating learning by freeing up time for more complex work. However, relying solely on AI risks hindering deeper technical growth, so LLMs should be seen as supportive tools while actively developing hands-on expertise.Senior developers and data scientists possess domain knowledge and problem-solving abilities beyond current AI capabilities. While automating standard solutions may save some time, their expertise is essential for guiding AI tools, ensuring reliable and scalable outcomes.The surge in AI productivity means that companies are in high demand for AI talent, and the competition for hiring and retaining this talent is fierce. There will also be a growing need for cybersecurity professionals who can protect AI systems from attack. Additionally, as AI systems become more prevalent, there may be more work in areas such as AI ethics, regulation, and public policy. Hence, investing in attracting and nurturing such talent is significant for companies to remain relevant in this rapidly evolving landscape.Creators in all sectors will be affected. In music, AI is aiding musicians across the creative process, from composing lyrics and melodies to digitally mastering and enhancing audio. Generative art tools allow visual artists to experiment with customized paintings catered to their unique styles.A Goldman Sachs study from March 2023 suggested that administrative and legal roles are most at risk. They estimated that about two thirds of current jobs will be exposed to automation by AI, and concluded that generative AI tools could impact 300 million full-time jobs worldwide, more than 20% of the current workforce. The pace of adoption is a critical unknown. McKinsey analysts estimated that automation could absorb between 60 to 70 percent of employee hours so that between 2030 and 2060 about half of today's work activities could be automated. According to PwC, by the mid-2030s, up to 30% of jobs could be automatable. But real-world adoption depends on many hard-to-predict factors like regulation, social acceptance and retraining policies.Knowledge work sectors such as software and app development are already seeing the effects of this transformation. Generative AI has been employed to streamline tasks ranging from initial code generation to image editing and design. It reduces repetitive manual work for developers and designers, enabling them to focus their efforts on higher-value innovation. However, meticulous monitoring and iterative correction of errors in auto-generated outputs remains critical.The large-scale automation of work activities could result in a major shift in labor demand, leading to substantial changes in occupation and necessitating employees to acquire new skills. Because their capabilities are fundamentally engineered to do cognitive tasks, Generative AI is likely to have the biggest impact on knowledge work, particularly activities involving decision making and collaboration, which previously had the lowest potential for automation. While previously, automation's impact was highest in lower-middle-income quintiles, Further, Generative AI could have the biggest impact on activities in high-wage jobs. A significant number of workers will need to substantially change the work they do, either in their existing occupations or in new ones. They will also need support in making transitions to new activities.Managing this turnover will require policy foresight to minimize hardship for displaced workers through retraining programs, job creation incentives and portable benefits. If worker transitions and other risks can be managed, generative AI could contribute substantively to economic growth and support a more sustainable, inclusive world, where people are liberated from repetitive work.If the efficiency gains from AI automation are reinvested well, new industries and jobs could be created in the long run. But smooth workforce transitions will require policy foresight and employee training in the interim. In summary, while certain jobs may be displaced by AI in the near term, especially routine cognitive tasks, it may automate certain activities rather than eliminate entire occupations. Technical experts like data scientists and programmers will remain key to developing AI tools and realizing their full business potential.

Law

Generative models like LLMs can automate routine legal tasks such as contract review, documentation generation, and brief preparation. They also enable faster, comprehensive legal research and analysis. Additional applications include explaining complex legal concepts in plain language and predicting litigation outcomes using case data. However, responsible and ethical use remains critical given considerations around transparency, fairness and accountability. Overall, properly implemented AI tools promise to boost legal productivity and access to justice, while requiring ongoing scrutiny regarding reliability and ethics.

Manufacturing

In the automotive sector, they are employed to generate 3D environments for simulations and aid in the development of cars. Additionally, generative AI is utilized for road testing autonomous vehicles using synthetic data. These models can also process object information to comprehend the surrounding environment, understand human intent through dialogues, generate natural language responses to human input, and create manipulation plans to assist humans in various tasks.

Medicine

A model that can accurately predict physical properties from gene sequences would represent a major breakthrough in medicine and could have profound impacts on society. It could further accelerate drug discovery and precision medicine, enable earlier disease prediction and prevention, provide a deeper understanding of complex diseases, and improve gene therapies. However, it also raises major ethical concerns around genetic engineering and could exacerbate social inequalities.New techniques with neural networks are already employed to lower long-read DNA sequencing error rates (Baid and colleagues; *DeepConsensus improves the accuracy of sequences with a gap-aware sequence transformer*, September 2022), and according to a report by ARK Investment Management (2023), in the short-term, technology like this can make it already possible to deliver the first high-quality, whole long-read genome for less than $1,000. This means that large-scale gene-to-expression models might not be far away either.

Military

Militaries worldwide are investing in research to develop lethal autonomous weapons systems (LAWS). Robots and drones can identify targets and deploy lethal force without any human supervision. Machines can process information and react faster than humans, removing emotion from lethal decisions. However, this raises significant moral questions. Allowing machines to determine if lives should be taken crosses a troubling threshold. Even with sophisticated AI, complex factors in war like proportionality and distinction between civilians and combatants require human judgment.If deployed, completely autonomous lethal weapons would represent an alarming step towards relinquishing control over life-and-death decisions.

They could violate international humanitarian law or be used by despotic regimes to terrorize populations. Once unleashed fully independently, the actions of autonomous killer robots would be impossible to predict or restrain.

Misinformation and cybersecurity

AI presents a dual-edged sword against disinformation. While it enables scalable detection, automation makes it easier to spread sophisticated, personalized propaganda. AI could help or harm security depending on whether it is used responsibly. It increases vulnerabilities to misinformation along with cyberattacks using generative hacking and social engineering. There are significant threats associated with AI techniques like micro-targeting and deepfakes. Powerful AI can profile users psychologically to deliver personalized disinformation that facilitates concealed manipulation, escaping broad examination Big data and AI could be leveraged to exploit psychological vulnerabilities and infiltrate online forums to attack and spread conspiracy theories. Disinformation has transformed into a multifaceted phenomenon, involving biased information, manipulation, propaganda, and intent to influence political behavior. For example, during the COVID-19 pandemic, the spread of misinformation and infodemics has been a major challenge. AI has a potential to influence public opinion on topics like elections, war, or foreign powers.It can also generate fake audio/video content to damage reputations and sow confusion. State and non-state actors are weaponizing these capabilities for propaganda, to damage reputations and sow confusion. AI can be used by political parties, governments, criminal groups, and even the legal system, launch lawsuits, and extract money. This likely will have far-reaching consequences in various domains. A significant portion of internet users may be obtaining the information they need without accessing external websites. There is a danger of large corporations being the gatekeepers of information and controlling public opinion, effectively being able to restrict certain actions or viewpoints.Careful governance and digital literacy are essential to build resilience. Though no single fix exists, collective efforts promoting responsible AI development can help democratic societies address emerging threats.

强大的人工智能可以在心理上分析用户，提供个性化的虚假信息，便于隐藏操纵，逃避广泛的审查。大数据和人工智能可以利用心理弱点，渗透在线论坛，攻击和传播阴谋论。

大公司有成为信息的守门人的危险，并控制着公众舆论，有效地能够限制某些行为或观点。

Practical Implementation Challenges

Realizing the potential of generative AI in a responsible manner involves addressing a number of practical legal, ethical and regulatory issues:

- **Legal**: Copyright laws remain ambiguous regarding AI-generated content. Who owns the output - the model creator, training data contributors, or end users? Replicating copyrighted data in training also raises fair use debates that need clarification.
- **Data Protection**: Collecting, processing and storing the massive datasets required to train advanced models creates data privacy and security risks. Governance models ensuring consent, anonymity and safe access are vital.
- **Oversight and Regulations**: Calls are mounting for oversight to ensure non-discrimination, accuracy and accountability from advanced AI systems. But flexible policies balancing innovation and risk are needed rather than burdensome bureaucracy.
- **Ethics**: Frameworks guiding development toward beneficial outcomes are indispensible. Integrating ethics through design practices focused on transparency, explicability and human oversight helps build trust.

越来越多的人呼吁进行监督，以确保先进人工智能系统的不歧视、准确性和问责制。但我们需要平衡创新和风险的灵活政策，而不是繁重的官僚主义

Overall, proactive collaboration between policymakers, researchers and civil society is essential to settle unresolved issues around rights, ethics and governance. With pragmatic guardrails in place, generative models can fulfill their promise while mitigating harm. But public interest must remain the compass guiding AI progress.There is a growing demand for algorithmic transparency. This means that tech companies and developers should reveal the source code and inner workings of their systems. However, there is resistance from these companies and developers who argue that disclosing proprietary information would harm their competitive advantage. Open-source models will continue to thrive and local legislation in EU and other countries will push for transparent use of AI.The consequence of AI bias includes potential harm to individuals or groups due to biased decisions made by AI systems. Incorporating ethics training into computer science curricula can help reduce biases in AI codes. By teaching developers how to build applications that are ethical by design applications, the probability of biases being embedded into the codes can be minimized. To stay on the right path, organizations need to prioritize transparency, accountability, and guardrails to prevent bias in their AI systems. AI bias prevention is a long-term priority for many organizations, however, without legislation driving it, it can take time to be introduced. Local legislation in EU countries, for example, such as the European Commission's proposal for harmonized rules on AI regulation, will drive more ethical use of language and imagery.A current German law on fake news, which imposes a 24-hour timeframe for platforms to remove fake news and hate speech, is impractical for both large and small platforms. Additionally, the limited resources of smaller platforms make it unrealistic for them to police all content. Further, online platforms should not have the sole authority to determine what is considered truth, as this could lead to excessive censorship. More nuanced policies are needed that balance free speech, accountability, and feasibility for a diversity of technology platforms to comply. Relying solely on private companies to regulate online content raises concerns around lack of oversight and due process. Broader collaboration between government, civil society, academics, and industry can develop more effective frameworks to counter misinformation while protecting rights.

德国现行关于假新闻的法律规定平台可以24小时删除假新闻和仇恨言论，这对大小平台都是不切实际的。

The Road Ahead

The forthcoming era of generative AI models offers a plethora of intriguing opportunities and unparalleled progression, yet it is interspersed with numerous uncertainties. As discussed in this book, many breakthroughs have been accomplished in the recent years but successive challenges continue to linger, mainly pertaining to precision, rationalizing ability, controllability and entrenched bias within these models. While grandiose claims of superintelligent AI on the horizon may seem hyperbolic, consistent trends predict sophisticated capabilities sprouting within a few decades. On an individual level, the proliferation of generative content raises valid concerns around misinformation, plagiarism in academia, and impersonation in online spaces. As these models become more adept at mimicking human expression, people may have difficulty discerning what is human-generated versus AI-generated, enabling new forms of deception. There are also fears about generative models exacerbating social media addiction due to their ability to produce endless customized content.From a societal perspective, the sheer pace of advancement creates unease surrounding human obsolescence and job displacement, which could further divide economic classes. Unlike physical automation of the past, generative AI threatens cognitive job categories previously considered safe from automation. Managing this workforce transition ethically and equitably will require foresight and planning. There are also philosophical debates around whether AI should be creating art, literature or music that has historically reflected the human condition.For corporations, effective governance frameworks have yet to be established around acceptable use cases. Generative models amplify risks of misuse, ranging from creating misinformation such as deepfakes to generating unsafe medical advice. Legal questions around content licensing and intellectual property arise. While these models can enhance business productivity, quality control and bias mitigation incur additional costs. While large tech firms currently dominate generative AI research and development, smaller entities may ultimately stand to gain the most from these technologies. As costs decline for computing, data storage, and AI talent, custom pre-training of specialized models could become feasible for small and mid-sized companies. Rather than relying on generic models from Big Tech, tailored generative AI fine-tuned on niche datasets could better serve unique needs. Startups and non-profits often excel at rapidly iterating to build cutting-edge solutions for specialized domains. Democratized access through cost reductions could enable such focused players to train performant models exceeding capabilities of generalized systems. Concerns have emerged about saturation as generative AI tools are relatively easy to build using foundation models. Customization of models and tools is will allow value creation, but it's unclear who will capture most upsides. While current market hype is high, investors are tempering decisions given lower valuations and skepticism following the 2021 AI boom/bust cycle. The long-term market impact and winning generative AI business models have yet to unfold.

> The **2021 AI boom/bust cycle** refers to a rapid acceleration in investment and growth in the AI startup space followed by a market cooldown and stabilization in 2022 as projections failed to materialize and valuations declined.

2021年人工智能的繁荣/萧条周期指的是人工智能初创企业领域的投资和增长迅速加速，随后是2022年的市场降温和稳定，因为预测未能实现，估值下降。

Here's a quick summary:

Boom Phase (2020-2021):

There was huge interest and skyrocketing investment in AI startups offering innovative capabilities like computer vision, natural language processing, robotics, and machine learning platforms. Total funding for AI startups hit record levels in 2021, with over $73 billion invested globally according to Pitchbook. Hundreds of AI startups were founded and funded during this period.

Bust Phase (2022):

In 2022, the market underwent a correction, with valuations of AI startups falling significantly from their 2021 highs. Several high-profile AI startups like Anthropic and Cohere faced valuation markdowns. Many investors became more cautious and selective with funding AI startups. Market corrections in the broader tech sector also contributed to the bust.

Key Factors:

Excessive hype, unrealistic growth projections, historically high valuations in 2021, and broader economic conditions all contributed to the boom-bust cycle. The cycle followed a classic pattern seen previously in sectors like dot-com and blockchain.

Looking decades ahead, perhaps the deepest challenges are ethical. As AI is entrusted with more consequential decisions, alignment with human values becomes critical. While accuracy, reasoning ability, controllability, and mitigating bias remain technical priorities, other priorities should include fortifying model robustness, promoting transparency and ensuring alignment with human values. In order to maximize benefits, companies need to ensure human oversight, diversity, and transparency in development. Policy makers may need to implement guardrails preventing misuse while providing workers with support to transition as activities shift. With responsible implementation, generative AI could propel growth, creativity and accessibility in a more prosperous society. Addressing potential risks early on and ensuring a just distribution of benefits designed to serve public welfare will cultivate a sense of trust among stakeholders, such as:

- **The Dynamics of Progress**: Fine-tuning the pace of transformation is critical to avoid any undesired repercussions. Moreover, excessively slow developments could stifle innovation, suggesting that determining an ideal pace through encompassing public discourse is crucial.
- **The Human-AI Symbiosis**: Rather than striving for outright automation, more advantageous systems would integrate and complement the creative prowess of humans with the productive efficiency of AI. Such a hybrid model will ensure optimal oversight.
- **Promoting Access and Inclusion**: Equitable access to resources, relevant education and myriad opportunities concerning AI is key to negating the amplification of disparities. Representativeness and diversity should be prioritized.
- **Preventive Measures and Risk Management**: Constant evaluation of freshly emerging capabilities via interdisciplinary insights is necessary to evade future dangers. Excessive apprehensions, however, should not impede potential progress.
- **Upholding Democratic Norms**: Collaborative discussions, communal efforts and reaching compromise will inevitably prove more constructive in defining the future course of AI, as compared to unilateral decrees imposed by a solitary entity. Public interest must take precedence.

通过跨学科的见解对新出现的能力进行持续评估是必要的。然而，过分的忧虑不应阻碍潜在的进展。

While future capabilities remain uncertain, proactive governance and democratization of access are essential to direct these technologies toward equitable, benevolent outcomes. Collaboration between researchers, policymakers and civil society around issues of transparency, accountability and ethics can help align emerging innovations with shared human values. The goal should be empowering human potential, not mere technological advancement.

虽然未来的能力仍然不确定，但积极的治理和准入的民主化对于指导这些技术走向公平、仁慈的结果至关重要。研究人员、决策者和公民社会就透明度、问责制和道德问题进行的合作，可以帮助新兴创新与共同的人类价值观相一致。其目标应该是增强人类的潜力，而不仅仅是技术的进步。