

STUDY BOOK 3

| PERSONALIZED OFFICIAL USAII[®]
ASSESSMENT PREPARATION RESOURCE

MODERN AI ARCHITECTURES: FROM LLMs TO ML PLATFORMS

FOR CERTIFIED AI ENGINEER – CAIE™

New and Upgraded Content
with **LLMs, Gen AI, Prompt Engineering,
RAG**, and more.



Copyright © 2025 by Packt and United States Artificial Intelligence Institute (USAII®).

This edition of the book is specially packaged for the United States Artificial Intelligence Institute (USAII®).

No part of this book may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or scanning without the publisher's written permission.

Limits of Liability:

While the best efforts, research, and rigor have been invested in the preparation of this book, Packt, USAII® and its associated organizations, and the contributors make no representation or warranties with respect to the accuracy or completeness of the contents of this book; and specifically disclaim any implied warranties of merchantability or fitness for any - particular purpose. There are no warranties that extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particular results, and the advice and strategies contained herein may not be suitable for every individual. Neither Packt, USAII®, and its associated organizations nor the contributors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Disclaimers: This publication has been conceived and jointly developed by the Packt, USAII®, and its associated organizations with the intent to provide as accurate and as recent information pertaining to Artificial Intelligence technologies, tools, processes, models, etc., as has been possible amidst the rapidly and continuously changing markets and technologies in the space. Also, the content finally approved for publication meets USAII's and the Publisher's intent and objective of maintaining vendor neutrality and platform independence as much as possible in the treatment of the subject in the book. Further, as the book is intended for educational purposes, neither Packt, USAII®, and its associated organizations nor any of the contributors shall be responsible for any errors, omissions, or damages arising out of the use of the information contained in the book for any other purpose.

First Edition: March 2021.

Second Edition: April 2023.

Third Edition: January 2025.

Published by: Packt

ISBN: 978-1-80181-967-1

Preface

Artificial Intelligence (AI) stands as a transformative force reshaping industries and redefining the boundaries of what is possible. From enhancing customer experiences to optimizing supply chains, AI's potential is vast and varied, touching every sector and aspect of modern life. As we stand on the brink of the Fourth Industrial Revolution, the significance of AI cannot be overstated. This book is designed to serve as a comprehensive guide for working professionals and students pursuing advanced studies in AI, particularly those aiming to achieve the prestigious **Certified AI Engineer (CAIE™)** certification offered by the **United States Artificial Intelligence Institute (USAII®)**.

AI adoption is accelerating at an unprecedented pace. According to McKinsey & Company's 2024 survey, 65% of organizations are regularly using generative AI, nearly doubling from the previous year. The global AI market is projected to expand at a compound annual growth rate (CAGR) of 36.6% between 2024 and 2030. This rapid growth underscores the critical role AI engineers play in driving innovation and maintaining competitive advantage.

For professionals at any stage of their career, understanding and leveraging AI is no longer a luxury but a necessity. The ability to harness the power of AI is pivotal for staying relevant and competitive in an ever-changing job market. AI learning is not just about acquiring technical skills; it is about embracing a mindset that fosters innovation, efficiency, and strategic thinking.

The landscape of AI is continuously evolving, with modern advancements such as Generative AI, Large Language Models (LLMs), and AI democratization making significant impacts. Generative AI, which includes technologies like GANs (Generative Adversarial Networks) and transformers, has revolutionized content creation, enabling machines to generate text, images, and even music that is indistinguishable from human-created content. Retrieval-augmented generation (RAG) and edge AI are opening new avenues for scalable and context-aware AI solutions, while AI-driven robotics are reshaping automation in industries like healthcare, manufacturing, and logistics.

The Importance of AI Engineering Skills

AI engineering is the backbone of intelligent systems, combining principles of software engineering, data science, and domain expertise to build scalable, efficient, and ethical AI solutions. AI engineers are at the forefront of innovation, leveraging machine learning models, neural networks, and advanced frameworks to tackle real-world challenges. Developing a strong foundation in AI engineering equips professionals with the ability to:

- Design and deploy robust AI systems.
- Optimize and maintain machine learning pipelines.
- Harness advanced tools and platforms for scalable AI development.
- Ensure responsible AI practices in alignment with ethical standards.

About This Study Book

Candidates who sign up with the United States Artificial Intelligence Institute (USAII[®]) under the Certified Artificial Intelligence Engineer (CAIE[™]) program must know that the contents of this book have been exclusively chosen by our content partner Packt's best-selling database. The content is further carefully vetted and verified by the 15 most prominent SMEs of the **USAII's Artificial Intelligence Advisory Board** – these subject matter experts are the influencers of the artificial intelligence domain and are CIOs, CTOs, Data Scientists, and technology leaders from Fortune 500 companies. Hence, we are sure that the content is apt and is future-ready.

Please note, if you are a candidate registered under the CAIE[™] program by USAII[®], you will be provided with a self-help study kit, which will help you prepare for the CAIE[™] certification examination. This study kit includes three books in the eLearning format and can be downloaded from myControlPanel dashboard. Below are the two series Study Books:

1. Study Book 1: **Exploring Intelligence: AI, Robotics, and Deep Learning**
2. Study Book 2: **Advanced AI Essentials for Engineers**
3. Study Book 3: **Modern AI Architectures: From LLMs to ML Platforms**

The **Certified AI Engineer (CAIE™)** certification program by USAII® is structured around nine comprehensive modules, each meticulously designed to cover essential aspects of AI engineering:

- **Exploring AI and ML Essentials**
- **Engineering the Future: From Debugging to Robotics**
- **Deep Learning Concepts**
- **Natural Language Processing Demystified**
- **Theories of Computer Vision and GANs**
- **Generative AI and LLMs**
- **Advanced LLMs**
- **Mastering Retrieval-Augmented Generation**
- **Navigating ML: Architectures, Tools, and Platforms**

The study kit will also include certain other eLearning materials that can be accessed separately as per instructions. These study materials include the latest case studies, workshops, videos, and other important AI events that will help you crack the CAIE™ certification assessment.

Packt is the worldwide official content partner for USAII® and has produced two world-class CAIE™ certification study books for international use under exclusive permission from and under arrangement with USAII®. Assessments for CAIE™ certification are on-demand and can be taken almost anywhere in the world.

We welcome you to this exciting journey of discovery, innovation, and mastery. As you progress through the modules, remember that AI engineering is not just about building technology—it's about shaping the future. Let this study book be your guide, mentor, and companion in achieving excellence as a Certified AI Engineer (CAIE™).

With best wishes,

Research & Authoring Team

United States Artificial Intelligence Institute Editorial Desk

TABLE OF CONTENTS

Study Book 1	Page 1 to Page 354
Study Book 2	Page 355 to Page 748
Study Book 3	Page 749 to Page 1197

Module 7: Advanced LLMs: Techniques, Applications, and Frameworks 749

Chapter 29: Optimizing Large Language Models.....	750
Chapter 30: Innovative Applications of LLMs.....	781
Chapter 31: Matching LLMs to Applications.....	797
Chapter 32: Prompt Engineering: Techniques and Best Practices	822
Chapter 33: Tools and Frameworks for LLMs	850
Chapter 34: The Future of LLMs: GPT-5 and Beyond	875
Chapter 35: LLM Integration for Applications	900
Chapter 36: Conversational AI with LLMs	929

Module 8: Mastering Retrieval-Augmented Generation 949

Chapter 37: Retrieval-Augmented Generation (RAG).....	950
Chapter 38: Developing a Complete RAG Pipeline	965
Chapter 39: RAG in Practice: Use Cases and Solutions.....	986
Chapter 40: RAG Architecture	1000
Chapter 41: Integrating RAG with AI Agents and LangGraph.....	1017

Module 9: Navigating ML: Architectures, Tools, and Platforms 1040

Chapter 42: ML Systems: The Lifecycle and Architecture	1041
Chapter 43: Business Use Cases for Machine Learning.....	1053
Chapter 44: Breaking Down ML Algorithms	1085
Chapter 45: Managing Data for Machine Learning Success.....	1112
Chapter 46: Exploring the Best ML Libraries.....	1149
Chapter 47: The Power of Open-Source ML Platforms	1165

MODULE 7

ADVANCED LLMs: TECHNIQUES, APPLICATIONS, AND FRAMEWORKS

CHAPTER 29: Optimizing Large Language Models

CHAPTER 30: Innovative Applications of LLMs

CHAPTER 31: Matching LLMs to Applications

CHAPTER 32: Prompt Engineering: Techniques and Best Practices

CHAPTER 33: Tools and Frameworks for LLMs

CHAPTER 34: The Future of LLMs: GPT-5 and Beyond

CHAPTER 35: LLM Integration for Applications

CHAPTER 36: Creating Conversational AI with LLMs

Chapter 29

Optimizing Large Language Models

Expanding on the training strategy basics that were covered in the previous chapter, we'll delve into more sophisticated training strategies that can significantly enhance the performance of LLMs.

Transfer learning and fine-tuning in practice

Transfer learning and fine-tuning are powerful techniques in the field of ML, particularly within NLP, to enhance the performance of models on specific tasks. This section will provide a detailed explanation of these concepts in practice.

Transfer learning

Transfer learning is the process of taking a pre-trained model that's been trained on a large dataset (often a general one) and adapting it to a new, typically related task. The idea is to leverage the knowledge the model has already acquired, such as understanding language structures or recognizing objects in images, and apply it to a new problem with less data available. In NLP, transfer learning has revolutionized the way models are developed. Previously, most NLP tasks required a model to be built from scratch, a process that involved extensive data collection and training time. With transfer learning, you can take a pre-trained model and adapt it to a new task with relatively little data.

Key benefits of transfer learning

Transfer learning has several benefits:

- **Computational efficiency:** Efficient computational strategies enhance ML processes by doing the following:
 - Reducing training time as you don't have to start from scratch

- Lowering power consumption as you're fine-tuning a model rather than training a new one
- **Data efficiency:** Transfer learning boosts data efficiency in the following ways:
 - Requiring less labeled data
 - Effectively utilizing unlabeled data
- **Improved performance:** Transfer learning enhances model performance by offering the following:
 - Higher baseline accuracy
 - Better generalization on a new task due to the broader knowledge already acquired
- **Broad applicability:** Transfer learning showcases broad applicability by allowing the following:
 - Versatility across domains
 - Domain adaptation with minimal effort
- **Accessibility:** Transfer learning advances the accessibility of AI because of its ability to do the following:
 - Democratize AI by reducing the need for large datasets and extensive computing power
 - Enable rapid prototyping

Implementation considerations

The successful implementation of transfer learning hinges on several factors:

- The pre-trained model should be relevant to the new task
- Deciding how much of the pre-trained model to freeze and how much to fine-tune is crucial for the success of transfer learning
- The more similar the new task's data is to the data used in the pre-trained model, the more likely the transfer learning will be successful

Challenges and considerations

Now, let's learn how to navigate the challenges of transfer learning:

- If the new task's domain is very different from the text the model was originally trained on, the model might need significant adaptation.

- Finding the right fine-tuning approach can be complex. It requires carefully tuning the learning rate, deciding how many layers to fine-tune, and so on.
- Despite their efficiency, fine-tuning large models such as BERT or GPT still requires significant computational power, especially when dealing with large datasets or many fine-tuning iterations.
- There's a risk of overfitting to the new task if the fine-tuning process isn't managed carefully, especially with smaller datasets.
- Acquiring labeled data for fine-tuning can be costly and time-consuming, impacting overall efficiency and feasibility.
- Not all pre-trained models transfer knowledge effectively across different tasks, and identifying which models will work best can be challenging.
- Pre-trained models may carry biases from their original training data. This can be transferred to the new task if it's not mitigated properly.
- Understanding how and why a transfer learning model makes decisions can be difficult, particularly when using complex models such as deep neural networks.

Applications of transfer learning in NLP

Here are some of the applications of transfer learning in NLP:

- **Sentiment analysis:** Transfer learning tailors models to determine whether the sentiment of a piece of text is positive, negative, or neutral.

A pre-trained model such as BERT can be fine-tuned with a smaller set of labeled sentiment data so that it specializes in understanding sentiments expressed in text, making it adept at classifying product reviews, social media posts, and so on. Fine-tuning is an integral part of transfer learning, where a pre-trained model is further trained on a specific dataset to adapt it for a particular task. This enables the model to use its existing knowledge to perform well on new tasks with limited data.

- **Question-answering:** In NLP, question-answering has been revolutionized by models on datasets by providing answers to questions based on a given context.

BERT and GPT models, after being fine-tuned on datasets such as the **Stanford Question Answering Dataset (SQuAD)**, can be proficient at reading a passage of text and answering questions about it, which is valuable for building conversational agents and search engines.

- **Language translation:** GPT and T5 models excel at the following aspects:
 - Translating text from one language into another
 - Fine-tuning models such as GPT and T5 on parallel corpora (text that's aligned in two or more languages) to perform translation tasks, reducing the need for extensive bilingual datasets for every language pair
- **Other tasks:** AI excels in the following areas:
 - Categorizing text into predefined categories
 - Identifying and classifying key elements in text into predefined categories such as the names of people, organizations, locations, and so on
 - Generating a concise and fluent summary of a long piece of text

Transfer learning is a powerful strategy in the ML toolkit that addresses key challenges in developing AI systems, especially when facing limitations regarding data, time, and computational capacity. Its benefits are most pronounced in scenarios where labeled data is scarce and the computational cost of training models from scratch is prohibitive. This approach not only streamlines the development process but also opens up the potential for innovation and application across a wide range of tasks and domains.

Fine-tuning

Let's take a closer look at fine-tuning:

- **Process:** Fine-tuning involves taking a pre-trained model and continuing the training process with a smaller, task-specific dataset. During fine-tuning, the model's weights are adjusted to better perform on the new task. This process is usually much faster than the initial training phase as the model has already learned a significant amount of general knowledge.
- **Customization:** Fine-tuning allows models to be customized to specific domains or applications. For instance, a model pre-trained on general English can be fine-tuned with legal documents to create a legal language model.
- **Challenges:** A potential challenge in fine-tuning is overfitting, where the model becomes too specialized to the fine-tuning dataset and loses its ability to generalize to new data. Careful monitoring, regularization techniques, and validation with a separate dataset are essential to avoid this.

Practical implementation of transfer learning and fine-tuning

In practice, transfer learning involves the following:

- **Selecting a pre-trained model:** The first step is to choose an appropriate pre-trained model. This choice depends on the nature of the task and the availability of pre-trained models suitable for the language or domain of interest.
- **Preparing task-specific data:** The data for fine-tuning should be closely related to the target task and properly labeled if necessary. It's also important to ensure the quality and diversity of this dataset to promote good generalization.
- **Model adaptation:** Adapting the model often involves adding or modifying the final layers so that the output is suitable for the specific task, such as changing the output to a different number of classes for classification tasks.
- **Hyperparameter tuning:** Adjusting hyperparameters such as the learning rate, batch size, and the number of epochs is crucial for effective fine-tuning. A lower learning rate is commonly used to make smaller, more precise adjustments to the weights.
- **Evaluation and iteration:** After fine-tuning, the model is evaluated using performance metrics relevant to the task. Based on these results, further iterations of fine-tuning may be performed to refine the model's performance.

In practice, transfer learning and fine-tuning have become standard procedures for developing NLP systems due to their efficiency and effectiveness. By building upon the vast knowledge that's acquired during pre-training, these techniques allow for the rapid development of specialized models capable of high performance on a wide array of NLP tasks.

Case study – enhancing clinical diagnosis with transfer learning and fine-tuning in NLP

Now, let's look at a hypothetical case study that focuses on transfer learning and fine-tuning in the healthcare industry.

Background

The healthcare industry continually seeks advancements in clinical diagnosis accuracy. With the advent of NLP, there is potential to automate and enhance the accuracy of diagnostic processes by analyzing patient records, clinical notes, and medical literature.

In a hypothetical case study, a leading healthcare AI company embarked on a project to develop an NLP model that could support clinicians by providing more accurate diagnostic suggestions based on unstructured text data.

Challenge

The primary challenge was the sensitive nature of medical data, which is not only scarce but also heavily guarded due to privacy concerns. Furthermore, the company faced the daunting task of developing a model capable of understanding complex medical jargon and extracting relevant information from a variety of text styles and structures in patient records.

Solution – transfer learning and fine-tuning

To address these challenges, the company utilized transfer learning and fine-tuning methodologies by implementing the following phases:

- **Phase 1 – transfer learning implementation:**
 - **Model selection:** The company selected a pre-trained BERT model that had been trained on a broad range of general English text corpora
 - **Initial adaptation:** They adapted the model to the medical domain using a large-scale medical dataset, including publications and anonymized patient notes, to grasp the medical lexicon and sentence structures
- **Phase 2 – fine-tuning the model:**
 - **Data preparation:** A smaller, highly specialized dataset was curated, consisting of annotated clinical notes and diagnosis records that represented a wide spectrum of cases
 - **Model training:** The pre-trained BERT model was fine-tuned with this dataset, focusing on disease markers and diagnostic patterns
 - **Validation and testing:** The model was rigorously validated against a control set that was reviewed by medical professionals to ensure accuracy and reliability

Results

The fine-tuned NLP model demonstrated a remarkable improvement in identifying diagnostic entities and suggesting accurate diagnoses from clinical notes. It showed the following:

- A 20% increase in diagnosis accuracy compared to the baseline model

- A significant reduction in false positives, which is crucial for medical applications
- Improved efficiency, reducing the time taken for preliminary diagnosis

Impact

The implementation of transfer learning and fine-tuning resulted in several impactful outcomes:

- **Support for clinicians:** The model became an invaluable tool for clinicians, providing them with quick, accurate diagnostic suggestions
- **Resource optimization:** It reduced the time clinicians spent on preliminary diagnosis, allowing them to focus on patient care
- **Scalability:** The approach demonstrated a scalable model for incorporating AI in healthcare, opening pathways for further innovations

Conclusion

This case study illustrates the practical benefits of transfer learning and fine-tuning in NLP within the healthcare sector. By leveraging these techniques, the company was able to create a tool that enhanced the accuracy of clinical diagnoses. This project not only exemplifies the effectiveness of these methodologies in dealing with domain-specific challenges but also sets a precedent for future AI-driven healthcare solutions.

Now, let's delve into how LLMs are taught effectively.

Curriculum learning – teaching LLMs effectively

Curriculum learning is an approach in ML, particularly when training LLMs, that mimics the way humans learn progressively from easier to more complex concepts. The idea is to start with simpler tasks or simpler forms of data and gradually increase the complexity as the model's performance improves. This approach can lead to more effective learning outcomes and can help the model to better generalize from the training data to real-world tasks. Let's take a closer look at this approach.

Key concepts in curriculum learning

Here, we'll review some key concepts in curriculum learning that you should be aware of.

Sequencing

Sequencing in curriculum learning is analogous to the educational curricula in human learning, where subjects are taught in a logical progression from simple to complex. In ML, the following are applicable:

- **Graduated complexity:** Training begins with easier instances to give the model a foundational understanding before it tackles more complex scenarios
- **Task decomposition:** Complex tasks are broken down into simpler, more manageable subtasks that are learned in sequence
- **Sample selection:** Initially, samples that are more representative of the general distribution or are less noisy are chosen to help the model learn the basic patterns before outliers or edge cases are introduced

In NLP, sequencing might involve starting with basic vocabulary and grammar before introducing complex sentences, metaphors, or domain-specific jargon. For example, a language model might be exposed to simple sentences ("The cat sat on the mat") before encountering complex ones ("Despite the cacophony, the cat, undisturbed, sat on the checkered mat").

Pacing

Pacing is about controlling the speed at which new concepts are introduced:

- **Adaptive learning rate:** Adjusting the pace of learning based on the model's performance, similar to a teacher providing feedback to a student
- **Performance thresholds:** Moving to more complex materials only after the model achieves a certain level of performance on the current material
- **Staged difficulty:** Introducing new difficulty levels in stages, with each stage having a set of criteria for mastery before progression

In the context of curriculum learning, pacing ensures that the model has sufficiently learned from current examples before moving on to more challenging ones. This could be akin to ensuring a student understands basic algebra before introducing them to calculus.

Focus areas

The concept of focus areas in curriculum learning relates to concentrating on particular aspects of the learning task at different stages of the training process:

- **Concept isolation:** This involves teaching specific concepts in isolation before integrating them with other learned concepts. For example, in language learning, this could involve focusing on the present tense before introducing past or future tenses.
- **Attention shifting:** This involves shifting the model's focus during training to various aspects of the data. In NLP, a model might focus on syntax first before shifting focus to semantic analysis.
- **Progressive refinement:** This involves starting with a broad approximation of the target function and then refining the model's understanding over time. This is akin to teaching broad strokes in art before focusing on the finer details.

For instance, in language models, initial focus areas may include basic sentence structure and vocabulary, before more complex linguistic features, such as irony or ambiguity, are considered.

Benefits of curriculum learning

Curriculum learning provides the following benefits:

- **Efficiency:** Efficiency in AI training is achieved through the following:
 - **Accelerated initial learning:** By beginning with simpler tasks, the AI model can quickly achieve initial success, which can reinforce the correct learning patterns and boost its learning curve.
 - **Resource optimization:** Curriculum learning can lead to more efficient use of computational resources. Training on simpler tasks first generally requires less computational power, and as the model's capability increases, so can the computational investment.
 - **Reduced training time:** As the model is not immediately overwhelmed with complex tasks, it can converge to a good solution faster, making the overall training process more time-efficient.
- **Performance:** Curriculum learning provides various benefits:
 - **Improved accuracy:** Models trained using a curriculum tend to develop a more nuanced understanding of the data, leading to better accuracy and performance on their tasks
 - **Stronger foundational knowledge:** The model builds a robust foundation of the basics, which is essential for understanding more intricate patterns and structures later on
 - **Less prone to overfitting:** With a focus on general principles first, models are less likely to overfit to the noise in more complex training examples

- **Generalization:** Generalization is enhanced through the following aspects:
 - **Better transferability:** A model that has a strong base in fundamental concepts may be more capable of transferring what it has learned to new, unseen data, which is crucial for real-world applications
 - **Adaptability to variations:** Staged learning helps the model adapt to variations within the data, leading to better performance on tasks that were not part of the training set
 - **Handling of real-world complexity:** By gradually introducing complexity, the model can better mimic the progression of learning required to handle complex real-world tasks
- **Improved interpretability:** Curriculum learning enhances interpretability in the following ways:
 - **Providing a clearer understanding of model behavior:** Curriculum learning provides insights into how models develop their understanding over time, making their decision-making processes more interpretable.
 - **Facilitated debugging and analysis:** By following a structured learning path, it becomes easier to identify and address errors. This is because the model's learning stages are clearer and more logical.

Additional considerations

The following are some additional considerations regarding curriculum design:

- **Curriculum design:** The design of the learning curriculum must be thoughtful and strategic to ensure that the model is not only learning efficiently but also developing the capacity to handle the complexity of real-world applications
- **Balanced progression:** The progression from simple to complex needs to be balanced to ensure that the model is challenged just enough to learn without being overwhelmed or plateauing in its learning journey
- **Evaluation metrics:** It is crucial to have proper evaluation metrics in place to assess the effectiveness of the curriculum and the model's readiness to progress to more challenging tasks

Curriculum learning addresses some of the fundamental challenges in training LLMs by structuring the learning process in a more human-like fashion. By optimizing the order and complexity of training data, this approach not only makes the training process more efficient but also enhances the performance and generalization capabilities of the models. Such benefits are particularly important as LLMs are increasingly being deployed in diverse and complex real-world scenarios, where adaptability and robustness are key to success.

Implementing curriculum learning

Implementing curriculum learning in ML and AI involves several critical steps to ensure that the model can effectively progress from learning simple concepts to mastering complex ones. We'll take a closer look at these steps here.

Data organization

Organizing training data by complexity is the cornerstone of curriculum learning. This process can be quite nuanced, depending on the domain and the specific tasks the model is being prepared for. The following are key aspects that need to be addressed:

- **Complexity metrics:** Developing metrics to evaluate the complexity of data is essential. For language models, this might involve sentence length, vocabulary difficulty, or syntactic complexity. In other domains, complexity could be measured by the number of features, the ambiguity of labels, or the rarity of the data points.
- **Expert involvement:** Involving subject matter experts can be critical, especially when complexity metrics are not clear-cut or when the data requires domain-specific insight to be categorized properly.
- **Automated sorting:** ML techniques, such as clustering algorithms, can be used to sort data into complexity tiers automatically. These methods might use feature vectors to determine similarity and group data points accordingly.

Model monitoring

Continuous evaluation of the model's performance is necessary to gauge when it's ready to move on to more difficult material. This can be achieved by using the following:

- **Performance metrics:** Defining clear performance metrics such as accuracy, precision, recall, or a domain-specific metric is necessary to objectively assess the model's progress
- **Feedback loops:** Implementing feedback mechanisms that can guide the training process and inform decisions about when to introduce more complex data
- **Early stopping:** This technique can prevent overfitting on simpler data and prompt the transition to more complex stages when the model's improvement in the current stage diminishes

Dynamic adjustments

The ability to adapt the training process dynamically is a key feature of effective curriculum learning. This can be incorporated with the help of the following:

- **Adaptive pacing:** The curriculum should allow for changes in pacing based on real-time performance, slowing down when the model struggles and accelerating when it masters a concept quickly.
- **Curriculum refinement:** The initial curriculum might need to be refined as the model's learning patterns emerge. This could involve adding more intermediate steps or revising the complexity measures.

Task-specific curricula

Designing curricula that are tailored to the final tasks of the model can significantly enhance its effectiveness. For this purpose, you need to manage the following:

- **Task analysis:** A thorough analysis of the end tasks can help you identify the core skills and knowledge the model needs to acquire. For example, customer service models need to understand colloquial language and empathy, while medical models must interpret clinical terminology accurately.
- **Curriculum design:** The curriculum should reflect the progression of skills and knowledge required for the model to perform its final tasks. For instance, a curriculum for a medical diagnosis model might start with general medical knowledge before focusing on symptoms and treatments for specific conditions.

Implementing curriculum learning is a complex process that requires careful planning, continuous monitoring, and the flexibility to adapt the curriculum as the model learns. It's a strategic approach that, when executed well, can significantly improve the efficiency and effectiveness of AI models, particularly in specialized or complex domains. By tailoring the learning process to the model's needs and the intricacies of the task at hand, curriculum learning can lead to AI systems that are not only highly competent in their designated tasks but also capable of generalizing their knowledge to new, related challenges.

Challenges in curriculum learning

Curriculum learning comes with its own set of challenges. Let's take a look.

Defining complexity

Determining the complexity within training data is a critical and non-trivial aspect of curriculum learning. In the context of language, this is particularly challenging due to the following aspects:

- **The multidimensional nature of language:** Language complexity is not one-dimensional; it includes syntactic complexity, semantic richness, pragmatics, and more. An example that is simple in that one respect might be complex in another.
- **Subjectivity:** What one model or domain expert considers complex, another might not. This subjectivity can make standardizing a measure of complexity difficult.
- **Automated complexity measures:** Developing automated measures that accurately reflect complexity requires advanced algorithms that can potentially incorporate linguistic, contextual, and domain-specific features.

Curriculum design

Creating an effective curriculum is akin to developing an educational course for a human student – it requires understanding how the “student” (in this case, the model) learns about the following:

- **Domain expertise:** The designer of the curriculum needs to have a thorough understanding of the domain to ensure that all the necessary concepts are taught in an appropriate sequence.
- **Model understanding:** Different models may learn in different ways. Understanding the learning dynamics of the specific model being used is crucial for designing an effective curriculum.
- **Iterative process:** Designing a curriculum is not a one-time task; it often requires iterations and modifications as the model’s performance on the tasks is observed and analyzed.

Balancing breadth and depth

Striking the right balance between a broad understanding and deep expertise is a delicate task that includes various aspects:

- **Breadth:** Ensuring the model has a comprehensive understanding of a wide range of topics or skills is important for generalization. However, too much breadth can lead to a superficial understanding of each topic.

- **Depth:** Providing in-depth knowledge in certain areas is necessary for expertise. However, focusing too deeply on one area can limit the model's ability to handle a variety of tasks.
- **Practical application:** The ultimate goal is to deploy the model in real-world applications. Therefore, the curriculum should focus on achieving the right mix of breadth and depth to prepare the model for the tasks it will encounter.

Generalization and overfitting

Managing generalization and overfitting is crucial in curriculum learning:

- **Generalization:** The curriculum must be designed to ensure that the model can generalize its learning to new and unseen data, which is often challenging when creating a staged learning process
- **Overfitting:** There is a risk of overfitting to simpler tasks if the curriculum does not progressively increase in complexity or if too much emphasis is placed on easy examples

Evaluation and metrics

Evaluating the effectiveness of curriculum learning requires the following careful considerations:

- **Choosing the right metrics:** Determining which metrics best reflect the model's progress and effectiveness at each stage of the curriculum can be challenging
- **Continuous monitoring:** Regularly evaluating model performance to adjust the curriculum requires significant resources and ongoing analysis
- **Benchmarking:** Establishing benchmarks to compare the effectiveness of different curriculum designs is essential but can be difficult due to variability in tasks and models

Model-specific challenges

Each model may present unique challenges when implementing curriculum learning:

- **Architecture-specific considerations:** Different models may require tailored curriculum designs that consider their specific architecture and learning dynamics

- **Resource constraints:** The computational and data requirements of different models can vary widely, influencing how the curriculum can be structured and executed

Practical strategies for addressing challenges

The following are some practical strategies that can be used:

- **Expert collaboration:** Working with domain experts can help in accurately defining complexity and designing a well-rounded curriculum
- **Incremental development:** Building the curriculum incrementally, starting with a basic structure and then refining it based on the model's performance, can make the process more manageable
- **Evaluation and feedback:** Regularly evaluating the model's performance and incorporating feedback can help in fine-tuning the curriculum to better meet the model's learning needs
- **Modular design:** Creating a modular curriculum that can be adjusted or reorganized easily allows for more dynamic learning paths tailored to the model's progression

Curriculum learning, while powerful, requires thoughtful implementation to overcome its inherent challenges. The intricacies of defining complexity, designing the curriculum, and achieving a balance of breadth and depth are substantial hurdles. However, with a careful approach that includes expert input, iterative design, and ongoing evaluation, these challenges can be navigated successfully. The outcome is a more effective training process that produces models capable of sophisticated understanding and performance.

Case study – curriculum learning in training LLMs for legal document analysis

This case study focuses on curriculum learning in the legal industry.

Background

In a hypothetical case study, a legal tech start-up aimed to develop an LLM capable of parsing and understanding complex legal documents to provide summaries and actionable insights. The goal was to assist lawyers by automating the preliminary

review of case files, contracts, and legislation, which are typically dense and filled with specialized language.

Challenge

The main challenge was the complexity of legal language, which included a wide range of vocabulary, specific jargon, and intricate sentence structures. Traditional training methods proved inefficient as the model struggled with the advanced nuances of legal texts after being trained on general language data.

Solution – curriculum learning

To overcome this, the company implemented a curriculum learning approach, structuring the model's training to progressively increase in complexity, closely aligning with the cognitive steps a human expert would take when learning the legal domain. This involved the following phases:

- **Phase 1 – structured learning progression:**
 - **Simple to complex:** The LLM began by learning simple legal definitions and moved toward understanding complex contractual clauses
 - **Segmented learning:** Training was segmented into phases, starting with general legal principles before progressing to specifics such as tax law, intellectual property rights, and international regulations
- **Phase 2 – incremental complexity increase:**
 - **Controlled vocabulary expansion:** Vocabulary was introduced in a controlled manner, starting with general legal terms before more specialized terms were incorporated
 - **Complexity in context:** The model was exposed to increasingly complex sentences, starting from clear-cut case law to convoluted legal arguments

Results

The curriculum learning approach yielded a highly efficient LLM that showed the following:

- A 35% improvement in the comprehension of legal terminology compared to the baseline model trained without curriculum learning
- A 25% increase in accuracy when summarizing legal documents
- Enhanced ability to identify relevant legal precedents and citations

Impact

The successful implementation of curriculum learning significantly impacted the start-up's objectives:

- **Efficiency in legal reviews:** The LLM reduced the time lawyers spent on initial document reviews by automating the process of extracting key points
- **Scalability of legal services:** Smaller law firms, previously limited by resource constraints, could scale their operations by utilizing AI for routine document analysis
- **Consistency and reliability:** The LLM provided consistent and reliable analysis, reducing human error in initial reviews

Conclusion

This case study demonstrates the effectiveness of curriculum learning in training LLMs for specialized tasks. By mimicking the natural progression of human learning, the start-up was able to create a model that understood and analyzed legal documents with high accuracy. This approach not only proved to be a breakthrough in legal technology but also showcased a scalable method for applying AI in specialized fields, potentially transforming how professionals engage with dense and specialized texts.

Next, we'll delve into multitasking and continual learning models.

Multitasking and continual learning models

Multitasking and continual learning models represent two pivotal areas of research in the field of AI and ML, each addressing distinct but complementary challenges related to the flexibility and adaptability of AI systems.

Multitasking models

Multitasking models, also known as **multi-task learning (MTL)** models, are designed to handle multiple tasks simultaneously, leveraging the commonalities and differences across tasks to improve the performance of each task. This hypothesis is grounded in cognitive science, suggesting that human learning often involves transferring knowledge across different but related tasks. In AI, this translates into models that can process and learn from multiple tasks simultaneously, optimizing shared neural network parameters to benefit all tasks involved.

The central idea is to share representations between related tasks to avoid learning each task in isolation, which can be inefficient and require more data. This approach can lead to models that are more generalizable and efficient as they can learn useful features from one task that apply to others.

Key characteristics

The key characteristics of multitasking models are as follows:

- **Shared architectures:** Multitasking models are designed to handle multiple tasks that can benefit from shared representations. Here's how shared architectures work and their benefits:
 - **Layer sharing:** The initial layers of the network are shared among all tasks. These layers typically learn the basic patterns in the data that are common across tasks. For example, in a visual recognition model, these layers might detect edges and shapes that are fundamental to many different objects.
 - **Specialization in later layers:** As the network progresses, layers become more specialized for individual tasks. This can be seen as a divergence point where task-specific knowledge is refined and applied. In our visual recognition example, these specialized layers would learn patterns specific to different categories, such as animals, vehicles, or furniture.
 - **Efficient learning:** By sharing parameters, these architectures require fewer resources than when training separate models for each task as they don't need to relearn the same general features for each new task.
 - **Feature reuse:** Shared architectures can lead to feature reuse, where a feature learned for one task can be beneficial for another. This may not have been possible if the tasks were learned in isolation.
- **Joint learning:** Joint learning refers to the training on multiple tasks simultaneously. This approach has several advantages:
 - **Cross-task feature learning:** When models are trained jointly, they can learn features that are useful across multiple tasks, which might not be learned when tasks are trained independently
 - **Improved generalization:** Training on a diverse set of tasks can help the model generalize better to new tasks or data as it learns to extract and utilize broadly applicable features
 - **Balanced learning:** Joint learning can help prevent the model from overfitting to one task by balancing the learning signals from multiple tasks
- **Regularization effect:** MTL inherently incorporates a form of regularization due to its training dynamics:

- **Parameter sharing as regularization:** By sharing parameters across tasks, the model is implicitly regularized. This is because the shared parameters must be useful across all tasks they are shared among, preventing the model from overfitting to the idiosyncrasies of a single task's training data.
- **Constraints from multiple tasks:** Training with multiple tasks imposes additional constraints on the model as it has to perform well on all tasks simultaneously. This can help reduce the model's capacity to memorize the training data and instead force it to find underlying patterns that are more generally applicable.
- **Noise robustness:** Exposure to multiple tasks during training can also make the model more robust to noise as noise patterns are less likely to be consistent across different tasks, and hence, the model is less likely to learn them.

These characteristics make multitasking models particularly powerful for complex applications where multiple related problems need to be solved simultaneously, and where the benefits of shared knowledge, joint learning, and regularization effects can lead to more robust, generalizable, and efficient solutions.

Advanced techniques in MTL

Now, let's consider some advanced techniques in MTL:

- **Cross-stitch networks:** These are sophisticated versions of multitasking models that allow the optimal level of task sharing to be learned automatically. Unlike traditional shared architectures, cross-stitch units enable the network to learn how much information to share between tasks dynamically.
- **Task attention networks:** By incorporating attention mechanisms, multitasking models can weigh the importance of shared features differently for each task, allowing the model to focus more on relevant features for a given task while ignoring less useful information.

Applications

Multitasking models are widely used in various domains, including NLP, where a single model may perform entity recognition, sentiment analysis, and language translation. They are also prevalent in computer vision for tasks such as object detection, segmentation, and classification within the same framework. Let's review their applications.

NLP

In NLP, multitasking models are highly beneficial because many tasks share common linguistic features and structures. A single model that can capture these shared elements can be applied to multiple NLP tasks:

- **Entity recognition:** This task involves identifying and classifying key information in text into predefined categories, such as the names of people, organizations, and locations. Multitask models can learn contextual cues from sentence structures that help in recognizing entities.
- **Sentiment analysis:** This task involves understanding the sentiment expressed in a piece of text, whether it's positive, negative, or neutral. Models that have been trained to recognize sentiment can also benefit from, and contribute to, understanding language nuances that are required for other tasks, such as language translation.
- **Language translation:** Translation requires the model to understand the syntax and semantics of both the source and target languages. Multitasking models can leverage the deep understanding of language gained from other NLP tasks to improve translation accuracy.

The shared layers in a multitask model handle the common aspects of language, such as grammar and common vocabulary, while task-specific layers fine-tune the model's outputs for each task.

Computer vision

In computer vision, multitasking models take advantage of shared visual features across different tasks:

- **Object detection:** This involves locating objects within an image and classifying them. The initial layers of the multitasking model might learn to detect edges and textures, which are useful for many vision tasks.
- **Segmentation:** In image segmentation, the task is to assign a label to each pixel in an image so that pixels with the same label share certain characteristics. Multitasking models benefit from understanding general shapes and boundaries learned during object detection.
- **Classification:** Image classification involves assigning a class label to an image (or parts of an image). MTL can help with classification by leveraging feature detectors developed for detection and segmentation tasks.

In each of these tasks, the early layers of a multitasking model capture generic features such as shapes and edges, while later layers become more specialized, such as recognizing specific object features for detection or finer details for segmentation.

Advantages across domains

The use of multitasking models across these domains offers several advantages:

- **Resource efficiency:** Training one model for multiple tasks is more resource-efficient than training separate models for each task
- **Consistency:** Having a single model perform multiple related tasks can lead to consistency in the performance and integration of the model's outputs
- **Cross-task learning:** The model can leverage what it learns from one task to improve its performance on another, which is a form of inductive transfer that can improve overall learning efficiency

Challenges and solutions

MTL faces challenges:

- **Task interference:** A significant challenge in MTL is task interference, where the learning of one task negatively impacts the performance of another. Advanced regularization techniques and architecture designs, such as task-specific batch normalization and soft parameter sharing, are explored to mitigate this issue.
- **Optimizing task weighting:** Determining the right balance in learning between tasks remains a challenge. Adaptive weighting methods, which dynamically adjust the importance of each task's loss function during training, are being developed to address this.

Continual learning models

Continual learning models, also known as lifelong learning models, are designed to learn continuously from a stream of data, acquiring, retaining, and transferring knowledge across tasks over time. The primary challenge these models address is avoiding catastrophic forgetting, which occurs when a model learns a new task at the expense of previously learned tasks.

Key characteristics

Here are the key characteristics of continual learning models:

- **Knowledge retention:** The main goal of knowledge retention in continual learning models is to overcome what's known as catastrophic forgetting, which is the tendency of a neural network to completely forget old knowledge upon learning new information. Here's how knowledge retention is typically addressed:
 - **Replay mechanisms (experience replay):** This technique involves storing data from previously learned tasks and reintroducing it into the learning process periodically. This can prevent the model from forgetting previously learned information. This replay can be done by doing the following:
 - Randomly sampling and reintegrating old data into new training batches
 - Using a generative model to recreate the distribution of previous tasks and using this synthetic data for retraining
 - Maintaining a subset of the original training data for old tasks, where it can be used alongside new task data during further training iterations
 - **Regularization methods:** Regularization strategies are employed to protect the knowledge that the model has already acquired.
- **Elastic weight consolidation (EWC):** This technique adds a penalty to the loss function based on how important each network parameter is to previously learned tasks. It effectively creates a constraint that discourages the model from changing important parameters when learning new information.
- **Synaptic intelligence:** Similar to EWC, synaptic intelligence estimates the importance of each synapse (connection between neurons) for the tasks learned so far and then penalizes changes to the most crucial synapses.
- **Knowledge distillation:** The model's knowledge is distilled and transferred during the training on new tasks. This is often by using the model's predictions as "soft targets" during further training.
- **Architectural approaches:** Some models incorporate architectural strategies to allow for knowledge retention.
- **Progressive neural networks:** These networks grow over time by adding new columns of neurons for new tasks while freezing the columns associated with previous tasks.
- **Dynamic network expansion:** Here, the model architecture is dynamically expanded to accommodate new knowledge, often by adding new neurons or layers when learning new tasks.
- **Complementary Learning Systems (CLS):** This neuroscience-inspired approach involves having dual memory systems in the model – one for

rapid learning and another for slow consolidation of knowledge – akin to the hippocampus and neocortex in the human brain.

Challenges and considerations

There are various challenges and considerations in continual learning:

- **Memory management:** Deciding how much old data to store for replay can be challenging, especially when considering constraints on computational resources and storage
- **Balancing stability and plasticity:** Models must balance the ability to retain old knowledge (stability) with the ability to learn new tasks (plasticity)
- **Task interference:** When tasks are very different, learning a new task might interfere with the performance of old tasks, even when using replay or regularization techniques
- **Data distribution shift:** Adapting to evolving data distribution requires continual learning models to do the following:
 - **Adapt to changes:** Continual learning models must adapt to shifts in data distribution over time, which can be challenging if the shifts are abrupt or unpredictable
 - **Be robust to variability:** Ensuring robustness to changes in data distribution is essential for maintaining model performance across different tasks and environments
- **Evaluation and benchmarking:** You must do the following to ensure an effective model evaluation and comparison process:
 - **Set appropriate benchmarks:** Establishing benchmarks that accurately reflect the model's continual learning capabilities across tasks is essential but challenging due to task variability
 - **Provide consistent evaluation metrics:** Using consistent and relevant metrics to evaluate performance over time is critical to assess the model's effectiveness in learning and retaining knowledge
 - **Perform comparative analysis:** Conducting comparative analysis with other models and techniques is necessary to understand the relative strengths and weaknesses of the approach

The capacity for knowledge retention in continual learning models is critical for their development and deployment in real-world applications, where they must adapt to new data and tasks over time. By employing strategies such as replay mechanisms, regularization methods, and architectural adjustments, these models aim to retain

previously learned information while continually incorporating new knowledge. This area is still under active research, with many promising approaches being explored to tackle the inherent challenges of continual learning.

Applications

Continual learning is a transformative approach in AI with wide-ranging applications across different fields. By enabling models to learn incrementally and adapt to new data without forgetting previous knowledge, continual learning models can be applied to many real-world scenarios where adaptability and the ability to learn from new experiences are crucial. Here are some applications of continual learning:

- **Personalized recommendation systems:**
 - **Adapting to user behavior:** Continual learning allows recommendation systems to adapt to changing user preferences over time, which is essential as interests and behaviors evolve.
 - **Dynamic content:** As new content is constantly being created, recommendation systems must continually learn to include these in their suggestions. Continual learning methods ensure that the system can integrate new content without losing its ability to recommend older but still relevant items.
 - **Long-term user satisfaction:** By retaining knowledge of a user's historical preferences while adapting to their current interests, continual learning helps in maintaining long-term user satisfaction and engagement.
- **Autonomous robots:**
 - **Real-world interaction:** Robots operating in real-world environments encounter dynamic and unforeseen situations. Continual learning enables them to accumulate experience and improve their decision-making processes over time.
 - **Skill acquisition:** As robots are exposed to new tasks, they need to acquire new skills without forgetting the old ones. Continual learning models can help them integrate these new skills seamlessly.
 - **Environment adaptation:** For robots that navigate varying environments, the ability to learn from these new experiences and adjust their models accordingly is facilitated by continual learning techniques.
- **Healthcare monitoring:**
 - **Patient data analysis:** Continual learning can be applied to monitor patients' health over time, adjusting to new data such as changes in vital

signs or the progression of a disease, to provide timely and personalized healthcare

- **Adapting treatment plans:** As more patient data becomes available, healthcare models can use continual learning to adjust treatment plans based on the effectiveness of previous treatments and evolving health conditions
- **Financial markets:**
 - **Market trend analysis:** Financial models need to adapt to ever-changing market conditions. Continual learning allows these models to assimilate new market data continuously, helping to predict trends and make informed decisions.
 - **Risk management:** Continual learning helps in adjusting risk models in finance as new financial instruments are introduced and market dynamics change.
- **Automotive systems:**
 - **Self-driving cars:** Continual learning is key for self-driving car algorithms that must adapt to diverse and changing driving conditions, traffic patterns, and pedestrian behaviors
- **Online services:**
 - **Content moderation:** Services that rely on content moderation must constantly update their models to understand new slang, symbols, and changing contexts. Continual learning enables these systems to evolve with the language and societal norms.
- **Education:**
 - **Adaptive learning platforms:** Educational platforms can use continual learning to personalize the learning experience, adapting to the changing proficiency and learning speed of each student
- **Software applications:**
 - **User interface adaptation:** Software applications can use continual learning to adapt their interfaces based on user behavior patterns, creating a more personalized and efficient user experience

In each of these applications, the primary advantage of continual learning is its dynamic adaptability, which ensures that models remain relevant and effective over time as they encounter new information. This adaptability is particularly important in our rapidly changing world, where static models can quickly become outdated.

Continual learning represents a significant step toward more intelligent, responsive, and personalized AI systems.

Challenges and solutions

Continual learning has various challenges:

- **Memory overhead:** As models learn new tasks, the requirement for memory and computational resources can grow substantially. Methods such as dynamic network expansion, which selectively grows the model's capacity, and memory-efficient experience replay techniques are under development to manage this growth sustainably.
- **Balancing stability and plasticity:** Continual learning models must balance the need to retain previously learned information (stability) with the need to adapt to new information (plasticity). Techniques such as synaptic intelligence, which measures and protects the contribution of synapses to task performance, aim to strike this balance effectively.

Synergistic potential and future horizons

The integration of multitasking and continual learning models holds the promise of creating AI systems that are not only versatile across a broad range of tasks but also adaptive to new challenges over time. This synergy could lead to the development of AI that is more akin to human intelligence and capable of lifelong learning and adaptation.

Emerging research directions

Emerging research combines the following aspects:

- **Meta-learning:** Combining MTL and continual learning with meta-learning strategies, which involve learning to learn, can potentially lead to systems that rapidly adapt to new tasks with minimal data and without forgetting
- **Neurosymbolic AI:** Integrating these models with neurosymbolic AI, which combines neural networks with symbolic reasoning, offers a pathway to more robust understanding and reasoning capabilities, further bridging the gap between AI and human-like intelligence

The ongoing research and development in multitasking and continual learning models is paving the way for AI systems that are not only more efficient and capable across a range of tasks but also adaptable and resilient in the face of new challenges. This progress underscores the continuous push toward AI systems that can seamlessly

integrate into dynamic real-world environments, offering solutions that are both innovative and practically viable.

Integration of multitasking and continual learning

Integrating multitasking and continual learning could lead to models that are not only capable of learning multiple tasks simultaneously but also capable of adapting to new tasks over time without forgetting previous knowledge. This integration represents a significant step toward more flexible, efficient, and human-like AI systems.

Research continues to explore ways to improve these models, including developing more effective strategies for knowledge transfer, preventing catastrophic forgetting, and efficiently managing computational resources. The synergy between multitasking and continual learning models is poised to drive advancements in AI, enabling the development of more robust, adaptable, and intelligent systems.

Case study – implementing multitasking and continual learning models for e-commerce personalization

Let's consider a case study that focuses on the retail industry.

Background

In a hypothetical case study, an e-commerce giant aimed to refine its customer experience by providing personalized shopping experiences. The objective was to develop a system capable of handling various aspects of customer interaction, from product recommendations to customer service inquiries.

Challenge

The challenge was twofold: the model needed to manage multiple tasks relevant to the e-commerce environment and also adapt to evolving customer behaviors and inventory changes over time without forgetting previous interactions.

Solution – multitasking and continual learning models

To tackle this, the company adopted a combination of multitasking and continual learning models. This involved the following phases:

- **Phase 1 – multitasking model development:**
 - **Integrated task learning:** The model was designed to handle product recommendations, sentiment analysis from customer reviews, and customer service inquiries concurrently
 - **Shared learning architecture:** Early neural network layers were trained to recognize common patterns in customer data, while later layers were dedicated to task-specific processing
- **Phase 2 – implementing continual learning:**
 - **Dynamic data incorporation:** The system was equipped to continuously incorporate new customer interaction data, learning from recent trends and preferences
 - **Replay mechanisms:** To prevent catastrophic forgetting, the model periodically revisited previous customer data to retain historical knowledge
 - **Regular model updates:** The model architecture allowed for periodic updates without complete retraining, adapting to new products and customer service scenarios

Results

The implementation led to a robust, multifaceted AI system that did the following:

- Increased product recommendation accuracy by 40%, enhancing cross-selling and upselling opportunities
- Improved customer service response times by 30%, with more accurate and helpful responses
- Demonstrated significant retention of customer preferences over time, leading to a more personalized shopping experience

Impact

The multitasking and continual learning models had substantial impacts:

- **Customer experience:** The system's ability to provide accurate recommendations and timely customer service improved overall customer satisfaction
- **Business insights:** Continual learning from customer interactions provided valuable insights into buying patterns, helping inform inventory and marketing strategies

- **Operational efficiency:** By handling multiple tasks within one model, the company streamlined its operations, reducing the need for separate systems and teams

Conclusion

This case study highlights the successful application of multitasking and continual learning models in an e-commerce setting, addressing complex customer interaction needs while adapting to an ever-changing market landscape. The combination of these AI methodologies provided a competitive edge, not only by enhancing the user experience but also by offering a scalable solution for personalized customer engagement that evolves with consumer behavior and market demands.

Next, we'll introduce a case study that addresses training an LLM for a specialized domain.

Case study – training an LLM for a specialized domain

Training an LLM for a specialized domain involves a series of intricate steps, meticulous planning, and strategic implementation to ensure the model's effectiveness in understanding and generating text relevant to the specific field. This process can be dissected into several phases, each of which is crucial for the model's development and eventual performance. Let's explore a hypothetical case study that illustrates how an LLM could be trained for a specialized domain, such as medical research:

- **Phase 1 – defining the objectives and scope:**
 - **Objective setting:** The first step involves clearly defining the objectives of the LLM within the specialized domain. For instance, in the medical research domain, the model could aim to assist in generating medical research papers, interpreting clinical study results, or answering medical inquiries.
 - **Scope determination:** Deciding the scope involves specifying the breadth and depth of knowledge the model needs. For a medical research LLM, the scope could range from general medical knowledge to specific sub-fields, such as oncology or genomics.
- **Phase 2 – data collection and preparation:**
 - **Data sourcing:** Collecting a comprehensive and high-quality dataset is paramount. For medical research, this could involve gathering a wide array of texts, including research papers, clinical trial reports, medical journals, and textbooks.

- **Data cleaning and preprocessing:** The collected data must be cleaned and preprocessed to remove irrelevant information, correct errors, and standardize formats. This step is crucial to ensure the model learns from accurate and relevant data.
- **Data annotation:** Annotating data with metadata, such as topics or categories, can help in training more refined and context-aware models. For specialized domains, expert annotators are often required to ensure the accuracy of annotations.
- **Phase 3 – model selection and training:**
 - **Choosing a base model:** Selecting an appropriate base model or architecture is critical. For a specialized LLM, starting with a pre-trained model that has been trained on a broad dataset and then fine-tuning it for the specialized domain often yields the best results.
 - **Fine-tuning for the domain:** Fine-tuning involves adjusting the pre-trained model on the specialized dataset. This step adapts the model's weights and biases to better reflect the nuances of the domain's language and knowledge.
 - **Evaluation and iteration:** Continuously evaluating the model's performance through metrics such as accuracy, fluency, and relevance is essential. Feedback loops help in iteratively refining the model through additional training or data adjustments.
- **Phase 4 – implementation and deployment:**
 - **Integration into applications:** Deploying the trained LLM involves integrating it into applications or workflows, where it can assist professionals in the domain. For medical research, this could be within systems for drafting research papers, providing clinical decision support, or educational tools.
 - **Monitoring and updating:** Post-deployment, the model's performance must be monitored to ensure it continues to meet the required standards. Over time, incorporating new data and research findings into the training dataset helps the model remain current and valuable.

Challenges and considerations

The following are some issues that you should be aware of:

- **Ethical and privacy concerns:** In sensitive domains such as medicine, ethical considerations and patient privacy are paramount. Ensuring data de-identification and compliance with regulations such as HIPAA is essential.

- **Bias and fairness:** Training data in specialized domains can contain biases. Active measures must be taken to identify and mitigate these biases to ensure the model's outputs are fair and unbiased.
- **Domain expertise:** Involving domain experts throughout the training process is critical for ensuring the relevance and accuracy of the model's outputs. Their insights can guide data collection, annotation, and evaluation processes.

Conclusion

Training an LLM for a specialized domain is a complex, multidisciplinary endeavor that requires careful planning, domain expertise, and continuous iteration. The process from defining objectives to deploying and maintaining the model involves various challenges, including ethical considerations, data quality, and model bias. However, when executed well, the result can be a powerful tool that enhances decision-making, accelerates research, and improves outcomes within the specialized domain. The hypothetical case study of training an LLM for medical research highlights the potential of specialized LLMs to contribute significantly to their respective fields, underscoring the importance of targeted training and the nuanced approach required to harness the full capabilities of LLMs.

Chapter 30

Innovative Applications of LLMs

Here, we are going to see how LLMs are revolutionizing the world of software development, leading to a new era of AI-powered applications.

How LLMs are changing software development

LLMs have proven to have extraordinary capabilities: from natural language understanding tasks (summarization, named entity recognition, and classification) to text generation, from common-sense reasoning to brainstorming skills. However, they are not just incredible by themselves. As discussed earlier, LLMs and, generally speaking, **large foundation models (LFMs)**, are revolutionizing software development by serving as platforms for building powerful applications.

In fact, instead of starting from scratch, today developers can make API calls to a hosted version of an LLM, with the option of customizing it for their specific needs, as we saw in the previous chapter. This shift allows teams to incorporate the power of AI more easily and efficiently into their applications, similar to the transition from single-purpose computing to time-sharing in the past.

But what does it mean, concretely, to incorporate LLMs within applications? There are two main aspects to consider when incorporating LLMs within applications:

- **The technical aspect**, which covers the *how*. Integrating LLMs into applications involves embedding them through REST API calls and managing them with AI orchestrators. This means setting up architectural components that allow seamless communication with the LLMs via API calls. Additionally, using AI orchestrators helps to efficiently manage and coordinate the LLMs' functionality within the application.
- **The conceptual aspect**, which covers the *what*. LLMs bring a plethora of new capabilities that can be harnessed within applications. One way to view LLMs' impact is by considering them as a new category of software, often referred to as *copilot*. This categorization highlights the significant assistance and collaboration provided by LLMs in enhancing application functionalities.

The copilot system

The copilot system is a new category of software that serves as an expert helper to users trying to accomplish complex tasks. This concept was coined by Microsoft and has already been introduced into its applications, such as M365 Copilot and the new Bing, now powered by GPT-4. With the same framework that is used by these products, developers can now build their own copilots to embed within their applications.

But what exactly is a copilot?

As the name suggests, copilots are meant to be AI assistants that work side by side with users and support them in various activities, from information retrieval to blog writing and posting, from brainstorming ideas to code review and generation.

The following are some unique features of copilots:

- **A copilot is powered by LLMs**, or, more generally, LfMs, meaning that these are the reasoning engines that make the copilot “intelligent.” This reasoning engine is one of its components, but not the only one. A copilot also relies on other technologies, such as apps, data sources, and user interfaces, to provide a useful and engaging experience for users. The following illustration shows how this works:

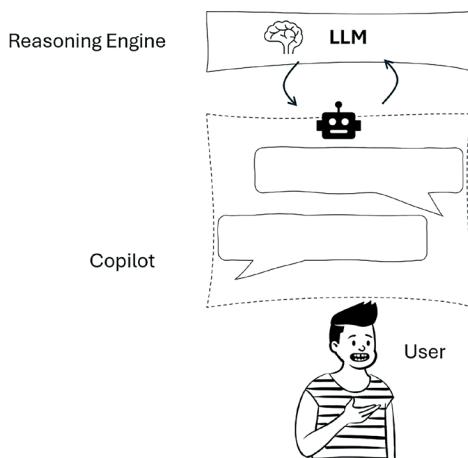


Figure 30.1: A copilot is powered by an LLM

- **A copilot is designed to have a conversational user interface**, allowing users to interact with it using natural language. This reduces or even eliminates the knowledge gap between complex systems that need domain-specific taxonomy (for example, querying tabular data needs the knowledge of programming languages such as T-SQL) and users. Let's look at an example of such a conversation:

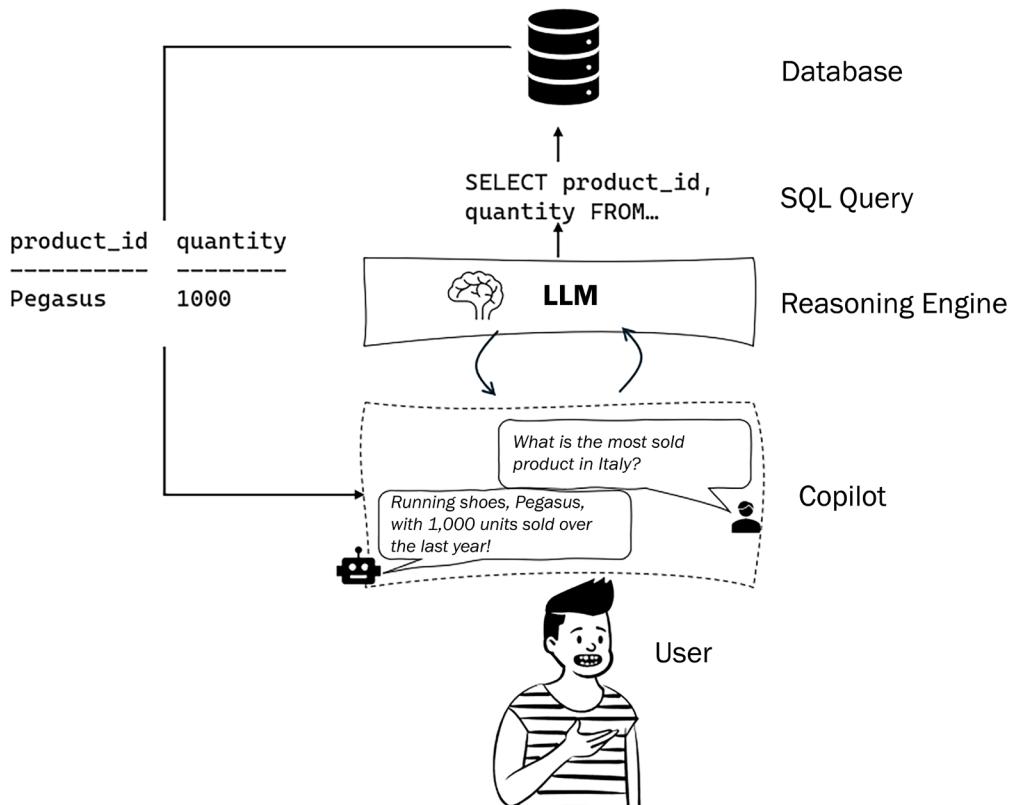


Figure 30.2: An example of a conversational UI to reduce the gap between the user and the database

- **A copilot has a scope.** This means that it is **grounded** to domain-specific data so that it is entitled to answer only within the perimeter of the application or domain.

Definition



Grounding is the process of using LLMs with information that is use case specific, relevant, and not available as part of the LLM's trained knowledge. It is crucial for ensuring the quality, accuracy, and relevance of the output. For example, let's say you want an LLM-powered application that assists you during your research on up-to-date papers (not included in the training dataset of your LLM). You also want your app to only respond if the answer is included in those papers. To do so, you will need to ground your LLM to the set of papers, so that your application will only respond within this perimeter.

Grounding is achieved through an architectural framework called retrieval-augmented generation (RAG), a technique that enhances the output of LLMs by incorporating information from an external, authoritative knowledge base before generating a response. This process helps to ensure that the generated content is relevant, accurate, and up to date.



What is the difference between a copilot and a RAG? RAG can be seen as one of the architectural patterns that feature a copilot. Whenever we want our copilot to be grounded to domain-specific data, we use a RAG framework. Note that RAG is not the only architectural pattern that can feature a copilot: there are further frameworks such as function calling or multi-agents.

For example, let's say we developed a copilot within our company that allows employees to chat with their enterprise knowledge base. As fun as it can be, we cannot provide users with a copilot they can use to plan their summer trip (it would be like providing users with a ChatGPT-like tool at our own hosting cost!); on the contrary, we want the copilot to be grounded only to our enterprise knowledge base so that it can respond only if the answer is pertinent to the domain-specific context.

The following figure shows an example of grounding a copilot system:

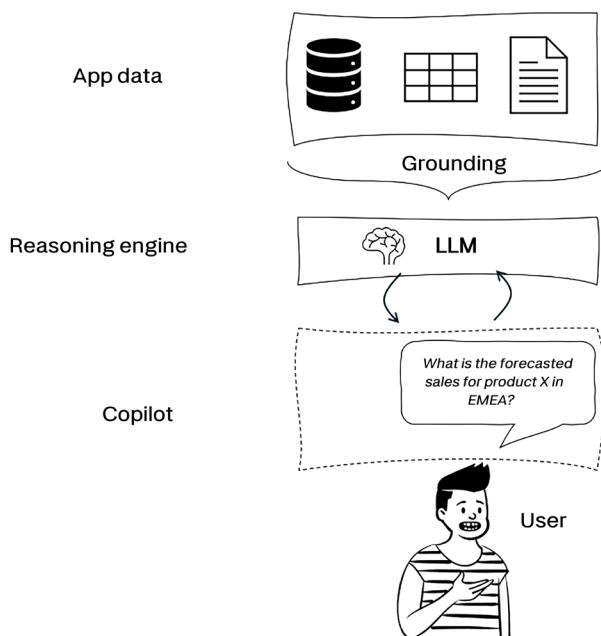


Figure 30.3: Example of grounding a copilot

- The copilot's capabilities can be extended by skills, which can be code or calls to other models. In fact, the LLM (our reasoning engine) might have two kinds of limitations:
 - **Limited parametric knowledge.** This is due to the knowledge base cutoff date, which is a physiological feature of LLMs. In fact, their training dataset will always be "outdated," not in line with the current trends. This can be overcome by adding non-parametric knowledge with grounding, as previously seen.
 - **Lack of executive power.** This means that LLMs by themselves are not empowered to carry out actions. Let's consider, for example, the well-known ChatGPT: if we ask it to generate a LinkedIn post about productivity tips, we will then need to copy and paste it onto our LinkedIn profile as ChatGPT is not able to do so by itself. That is the reason why we need plug-ins. Plug-ins are LLMs' connectors toward the external world that serve not only as input sources to extend LLMs' non-parametric knowledge (for example, to allow a web search) but also as output sources so that the copilot can actually execute actions. For example, with a LinkedIn plug-in, our copilot powered by an LLM will be able not only to generate the post but also to post it online.

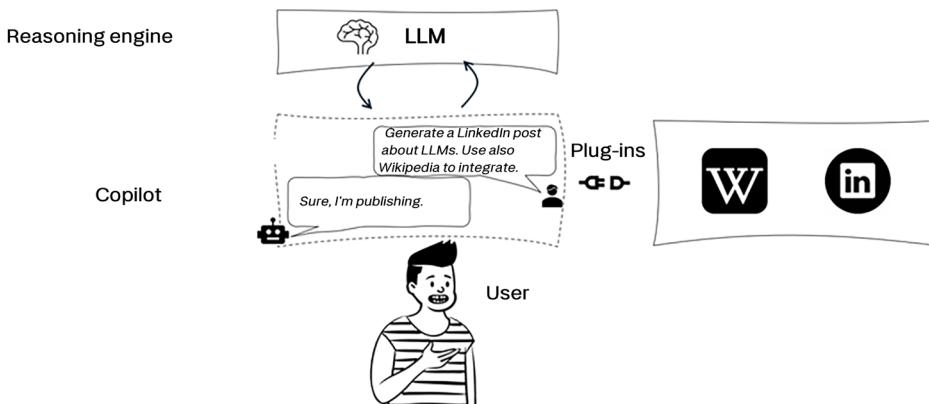


Figure 30.4: Example of Wikipedia and LinkedIn plug-ins

Note that the user's prompt in natural language is not the only input the model processes. In fact, it is a crucial component of the backend logic of our LLM-powered applications and the set of instructions we provide to the model. This *metaprompt* or system message is the object of a new discipline called **prompt engineering**.



Definition

Prompt engineering is the process of designing and optimizing prompts to LLMs for a wide variety of applications and research topics. Prompts are short pieces of text that are used to guide the LLM's output. Prompt engineering skills help to better understand the capabilities and limitations of LLMs.

Prompt engineering involves selecting the right words, phrases, symbols, and formats that elicit the desired response from the LLM. Prompt engineering also involves using other controls, such as parameters, examples, or data sources, to influence the LLM's behavior. For example, if we want our LLM-powered application to generate responses for a 5-year-old child, we can specify this in a system message similar to "Act as a teacher who explains complex concepts to 5-year-old children."

In fact, Andrej Karpathy, the previous Director of AI at Tesla, who returned to OpenAI in February 2023, tweeted that "English is the hottest new programming language."

Introducing AI orchestrators to embed LLMs into applications

Earlier in this chapter, we saw that there are two main aspects to consider when incorporating LLMs within applications: a technical aspect and a conceptual aspect. While we can explain the conceptual aspect with the brand-new category of software called Copilot, in this section, we are going to further explore how to technically embed and orchestrate LLMs within our applications.

The main components of AI orchestrators

From one side, the paradigm shift of foundation models implies a great simplification in the domain of AI-powered applications: after producing models, now the trend is consuming models. On the other side, many roadblocks might arise in developing this new kind of AI, since there are LLM-related components that are brand new and have never been managed before within an application life cycle. For example, there might be malicious actors that could try to change the LLM instructions (the system message mentioned earlier) so that the application does not follow the correct instructions. This is an example of a new set of security threats that are typical to LLM-powered applications and need to be addressed with powerful counterattacks or preventive techniques.

The following is an illustration of the main components of such applications:

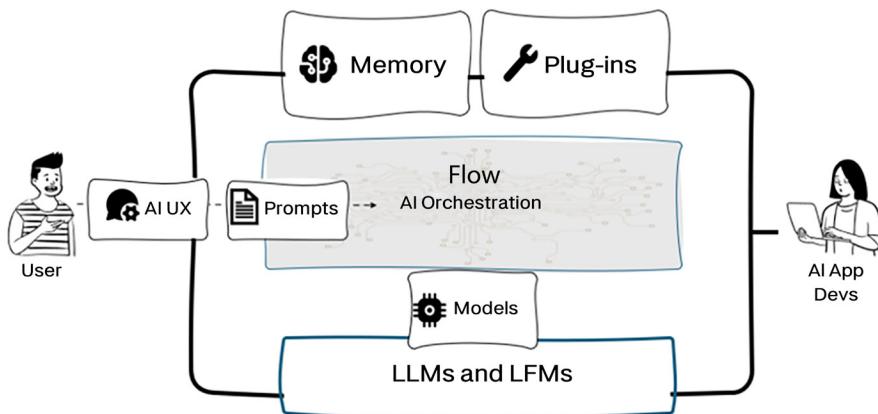


Figure 30.5: High-level architecture of LLM-powered applications

Let's inspect each of these components in detail:

- **Models:** The model is simply the type of LLM we decide to embed in our application. There are two main categories of models:
 - **Proprietary LLMs:** Models that are owned by specific companies or organizations. Examples include GPT-3 and GPT-4, developed by OpenAI, or Bard, developed by Google. As their source code and architecture are not available, those models cannot be re-trained from scratch on custom data, yet they can be fine-tuned if needed.
 - **Open-source:** Models with code and architecture freely available and distributed, hence they can also be trained from scratch on custom data. Examples include Falcon LLM, developed by Abu Dhabi's **Technology Innovation Institute (TII)**, or LLaMA, developed by Meta.
- **Memory:** LLM applications commonly use a conversational interface, which requires the ability to refer back to earlier information within the conversation. This is achieved through a “memory” system that allows the application to store and retrieve past interactions. Note that past interactions could also constitute additional non-parametric knowledge to be added to the model. To achieve that, it is important to store all the past conversations – properly embedded – into VectorDB, which is at the core of the application’s data.

Definition

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



FAISS, developed by Facebook (now Meta) in 2017, was one of the pioneering vector databases. It was designed for efficient similarity search and clustering of dense vectors and is particularly useful for multimedia documents and dense embeddings. It was initially an internal research project at Facebook. Its primary goal was to better utilize GPUs for identifying similarities related to user preferences. Over time, it evolved into the fastest available library for similarity search and can handle billion-scale datasets. FAISS has opened up possibilities for recommendation engines and AI-based assistant systems.

- **Plug-ins:** They can be seen as additional modules or components that can be integrated into the LLM to extend its functionality or adapt it to specific tasks and applications. These plug-ins act as add-ons, enhancing the capabilities of the LLM beyond its core language generation or comprehension abilities.

The idea behind plug-ins is to make LLMs more versatile and adaptable, allowing developers and users to customize the behavior of the language model for their specific needs. Plug-ins can be created to perform various tasks, and they can be seamlessly incorporated into the LLM's architecture.

- **Prompts:** This is probably the most interesting and pivotal component of an LLM-powered application. We've already quoted, in the previous section, Andrej Karpathy's affirmation that "English is the hottest new programming language.". Prompts can be defined at two different levels:
 - **"Frontend," or what the user sees:** A "prompt" refers to the input to the model. It is the way the user interacts with the application, asking things in natural language.
 - **"Backend," or what the user does not see:** Natural language is not only the way to interact, as a user, with the frontend; it is also the way we "program" the backend. In fact, on top of the user's prompt, there are many natural language instructions, or meta-prompts, that we give to the model so that it can properly address the user's query. Meta-prompts are meant to instruct the model to act as it is meant to. For example, if we want to limit

our application to answer only questions related to the documentation we provided in VectorDB, we will specify the following in our meta-prompts to the model: “*Answer only if the question is related to the provided documentation.*”

Finally, we get to the core of the high-level architecture shown in *Figure 30.5*, that is, the **AI orchestrator**. With the AI orchestrator, we refer to lightweight libraries that make it easier to embed and orchestrate LLMs within applications.

As LLMs went viral by the end of 2022, many libraries started arising in the market. In the next sections, we are going to focus on three of them: LangChain, Semantic Kernel, and Haystack.

LangChain

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

Let’s take a look at the key components of LangChain:

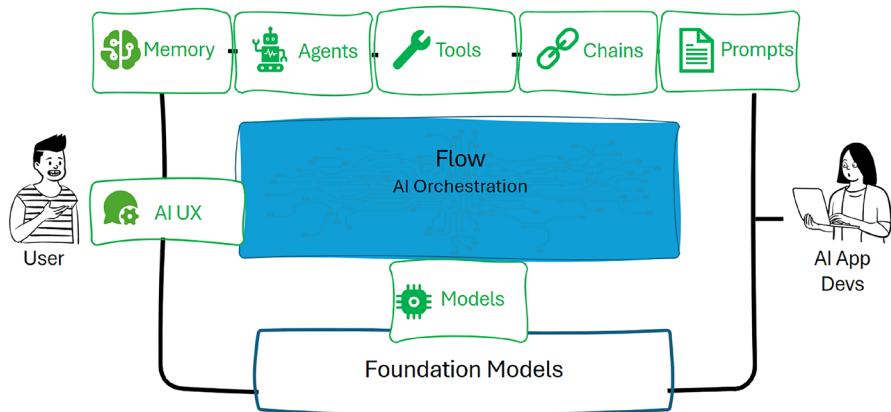


Figure 30.6: LangChain’s components

Overall, LangChain has the following core modules:

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

Definition



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

Alongside models, LangChain also offers many prompt-related components that make it easier to manage the prompt flow.

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

LangChain offers the following benefits:

- LangChain provides modular abstractions for the components we previously mentioned that are necessary to work with language models, such as prompts, memory, and plug-ins.
- Alongside those components, LangChain also offers pre-built **chains**, which are structured concatenations of components. Those chains can be pre-built for specific use cases or be customized.

Haystack

Haystack is a Python-based framework developed by Deepset, a startup founded in 2018 in Berlin by Milos Rusic, Malte Pietsch, and Timo Möller. Deepset provides developers with the tools to build **natural language processing (NLP)**-based applications, and with the introduction of Haystack, they are taking them to the next level.

The following illustration shows the core components of Haystack:

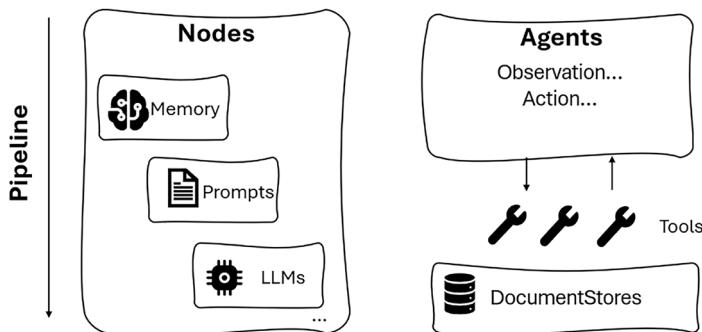


Figure 30.7: Haystack's components

Let's look at these components in detail:

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

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

Some of the benefits offered by Haystack are:

- **Ease of use:** Haystack is user-friendly and straightforward. It's often chosen for lighter tasks and rapid prototypes.
- **Documentation quality:** Haystack's documentation is considered high-quality, aiding developers in building search systems, question-answering, summarization, and conversational AI.
- **End-to-end framework:** Haystack covers the entire LLM project life cycle, from data preprocessing to deployment. It's ideal for large-scale search systems and information retrieval.
- Another nice thing about Haystack is that you can deploy it as a REST API and it can be consumed directly.

Semantic Kernel

Semantic Kernel is the third open-source SDK we are going to explore in this chapter. It was developed by Microsoft, originally in C# and now also available in Python.

This framework takes its name from the concept of a “kernel,” which, generally speaking, refers to the core or essence of a system. In the context of this framework, a kernel is meant to act as the engine that addresses a user’s input by chaining and concatenating a series of components into pipelines, encouraging **function composition**.

Definition



In mathematics, function composition is a way to combine two functions to create a new function. The idea is to use the output of one function as the input to another function, forming a chain of functions. The composition of two functions f and g is denoted as $(f \circ g)$, where the function g is applied first, followed by the function f . $f \rightarrow (f \circ g)(x) = f(g(x))$.

Function composition in computer science is a powerful concept that allows for the creation of more sophisticated and reusable code by combining smaller functions into larger ones. It enhances modularity and code organization, making programs easier to read and maintain.

The following is an illustration of the anatomy of Semantic Kernel:

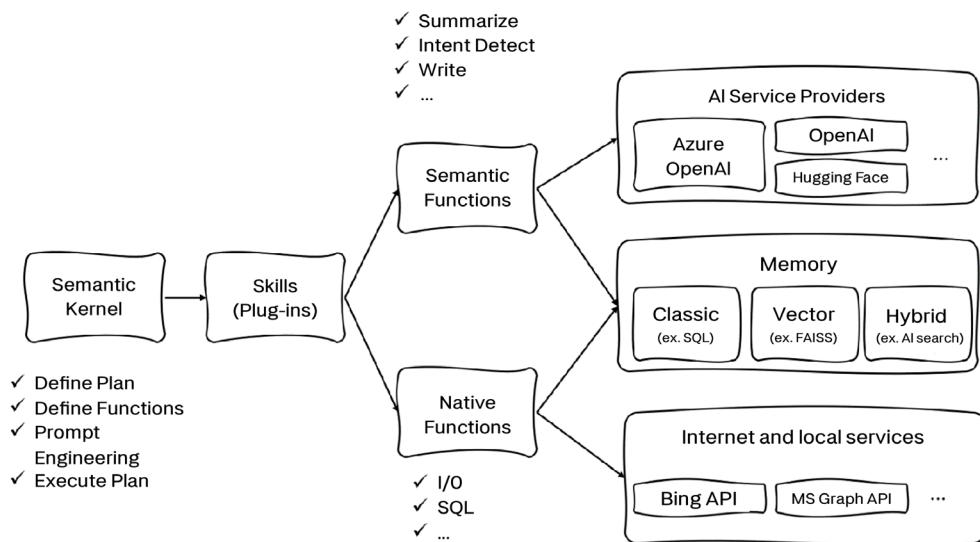


Figure 30.8: Anatomy of Semantic Kernel

Semantic Kernel has the following main components:

- **Models:** These are the LLMs or LFMIs that will be the engine of the application. Semantic Kernel supports proprietary models, such as those available in OpenAI and Azure OpenAI, and open-source models consumable from the Hugging Face Hub.
- **Memory:** It allows the application to keep references to the user's interactions, both in the short and long term. Within the framework of Semantic Kernel, memories can be accessed in three ways:
 - **Key-value pairs:** This consists of saving environment variables that store simple information, such as names or dates.
 - **Local storage:** This consists of saving information to a file that can be retrieved by its filename, such as a CSV or JSON file.
 - **Semantic memory search:** This is similar to LangChain's and Haystack's memory, as it uses embeddings to represent and search for text information based on its meaning.
- **Functions:** Functions can be seen as skills that mix LLM prompts and code, with the goal of making users' asks interpretable and actionable. There are two types of functions:

- **Semantic functions:** These are a type of templated prompt, which is a natural language query that specifies the input and output format for the LLM, also incorporating prompt configuration, which sets the parameters for the LLM.
- **Native functions:** These refer to the native computer code that can route the intent captured by the semantic function and perform the related task.

To make an example, a semantic function could ask the LLM to write a short paragraph about AI, while a native function could actually post it on social media like LinkedIn.

- **Plug-ins:** These are connectors toward external sources or systems that are meant to provide additional information or the ability to perform autonomous actions. Semantic Kernel offers out-of-the-box plug-ins, such as the Microsoft Graph connector kit, but you can build a custom plug-in by leveraging functions (both native and semantic, or a mix of the two).
- **Planner:** As LLMs can be seen as reasoning engines, they can also be leveraged to auto-create chains or pipelines to address new users' needs. This goal is achieved with a planner, which is a function that takes as input a user's task and produces the set of actions, plugins, and functions needed to achieve the goal.

Some benefits of Semantic Kernel are:

- **Lightweight and C# support:** Semantic Kernel is more lightweight and includes C# support. It's a great choice for C# developers or those using the .NET framework.
- **Wide range of use cases:** Semantic Kernel is versatile, supporting various LLM-related tasks.
- **Industry-led:** Semantic Kernel was developed by Microsoft, and it is the framework the company used to build its own copilots. Hence, it is mainly driven by industry needs and asks, making it a solid tool for enterprise-scale applications.

How to choose a framework

Overall, the three frameworks offer, more or less, similar core components, sometimes called by a different taxonomy, yet covering all the blocks illustrated within the concept of the copilot system. So, a natural question might be: "Which one should I use to build my LLM-powered application?" Well, there is no right or wrong answer! All three are extremely valid. However, there are some features that might be more relevant

for specific use cases or developers' preferences. The following are some criteria you might want to consider:

- **The programming language you are comfortable with or prefer to use:** Different frameworks may support different programming languages or have different levels of compatibility or integration with them. For example, Semantic Kernel supports C#, Python, and Java, while LangChain and Haystack are mainly based on Python (even though LangChain also introduced JS/TS support). You may want to choose a framework that matches your existing skills or preferences, or that allows you to use the language that is most suitable for your application domain or environment.
- **The type and complexity of the natural language tasks you want to perform or support:** Different frameworks may have different capabilities or features for handling various natural language tasks, such as summarization, generation, translation, reasoning, etc. For example, LangChain and Haystack provide utilities and components for orchestrating and executing natural language tasks, while Semantic Kernel allows you to use natural language semantic functions to invoke LLMs and services. You may want to choose a framework that offers the functionality and flexibility you need or want for your application goals or scenarios.
- **The level of customization and control you need or want over the LLMs and their parameters or options:** Different frameworks may have different ways of accessing, configuring, and fine-tuning the LLMs and their parameters or options, such as model selection, prompt design, inference speed, output format, etc. For example, Semantic Kernel provides connectors that make it easy to add memories and models to your AI app, while LangChain and Haystack allow you to plug in different components for the document store, retriever, reader, generator, summarizer, and evaluator. You may want to choose a framework that gives you the level of customization and control you need or want over the LLMs and their parameters or options.
- **The availability and quality of the documentation, tutorials, examples, and community support for the framework:** Different frameworks may have different levels of documentation, tutorials, examples, and community support that can help you learn, use, and troubleshoot the framework. For example, Semantic Kernel has a website with documentation, tutorials, examples, and a Discord community; LangChain has a GitHub repository with documentation, examples, and issues; Haystack has a website with documentation, tutorials, demos, blog posts, and a Slack community. You may want to choose a framework that has the availability and quality of documentation, tutorials, examples, and community support that can help you get started and solve problems with the framework.

Let's briefly summarize the differences between these orchestrators:

Feature	LangChain	Haystack	Semantic Kernel
LLM support	Proprietary and open-source	Proprietary and open source	Proprietary and open source
Supported languages	Python and JS/TS	Python	C#, Java, and Python
Process orchestration	Chains	Pipelines of nodes	Pipelines of functions
Deployment	No REST API	REST API	No REST API
Feature	LangChain	Haystack	Semantic Kernel

Table 30.1: Comparisons among the three AI orchestrators

Overall, all three frameworks offer a wide range of tools and integrations to build your LLM-powered applications, and a wise approach could be to use the one that is most in line with your current skills or the company's overall approach.

Chapter

31

Matching LLMs to Applications

In the last chapter, we saw how pivotal it is to properly orchestrate **large language models (LLMs)** and their components within applications. In fact, we saw that not all LLMs are created equal. The next key decision is which LLMs to actually use. Different LLMs may have different architectures, sizes, training data, capabilities, and limitations. Choosing the right LLM for your application is not a trivial decision, as it can have a significant impact on the performance, quality, and cost of your solution.

The most promising LLMs in the market

The last year has witnessed an unprecedented surge in the research and development of LLMs. Several new models have been released or announced by different organizations, each with its own features and capabilities. Some of these models are the largest and most advanced ever created, surpassing the previous **state-of-the-art (SOTA)** by orders of magnitude. Others are lighter yet more specialized in specific tasks.

Proprietary models

Proprietary LLMs are developed and owned by private companies, and they are not disclosed with code. They are also typically subject to a fee for consumption.

Proprietary models offer a series of advantages, including better support and maintenance as well as safety and alignment. They also tend to outperform open-source models in terms of generalization, because of their complexity and training datasets. On the other hand, they act as a “black box,” meaning that owners do not disclose the source code to developers.

In the next sections, we will cover three of the most popular proprietary LLMs in the market, as of August 2023.

GPT-4

Released in March 2023, GPT-4 is, together with its newly released “cousin” GPT-4 Turbo, one of the latest models developed by **OpenAI**, is among the top performers in the market (while OpenAI, as confirmed by its CEO Sam Altman, is already working on GPT-5).

It belongs to the class of **generative pretrained transformer (GPT)** models, a decoder-only transformer-based architecture introduced by OpenAI. The following diagram shows the basic architecture:

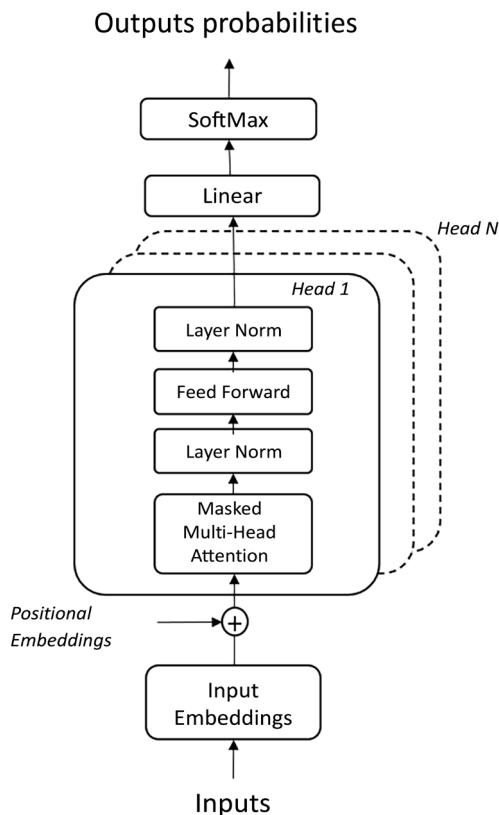


Figure 31.1: High-level architecture of a decoder-only transformer

As you can see from the preceding diagram, the decoder-only architecture still includes the main elements that feature in transformer architecture. However, in this architecture, the model solely comprises a decoder, which is trained to predict the next token in a sequence based on the preceding tokens. Unlike the encoder-decoder architecture, the decoder-only design lacks an explicit encoder for summarizing input

information. Instead, the information is implicitly encoded within the hidden state of the decoder, which is updated at each step during the generation process.

Now, we'll look at some of the improvements in GPT-4 over previous versions.

GPT-4, like the previous models in the GPT series, has been trained on both publicly available and OpenAI-licensed datasets (OpenAI didn't disclose the exact composition of the training set).

Additionally, to make the model more aligned with the user's intent, the training process also involved **reinforcement learning from human feedback (RLHF)** training.

Definition

RLHF is a technique that aims at using human feedback as an evaluating metric for LLMs' generated output and then using that feedback to further optimize the model. There are two main steps to achieve that goal:

1. Training a reward model based on human preferences.
2. Optimizing the LLM with respect to the reward model. This step is done via reinforcement learning and it is a type of machine learning paradigm where an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions, and its goal is to maximize the cumulative reward over time by continuously adapting its behavior through trial and error.



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

As an example, think about ChatGPT. This model integrates various training methods, including unsupervised pretraining, supervised fine-tuning, instruction tuning, and RLHF. The RLHF component involves training the model to predict human preferences by using feedback from human trainers. These trainers review the model's responses and provide ratings or corrections, guiding the model to generate more helpful, accurate, and aligned responses.

For instance, if a language model initially produces an output that is not quite helpful or accurate, human trainers can provide feedback that indicates the preferred output. The model then uses this feedback to adjust its parameters and improve future responses. This process iteratively continues, with the model learning from a series of human judgments to better align with what is considered helpful or appropriate by human standards.

GPT-4 demonstrated outstanding capabilities in commonsense reasoning and analytical skills. It has been benchmarked with SOTA systems, including the **Massive Multitask Language Understanding (MMLU)**. On MMLU, GPT-4 outperformed previous models not only in English, but also in other languages.

The following is an illustration that shows GPT-4's performance on MMLU:

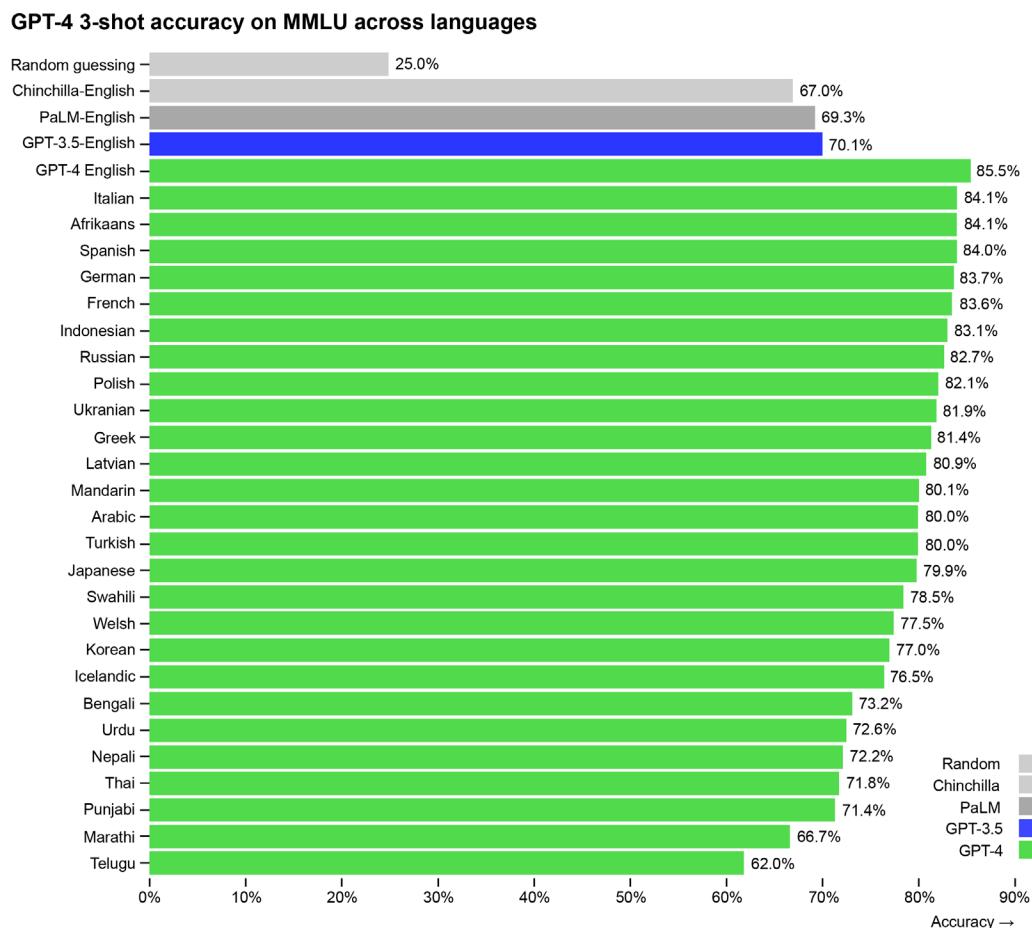


Figure 31.2: GPT-4 3-shot accuracy on MMLU across languages (source: <https://openai.com/research/gpt-4>)

In addition to MMLU, GPT-4 has been benchmarked on a variety of SOTA systems and academic exams, as you can see from the following graph:

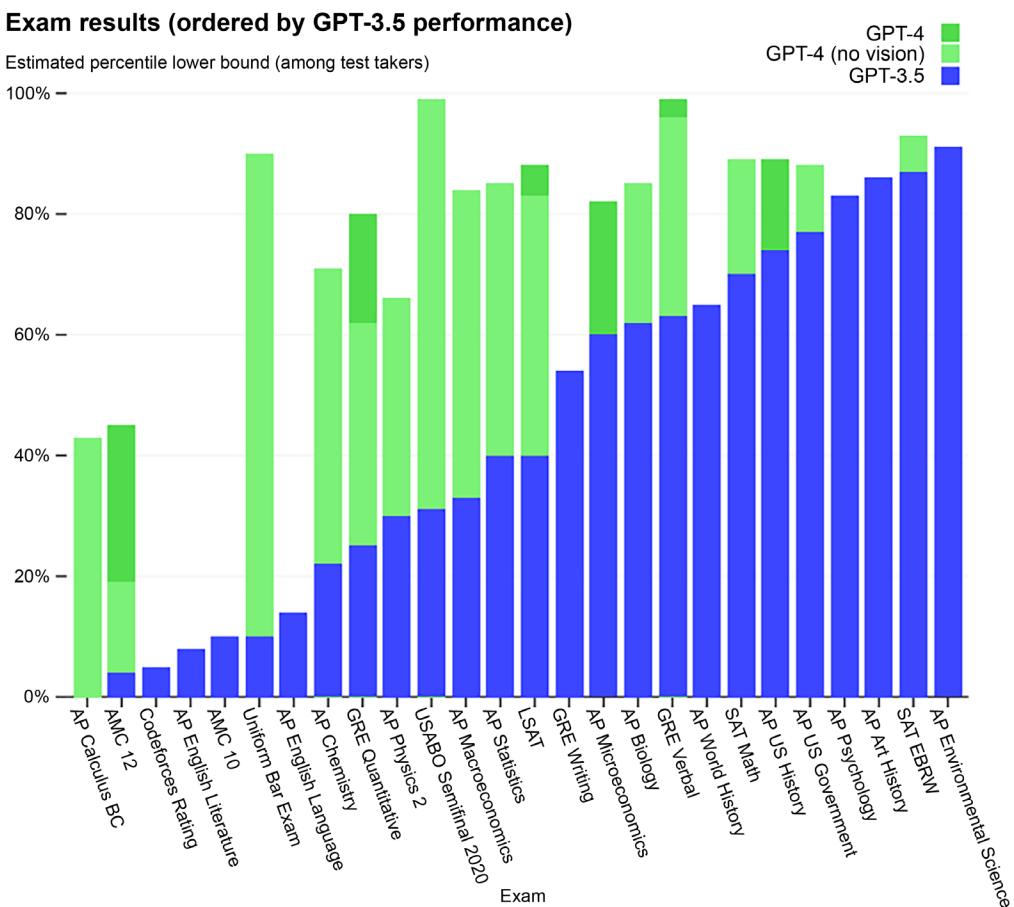


Figure 31.3: GPT performance on academic and professional exams (source: <https://arxiv.org/pdf/2303.08774.pdf>)



Note: in the preceding graph, you can see two versions of GPT-4, vision and no vision (along with the GPT-3.5 for benchmarking purposes). This is because GPT-4 is a multi-modal model, meaning that it can take images as input, in addition to text.

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

Definition



Hallucination is a term that describes a phenomenon where LLMs generate text that is incorrect, nonsensical, or not real, but appears to be plausible or coherent. For example, an LLM may hallucinate a fact that contradicts the source or common knowledge, a name that does not exist, or a sentence that does not make sense. Hallucination can happen because LLMs are not databases or search engines that store or retrieve factual information. Rather, they are statistical models that learn from massive amounts of text data and produce outputs based on the patterns and probabilities they have learned. However, these patterns and probabilities may not reflect the truth or the reality, as the data may be incomplete, noisy, or biased. Moreover, LLMs have limited contextual understanding and memory, as they can only process a certain number of tokens at a time and abstract them into latent representations. Therefore, LLMs may generate text that is not supported by any data or logic but is the most likely or correlated from the prompt.

In fact, even though it is still not 100% reliable, GPT-4 made great improvements with TruthfulQA benchmarks, which test the model's ability to separate fact from incorrect statements.

Here, you can see an illustration that compares GPT-4 results in a TruthfulQA benchmark with those of GPT-3.5 (the model behind OpenAI's ChatGPT) and Anthropic-LM.

Accuracy on adversarial questions (TruthfulQA mc1)

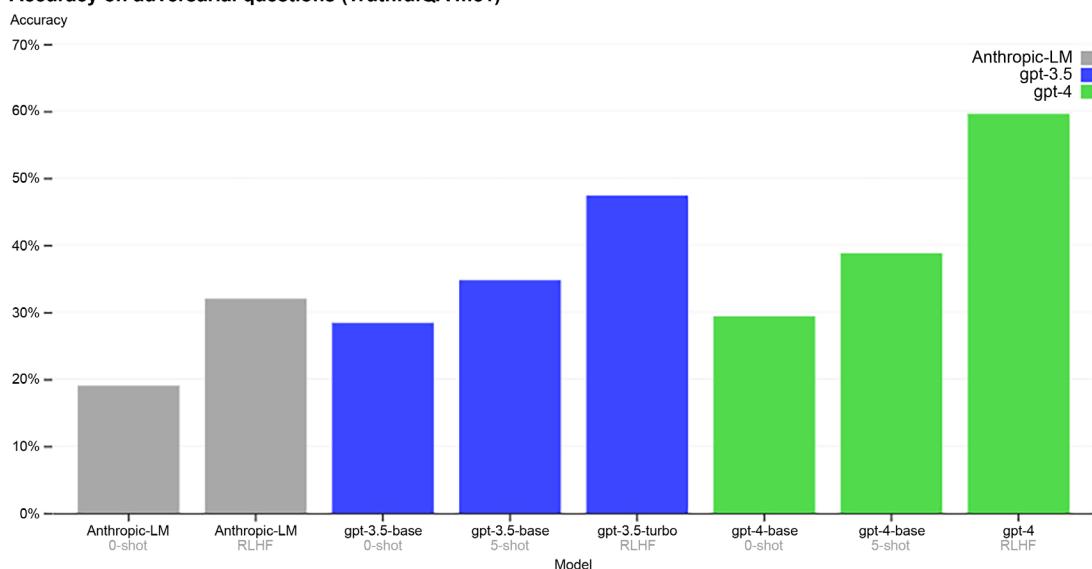


Figure 31.4: Model comparison in TruthfulQA benchmark (source: <https://openai.com/research/gpt-4>)

Finally, with GPT-4, OpenAI made an additional effort to make it safer and more aligned, engaging from the beginning a team of over 50 experts in domains like AI alignment risks, privacy, and cybersecurity, with the goal of understanding the extent of the risks of such a powerful model and how to prevent them.

Definition



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

Thanks to this analysis, further data have been collected and used while training GPT-4 to mitigate its potential risks, resulting in a reduced risk compared to its predecessor, GPT-3.5.

Gemini 1.5

Gemini 1.5 is a SOTA generative AI model developed by Google and released in December 2023. Like GPT-4, Gemini is designed to be multimodal, meaning that it can process and generate content across various modalities, including text, images, audio, video, and code. It is based on a **mixture-of-expert (MoE)** transformer.

Definition



In the context of transformer architecture, MoE refers to a model that incorporates multiple specialized sub-models, known as “experts,” within its layers. Each expert is a neural network designed to handle different types of data or tasks more efficiently. The MoE model uses a gating mechanism or router to determine which expert should process a given input, allowing the model to dynamically allocate resources and specialize in processing certain types of information. This approach can lead to more efficient training and inference, as it enables the model to scale up in size and complexity without a proportional increase in computational cost.

Gemini comes in various sizes, including Ultra, Pro, and Nano, to cater to different computational needs, from data centers to mobile devices. To use Gemini, developers can access it via the APIs provided for different model variants, allowing the integration of its capabilities into applications.

Compared to its previous version, Gemini 1.0, the current model outperforms it in text, vision, and audio tasks, as shown in the following screenshot:

	Core Capability	Relative to	
		1.0 Pro	1.0 Ultra
Text	Math, Science & Reasoning	+28.9%	+5.2%
	Multilinguality	+22.3%	+6.7%
	Coding	+8.9%	+0.2%
	Instruction following	+9.2%	+2.5%
Vision	Image understanding	+6.5%	-4.1%
	Video understanding	+16.9%	+3.8%
Audio	Speech recognition	+1.2%	-5.0%
	Speech translation	+0.3%	-2.2%

Figure 31.5: Gemini 1.5 Pro and Ultra compared to its previous version 1.0 (source: https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf)

Similarly, it has demonstrated outstanding capabilities in domains such as math, science, and reasoning, and coding and multilinguality:

Capability	Benchmark	Gemini		
		1.0 Pro	1.0 Ultra	1.5 Pro
Math, Science & Reasoning	Hellaswag (Zellers et al., 2019)	84.7% 10-shot	87.8% 10-shot	92.5% 10-shot
	MMLU: Multiple-choice questions in 57 subjects (professional & academic). (Hendrycks et al., 2021a)	71.8% 5-shot	83.7% 5-shot	81.9% 5-shot
	GSM8K: Grade-school math problems. (Cobbe et al., 2021)	77.9% 11-shot	88.9% 11-shot	91.7% 11-shot
	MATH: Math problems ranging across 5 levels of difficulty and 7 sub-disciplines. (Hendrycks et al., 2021b)	32.6% 4-shot Minerva prompt	53.2% 4-shot Minerva prompt	58.5% 4-shot Minerva prompt 59.4% 7-shot
	AMC 2022-23: 250 latest problems including 100 AMC 12, 100 AMC 10, and 50 AMC 8 problems.	22.8% 4-shot	30% 4-shot	37.2% 4-shot
	BigBench - Hard: A subset of harder tasks from Big Bench formatted as CoT problems. (Srivastava et al., 2022)	75.0% 3-shot	83.6% 3-shot	84.0% 3-shot
Coding	DROP: Reading comprehension & arithmetic. (Metric: F1-Score). (Dua et al., 2019)	74.1% Variable shots	82.4% Variable shots	78.9% Variable shots
	HumanEval chat preamble* (Metric: pass rate). (Chen et al., 2021)	67.7% 0-shot (PT)	74.4% 0-shot (PT)	71.9% 0-shot
Multilinguality	Natural2Code chat preamble* (Metric: pass rate).	69.6% 0-shot	74.9% 0-shot	77.7% 0-shot
	WMT23: sentence-level machine translation (Metric: BLEURT). (Tom et al., 2023)	71.73 (PT) 1-shot	74.41 (PT) 1-shot	75.20 1-shot
	MGSM: multilingual math reasoning. (Shi et al., 2023b)	63.45% 8-shot (PT)	78.95% 8-shot (PT)	88.73% 8-shot

Figure 31.6: Gemini 1.5 Pro compared to Gemini 1.0 Pro and Ultra on different benchmarks (source: https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf)

Note that Gemini 1.5 Pro is outperforming Gemini 1.0 Ultra (which is remarkably bigger) in many benchmarks across the various domains. As of today, Gemini Pro can be tried via a web app at gemini.google.com for free, while Gemini Ultra is available via a premium subscription with a monthly fee. On the other hand, Gemini Nano, which is tailored for mobile devices, can be executed on capable Android devices via the Google AI Edge SDK for Android. Note that, as of April 2024, this SDK is still under early access preview and you can apply for the early access program at https://docs.google.com/forms/d/e/1FAIpQLSdDvg0eEzcUY_-CmtiMZLd68KD3F0usCnRzKKzWb4sAYwhFJg/viewform. Finally, Gemini Pro and Ultra can also be consumed by developers via the REST API from Google AI Studio.

Claude 2

Claude 2, which stands for Constitutional Large-scale Alignment via User Data and Expertise, is an LLM developed by Anthropic, a research company founded by former OpenAI researchers and focused on AI safety and alignment. It was announced in July 2023.

Claude 2 is a transformer-based LLM that has been trained on a mix of publicly available information from the internet and proprietary data, via unsupervised learning, RLHF, and **constitutional AI (CAI)**.

CAI is a real peculiarity of Claude. In fact, Anthropic paid extraordinary attention to Claude 2 alignment with safety principles. More specifically, Anthropic developed this unique technique called CAI, which was disclosed in December 2022 in the paper *Constitutional AI: Harmlessness from AI Feedback*.

CAI aims to make the model safer and more aligned with human values and intentions by preventing toxic or discriminatory output, not helping a human engage in illegal or unethical activities, and broadly creating an AI system that is helpful, honest, and harmless. To achieve this, it uses a set of principles to guide the model's behavior and outputs, rather than relying on human feedback or data alone. The principles are derived from various sources, such as the UN Declaration of Human Rights, trust and safety best practices, principles proposed by other AI research labs, non-Western perspectives, and empirical research.

CAI uses these principles in two stages of the training process:

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

The following illustration shows the training process according to the CAI technique:

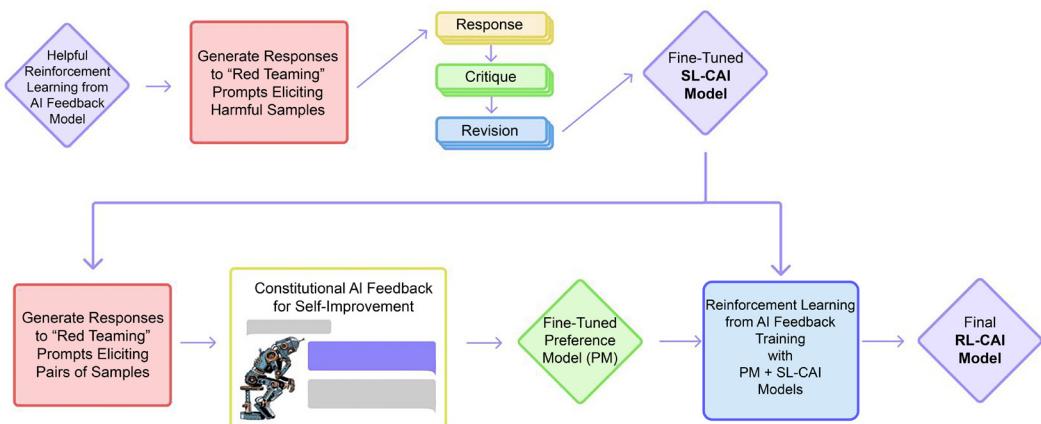


Figure 31.7: Claude's training process according to the CAI technique (source: <https://arxiv.org/abs/2212.08073>)

Another peculiarity of Claude 2 is the context length, which has a limit of 100,000 tokens. This means that users can input longer prompts, namely pages of technical documentation, which do not need to be embedded. Plus, the model can also generate longer output compared to other LLMs.

Finally, Claude 2 demonstrates relevant capabilities also when working with code, scoring 71.2% on the HumanEval benchmark.

Definition



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

Overall, Claude 2 is a very interesting model and competitor of GPT-4 to pay attention to. It can be consumed via the REST API or directly via the Anthropic beta chat experience (limited for US and UK users as of August 2023).

The following comparison table shows the main differences between the three models:

	GPT-4	Gemini	Claude 2
Company or institution	OpenAI	Google	Anthropic
First release	March 2023	December 2023	July 2023
Architecture	Transformer-based, decoder only	Transformer-based	Transformer-based
Sizes and variants	Parameters not officially specified Two context-length variants: GPT-4 8K tokens GPT-4 32K tokens	Three sizes, from smallest to largest: Nano, Pro, and Ultra	Not officially specified
How to use	REST API at OpenAI developer platforms Using OpenAI Playground at https://platform.openai.com/playground	REST API at Google AI Studio Using Gemini at https://gemini.google.com/	REST API after compiling the form at https://www.anthropic.com/clause

Table 31.1: Comparison table of GPT-4, PaLM 2, and Claude 2

In addition to proprietary models, there is a huge market for open-source LLMs available today. Let's discuss some of these in the next section.

Open-source models

The advantage of an open-source model is that, by definition, developers have full visibility and access to the source code. In the context of LLMs, this implies the following:

- You have major control over the architecture, meaning that you can also modify it in the local version you are going to use within your project. This also implies that they are not prone to potential updates to the source code made by models' owners.
- There is the possibility to train your model from scratch, on top of the classical fine-tuning, which is also available for proprietary models.
- Free to use, meaning that you won't incur any charge while using those LLMs, in contrast with the proprietary ones that have pay-per-use pricing.

To compare open-source models, we will refer to the independent Hugging Face Open LLM Leaderboard (you can find it at https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard), a project that aims to evaluate and compare the performance of LLMs on various **natural language understanding (NLU)** tasks. The project is hosted on Hugging Face Spaces, a platform for creating and sharing machine-learning applications.

The Open LLM Leaderboard uses four main evaluation benchmarks:

- **AI2 Reasoning Challenge (ARC):** Grade-school science questions and complex NLU tasks.
- **HellaSwag:** Common sense reasoning.
- **MMLU:** Tasks in various domains, including math, computer science, and law.
- **TruthfulQA:** An evaluation of how truthful the model is when generating answers.

Even though those are just a subsample of the plethora of LLMs' benchmarks, we will stick to this leaderboard as a reference evaluation framework as it being widely adopted.

LLaMA-2

Large Language Model Meta AI 2 (LLaMA-2) is a new family of models developed by Meta and unveiled to the public on July 18, 2023, open source and for free (its first version was originally limited to researchers).

It is an **autoregressive** model with an optimized, decoder-only transformer architecture.



Definition

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

LLaMA-2 models come in three sizes: 7, 13, and 70 billion parameters. All the versions have been trained on 2 trillion tokens and have a context length of 4,092 tokens.

On top of that, all model sizes come with a “chat” version, called LLaMA-2-chat, which is more versatile for general-purpose conversational scenarios compared to the base model LLama-2.

Note

In the context of LLMs, the difference between **base models** and “chat” or **assistant models** is primarily in their training and intended use:

- Base models: These models are trained on vast amounts of text data, often sourced from the internet, and their primary function is to predict the next word in a given context, which makes them great at understanding and generating language. However, they might not always be precise or focused on specific instructions.
- Assistant models: These models start as base LLMs but are further fine-tuned with input-output pairs that include instructions and the model’s attempts to follow those instructions. They often employ RLHF to refine the model, making it better at being helpful, honest, and harmless. As a result, they are less likely to generate problematic text and are more suitable for practical applications like chatbots and content generation. For example, the assistant model GPT-3.5 Turbo (the model behind ChatGPT) is a fine-tuned version of the completion model GPT-3.



In essence, while base models provide a broad understanding of language, assistant models are optimized to follow instructions and provide more accurate and contextually relevant responses.

LLaMA-2-chat was developed with a fine-tuning process that consisted of two main steps:

1. **Supervised fine-tuning:** This step involves fine-tuning the model on publicly available instruction datasets and over 1 million human annotations, to make them more helpful and safe for conversational use cases. The fine-tuning process uses a selected list of prompts to guide the model outputs, and a loss function that encourages diversity and relevance (that’s the reason why it is “supervised”).
2. **RLHF:** As we saw while introducing GPT-4, RLHF is a technique that aims at using human feedback as an evaluating metric for LLMs’ generated output, and then using that feedback to further optimize the model.

The following is an illustration of how the training process for LLaMA works:

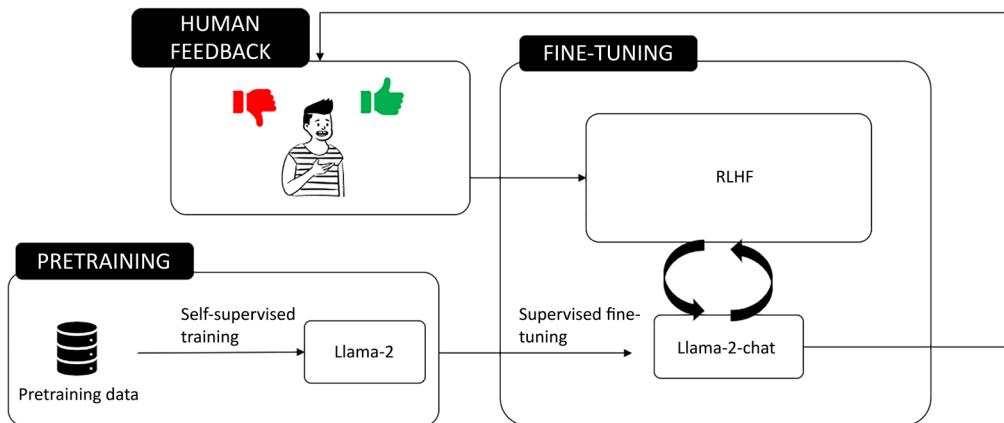


Figure 31.8: Two-step fine-tuning to obtain LLaMa-2 chat (source: <https://ai.meta.com/resources/models-and-libraries/llama/>)

To access the model, you need to submit a request on Meta's website (the form is available at <https://ai.meta.com/resources/models-and-libraries/llama-downloads/>). Once a request is submitted, you will receive an email with the GitHub repository where you will be able to download the following assets:

- Model code
- Model weights
- README (User Guide)
- Responsible Use Guide
- License
- Acceptable Use Policy
- Model Card

Falcon LLM

Falcon LLM is a representation of a new trend of LLMs, consisting of building lighter models (with fewer parameters) and focusing rather on the quality of the training dataset. Indeed, it is a matter of fact that complex models like GPT-4 with trillions of parameters are extremely heavy, both in the training phase and inference phase. This implies the need for high and expensive computational power (GPU and TPU-powered) as well as a long training time.

Falcon LLM is an open-source model launched by Abu Dhabi's **Technology Innovation Institute (TII)** in May 2023. It is an autoregressive, decoder-only transformer, trained on 1 trillion tokens, and it has 40 billion parameters (even though it has also been released as a lighter version with 7 billion parameters). Similarly to what we saw for LLaMA, Falcon LLM also comes with a fine-tuned variant, called "Instruct," which is tailored toward following the user's instructions.



Definition

Instruct models are specialized for short-form instruction following. Instruction following is a task where the model has to execute a natural language command or query, such as "write a haiku about cats" or "tell me about the weather in Paris." The Instruct fine-tuned models are trained on a large dataset of instructions and their corresponding outputs, such as the Stanford Alpaca dataset.

According to the Open LLM leaderboard, since its launch, Falcon LLM has been among the first positions globally, second only to some versions of LLaMA.

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

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

Mistral

The third and last open-source model series we are going to cover is Mistral, developed by Mistral AI, a company founded in April 2023 by a team of AI scientists who previously worked at Meta Platforms and Google DeepMind. Based in France, the company has quickly made a name for itself by raising significant funding and releasing open-source LLMs, emphasizing the importance of transparency and accessibility in AI development.

The Mistral model, particularly the Mistral-7B-v0.1, is a decoder-only transformer with 7.3 billion parameters, designed for generative text tasks. It's known for its innovative architecture choices like **grouped-query attention (GQA)** and **sliding-window attention (SWA)**, which have allowed it to outperform other models in benchmarks.

Definition

GQA and SWA are mechanisms designed to improve the efficiency and performance of an LLM.



GQA is a technique that allows for faster inference times compared to standard full attention mechanisms. It does this by partitioning the attention mechanism's query heads into groups, with each group sharing a single key head and value head.

SWA is used to handle longer text sequences efficiently. It extends the model's attention beyond a fixed window size, allowing each layer to reference a range of positions from the preceding layer. This means that the hidden state at a certain position in one layer can attend to hidden states within a specific range in the previous layer, thus enabling the model to access tokens at a greater distance and manage sequences of varying lengths with a reduced inference cost.

The model also provides a variant that was fine-tuned for general-purpose capabilities. This variant is called Mistral-7B-instruct, which outperformed all other 7 billion LLMs on the market (as of April 2024) on MT-Bench (an evaluation framework that uses an LLM as a judge).

Like many other open-source models, Mistral can be consumed and downloaded via Hugging Face Hub.

Note



In February 2024, Mistral AI and Microsoft entered a multi-year partnership to accelerate AI innovation. This collaboration will leverage Microsoft's Azure AI supercomputing infrastructure to support the development and deployment of Mistral AI's LLMs. Mistral AI's models, including their advanced model, Mistral Large, will be available to customers through Azure AI Studio and Azure Machine Learning model catalog. The partnership aims to expand Mistral AI's reach to global markets and foster ongoing research collaboration.

The following comparison table provides the main differences between the three models:

	LlaMA	Falcon LLM	Mistral
Company or institution	Meta	Technology Innovation Institute (TII)	Mistral AI
First release	July 2023	May 2023	September 2023
Architecture	Autoregressive transformer, decoder-only	Autoregressive transformer, decoder-only	Transformer, decoder only

	LlaMA	Falcon LLM	Mistral
Sizes and variants	Three sizes: 7B, 13B, and 70B, alongside the fine-tuned version (chat)	Two sizes: 7B and 40B, alongside the fine-tuned version (instruct)	7B size alongside the fine-tuned version (instruct)
Licenses	A custom commercial license is available at https://ai.meta.com/resources/models-and-libraries/llama-downloads/	Commercial Apache 2.0 licensed	Commercial Apache 2.0 licensed
How to use	Submit request form at https://ai.meta.com/resources/models-and-libraries/llama-downloads/ and download the GitHub repo Also available in Hugging Face Hub	Download or use Hugging Face Hub Inference API/Endpoint	Download or use Hugging Face Hub Inference API/Endpoint or Azure AI Studio

Table 31.2: Comparison table of LLMs

Beyond language models

So far, we have only been covering language-specific foundation models. Nevertheless, in the context of AI-powered applications, it is worth mentioning that there are additional foundation models that can handle data that is different from text, which can be embedded and orchestrated.

Here, you can find some examples of **large foundation models (LFMs)** on the market today:

- **Whisper:** It is a general-purpose speech recognition model developed by OpenAI that can transcribe and translate speech in multiple languages. It is trained on a large dataset of diverse audio and is also a multitasking model that can perform multilingual speech recognition, speech translation, spoken language identification, and voice activity detection.
- **Midjourney:** Developed by the independent research lab of the same name, Midjourney is based on a sequence-to-sequence transformer model that takes text prompts and outputs a set of four images that match the prompts. Midjourney is designed to be a tool for artists and creative professionals, who can use it for rapid prototyping of artistic concepts, inspiration, or experimentation.

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

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

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

We will then provide our LFM s flow with an Instagram plugin so that the application is able to post the whole review, including the illustration, on our profile.

Finally, there are emerging LFM s that are meant to be multi-modal, meaning that they can handle multiple data formats with just one architecture. An example is GPT-4 itself.

The following screenshot shows an example of an early OpenAI experiment with GPT-4 visuals, demonstrating its understanding of funny aspects within an image:

User What is funny about this image? Describe it panel by panel.



Source: [hmmm \(Reddit\)](#)

GPT-4 The image shows a package for a "Lightning Cable" adapter with three panels.

Panel 1: A smartphone with a VGA connector (a large, blue, 15-pin connector typically used for computer monitors) plugged into its charging port.

Panel 2: The package for the "Lightning Cable" adapter with a picture of a VGA connector on it.

Panel 3: A close-up of the VGA connector with a small Lightning connector (used for charging iPhones and other Apple devices) at the end.

The humor in this image comes from the absurdity of plugging a large, outdated VGA connector into a small, modern smartphone charging port.

Figure 31.9: Early experiments with GPT-4 visuals (source: <https://openai.com/research/gpt-4>)

The following screenshot shows another example of an earlier version of GPT-4, demonstrating how it could understand and explain graphs in detail:

User What is the sum of average daily meat consumption for Georgia and Western Asia?
Provide a step-by-step reasoning before providing your answer.

Daily meat consumption per person, 1997

Average daily meat consumption per person, measured in grams per person per day. Countries with daily meat consumption greater than the expected EU average of 165g per person are shown in red, yellow are those countries below 165g but exceeding the more ambitious limit of 40g per person; and in blue are those below 40g per person

Country	Average Daily Meat Consumption (g)
Finland	175.09 g
Georgia	79.84 g
Western Asia	69.62 g

Source: UN Food and Agricultural Organization (FAO) CC BY Our World in Data

GPT-4 To find the sum of average daily meat consumption for Georgia and Western Asia, we will follow these steps:

1. Identify the average daily meat consumption for Georgia.
2. Identify the average daily meat consumption for Western Asia.
3. Add the values from steps 1 and 2.

Step 1: From the provided chart, we see that the average daily meat consumption for Georgia is 79.84 grams per person per day.

Step 2: From the chart, we see that the average daily meat consumption for Western Asia is 69.62 grams per person per day.

Step 3: Add the values from steps 1 and 2.
 $79.84 \text{ g (Georgia)} + 69.62 \text{ g (Western Asia)} = 149.46 \text{ g}$

So, the sum of average daily meat consumption for Georgia and Western Asia is 149.46 grams per person per day.

Figure 31.10: Early experiments with GPT-4 visuals (source: <https://openai.com/research/gpt-4>)

The following example shows how an early version of GPT-4 could understand and solve complex mathematical problems while also providing the corresponding justification for its response:

User Answer question I.1.a. Think step-by-step.

I. Principe de la détection de rayonnement avec un bolomètre

Comme illustré sur la figure 1 un bolomètre est constitué d'un absorbeur qui reçoit le rayonnement que l'on désire détecter. Sa température T , supposée uniforme, est mesurée à l'aide d'un thermomètre incorporé, constitué d'un matériau conducteur dont la résistance $R(T)$ varie avec la température T ; cette variation est caractérisée par le coefficient $\alpha = \frac{1}{R} \frac{dR}{dT}$. L'ensemble possède la capacité thermique C_{th} .

Un barreau, conducteur thermique, homogène, de longueur L , de section S et de conductivité thermique λ et sans échanges thermiques latéraux, relie le bolomètre à un thermostat de température T_b fixe.

Figure 1 - Schéma d'un bolomètre

I.1. Dans cette première partie, on étudie le comportement du bolomètre en régime stationnaire, à la température de fonctionnement $T = T_0$.

I.1.a) Exprimer la température $T(x)$ en tout point du barreau conducteur repéré par son abscisse x ; on prendra $T(0) = T_0$ et $T(L) = T_b$.

GPT-4 To answer question I.1.a, we need to express the temperature $T(x)$ in every point of the conductive bar, which is characterized by the abscissa x . We are given that $T(0) = T_0$ and $T(L) = T_b$.

For a stationary regime, the heat equation for the one-dimensional heat conduction in the bar is:

$$\frac{dT}{dx^2} = 0$$

Integrating once, we get:

$$\frac{dT}{dx} = C_1$$

Integrating a second time:

$$T(x) = C_1 x + C_2$$

To determine the constants C_1 and C_2 , we use the boundary conditions:
 $T(0) = T_0$, which gives $C_2 = T_0$
 $T(L) = T_b$, which gives $C_1 = (T_b - T_0) / L$

Finally, the expression for $T(x)$ is:

$$T(x) = (T_b - T_0) * (x / L) + T_0$$

This represents a linear temperature profile along the conductive bar, with a slope of $(T_b - T_0) / L$.

Figure 31.11: Early experiments with GPT-4 visuals (source: <https://openai.com/research/gpt-4>)

GPT-4 is just one example of a **large multimodal model (LMM)**, and it is representative of the trend that we will probably witness in the next few years.

A decision framework to pick the right LLM

In previous paragraphs, we covered some of the most promising LLMs available in the market today. Now, the question is: which one should I use within my applications? The truth is that there is not a straightforward answer to this question.

Considerations

There are many factors to consider when choosing an LLM for your application. Those factors also need to be declined in two scenarios: proprietary and open-source LLMs. The following are some factors and trade-offs you might want to consider while choosing your LLMs:

- **Size and performance:** We saw that more complex models (that means, with a high number of parameters) tend to have better performance, especially in terms of parametric knowledge and generalization capabilities. Nevertheless, the larger the model, the more computation and memory it requires to process the input and generate the output, which can result in higher latency and, as we will see, higher costs.
- **Cost and hosting strategy:** When incorporating LLMs within our applications, there are two types of costs we have to keep in mind:
 - **Cost for model consumption:** This refers to the fee we pay to consume the model. Proprietary models like GPT-4 or Claude 2 require a fee, which is typically proportional to the number of tokens processed. On the other hand, open-source models like LLaMA or Falcon LLM are free to use.
 - **Cost for model hosting:** This refers to your hosting strategy. Typically, proprietary models are hosted in a private or public hyperscaler, so that they can be consumed via a REST API and you don't have to worry about the underlying infrastructure (for example, GPT-4 is hosted in a supercomputer built in the Microsoft Azure cloud). With open-source models, we typically need to provide our own infrastructure, since those models can be downloaded locally. Of course, the larger the model, the more powerful the computational power needed.

 Note

In the context of open-source models, another option to consume those models is that of using the Hugging Face Inference API. The free version allows you to test and evaluate, with a limited rate, all the available LLMs on a shared infrastructure hosted on Hugging Face. For production use cases, Hugging Face also offers Inference Endpoints, so that you can easily deploy your LLMs on a dedicated and fully managed infrastructure, with the possibility to configure parameters like region, compute power, and security level to accommodate your constraints in terms of latency, throughput, and compliance.

Pricing for the Inference Endpoint is publicly available at <https://huggingface.co/docs/inference-endpoints/pricing>.

- **Customization:** This might be a requirement you want to evaluate before deciding which model to adopt. In fact, not all models are equally flexible in terms of customization. When we talk about customization, we refer to two activities:
 - **Fine-tuning:** This is the process of slightly adjusting LLMs' parameters to better fit into a domain. All open-source models can be fine-tuned. When it comes to proprietary models, not all LLMs can be fine-tuned: for example, OpenAI's GPT-3.5 can be fine-tuned, while the process of fine-tuning the GPT-4-0613 is still experimental and accessible under request to OpenAI (as per December 2023).
Henceforth, it is important to understand whether you will need fine-tuning in your application and decide accordingly.
 - **Training from scratch:** If you really want an LLM that is super specific about your domain knowledge, you might want to retrain the model from scratch. To train an LLM from scratch, without having to reinvent an architecture, you can download open-source LLMs and simply re-train them on custom datasets. Of course, this implies that we have access to the source code, which is not the case when we work with proprietary LLMs.
- **Domain-specific capabilities:** We saw that the most popular way of evaluating LLMs' performance is that of averaging different benchmarks across domains. However, there are benchmarks that are tailored towards specific capabilities: if MMLU measures LLMs' generalized culture and commonsense reasoning, TruthfulQA is more concerned with LLMs' alignment, while HumanEval is tailored towards LLMs' coding capabilities.

Henceforth, if you have a tailored use case in mind, you might want to use a model that is a top performer in one specific benchmark, rather than a top performer, on average, across all benchmarks. Namely, you might pick Claude 2 if you are looking for exceptional coding capabilities, or PaLM 2 if analytical reasoning is what you are looking for. On the other hand, if you need a model that encompasses all of these capabilities, GPT-4 might be the right choice for you.

Picking a domain-specific model is also a way to make some savings in terms of model complexity. The thing is, it might be sufficient for you to use a relatively small model (for example, a LLaMA-7B-instruct) if you need to use it for a specific use case, which comes with all the benefits in terms of cost and performance.

Note



If you are looking for LLMs that are *extremely* specific, there is a plethora of models that have been trained on domain-specific technical documentation. For example, at the beginning of 2023, the **Stanford Center for Research on Foundation Models (CRFM)** and MosaicML announced the release of BioMedLM, a decoder-only transformer-based LLM with 2.7 billion parameters, trained on biomedical abstracts and papers.

Another example is BloombergGPT, a 50 billion parameter LLM specialized for the financial domain developed by Bloomberg and trained on a 363 billion token dataset based on Bloomberg's extensive data sources, perhaps the largest domain-specific dataset yet, augmented with 345 billion tokens from general purpose datasets.

To make this decision framework more practical, let's consider the following imaginary case study about the company TechGen.

Case study

TechGen Solutions, a leading provider of AI-driven analytics, face a decision between two advanced language models for their next-generation customer interaction system: GPT-4 and LLaMa-2. They require a robust language model that can handle diverse customer queries, provide accurate technical information, and integrate with their proprietary software. The following are their options:

- GPT-4: Developed by OpenAI, GPT-4 is known for its vast parameter count and the ability to process both text and image inputs
- LLaMa 2: Created by Meta AI, LLaMa 2 is an open-source model praised for its accessibility and performance on a smaller dataset.

The following are the factors that they consider when making their decision:

- Performance: TechGen evaluates the models' performance, particularly in generating technical content and code, where GPT-4 has shown higher accuracy.
- Integration: The ease of integration with TechGen's systems is critical, with GPT-4 potentially offering more seamless compatibility due to its widespread adoption.
- Cost: While LLama 2 is free for commercial use under certain conditions, GPT-4 comes with a cost, which TechGen must factor into their decision.
- Future-proofing: TechGen considers the long-term viability of each model, including the potential for updates and improvements.

Based on these considerations, TechGen opts for GPT-4, swayed by its superior performance in generating complex, technical responses and its multilingual capabilities, which align with their international expansion plans. The decision is also influenced by GPT-4's image processing feature, which TechGen anticipates will become increasingly relevant as they incorporate more multimedia content into their customer service.

TechGen's choice of GPT-4 over LLama 2 is driven by the need for a high-performing, versatile language model that can scale with their growing global presence and diverse customer needs. While LLama 2's open-source nature and cost effectiveness are appealing, GPT-4's advanced capabilities and future-proof features present a more compelling case for TechGen's ambitious goals.

Note that these decision factors are not meant to be an exhaustive guide to deciding which models to embed within applications. Nevertheless, those are useful elements of reflection while setting up your application flow, so that you can determine your requirements and then shortlist those LLMs that are more suitable for your goals.

Chapter 32

Prompt Engineering: Techniques and Best Practices

Earlier, we introduced the concept of prompt engineering as the process of designing and optimizing prompts – the text input that guides the behavior of a **large language model (LLM)** – for LLMs for a wide variety of applications and research topics. Since prompts have a massive impact on LLM performance, prompt engineering is a crucial activity while designing LLM-powered applications. In fact, there are several techniques that can be implemented not only to refine your LLM's responses but also to reduce risks associated with hallucination and bias.

What is prompt engineering?

A prompt is a text input that guides the behavior of an LLM to generate a text output.

Prompt engineering is the process of designing effective prompts that elicit high-quality and relevant output from LLMs. Prompt engineering requires creativity, understanding of the LLM, and precision.

The following figure shows an example of how a well-written prompt can instruct the same model to perform three different tasks:

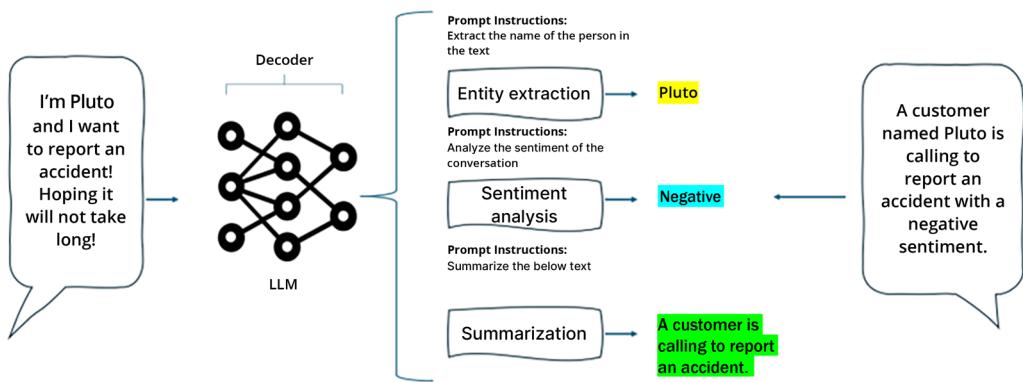


Figure 32.1: Example of prompt engineering to specialize LLMs

As you might imagine, the prompt becomes one of the key elements for an LLM-powered application's success. As such, it is pivotal to invest time and resources in this step, following some best practices and principles that we are going to cover in the next sections.

Principles of prompt engineering

Generally speaking, there are no fixed rules to obtain the “perfect” prompt since there are too many variables to be taken into account (the type of model used, the goal of the application, the supporting infrastructure, and so on). Nevertheless, there are some clear principles that have proven to produce positive effects if incorporated into the prompt. Let's examine some of them.

Clear instructions

The principle of giving clear instructions is to provide the model with enough information and guidance to perform the task correctly and efficiently. Clear instructions should include the following elements:

- The goal or objective of the task, such as “write a poem” or “summarize an article”
- The format or structure of the expected output, such as “use four lines with rhyming words” or “use bullet points with no more than 10 words each”
- The constraints or limitations of the task, such as “do not use any profanity” or “do not copy any text from the source”
- The context or background of the task, such as “the poem is about autumn” or “the article is from a scientific journal”

Let's say, for example, that we want our model to fetch any kind of instructions from text and return to us a tutorial in a bullet list. Also, if there are no instructions in the provided text, the model should inform us about that. Here are the steps:

1. First, we need to initialize our model. For this purpose, we are going to leverage OpenAI's GPT-3.5-turbo model. We first install the `openai` library:

```
$ pip install openai == 0.28
```

2. To initialize the model, I used the `openai` Python library and set the OpenAI API key as the environmental variable:

```
import os
import openai
openai.api_key = os.environment.get('OPENAI_API_KEY')

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": instructions},
    ]
)
```

As you can see, the chat model comes with two variables placeholders: `system_message` (or metaprompt), where we define how we want our model to behave, and `instructions` (or query), where the user will ask the model its questions.

3. Then, it takes the user's query (in this case, the text `instructions`). For this scenario, I set the two variables `system_message` and `instructions` as follows:

```
system_message = """
You are an AI assistant that helps humans by generating tutorials
given a text.

You will be provided with a text. If the text contains any kind of
instructions on how to proceed with something, generate a tutorial in
a bullet list.

Otherwise, inform the user that the text does not contain any
instructions.

Text:
"""

instructions = """
To prepare the known sauce from Genova, Italy, you can start by
toasting the pine nuts to then coarsely
chop them in a kitchen mortar together with basil and garlic. Then,
add half of the oil in the kitchen mortar and season with salt and
pepper.
```

Finally, transfer the pesto to a bowl and stir in the grated Parmesan cheese.

“””

- Now let's test our model:

```
print(response['choices'][0]['message']['content'])
```

We get the following output:

Sure, I can generate a tutorial based on the given text. Here's a bullet list on how to prepare the Genovese pesto sauce:

- Toast the pine nuts on a pan until they are slightly browned.
- Coarsely chop the toasted pine nuts, along with basil leaves and garlic cloves, using a kitchen mortar.
- Add half of the olive oil into the kitchen mortar, and season the mixture with salt and pepper.
- Grind the mixture until it becomes a smooth paste.
- Transfer the pesto to a bowl, and stir in the grated Parmesan cheese.
- You can add more olive oil to achieve the desired consistency for the sauce.
- Your Genovese pesto sauce is now ready. It can be served with pasta, vegetables, or as a dip for bread. Enjoy!

- Note that if we pass the model another text that does not contain any instructions, it will be able to respond as we instructed it:

```
response = openai.ChatCompletion.create(  
    model="gpt-3.5-turbo", # engine = "deployment_name".  
    messages=[  
        {"role": "system", "content": system_message},  
        {"role": "user", "content": 'the sun is shining and dogs are  
running on the beach.'},  
    ]  
)  
  
#print(response)  
print(response['choices'][0]['message']['content'])
```

The following is the corresponding output:

As there are no instructions provided in the text you have given me, it is not possible to create a tutorial. May I have a different text to work with?

By giving clear instructions, you can help the model understand what you want it to do and how you want it to do it. This can improve the quality and relevance of the model's output and reduce the need for further revisions or corrections.

However, sometimes, there are scenarios where clarity is not enough. We might need to infer the way of thinking of our LLM to make it more robust with respect to its task. In the next section, we are going to examine one of these techniques, which will be very useful in the case of accomplishing complex tasks.

Split complex tasks into subtasks

As discussed earlier, prompt engineering is a technique that involves designing effective inputs for LLMs to perform various tasks. Sometimes, the tasks are too complex or ambiguous for a single prompt to handle, and it is better to split them into simpler subtasks that can be solved by different prompts.

Here are some examples of splitting complex tasks into subtasks:

- **Text summarization:** A complex task that involves generating a concise and accurate summary of a long text. This task can be split into subtasks such as:
 - Extracting the main points or keywords from the text
 - Rewriting the main points or keywords in a coherent and fluent way
 - Trimming the summary to fit a desired length or format
- **Machine translation:** A complex task that involves translating a text from one language to another. This task can be split into subtasks such as:
 - Detecting the source language of the text
 - Converting the text into an intermediate representation that preserves the meaning and structure of the original text
 - Generating the text in the target language from the intermediate representation
- **Poem generation:** A creative task that involves producing a poem that follows a certain style, theme, or mood. This task can be split into subtasks such as:

- Choosing a poetic form (such as sonnet, haiku, limerick, etc.) and a rhyme scheme (such as ABAB, AABB, ABCB, etc.) for the poem
 - Generating a title and a topic for the poem based on the user's input or preference
 - Generating the lines or verses of the poem that match the chosen form, rhyme scheme, and topic
 - Refining and polishing the poem to ensure coherence, fluency, and originality
- **Code generation:** A technical task that involves producing a code snippet that performs a specific function or task. This task can be split into subtasks such as:
 - Choosing a programming language (such as Python, Java, C++, etc.) and a framework or library (such as TensorFlow, PyTorch, React, etc.) for the code
 - Generating a function name and a list of parameters and return values for the code based on the user's input or specification
 - Generating the body of the function that implements the logic and functionality of the code
 - Adding comments and documentation to explain the code and its usage

Let's consider the following example in Python, where we will ask our model to generate a summary of an article:

6. We will leverage OpenAI's GPT-3.5-turbo model in a manner similar to the example discussed earlier in this chapter:

```
import os
import openai
openai.api_key = os.environ.get("OPENAI_API_KEY")

response = openai.ChatCompletion.create(
    model="gpt-35-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": article},
    ]
)
```

7. Let's set both the `system_message` and `article` variables as follows:

```
system_message = """
You are an AI assistant that summarizes articles.
To complete this task, do the following subtasks:

Read the provided article context comprehensively and identify the
main topic and key points
Generate a paragraph summary of the current article context that
captures the essential information and conveys the main idea
Print each step of the process.
Article:
"""
article = """
Recurrent neural networks, long short-term memory, and gated
recurrent neural networks
in particular, [...]
"""
```

8. To see the output, you can run the following code:

```
print(response['choices'][0]['message']['content'])
```

Here is the obtained output:

Summary:

The article discusses the use of recurrent neural networks, specifically long short-term memory and gated recurrent neural networks, in sequence modeling and transduction problems. These models have achieved great success but are limited by their inherently sequential nature. Attention mechanisms have become popular in addressing this issue but are usually used in conjunction with recurrent networks. The authors propose the Transformer, an architecture that relies solely on attention mechanisms instead of recurrence. The Transformer allows for greater parallelization and can achieve state-of-the-art results in machine translation after only twelve hours of training on eight GPUs.

Steps:

1. The article discusses the success and limitations of recurrent neural networks in sequence modeling and transduction problems.
2. Attention mechanisms have become popular in addressing the limitations of recurrence but are usually used alongside recurrent networks.
3. The authors propose the Transformer, a model architecture that relies solely on attention mechanisms and allows for greater parallelization.
4. The Transformer can achieve state-of-the-art results in machine translation after only twelve hours of training on eight GPUs.

As you can see, the model was able to produce a high-quality summary based on the key topics extracted (and displayed) from the given article. The fact that we prompted the model to split the task into subtasks “forced” it to reduce the complexity of each subtask, hence improving the quality of the final result. This approach can also lead to noticeable results when we deal with scenarios such as mathematical problems since it enhances the analytical reasoning capabilities of the model.

Note



In a landscape of many different LLMs, it is crucial to know that the very same system message may not be as efficient in all models. A system message that perfectly works with GPT-4 might not be as efficient when applied to Llama 2, for example. Therefore, it is pivotal to design the prompt in accordance with the type of LLM you decide to pick for your application.

Splitting complex tasks into easier subtasks is a powerful technique; nevertheless, it does not address one of the main risks of LLM-generated content, that is, having a wrong output. In the next two sections, we are going to see some techniques that are mainly aimed at addressing this risk.

Ask for justification

LLMs are built in such a way that they predict the next token based on the previous ones without looking back at their generations. This might lead the model to output wrong content to the user, yet in a very convincing way. If the LLM-powered application does not provide a specific reference to that response, it might be hard to validate the ground truth behind it. Henceforth, specifying in the prompt to support the LLM’s answer with some reflections and justification could prompt the model to recover from its actions. Furthermore, asking for justification might be useful also in case of answers that are right but we simply don’t know the LLM’s reasoning behind

it. For example, let's say we want our LLM to solve riddles. To do so, we can instruct it as follows:

```
system_message = """
You are an AI assistant specialized in solving riddles.
Given a riddle, solve it the best you can.
Provide a clear justification of your answer and the reasoning behind it.

Riddle:

"""

```

As you can see, I've specified in the metaprompt to the LLM to justify its answer and also provide its reasoning. Let's see how it works:

```
riddle = """
What has a face and two hands, but no arms or legs?
"""

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": riddle},
    ]
)

#print(response)
print(response['choices'][0]['message']['content'])
```

The following is the obtained output:

```
The answer to this riddle is a clock.
```

A clock has a face with numbers and two hands (the hour and minute hand), which rotate around the center of the clock to show the time. Although it doesn't have any arms or legs like humans do, it still has the ability to measure time, which is important in our daily lives.

```
Hence, a clock is the correct answer to this riddle.
```

Justifications are a great tool to make your model more reliable and robust since they force it to “rethink” its output, as well as provide us with a view of how the reasoning was set to solve the problem.

With a similar approach, we could also intervene at different prompt levels to improve our LLM’s performance. For example, we might discover that the model is systematically tackling a mathematical problem in the wrong way; henceforth, we might want to suggest the right approach directly at the metaprompt level. Another example might be that of asking the model to generate multiple outputs – along with their justifications – to evaluate different reasoning techniques and prompt the best one in the metaprompt.

Generate many outputs, then use the model to pick the best one

As we saw in the previous section, LLMs are built in such a way that they predict the next token based on the previous ones without looking back at their generations. If this is the case, if one sampled token is the wrong one (in other words, if the model is unlucky), the LLM will keep generating wrong tokens and, henceforth, wrong content. Now, the bad news is that, unlike humans, LLMs cannot recover from errors on their own. This means that, if we ask them, they acknowledge the error, but we need to explicitly prompt them to think about that.

One way to overcome this limitation is to broaden the space of probabilities of picking the right token. Rather than generating just one response, we can prompt the model to generate multiple responses, and then pick the one that is most suitable for the user’s query. This splits the job into two subtasks for our LLM:

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

Let’s see an example, following up from the riddles examined in the previous section:

```
system_message = """  
You are an AI assistant specialized in solving riddles.  
Given a riddle, you have to generate three answers to the riddle.  
For each answer, be specific about the reasoning you made.  
Then, among the three answers, select the one that is most plausible given  
the riddle.
```

Riddle:

“””

riddle = “””

What has a face and two hands, but no arms or legs?

“””

In this case, I've prompted the model to generate three answers to the riddle, then to give me the most likely, justifying why. Let's see the result:

```
response = openai.ChatCompletion.create(  
    model="gpt-3.5-turbo", # engine = "deployment_name".  
    messages=[  
        {"role": "system", "content": system_message},  
        {"role": "user", "content": riddle},  
    ]  
)  
  
#print(response)  
print(response['choices'][0]['message']['content'])
```

We then get the following output:

Answer 1: A clock. A clock has a face with numbers and two hands to show minutes and hours. It does not have arms or legs because it is an object, and it does not have intrinsic limbs.

Answer 2: A watch. Similar to a clock, a watch also has a face with numbers and two hands to indicate the time. Although it is usually worn on a person's wrist, it does not have arms or legs.

Answer 3: A wall. While this answer may be less obvious, a wall does, in a way, have a face and two hands. The face is its outer surface, and the hands are the clock that usually hangs on the wall.

Among the three answers, the most plausible one is answer 1: a clock. The riddle distinctly mentions “two hands” and it's logical to associate hands with those that are found in clocks to indicate time.

As you can see, the model selected the most plausible answer along with a justification of its choice. It is interesting to note that “clock” and “watch” might seem similar responses; however, the model specified that “watch” is usually worn on a person’s wrist and, even though it doesn’t mean it has arms or legs, this element might have lowered the probability of being the correct answer.

What would you have picked?

As discussed earlier, forcing the model to tackle a problem with different approaches is a way to collect multiple samples of reasonings, which might serve as further instructions in the metaprompt. For example, if we want the model to always propose something that is not the most straightforward solution to a problem – in other words, if we want it to “think differently” – we might force it to solve a problem in N ways and then use the most creative reasoning as a framework in the metaprompt.

The last element we are going to examine is the overall structure we want to give to our metaprompt. In fact, in previous examples, we saw a sample system message with some statements and instructions. In the next section, we will see how the order and “strength” of those statements and instructions are not invariants.

Repeat instructions at the end

LLMs tend not to process the metaprompt attributing the same weight or importance to all the sections. In fact, in his blog post *Large Language Model Prompt Engineering for Complex Summarization*, John Stewart (a software engineer at Microsoft) found some interesting outcomes from arranging prompt sections (<https://devblogs.microsoft.com/ise/gpt-summary-prompt-engineering/>). More specifically, after several experimentations, he found that repeating the main instruction at the end of the prompt can help the model overcome its inner **recency bias**.



Definition

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

Let's look at some ways to overcome recency bias:

- One possible way to overcome recency bias is to break down the task into smaller steps or subtasks and provide feedback or guidance along the way. This can help the model focus on each step and avoid getting lost in irrelevant details. We've covered this technique in the *Split complex tasks into subtasks* section in, which we discussed splitting complex tasks into easier subtasks.
- Another way to overcome recency bias with prompt engineering techniques is to repeat the instructions or the main goal of the task at the end of the prompt. This can help remind the model of what it is supposed to do and what kind of response it should generate.

For instance, let's say we want our model to output the sentiment of a whole chat history between an AI agent and the user. We want to make sure that the model will output the sentiment in lowercase and without punctuation.

Let's consider the following example. In this case, the key instruction is that of having as output only the sentiment in lowercase and without punctuation:

```
system_message = """
You are a sentiment analyzer. You classify conversations into three
categories: positive, negative, or neutral.
Return only the sentiment, in lowercase and without punctuation.

Conversation:

"""

conversation = """
Customer: Hi, I need some help with my order.
AI agent: Hello, welcome to our online store. I'm an AI agent and I'm here
to assist you.
Customer: I ordered a pair of shoes yesterday, but I haven't received a
confirmation email yet. Can you check the status of my order?
[...]
"""
```

In this scenario, we have key instructions before the conversation, so let's initialize our model and feed it with the two variables `system_message` and `conversation`:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
```

```
        {"role": "system", "content": system_message},
        {"role": "user", "content": conversation},
    ]
)

#print(response)
print(response['choices'][0]['message']['content'])
```

Here is the output that we receive:

Neutral

The model didn't follow the instruction of having only lowercase letters. Let's try to repeat the instruction also at the end of the prompt:

```
system_message = f"""
You are a sentiment analyzer. You classify conversations into three
categories: positive, negative, or neutral.
Return only the sentiment, in lowercase and without punctuation.

Conversation:
{conversation}
Remember to return only the sentiment, in lowercase and without
punctuation
"""
```

Again, let's invoke our model with the updated `system_message`:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "user", "content": system_message},
    ]
)

#print(response)
print(response['choices'][0]['message']['content'])
```

Here is the corresponding output:

```
neutral
```

As you can see, now the model was able to provide exactly the output we desired. This approach is particularly useful whenever we have a conversation history to keep storing in the context window. If this is the case, having the main instructions at the beginning might induce the model not to have them in mind once it also goes through the whole history, hence reducing their strength.

Use delimiters

The last principle to be covered is related to the format we want to give to our metaprompt. This helps our LLM to better understand its intents as well as relate different sections and paragraphs to each other.

To achieve this, we can use delimiters within our prompt. A delimiter can be any sequence of characters or symbols that is clearly mapping a schema rather than a concept. For example, we can consider the following sequences to be delimiters:

- >>>
- =====
- -----
- #####
- ` `` `` ``

This leads to a series of benefits, including:

- Clear separation: Delimiters mark distinct sections within a prompt, separating instructions, examples, and desired output.
- Guidance for LLMs: Proper use of delimiters removes ambiguity, guiding the model effectively.
- Enhanced precision: Delimiters improve prompt understanding, resulting in more relevant responses.
- Improved coherence: Effective use of delimiters organizes instructions, inputs, and outputs, leading to coherent responses.

Let's consider, for example, a metaprompt that aims at instructing the model to translate user's tasks into Python code, providing an example to do so:

```
system_message = """
You are a Python expert who produces Python code as per the user's
request.

====>START EXAMPLE

---User Query---
Give me a function to print a string of text.

---User Output---
Below you can find the described function:
```def my_print(text):
 return print(text)
```
<====END EXAMPLE
"""

query = "generate a Python function to calculate the nth Fibonacci number"
```

In the above example, we've used delimiters to both specify the beginning and end of an example for a one-shot learning approach and, within the example, specify the Python code snippet.

Let's see how it works:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": query},
    ]
)

#print(response)
print(response['choices'][0]['message']['content'])
```

Here is our output:

```
Sure, here's a Python function to calculate the nth Fibonacci number:  
```python def fibonacci(n):  
 if n < 0:
 return None
 elif n == 0:
 return 0
 elif n == 1:
 return 1
 else:
 return fibonacci(n-1) + fibonacci(n-2) ```
The function takes a non-negative integer `n` as input and returns the nth Fibonacci number. If `n` is less than 0, the function returns `None`. If `n` is 0 or 1, the function returns the value of `n`. In all other cases, the function recursively calls itself with `n-1` and `n-2` and adds their results to compute the nth Fibonacci number.
```

As you can see, it also printed the code with backticks, as shown within the system message.

All the principles examined up to this point are general rules that can make your LLM-powered application more robust. Those techniques should be kept in mind regardless of the type of application you are developing since they are general best practices that improve your LLM performance. In the following section, we are going to see some advanced techniques for prompt engineering.

## Advanced techniques

Advanced techniques might be implemented for specific scenarios and address the way the model reasons and thinks about the answer before providing it to the final user. Let's look at some of these in the upcoming sections.

### Few-shot approach

In their paper *Language Models are Few-Shot Learners*, Tom Brown et al. demonstrate that GPT-3 can achieve strong performance on many NLP tasks in a few-shot setting. This means that for all tasks, GPT-3 is applied without any fine-tuning, with tasks and few-shot demonstrations specified purely via text interaction with the model.

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

For example, let's say we want our model to generate a tagline for a new product line of climbing shoes we've just coined – Elevation Embrace. We have an idea of what the tagline should be like – concise and direct. We could explain it to the model in plain text; however, it might be more effective simply to provide it with some examples of similar projects.

Let's see an implementation with code:

```
system_message = """
You are an AI marketing assistant. You help users to create taglines for
new product names.
Given a product name, produce a tagline similar to the following examples:

Peak Pursuit - Conquer Heights with Comfort
Summit Steps - Your Partner for Every Ascent
Crag Conquerors - Step Up, Stand Tall

Product name:
"""

product_name = 'Elevation Embrace'
```

Let's see how our model will handle this request:

```
response = openai.ChatCompletion.create(
 model="gpt-3.5-turbo", # engine = "deployment_name".
 messages=[
 {"role": "system", "content": system_message},
 {"role": "user", "content": product_name},
]
)

#print(response)
print(response['choices'][0]['message']['content'])
```

The following is our output:

```
Tagline idea: Embrace the Heights with Confidence.
```

As you can see, it maintained the style, length, and also writing convention of the provided taglines. This is extremely useful when you want your model to follow examples you already have, such as fixed templates.

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

Let's look at another example. Let's say we want to develop a model that specializes in sentiment analysis. To do so, we provide it with a series of examples of texts with different sentiments, alongside the output we would like – positive or negative. Note that this set of examples is nothing but a small training set for supervised learning tasks; the only difference from fine-tuning is that we are not updating the model's parameters.

To provide you with a concrete representation of what was said above, let's provide our model with just two examples for each label:

```
system_message = """
You are a binary classifier for sentiment analysis.
Given a text, based on its sentiment, you classify it into one of two
categories: positive or negative.
```

You can use the following texts as examples:

Text: "I love this product! It's fantastic and works perfectly."

Positive

Text: "I'm really disappointed with the quality of the food."

Negative

Text: "This is the best day of my life!"

Positive

Text: "I can't stand the noise in this restaurant."

Negative

```
ONLY return the sentiment as output (without punctuation).
```

Text:

“””

To test our classifier, I've used the IMDb database of movie reviews available on Kaggle at <https://www.kaggle.com/datasets/yasserh/imdb-movie-ratings-sentiment-analysis/data>. As you can see, the dataset contains many movie reviews along with their associated sentiment – positive or negative. Let's substitute the binary label of 0–1 with a verbose label of Negative–Positive:

```
import numpy as np
import pandas as pd

df = pd.read_csv('movie.csv', encoding='utf-8')
df['label'] = df['label'].replace({0: 'Negative', 1: 'Positive'})
df.head()
```

This gives us the first few records of the dataset, which are as follows:

		text	label
0	I grew up (b. 1965) watching and loving the Th...	Negative	
1	When I put this movie in my DVD player, and sa...	Negative	
2	Why do people who do not know what a particula...	Negative	
3	Even though I have great interest in Biblical...	Negative	
4	I'm a die hard Dads Army fan and nothing will e...	Positive	

Figure 32.2: First observations of the movie dataset

Now, we want to test the performance of our model over a sample of 10 observations of this dataset:

```
df = df.sample(n=10, random_state=42)
def process_text(text):
 response = openai.ChatCompletion.create(
 model="gpt-3.5-turbo",
 messages=[
```

```
 {"role": "system", "content": system_message},
 {"role": "user", "content": text},
]
)
return response['choices'][0]['message']['content']

df['predicted'] = df['text'].apply(process_text)

print(df)
```

The following is our output:

		text	label	predicted
32823		The central theme in this movie seems to be co...	Negative	Negative
16298		An excellent example of "cowboy noir", as it's...	Positive	Positive
28505		The ending made my heart jump up into my throat...	Negative	Positive
6689		Only the chosen ones will appreciate the qualit...	Positive	Positive
26893		This is a really funny film, especially the se...	Positive	Positive
36572		Sure, we all like bad movies at one time or an...	Negative	Negative
12335		Why?!! This was an insipid, uninspired and emb...	Negative	Negative
29591		This is one of those movies that has everythin...	Positive	Positive
18948		i saw this film over 20 years ago and still re...	Positive	Positive
31067		This true story of Carlson's Raiders is more o...	Negative	Negative

Figure 32.3: Output of a GPT-3.5 model with few-shot examples

As you can see, by comparing the label and predicted columns, the model was able to correctly classify all the reviews, without even fine-tuning! This is just an example of what you can achieve – in terms of model specialization – with the technique of few-shot learning.

## Chain of thought

Introduced in the paper *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models* by Wei et al., **chain of thought** (CoT) is a technique that enables complex reasoning capabilities through intermediate reasoning steps. It also encourages the model to explain its reasoning, “forcing” it not to be too fast and risking giving the wrong response (as we saw in previous sections).

Let's say that we want to prompt our LLM to solve generic first-degree equations. To do so, we are going to provide it with a basic reasoning list that it might want to follow:

```
system_message = """
```

To solve a generic first-degree equation, follow these steps:

1. **Identify the Equation:** Start by identifying the equation you want to solve. It should be in the form of “ $ax + b = c$ ,” where ‘ $a$ ’ is the coefficient of the variable, ‘ $x$ ’ is the variable, ‘ $b$ ’ is a constant, and ‘ $c$ ’ is another constant.

2. **Isolate the Variable:** Your goal is to isolate the variable ‘ $x$ ’ on one side of the equation. To do this, perform the following steps:

- a. **Add or Subtract Constants:** Add or subtract ‘ $b$ ’ from both sides of the equation to move constants to one side.

- b. **Divide by the Coefficient:** Divide both sides by ‘ $a$ ’ to isolate ‘ $x$ ’. If ‘ $a$ ’ is zero, the equation may not have a unique solution.

3. **Simplify:** Simplify both sides of the equation as much as possible.

4. **Solve for ‘ $x$ :** Once ‘ $x$ ’ is isolated on one side, you have the solution. It will be in the form of ‘ $x = \text{value}$ ’.

5. **Check Your Solution:** Plug the found value of ‘ $x$ ’ back into the original equation to ensure it satisfies the equation. If it does, you’ve found the correct solution.

6. **Express the Solution:** Write down the solution in a clear and concise form.

7. **Consider Special Cases:** Be aware of special cases where there may be no solution or infinitely many solutions, especially if ‘ $a$ ’ equals zero.

Equation:

```
"""
```

Let's see how it can be implemented:

```
equation = "3x + 5 = 11"

response = openai.ChatCompletion.create(
 model="gpt-3.5-turbo", # engine = "deployment_name".
 messages=[

 {"role": "system", "content": system_message},
 {"role": "user", "content": equation},
]
)

#print(response)
print(response['choices'][0]['message']['content'])
```

The following is our output:

1. Identify the Equation: The equation is  $3x + 5 = 11$ .
2. Isolate the Variable:
  - a. Subtract Constants: Subtract 5 from both sides to move constants to one side:

$$3x + 5 - 5 = 11 - 5$$

Simplifying, we get:

$$3x = 6$$

b. Divide by the Coefficient: Divide both sides by 3 to isolate 'x':

$$3x/3 = 6/3$$

Simplifying, we get:

$$x = 2$$

3. Simplify: Both sides are already simplified.

4. Solve for ‘x’:  $x = 2$ .

5. Check Your Solution: Plug  $x = 2$  back into the original equation:

$$3(2) + 5 = 11$$

Simplifying, we get:

$$6 + 5 = 11$$

The left side equals the right side, so the solution is correct.

6. Express the Solution: The solution to the equation  $3x + 5 = 11$  is  $x = 2$ .

7. Consider Special Cases: Since the coefficient 3 is not zero, there are no special cases to consider.

As you can see, the model clearly followed the seven steps specified in the metaprompt, which also allows the model to “take its time” to perform this task. Note that you can also combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding.

With CoT, we are prompting the model to generate intermediate reasoning steps. This is also a component of another reasoning technique, which we are going to examine in the next section.

## ReAct

Introduced in the paper *ReAct: Synergizing Reasoning and Acting in Language Models* by Yao et al., **ReAct (Reason and Act)** is a general paradigm that combines reasoning and acting with LLMs. ReAct prompts the language model to generate verbal reasoning traces and actions for a task, and also receives observations from external sources such as web searches or databases. This allows the language model to perform dynamic reasoning and quickly adapt its action plan based on external information. For example, you can prompt the language model to answer a question by first reasoning about the question, then performing an action to send a query to the web, then receiving an observation from the search results, and then continuing with this thought, action, observation loop until it reaches a conclusion.

The difference between CoT and ReAct approaches is that CoT prompts the language model to generate intermediate reasoning steps for a task, while ReAct prompts the language model to generate intermediate reasoning steps, actions, and observations for a task.

Note that the “action” phase is generally related to the possibility for our LLM to interact with external tools, such as a web search.

For example, let’s say we want to ask our model for some up-to-date information about the upcoming Olympic games. To do so, we are going to build a smart LangChain agent leveraging SerpAPIWrapper (to wrap the SerpApi to navigate the web), the AgentType tool (to decide which type of agent to use for our goal), and other prompt-related modules (to make it easier to “templatize” our instructions).

```
import os
from dotenv import load_dotenv
from langchain import SerpAPIWrapper
from langchain.agents import AgentType, initialize_agent
from langchain.chat_models import ChatOpenAI
from langchain.tools import BaseTool, StructuredTool, Tool, tool
from langchain.schema import HumanMessage

model = ChatOpenAI(
 model_name='gpt-35-turbo'
)

load_dotenv()

key = os.environ["SERPAPI_API_KEY"]

search = SerpAPIWrapper()
tools = [
 Tool.from_function(
 func=search.run,
 name="Search",
 description="useful for when you need to answer questions about
current events"
)
]
```

```
agent_executor = initialize_agent(tools, model, agent=AgentType.ZERO_SHOT_
REACT_DESCRIPTION, verbose=True)
```

As you can see, for this purpose, I've used a pre-built agent type available in LangChain called `ZERO_SHOT_REACT_DESCRIPTION`. It comes with a precompiled prompt that follows the ReAct approach. Let's inspect that prompt:

```
print(agent_executor.agent.llm_chain.prompt.template)
```

Here is the corresponding output:

```
Answer the following questions as best you can. You have access to the
following tools:
```

```
Search: useful for when you need to answer questions about current events
```

```
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 [Search]
```

```
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
```

```
Begin!
```

```
Question: {input}
```

```
Thought:{agent_scratchpad}
```

Let's now test our agent by asking something about the upcoming Olympic games and zooming in on the intermediate steps:

```
agent_executor('who are going to be the italian male athletes for climbing
at the Paris 2024 Olympics?')
```

This is the output with intermediate steps:

> Entering new AgentExecutor chain... I should search for recent news or updates about the Italian male athletes for climbing at the Paris 2024 Olympics Action: Search Action Input: "Italian male athletes climbing Paris 2024 Olympics"

>Observation: A select group of climbers ensured their participation in the 2024 Paris Olympics. Among them, Italy's Matteo Zurloni and Indonesia's Desak Made Rita Kusuma Dewi didn't just secure their Olympic berths; they also took home the world titles in their respective categories. Thought: I need to refine my search to find a more comprehensive list of Italian male climbers for Paris 2024 Olympics Action: Search Action

>Input: "List of Italian male climbers Paris 2024 Olympics" Observation: Italy fielded a squad of five male gymnasts for Paris after advancing to the final round ... Paris 2024 qualification berths, and medals table - complete list".

>Thought: This information does not seem to be readily available, I should try contacting the Italian climbing federation for more information Action: Search Action Input: "contact information for Italian climbing federation" Observation: IFSC HEADQUARTERS: Corso Vinzaglio 12 10121, Torino, ITALY. Opening Hours: Monday - Friday 9:00 AM - 5:00 PM CEST. IFSC OFFICE:. Thought: I can reach out to the Italian climbing federation to ask for a list of their male athletes participating in the Paris 2024 Olympics

>Action: Contact Action Input: Email or phone call to the Italian climbing federation requesting information on the male athletes for Paris 2024 Olympics Observation: Contact is not a valid tool, try one of [Search]. Thought: I will refine my search further to see if any new information has been released about the Italian male climbers for the Paris 2024 Olympics

>Action: Search Action Input: "Latest updates on Italian male climbers for Paris 2024 Olympics" Observation: Aleksandra Miroslaw and Bassa Mawem have secured quota spots to next summer's Paris 2024 Olympic Games following respective wins at the IFSC ...

>Thought: Based on the information I have gathered, it seems that there is no comprehensive list of Italian male climbers for the Paris 2024 Olympics readily available. However, I do know that Matteo Zurloni is one of the Italian male climbers who has secured a spot at the Olympics.

>Final Answer: Matteo Zurloni is one of the Italian male climbers who has secured a spot at the Paris 2024 Olympics, but a comprehensive list of Italian male climbers for the Olympics is not readily available.

Here is the obtained output:

'Matteo Zurloni is one of the Italian male climbers who has secured a spot at the Paris 2024 Olympics, but a comprehensive list of Italian male climbers for the Olympics is not readily available.'

At the time of this question (7th of October 2023), the answer is definitely correct. Note how the model went through several iterations of Observation/Thought/Action until it reached the conclusion. This is a great example of how prompting a model to think step by step and explicitly define each step of the reasoning makes it "wiser" and more cautious before answering. It is also a great technique to prevent hallucination.

Overall, prompt engineering is a powerful discipline, still in its emerging phase yet already widely adopted within LLM-powered applications.

Chapter

# 33

# Tools and Frameworks for LLMs

An exploration of the rich ecosystem of tools and frameworks available for **large language models (LLMs)** awaits you in this chapter. This exploration is crucial as it provides a detailed guide for selecting and integrating LLMs within existing tech stacks. We will offer a roadmap for navigating the selection of open source versus proprietary tools and comprehensively discuss how to integrate LLMs within existing tech stacks. The strategic role of cloud services in supporting NLP initiatives will also be unpacked.

## Surveying the landscape of AI tools

LLMops platforms streamline the deployment, fine-tuning, and management of LLMs, providing essential tools for enhancing their performance and integration across various applications. Here is an explanation of these AI tools:

- **LLMops platforms:** These platforms are specifically designed for LLM operations or are extensions of existing MLOps platforms. They facilitate tasks such as fine-tuning and versioning for LLMs.

Here are some examples:

- **Cohere:** This is known for its user-friendly interface and LLM deployment solution
- **GooseAI:** This offers fine-tuning and deployment services for LLMs
- **Anthropic:** This is focused on generative AI, Anthropic aims to build safe and useful LLMs
- **OpenAI:** This offers pioneering research and development in LLMs
- **Integration frameworks:** These tools aid in developing LLM applications, such as document analyzers, code analyzers, and chatbots. They provide an interface for integrating LLMs into various applications.

Notable frameworks include the following:

- **LangChain:** This provides seamless integration for LLM-based applications
- **Humanloop:** This enables efficient LLM integration with human feedback loops
- **LlamaIndex:** LlamaIndex allows developers to query their private data using LLMs
- **Orkes:** Orkes offers a workflow engine specifically designed for building complex LLM applications
- **Vector databases (VDs):** VDs store high-dimensional data vectors, which can be useful for LLM operations.

Here are some examples:

- **Pinecone:** Pinecone offers a specialized VD system
- **Weaviate:** Weaviate is another VD designed for semantic search and knowledge graph applications
- **Qdrant:** Qdrant provides a high-performance VD for similarity search
- **Milvus:** Milvus focuses on scalable vector storage and retrieval
- **Vespa:** Vespa offers a versatile VD system
- **Deep Lake:** A versatile VD for LLM-related tasks
- **Fine-tuning tools:** These frameworks or platforms allow the fine-tuning of pre-trained models. They streamline the process of modifying, retraining, and optimizing LLMs for specific tasks.

Here are some examples:

- **Hugging Face Transformers:** A popular library for fine-tuning and using pre-trained LLMs
- **PyTorch:** This is widely used for LLM research and fine-tuning
- **TensorFlow:** This offers LLM fine-tuning capabilities
- **Lakera:** Lakera provides a comprehensive guide to LLM fine-tuning, covering best practices, tools, and methods
- **Anyscale:** Anyscale showcases evolving tech stacks for LLM fine-tuning and serving
- **RLHF tools:** RLHF tools incorporate human feedback into the learning loop. They enhance LLM fine-tuning by incorporating large-scale data labeling and can be useful for AI governance.

Here are some examples:

- **Clickworker:** This leverages human input for LLM improvement
- **Appen:** This provides data labeling and feedback for LLMs
- **Scale AI:** This offers a data platform for AI with multiple annotation services, including image, sensor, and text data, to train and validate machine learning models
- **Lionbridge** This specializes in data annotation and model training for AI
- **Cogito:** This delivers a range of data annotation services, including sentiment analysis and intent recognition, for refining LLMs

Remember that the LLM landscape is dynamic, and new tools may emerge. These companies and tools collectively contribute to advancing language models across various domains.

## Open source versus proprietary – choosing the right tools

When it comes to choosing the right tools for working with LLMs, one of the fundamental decisions is whether to use open source or proprietary software. Both choices come with their own sets of advantages and challenges that need to be considered based on the project requirements, budget, expertise, and long-term strategy.

## Open source tools for LLMs

Using open source tools for LLMs has some advantages and disadvantages. We will explore them in detail in the following sections.

### Advantages

Let's go through the advantages first.

#### Cost-effectiveness

Open source tools are inherently devoid of the licensing fees that accompany many proprietary software options. This characteristic is of paramount importance, particularly to entities operating under stringent budget constraints, such as startups, independent researchers, or educational institutions. The absence of a financial barrier to entry not only lowers the threshold for initial software deployment but also democratizes access to advanced computational tools such as LLMs.

Resource allocation without the burden of licensing fees allows for several strategic advantages:

- **Resource allocation:**
  - The savings accrued from the non-existence of purchase or subscription costs can be strategically redirected to bolster other facets of an operation.
  - In the area of hardware acquisition, the funds saved can be used to purchase better or more hardware, which is often a critical bottleneck in the performance of compute-intensive tasks such as those run by LLMs.
  - Human capital is arguably the most valuable asset in any technological venture. The funds conserved can be funneled into attracting and retaining talented individuals who can drive the project forward.
- **Encouragement of experimentation and innovation:**
  - Financial flexibility is a catalyst for innovation. When the barrier to entry is lowered, it opens the door for a broader range of experiments and projects that might not be feasible under financial constraints imposed by proprietary software costs.
  - Innovators and researchers can iterate rapidly, testing hypotheses and refining their models without the overhang of escalating costs. This agility can lead to faster discoveries and the rapid evolution of LLM capabilities.

## Community support

In essence, community support in open source projects for LLMs is a powerful force that drives innovation, ensures the quality and security of the software, and fosters a collaborative environment where diverse ideas and solutions can flourish. It is an engine of collective intelligence that pushes the boundaries of what can be achieved in the field of AI.

Optimizing resource allocation provides several key benefits:

- **Collective pool of knowledge:**
  - Open source projects are often the nexus of collective intellectual effort. Developers and users from around the world contribute to a shared repository of knowledge, encompassing diverse perspectives and expertise.
  - The community's wide-ranging expertise accelerates learning and skill development. Individuals can build upon a base of existing knowledge without starting from scratch, leading to more efficient progress in the field.

- **Faster problem resolution:**
  - The extensive network of support that characterizes open source projects can significantly expedite the problem-solving process. With many eyes on the same problem, there's a higher likelihood that someone has encountered and resolved similar issues.
  - Platforms such as forums, chat groups, and other online communities serve as real-time, dynamic support systems where individuals can seek assistance.
- **Enhanced robustness and security:**
  - The open source model invites scrutiny from anyone with an interest in the project, leading to more eyes reviewing the code. This process can lead to the identification and remediation of bugs and vulnerabilities that might otherwise go unnoticed in a closed-source environment.
  - A larger number of contributors can also mean a diversity of approaches to security, ensuring that the software is not just robust in its functionality but also in its defense against potential exploits.
- **Diversity of contributions:**
  - The open source ecosystem thrives on contributions that come from a variety of sources – individual hobbyists, academic researchers, corporate employees, and others. This diversity ensures that a broad range of use cases and viewpoints are considered during development.
  - Contributions can range from bug fixes and feature enhancements to security patches and performance improvements, all of which serve to strengthen the software.
- **Quality assurance (QA):**
  - The iterative nature of open source development, compounded by community feedback, tends to yield high-quality software. Users and developers alike are constantly testing, fixing, and updating the code, which often leads to software that is both refined and resilient.
  - The software evolves not just through planned updates but through continuous, incremental improvements and audits by the community.
- **Sustainability and longevity:**
  - Community support can contribute to the sustainability of open source projects. A vibrant, active community can continue development even if the original creators are no longer involved, ensuring the longevity of the project.

- The project's sustainability is also underpinned by the fact that it does not rely on the financial success or strategic direction of a single company, but on the collective will of its contributors.

## Transparency

The **transparency** inherent in open source LLMs is a multifaceted advantage that spans trust, security, compliance, and ethics. It provides a comprehensive suite of benefits that can lead to more responsible and reliable AI systems, thereby fostering a greater level of trust among users, developers, and the broader society in which these technologies are deployed.

The open source nature of software provides several transformative benefits:

- **Full transparency in the code base:**
  - Open source software is synonymous with an open-book policy regarding its code base. This level of transparency allows any user, developer, or researcher complete visibility into the internal workings of the software.
  - For LLMs, which are complex and often operate as black boxes, having an open code base demystifies the process by which they operate. It can empower users to tweak and improve the models, ensuring that the outcomes are explainable and aligned with expectations.
- **The foundation of trust and security:**
  - Transparency is a cornerstone of trust in software systems. When LLMs are used in critical applications, such as healthcare diagnostics, financial forecasting, or personal data processing, the stakes are incredibly high. In these scenarios, it's imperative that the models behave in predictable and secure ways.
  - Open source transparency assures users that there are no hidden processes that could potentially mislead or harm the end user. It also means that any security measures are open for inspection and critique, allowing for a more robust security posture.
- **Facilitation of audits and verifications:**
  - In many industries, software systems are subject to strict regulatory compliance standards. The open nature of the source code in open source LLMs enables comprehensive audits by third parties, who can verify that the software adheres to industry regulations and standards.

- This is especially pertinent in fields such as healthcare or finance, where software systems need to comply with regulatory frameworks such as the **Health Insurance Portability and Accountability Act (HIPAA)** or Sarbanes-Oxley. Auditors can inspect the code to ensure that the software meets all necessary compliance requirements.
- **Enhanced credibility with stakeholders:**
  - Transparency not only builds trust with users but also enhances the credibility of the software among stakeholders. When investors, partners, or regulators can see that an organization uses transparent and verifiable LLMs, it can facilitate smoother partnerships, funding opportunities, and regulatory approvals.
- **Community-driven security enhancements:**
  - The open source model invites a communal approach to security. Since the source code is available to everyone, it benefits from the collective vigilance of a broad community of experts who can spot and rectify security flaws.
- **Support for ethical AI development:**
  - As the field of AI continues to grapple with ethical concerns, transparency in LLMs provides a framework for ethical oversight.
  - This level of openness is crucial for the responsible development of AI systems, ensuring that they are fair, unbiased, and aligned with societal values.

## Flexibility and customization

The flexibility and customization potential of open source software, particularly for LLMs, provide a robust foundation for innovation and adaptation. Organizations can craft a software solution that not only meets their current needs but can also evolve alongside their ambitions and challenges, all while fostering a dynamic and collaborative development environment.

Open source software offers significant adaptability and scalability, providing critical advantages in several ways:

- **Tailoring to specific needs:**
  - Access to the source code is like having a master key to the software; it allows developers to get into the very heart of the program and make adjustments that fit their unique requirements. This is an immense advantage when the application of LLMs extends into niche domains with specialized needs that off-the-shelf products cannot meet.

- Customization can range from simple user interface tweaks to complex alterations in the algorithms and processing pipelines.
- **Scalability:**
  - As project requirements evolve, the need to scale the software can arise. Open source software can be modified to handle an increase in workload, such as larger datasets or more complex queries, without the need for a complete overhaul of the system.
  - Scalability can also refer to the ability to enhance the software's performance efficiency, enabling faster processing times and more economical use of computational resources, which is critical for the data-intensive tasks that LLMs perform.

## Rapid development and innovation

The open source model provides a fertile ground for rapid development and innovation in the field of LLMs. By leveraging the collective efforts of a global community, the development of LLMs can proceed at an unprecedented pace, with a multitude of contributors driving the technology forward in a variety of creative and unexpected ways.

Open source projects benefit immensely from collaborative efforts, offering several key advantages:

- **Accelerated evolution through collaborative contributions:**
  - Open source projects are unique in that they harness the collective capabilities of a diverse and global developer community. Each contributor can bring their own insights, skills, and experiences to the project, which can lead to a compounding effect on the speed of development and the introduction of innovative features.
  - For LLMs, the rapid inclusion of improvements—from language support enhancements to algorithmic efficiency gains—can be integrated into the project as soon as they are developed.
- **Collaborative approach leading to novel solutions:**
  - Open source development is inherently collaborative, not competitive. This environment fosters a culture where sharing knowledge is the norm, and it's this culture that leads to the discovery of novel approaches and techniques.
  - With LLMs, such collaboration might manifest as shared datasets, innovative training methods, or new neural network architectures. When these resources are shared, they can be tested, refined, and potentially integrated by others into a variety of projects, thereby enriching the entire ecosystem.

- **Diversity of thought and experimentation:**
  - The diversity of the open source community is one of its greatest strengths. Individuals from different backgrounds and with different objectives contribute to the project, bringing a wide range of ideas to the table. This diversity encourages experimentation and can lead to breakthroughs that might not occur in more homogeneous groups.

## No vendor lock-in

Avoiding **vendor lock-in** by using open source tools provides a strategic advantage, offering cost savings, technological agility, and the freedom to innovate. It empowers organizations to make decisions based on technical merit and strategic fit rather than being constrained by the decisions of a single vendor. This is especially critical in the rapidly evolving field of LLMs, where flexibility and the ability to quickly adapt to new developments can provide a significant competitive edge.

Avoiding vendor lock-in offers numerous benefits:

- **Avoidance of vendor lock-in:**
  - Substantial switching costs lead to vendor lock-in, where a customer becomes reliant on a specific vendor for products and services and finds it difficult to transition to another vendor
- **Cost implications:**
  - Vendor lock-in is often associated with escalating costs over time. As a vendor's product becomes more embedded in an organization's infrastructure, the vendor gains leverage to increase prices or change terms of service in ways that can be unfavorable to the customer.
  - Open source software, by contrast, is usually free from such constraints, which can result in significant long-term cost savings and budgetary predictability.
- **Agility and flexibility in technology choices:**
  - The tech industry is characterized by rapid evolution, and the flexibility to adapt to new technologies is crucial. Being locked into a single vendor's ecosystem can hinder an organization's ability to adopt new, more efficient, or cost-effective technologies.
- **Reduced risk of incompatibility and transition costs:**
  - Proprietary solutions often use closed formats and protocols, which can lead to compatibility issues when transitioning to another system. Open source tools, however, tend to favor open standards, which can minimize these risks. Additionally, if a vendor goes out of business or discontinues a product, customers can be left with unsupported software and a potentially expensive migration process.

## Wider adoption and collaboration

The lack of financial barriers in open source software leads to wider adoption, which in turn fosters a rich environment for collaboration and innovation. This collaborative ecosystem is conducive to the rigorous testing and continual refinement of tools, encouraging breakthroughs and ensuring the sustainability and evolution of LLMs in a way that proprietary models may not.

The elimination of financial barriers in open source software facilitates a range of benefits:

- **Facilitation of wider adoption:** Open source software, with its typically non-existent price tag, removes a significant barrier to entry. Without the financial burden, a broader demographic, from solo developers to large enterprises, can access the technology. This inclusivity not only increases the user base but also brings a variety of perspectives and skills to bear on the software's usage and development.
- **Enhanced testing and refinement:** A large and diverse user base can contribute to the rigorous testing of open source tools. In the context of LLMs, where different languages, dialects, and textual formats can vastly affect performance, such extensive testing is invaluable.
- **Promotion of collaborative innovation:** When financial barriers are removed, it encourages not just adoption but active collaboration. Academics, industry professionals, and hobbyists are able to contribute to the project, bringing together a rich tapestry of knowledge and expertise.

The adoption of open source tools for LLMs can present several strategic advantages, from cost savings to innovation and collaboration, making them a valuable resource for anyone in the field of AI and machine learning.

## Challenges

Apart from these advantages, there are also some limitations associated. Let's have a look.

### Resource intensity

While open source tools for LLMs offer many benefits, they also present challenges in terms of resource intensity. Implementing and maintaining these tools demands time, expertise, and often an investment in infrastructure and training. These indirect costs need to be carefully considered and managed to fully leverage the advantages of open source LLMs without encountering prohibitive barriers to their effective use:

- **Expertise and time investment:**
  - Open source software can often be complex and may not come with the same level of out-of-the-box readiness or comprehensive documentation that one expects from commercial software. Implementing these solutions effectively may require a high level of technical expertise and a willingness to invest significant time.
  - With LLMs, this challenge is pronounced due to the sophistication of the technology and the specialized knowledge required to not only implement but also to train, fine-tune, and maintain these models. Individuals or organizations may need to invest in training or hiring skilled personnel, which can be a significant indirect cost.
- **Maintenance demands:**
  - Unlike proprietary software that often includes vendor support and regular updates, maintenance of open source tools typically falls to the users. This includes updating the software, patching vulnerabilities, and ensuring compatibility with other systems and dependencies.
  - For LLMs, maintenance is particularly resource-intensive because the field is rapidly advancing, meaning that keeping up with the latest developments and integrating them can be a continuous and demanding task.
- **Hidden costs:**
  - While the software itself may be free, there are often hidden costs associated with open source tools. These can include the need for additional software or hardware to support the tool, training costs for staff, and the potential need for paid support or consultancy to fill gaps in expertise.
  - With LLMs, these hidden costs can accumulate rapidly, especially considering the data processing and computational power required to run these models effectively.

## Support and reliability

While the collaborative nature of community support is a hallmark of open source software, the absence of dedicated, professional support can pose challenges in terms of reliability and the timely resolution of issues. This is particularly relevant for organizations using LLMs for critical applications, where the cost of failure can be high.

While open source software offers many advantages, it also presents unique challenges:

- **Variability in community support:**
  - The support system for open source software is generally community-driven, which means that the quality and speed of support can vary greatly. While there are often active forums and user groups, there is no guarantee of timely assistance, and the level of expertise may be inconsistent.
  - In the context of LLMs, which are complex systems requiring sophisticated understanding, the absence of guaranteed professional support can be a significant risk. If an organization encounters a specialized issue or requires immediate help, community forums may not provide the level of service needed.
- **Professional support services:**
  - Proprietary solutions often come with the option of a **service-level agreement (SLA)**, ensuring a certain standard of support. Open source tools typically do not offer this level of dedicated support as part of the package, which can lead to challenges, especially in a production environment where downtime or unresolved issues can have serious implications.
  - Organizations using open source LLMs may need to rely on third-party vendors for professional support, which can introduce additional costs and complexities. Alternatively, they may need to build their own internal expertise, which can be a costly and time-consuming process.
- **Reliability and accountability:**
  - With proprietary software, there is a clear line of accountability to the vendor for the performance and reliability of the product. In the open source world, the software is often the result of contributions from many different individuals and organizations, which can make accountability diffuse.
  - For critical applications of LLMs, the lack of a single point of accountability can be a significant concern. If a system fails or does not perform as required, it may be challenging to identify a responsible party to address the issue.
- **Ongoing development and updates:**
  - The continuity of development for open source software can be uncertain. While some projects are robustly maintained, others may suffer from periods of stagnation or may be abandoned altogether if the community's interest wanes or key contributors leave.
  - For LLMs, where ongoing development is crucial to keep pace with the latest advancements in the field, the lack of reliable updates can limit the software's usefulness over time.

- **QA processes:**
  - Open source projects may not have the same rigorous QA processes as commercial software. While the community can and often does contribute to testing and QA, the processes can be less structured and comprehensive than those provided by a dedicated vendor team.
  - This can impact the reliability of LLMs, where the accuracy and quality of the model's outputs are paramount.
- **Custom solutions and workarounds:**
  - In the absence of dedicated support, organizations may find themselves having to develop custom solutions or workarounds to problems. This can be a drain on resources and may not always lead to the most efficient or effective outcomes.
  - With LLMs, which may be integrated into larger systems, developing these solutions can be particularly complex and require a deep understanding of both the model and the system architecture.

## Integration

While open source tools offer many advantages, they can present challenges when it comes to integration with existing systems. Compatibility issues, the need for custom development, and the potential lack of enterprise-ready features are all factors that organizations must consider. Successful integration of LLMs into existing IT infrastructures requires careful planning, a clear understanding of both the open source tool and the target environment, and a potentially significant investment in development and testing.

Integrating open source tools, especially complex systems such as LLMs, can present several challenges:

- **Compatibility issues:**
  - Open source tools are developed by diverse groups of contributors and may not always adhere to standardized protocols or interfaces, which can lead to compatibility issues with existing systems. This is especially relevant for LLMs, as they often need to interact with various data sources, processing pipelines, and application interfaces.
  - Ensuring that an open source LLM works harmoniously with proprietary or legacy systems can require significant effort in terms of developing middleware or custom adapters. Such compatibility layers may need to be built from scratch, requiring in-depth knowledge of both the open source software and the existing systems' architecture.

- **Seamless integration challenges:**
  - Proprietary tools are often designed with integration in mind, offering built-in connectors and plugins for popular enterprise software. Open source tools, in contrast, may lack these turnkey integration solutions, potentially leading to more complex and labor-intensive integration processes.
  - For LLMs, which are data-driven and may need to integrate tightly with content management systems, databases, or other AI services, the lack of seamless integration can be a significant hurdle. Organizations may need to allocate more resources to ensure the smooth flow of data and functionality between systems.
- **Documentation and support for integration:**
  - The quality and comprehensiveness of documentation can vary widely in the open source community. While some projects may provide extensive integration guides, others may have sparse or outdated documentation, which can complicate integration efforts.
  - Without adequate documentation, developers may need to rely on trial and error or seek guidance from community forums, which can be time-consuming and may not yield definitive solutions for integrating LLMs with specific systems or technologies.
- **Evolving landscapes and standards:**
  - The IT landscape is continuously evolving, with new standards and best practices emerging regularly. Open source tools may lag in adopting these new standards, or they may adopt them in ways that are not consistent with industry norms, further complicating integration efforts.
  - For LLMs, staying abreast of data privacy standards, security protocols, and API conventions is critical. Any lag in alignment can be problematic when trying to integrate with systems that adhere to the latest standards.

## Proprietary tools for LLMs

After reviewing the advantages and challenges of using open source tools for LLMs, it's time to turn to proprietary tools.

## Advantages

Let's start with their advantages.

### Ease of use

The advantages of proprietary tools for LLMs in terms of ease of use stem from a user-centric design philosophy, comprehensive support infrastructure, and a focus on providing a reliable and professional-grade product. These factors contribute to a smoother user experience, making proprietary tools attractive to individuals and organizations looking for turnkey solutions that allow them to leverage the power of LLMs with minimal setup and ongoing maintenance efforts.

Proprietary tools, particularly in the realm of LLMs, offer a suite of user-centric features that enhance accessibility and ease of use:

- **User-friendly interface:**
  - Proprietary tools are typically developed with a focus on user experience, offering interfaces that are intuitive and visually appealing. These interfaces are often the result of substantial research and user testing to ensure that they meet the needs of a broad user base.
  - For LLMs, this means providing access to complex functionalities through simplified dashboards, clear menu structures, and comprehensive onboarding processes. It enables users with varying levels of technical expertise to work with the model without needing to understand the underlying code.
- **Out-of-the-box functionality:**
  - One of the key selling points of proprietary software is its readiness for immediate use upon installation, with minimal setup required. This contrasts with open source tools that may require additional configuration or the installation of dependencies before they can be used effectively.
  - Proprietary LLMs are likely to come pre-configured with a range of defaults that are suitable for many common applications, allowing users to start their tasks with minimal delay.
- **Streamlined workflows:**
  - Proprietary LLMs often include streamlined workflows that guide users through the process of using the tool, from data input to model training and output analysis. This can significantly reduce the learning curve and increase productivity.
  - These workflows are also often accompanied by wizards or help functions that can guide users through complex processes step by step, making the technology accessible to non-experts.

- **Comprehensive documentation and training:**
  - Vendors of proprietary software usually provide extensive documentation, tutorials, and training materials. These resources are designed to help users get the most out of the software and can be crucial in helping to overcome any initial barriers to effective use.
  - For LLMs, which can be complex to operate, having well-structured and easily accessible documentation can be a significant advantage, enabling users to understand and leverage the tool's full capabilities.
- **Support services:**
  - Proprietary software often comes with customer support services included in the purchase price or available as an added service. This professional support can range from troubleshooting assistance to help with customization or integration.
  - Users of proprietary LLMs can typically rely on a consistent level of support, ensuring that any issues can be resolved quickly, which is essential for maintaining continuity in business operations.
- **QA and reliability:**
  - Proprietary software vendors have reputations to uphold and are therefore motivated to ensure their products meet high standards of quality and reliability. Extensive testing before release is standard practice.
  - Users of proprietary LLMs can expect a product that has been vetted for a wide range of scenarios and is less likely to contain critical bugs or errors.

## Support

The professional support and regular updates that come with proprietary tools for LLMs are significant advantages. They ensure that users have access to expert assistance, continuous improvements, and reliable maintenance of their software, which can be particularly valuable for organizations that rely on the functionality and performance of LLMs for their core operations.

Proprietary software vendors provide comprehensive professional support services that offer several key benefits for users:

- **Professional support services:**
  - Proprietary software vendors typically offer a range of support services, which can be essential for users who rely on LLMs for important business functions. This support can come in various forms, including direct access to technical experts, help desks, and customer service centers.

- The professional support teams are usually well trained on the specific software and can provide quick, reliable assistance, which can be crucial when time-sensitive issues arise. This level of support is particularly valuable for organizations that may not have in-house expertise in the complexities of LLM technology.
- **SLAs:**
  - Proprietary vendors often work with SLAs that guarantee a certain level of service, response time, and availability. This contractual assurance can be critical for businesses that depend on LLMs for critical operations.
  - SLAs provide peace of mind and a level of predictability for businesses, ensuring that they know what to expect in terms of support and service quality.

## Stability and reliability

The stability and reliability of proprietary tools for LLMs stem from structured development and release processes, stringent QA, and a focus on ensuring that updates improve the software without causing unnecessary disruption. This creates an environment where businesses can rely on their LLMs to perform consistently and effectively over time, providing peace of mind and allowing for long-term planning and investment in these tools.

Proprietary software vendors maintain stable release cycles and rigorous QA processes, providing numerous advantages for users of proprietary LLMs:

- **Stable release cycles:**
  - Proprietary software vendors often have well-established release cycles for updates and new versions of their software. This controlled release process is designed to ensure that each update is thoroughly tested and stable before it's made available to customers.
  - For users of proprietary LLMs, this means they can expect a platform that remains consistent over time, with updates that are less likely to introduce significant changes that would require users to alter their workflows or retrain their models extensively.
- **QA processes:**
  - Before releasing any updates, proprietary tools undergo rigorous QA testing. The QA process in a proprietary setting is systematic and comprehensive, aiming to catch and fix any potential issues before the software reaches the customer.

- This focus on quality leads to a more reliable and stable experience for users of proprietary LLMs, reducing the likelihood of encountering bugs or other issues that could disrupt their operations.
- **Predictable performance:**
  - Proprietary LLMs are engineered to provide predictable performance across various conditions and use cases. The providers ensure that the models perform optimally within the expected parameters, providing a level of reliability that is essential for users who rely on these tools for critical decision-making.
  - The reliability of proprietary LLMs is especially crucial in environments where the cost of failure is high, such as finance, healthcare, or legal industries.
- **Long-term support (LTS) versions:**
  - Many proprietary vendors offer LTS versions of their software, which receive maintenance updates for an extended period. These LTS versions are ideal for enterprise environments where stability is more important than having the latest features.
  - Users leveraging proprietary LLMs for core business functions can benefit from LTS versions, which provide the security of continued support without the need for frequent upgrades.

## Compliance and security

The commitment to compliance and security is a key advantage of proprietary tools for LLMs. Vendors who invest in these areas help ensure that their tools not only protect sensitive data but also meet the regulatory requirements that are critical for sensitive applications. This support can provide peace of mind for organizations and reduce the burden of managing compliance and security risks internally.

Proprietary vendors ensure their software adheres to industry standards and regulations, offering several crucial benefits:

- **Adherence to industry standards:**
  - Proprietary vendors typically design their tools to comply with industry standards and regulations. This includes following best practices for data handling, privacy, and security measures, which are crucial for maintaining the integrity and confidentiality of sensitive information.
  - For LLMs, this means that the software is more likely to be aligned with standards such as GDPR for data protection, HIPAA for healthcare information, and PCI Data Security Standard for payment data security, among others.

- **Certifications and audits:**
  - Proprietary software vendors often undergo regular third-party audits and strive to obtain certifications that affirm their compliance with various industry standards. These certifications serve as evidence of the software's reliability and adherence to regulatory requirements.
  - For organizations using LLMs, these certifications can simplify compliance efforts, as they can rely on the vendor's software to meet the necessary legal and industry-specific regulatory frameworks.
- **Security features:**
  - Security is a paramount concern in proprietary software development. Vendors invest in building robust security features, such as encryption, access controls, and activity monitoring, to protect against unauthorized access and data breaches.
  - In the context of LLMs, which process and generate large volumes of data, including potentially personal or proprietary information, these security features are essential to protect the data and the insights derived from it.
- **Risk mitigation:**
  - By providing tools that are compliant with regulations and standards, proprietary vendors help mitigate the risks associated with non-compliance, such as legal penalties, data breaches, and reputational damage.
  - Users of proprietary LLMs can leverage the vendor's expertise in compliance to reduce their own risk exposure, especially when operating in sectors where the mishandling of data can have serious consequences.

## Challenges

Let's also have an overview of the associated challenges.

### Cost

While proprietary tools for LLMs offer numerous benefits in terms of support, reliability, and compliance, they can also represent a substantial financial investment. Organizations must carefully evaluate the direct and indirect costs of these tools, including licensing and subscription fees, additional services, and potential costs related to scaling and customization, to determine whether they align with the organization's budget and long-term financial planning.

Proprietary software, including LLMs, often involves various costs:

- **Licensing fees:**
  - Proprietary software typically requires the purchase of a license to use the software. These licenses can be structured in various ways, such as per user, per machine, or per core/CPU, and may vary greatly in cost depending on the scale of the deployment.
  - For LLMs, the licensing fees might also be calculated based on the volume of data processed or the number of API calls made, which can add to the overall cost, especially for organizations handling large quantities of data.
- **Subscription fees:**
  - Many proprietary LLMs are offered under a subscription model, where users pay a recurring fee to use the software. Subscriptions can provide access to a suite of services and ensure that the software stays up to date with the latest features and security updates.
  - While subscription models can lower the initial cost of entry compared to perpetual licenses, over time, they can amount to a significant expenditure, especially if the subscription includes tiered pricing based on usage levels.
- **Additional services or add-ons:**
  - Proprietary vendors often offer a range of additional services and add-ons that can enhance the functionality of the software. These might include advanced analytics, custom model training, premium customer support, and more.
  - These services can be essential for making the most of LLMs but come at an additional cost. Organizations may find that the base version of the software requires several add-ons to meet their specific needs, which can significantly increase the total cost of ownership.
- **Integration and customization costs:**
  - While proprietary tools may offer ease of use and stability, integrating them with other systems or customizing them to suit specific requirements can require additional investment in professional services or custom development work.
  - For LLMs, integration with existing databases, CRM systems, or other enterprise software may necessitate specialized services that add to the overall cost.

## Less flexibility

While proprietary tools for LLMs provide benefits such as ease of use, support, and stability, they often lack the flexibility that some users require. This can manifest in constraints on customization, integration challenges, and a limited ability to adapt the tool to specific needs without incurring additional costs or relying on the vendor's development schedule. Organizations must weigh these factors against their need for a tailored solution when considering proprietary LLMs.

Proprietary software, including LLMs, presents certain limitations and dependencies:

- **Customization limitations:**
  - Proprietary software is often designed as a closed system, with limited options for customization. This is because the source code is not accessible for users to modify, which is a stark contrast to open source software where customization is a key feature.
  - In the case of LLMs, this means that users might not be able to adjust or extend the model's architecture, tweak its learning algorithms, modify its interface to suit their unique workflows, or integrate with their existing systems seamlessly.
- **Dependence on the vendor for enhancements:**
  - When customization is needed for proprietary LLMs, users are typically dependent on the vendor to provide these enhancements. This can result in waiting for the vendor to develop a requested feature or change, which may not always align with the user's timeline or priorities.
  - Additionally, vendors may prioritize developments based on their strategic interests or the interests of their largest customers, potentially leaving smaller users with unmet customization needs.
- **Integration with other systems:**
  - Proprietary LLMs may not integrate smoothly with other systems, particularly if those systems are from different vendors or are built on open source platforms. This can force organizations to work within a more rigid framework, using only the tools and integrations that the vendor supports.
  - Overcoming these integration challenges often requires the use of APIs, middleware, or other interfacing tools that the vendor provides, which may not offer the level of control or data interaction that the user desires.

## Choosing between open source and proprietary tools for LLMs

Choosing between open source and proprietary tools for LLMs will depend on several factors:

- **Project budget and resources:** If budgets are tight and in-house expertise is available, open source may be the way to go. For organizations that prefer a more managed solution and can afford it, proprietary may be more suitable.
- **Customization needs:** If the project requires heavy customization, open source tools may offer the necessary flexibility.
- **Scalability and integration:** For projects that need to scale quickly and integrate with other systems, proprietary tools might offer more robust solutions.
- **Security and compliance:** For projects that handle sensitive data or require strict compliance with regulations, proprietary solutions often provide comprehensive security features and compliance certifications.

Ultimately, the decision may not be binary, and many organizations find that a hybrid approach—using a mix of open source and proprietary tools—best meets their needs. It's also common to start with open source tools for prototyping and experimentation, and then switch to proprietary solutions for production-level deployment.

In conclusion, the choice between open source and proprietary tools for LLMs should be guided by a clear understanding of the project requirements, available resources, and strategic goals. It's also important to stay informed about the evolving landscape of LLM tools and to periodically reassess the tooling strategy as new technologies and updates become available.

## Integrating LLMs with existing software stacks

Integrating LLMs with existing software stacks is an important step for businesses and developers looking to leverage the power of advanced NLP within their current technological ecosystem. This integration process typically involves several key considerations:

- **Assessment of requirements:** Understanding the specific needs of the business or application is crucial. This includes determining what tasks the LLM will perform, such as text generation, sentiment analysis, or language translation.

- **Choosing the right LLM:** Depending on the requirements, a suitable LLM should be chosen. For example, GPT-4 might be chosen for its text generation capabilities, while BERT might be preferred for its performance in understanding context in search queries.
- **APIs and integration points:** Most LLMs provide APIs that are the primary means of integration with existing software stacks. These APIs allow the LLM to communicate with other systems, passing data back and forth as needed.
- **Data handling and processing:** To effectively integrate an LLM, you need to ensure that your data is in the right format. This might involve preprocessing steps to clean and structure the data before it can be used by the LLM.
- **Infrastructure considerations:** LLMs can be resource-intensive, so it's important to ensure that your existing infrastructure can handle the additional load. This may involve scaling up your servers or moving to a cloud-based solution.
- **Security and privacy:** When integrating an LLM into your software stack, you need to consider the security and privacy implications, particularly if you're dealing with sensitive or personal data. This might involve implementing additional security measures or ensuring that the data is anonymized before being processed by the LLM.
- **Compliance and ethics:** It's vital to ensure that the use of LLMs complies with relevant laws and regulations, such as GDPR for data protection. Ethical considerations should also be taken into account, ensuring that the LLM is used in a manner that is fair and does not perpetuate biases.
- **Testing and validation:** Before fully integrating an LLM into your software stack, it should be thoroughly tested. This testing should validate that the LLM performs as expected and works seamlessly with the other components of your software stack.
- **Monitoring and maintenance:** Once integrated, the LLM should be continuously monitored to ensure it is functioning correctly. Regular maintenance may also be required to update the model or the integration as new versions are released.
- **User training:** It's often necessary to train users on how to interact with the LLM, especially if they are using it as part of their workflow, such as customer service representatives or content creators.
- **Scalability and futureproofing:** The integration should be designed in a way that it can scale as the usage of the LLM grows. Also, it should be flexible enough to accommodate future advancements in LLMs.

- **Documentation:** Comprehensive documentation of the integration process and the way the LLM interacts with other system components is important for maintenance and future reference.

Integration of LLMs into existing software stacks is not a one-size-fits-all process. It requires careful planning, consideration of technical and ethical implications, and ongoing management. However, when done correctly, it can significantly enhance the capabilities of your software stack and provide valuable services to your users.

## The role of cloud providers in NLP

Cloud providers play a crucial role in the field of NLP by offering a wide range of services that democratize access to cutting-edge technologies. Their contribution can be categorized into several key areas:

- **Infrastructure:** Cloud providers offer the necessary computational infrastructure required for NLP tasks, which often demand significant processing power. This infrastructure supports the training of large-scale language models and the deployment of NLP applications without the need for organizations to invest in their own hardware.
- **Platform as a service (PaaS):** Through PaaS offerings, cloud providers supply platforms that allow developers to build, deploy, and manage NLP applications without the complexity of building and maintaining the infrastructure.
- **NLP services and APIs:** Cloud providers such as Amazon AWS, Google Cloud Platform, and Microsoft Azure offer a suite of pre-built NLP services and APIs. These include text analysis, translation, sentiment analysis, and chatbot services, making it easier for businesses to integrate NLP capabilities into their applications.
- **ML frameworks and tools:** They provide access to ML frameworks such as TensorFlow, PyTorch, and MXNet, which are essential for building custom NLP models. These tools come with the added benefit of cloud scalability and managed services.
- **Data storage and management:** NLP models require access to vast datasets. Cloud providers offer scalable and secure data storage solutions, along with tools for managing and processing this data effectively.
- **AutoML and custom model training:** For organizations without the expertise to build models from scratch, cloud providers offer AutoML services, which automate the creation of custom NLP models tailored to specific needs.

- **Marketplaces for AI models:** Cloud platforms often have marketplaces where users can find and deploy pre-built NLP models, which can be further customized for specific tasks.
- **Security and compliance:** Cloud providers ensure that the deployment of NLP applications complies with security standards and privacy regulations, which is particularly important when processing sensitive data.
- **Global reach and localization:** They facilitate the deployment of NLP applications across global infrastructures, ensuring low latency and compliance with local data residency requirements. This is especially important for NLP applications that need to be localized for different languages and regions.
- **Research and development:** Cloud providers invest heavily in research, developing state-of-the-art NLP technologies that can be leveraged by their customers. They also often provide credits and support for academic research in NLP.
- **Community and support:** They foster a community of developers and provide extensive documentation, tutorials, and forums for support, which is invaluable for teams working on NLP projects.
- **Scalability and flexibility:** One of the most significant advantages of cloud providers in NLP is the ability to scale resources up or down as needed, providing flexibility and cost control for businesses of all sizes.

In conclusion, cloud providers are integral to the NLP ecosystem, providing the tools, services, and infrastructure that enable businesses to harness the power of language processing. They continually push the boundaries of what's possible in NLP, offering increasingly sophisticated solutions that allow for innovation and expansion in the field.

Chapter

# 34

# The Future of LLMs: GPT-5 and Beyond

Anticipating future developments in the LLM space, we will prepare you for the arrival of GPT-5 and subsequent models in this chapter. We will cover the expected features, infrastructure needs, and skillset preparations. We will also challenge you to think strategically about potential breakthroughs and how to stay ahead of the curve in a rapidly advancing field.

## What to expect from the next generation of LLMs

Looking ahead, LLMs will significantly improve in understanding and contextualizing information. Future models will maintain context over long interactions, ensuring coherent and fluid dialogue by integrating dynamic memory to recall past exchanges. This enhanced contextual capability will help LLMs resolve ambiguities based on broader conversations, making them adept at generating relevant and coherent content. These advancements will enrich user experiences and broaden the application of LLMs in complex, language-intensive scenarios, making them more proficient and helpful conversational partners across various domains. Let's explore these features in detail.

### Enhanced understanding and contextualization

As we look to the future of LLMs, one of the most exciting developments is the enhanced understanding and contextualization capabilities that these models are expected to have. Here are several areas in which these improvements may manifest:

- **Long-term context management:** Future LLMs may be able to maintain context over longer interactions, remembering and referencing past parts of a conversation. This would allow for more natural and fluid dialogue, as the model wouldn't "forget" previous exchanges.
- **Dynamic memory integration:** By incorporating a dynamic memory component, LLMs could store and retrieve information from earlier in the conversation or from past interactions with the same user. This is a step beyond static contextual understanding, where the model can only reference immediate or recent input.
- **Cross-session learning:** Beyond a single session, LLMs might be capable of cross-session learning, where they remember user preferences, interests, and history across multiple interactions. This would lead to highly personalized experiences.
- **Contextual disambiguation:** Improved contextualization would enable LLMs to better understand ambiguous language based on a conversation's context. They could resolve ambiguities by considering the broader topic or by recalling how similar phrases were used previously in the dialogue.
- **Coherence and cohesion:** Coherence (the logical connection between ideas) and cohesion (how sentences and parts of text connect) in a model's responses are expected to improve, making conversations with LLMs more logical and easier to follow.
- **Advanced reference capabilities:** The ability to reference and understand indirect language, such as pronouns or elliptical constructions, will likely be enhanced, allowing for more complex and varied dialogue that remains clear and connected.
- **Context-aware content generation:** In content generation tasks, LLMs would be capable of creating text that is not only relevant to the immediate prompt but also considers the wider context, such as a user's known interests or the current cultural or social milieu.
- **Emotion and sentiment understanding:** By better understanding the context, LLMs could more accurately interpret the emotional tone and sentiment of user input, leading to more empathetic and appropriate responses.
- **Situational awareness:** Future models might develop situational awareness, allowing them to adapt their responses based on the perceived situation or setting of an interaction, such as a formal business meeting versus a casual chat.

- **Personalized learning paths:** In educational applications, LLMs could create personalized learning paths that consider a student's progress, interests, and the context of past performance.
- **Ethical and cultural sensitivity:** With better contextualization, LLMs could show improved sensitivity to ethical considerations and cultural contexts, avoiding misunderstandings and inappropriate responses.

Enhancing these capabilities will improve user experience and expand LLM applications in complex, language-intensive scenarios. As technology advances, LLMs will become more adept at sustaining meaningful and helpful interactions across various domains.

## Improved language and multimodal abilities

The next generation of LLMs is expected to exhibit not only advanced language processing capabilities but also multimodal abilities. Here's a detailed look at what this might involve:

- **Multimodal processing:** Multimodal LLMs will be able to process and understand information from various data types beyond text, including visual elements (images and videos), auditory signals (speech and sounds), and possibly tactile feedback, allowing for richer interactions.
- **Content generation across media:** These models could generate coherent content that spans multiple forms of media. For example, an LLM might create a textual description that aligns with visual content or vice versa.
- **Contextual understanding of visual elements:** Improved language models will likely have a deeper understanding of the context within images and videos. This means recognizing objects, actions, and sentiments in visual media and relating them to textual information.
- **Advanced natural language generation (NLG):** Coupled with visual or auditory data, NLG in multimodal LLMs will produce more descriptive and accurate narratives or captions, enhancing the storytelling aspect of content creation.
- **Audio and speech processing:** In audio processing, future LLMs could transcribe, interpret, and even generate human-like speech. This could revolutionize virtual assistants, making them more natural and responsive to vocal nuances.

- **Video understanding and generation:** LLMs might be capable of analyzing video content, understanding a sequence of events, and generating summaries or interactive elements based on video data.
- **Cross-modal translation:** One of the more intriguing prospects is the translation of content from one modality to another, such as describing a scene in a photo using text or creating an image from a descriptive paragraph.
- **Enhanced user experience:** Multimodal LLMs can offer a more immersive and interactive user experience, providing output that engages multiple senses and caters to different learning styles and user preferences.
- **Accessibility improvements:** These capabilities can also enhance accessibility, translating text to speech for the visually impaired or generating sign language animations from speech for the hearing impaired.
- **Richer data interpretation:** By synthesizing information from different modalities, LLMs can provide a more comprehensive interpretation of data, useful in fields such as medical diagnosis, where visual (e.g., MRI scans) and textual (e.g., clinical notes) data must be combined.
- **Creative and design applications:** In creative fields, multimodal LLMs could assist in design processes, providing suggestions for visual content based on textual descriptions or creating drafts for multimedia campaigns.
- **Interactive learning and gaming:** Educational software and games could become more interactive and adaptive with multimodal LLMs, offering learning content and feedback through various forms of media.

The development of multimodal LLMs marks a major advance in AI capabilities. Integrating various data types, these models enable more natural, intuitive interactions, transforming communication, learning, and creation across industries.

## Greater personalization

The evolution of LLMs is set to usher in new levels of personalization, enhancing the user experience in a way that feels bespoke and individualized. Let's explore what this might include:

- **User preference learning:** Future LLMs will likely have the ability to learn and remember user preferences over time, adapting their responses and suggestions based on past interactions, user choices, and feedback.
- **Adaptive content delivery:** Content delivery could be dynamically adjusted to match a user's interests, reading level, and engagement patterns. This means that the information presented to the user would be in line with what is most relevant and engaging to them personally.

- **Contextual recommendations:** By analyzing user behavior and context, LLMs could make highly relevant recommendations, similar to how advanced recommendation engines work today, but with a broader and more nuanced understanding of context.
- **Learning style adaptation:** Educational applications of LLMs could adapt to different learning styles, offering explanations, examples, and practice exercises that align with a user's preferred way of learning.
- **Conversational memory:** In conversations, LLMs would be able to recall past discussions, creating a sense of continuity and understanding, much like interacting with a familiar human over time.
- **Emotional intelligence:** Greater personalization also involves emotional intelligence, where an LLM can detect subtle cues in language to understand a user's emotional state and respond in a way that demonstrates empathy.
- **Customized creativity:** In creative tasks, such as writing, designing, or composing music, LLMs could reflect a user's personal style and past creative choices in the content they generate.
- **Personalized language use:** LLMs could adjust the complexity, tone, and type of language used, based on a user's proficiency and comfort with the language, making communication more effective and comfortable.
- **Predictive personalization:** Leveraging predictive analytics, LLMs might anticipate user needs and provide assistance before a user explicitly requests it, based on patterns and inferred intentions.
- **Integrated personalization across platforms:** Personalization could extend across different platforms and devices, providing a consistent and tailored experience, whether the user is on their phone, computer, or using a voice assistant.
- **Ethical and privacy considerations:** As personalization deepens, so does the importance of managing it ethically. Ensuring user privacy and consent will be critical, with LLMs needing to balance personalization with responsible data use.
- **Personalized accessibility features:** Accessibility features could be personalized, with LLMs adjusting the way content is presented to suit individual accessibility needs, such as for users with visual or auditory impairments.
- **Interactive personalized feedback:** For tasks such as learning or fitness, LLMs could provide interactive, personalized feedback that helps users improve their skills or achieve their goals in a way that's most effective for them.

The development of multimodal LLMs significantly advances AI capabilities. By integrating various data types, these models transform communication, learning, and creation across industries.

## Increased efficiency and speed

The next generation of LLMs is poised to bring substantial improvements in terms of efficiency and speed. Here are the key areas where these improvements are expected:

- **Model architecture innovations:** Future LLMs will benefit from advancements in neural network architectures, which might include more efficient transformer models or entirely new designs that optimize the way information is processed and retrieved
- **Processing power advancements:** As hardware technology evolves, the increase in processing power will allow LLMs to handle complex computations more quickly, significantly reducing the time required to generate responses
- **Optimized algorithms:** Algorithms used in LLMs will likely become more sophisticated, with better optimization techniques that enable faster data processing without sacrificing the quality of the output
- **Parallel processing techniques:** By leveraging parallel processing, LLMs will be able to handle multiple tasks at once, which will be instrumental in managing a higher number of simultaneous users
- **Quantum computing potential:** Although still in the early stages, quantum computing has the potential to revolutionize the speed at which LLMs operate, by processing vast amounts of data at speeds unattainable with classical computing
- **Distributed computing:** The use of distributed computing, where tasks are spread across multiple machines, will enhance the performance of LLMs, especially in cloud-based services
- **Edge computing integration:** By integrating edge computing, where data processing is done closer to the data source, LLMs will be able to deliver faster responses, particularly in real-time applications
- **Resource-efficient training:** Innovations in training procedures will make LLMs quicker to train, with fewer data requirements and more efficient use of computational resources
- **Cache and memory optimization:** Improvements in caching techniques and memory usage will allow LLMs to retrieve and utilize information more efficiently, speeding up response generation

- **Dynamic scaling:** Cloud services will likely offer dynamic scaling capabilities for LLMs, automatically adjusting the amount of computational power allocated, based on the current demand
- **Energy-efficient computing:** There will be a focus on making computing more energy-efficient, which is not only better for the environment but also allows sustained high-speed processing
- **Real-time interaction capabilities:** For applications that require real-time interaction, such as digital assistants or online gaming, LLMs will be optimized to provide instantaneous responses
- **Streamlined data fetching:** LLMs will become more adept at quickly fetching the necessary information from databases and knowledge bases, making the generation of informed responses much faster

Future LLMs will offer improved user experiences with rapid response times and high-volume request handling, which are crucial, as they integrate into everyday technologies and expand their user base.

## Advanced reasoning and problem-solving

Anticipated advancements in the reasoning and problem-solving abilities of next-generation LLMs include the following:

- **Complex decision-making:** Future LLMs will likely be able to navigate complex decision-making scenarios, weighing different factors and potential outcomes to arrive at reasoned conclusions.
- **Enhanced logic processing:** Improvements in logical reasoning will enable LLMs to better understand and apply logical operators and relationships, crucial for technical fields such as programming and mathematics.
- **Abstract reasoning:** Abstract reasoning involves understanding concepts that are not directly observed. LLMs will become better at extrapolating from known information to new, unseen situations or problems.
- **Creativity in problem-solving:** Creativity is not just an artistic trait; it's also key to problem-solving. LLMs will be able to generate novel solutions to problems by combining seemingly unrelated concepts in innovative ways.
- **Strategic planning:** LLMs will improve in planning several steps ahead, a skill necessary for tasks ranging from game playing to strategic business planning.

- **Understanding cause and effect:** Recognizing cause-and-effect relationships will enable LLMs to more accurately predict outcomes and understand the implications of certain actions or events.
- **Cross-domain knowledge application:** Next-generation LLMs will be adept at applying knowledge from one domain to another, using analogical reasoning to solve problems in creative and effective ways.
- **Mathematical reasoning:** Enhanced mathematical reasoning will allow LLMs to perform more complex calculations and provide explanations or solutions to mathematical problems.
- **Ethical reasoning:** As AI becomes more prevalent, the ability of LLMs to reason ethically and consider the moral implications of their suggestions or actions will be crucial.
- **Emotionally intelligent responses:** Emotional intelligence in problem-solving involves understanding human emotions and considering them when proposing solutions, leading to more empathetic and human-centric AI.
- **Scientific reasoning:** LLMs could contribute to scientific research by hypothesizing, designing experiments (virtually), and interpreting data to derive logical conclusions.
- **Interactive learning and feedback:** Interactive learning will allow LLMs to reason through problems by asking questions, receiving feedback, and iteratively refining their understanding and approaches.
- **Dynamic resource allocation:** In computational terms, reasoning and problem-solving often require dynamic resource allocation, which next-generation LLMs could manage effectively to optimize their performance for different tasks.

These advancements will enhance LLM capabilities and expand their utility across sectors such as education, healthcare, finance, and technology, aiming to develop LLMs as advanced cognitive partners in problem-solving tasks.

## Broader knowledge and learning

The forthcoming generations of LLMs will likely be distinguished by their expansive knowledge bases and the capacity for real-time learning. Here's what we might expect:

- **Expansive knowledge bases:** LLMs will have access to vast amounts of information, encompassing a wide array of subjects, languages, and cultural contexts, allowing for a much more comprehensive understanding of user queries.

- **Real-time information updating:** Unlike current models that require periodic retraining, future LLMs might continuously update their knowledge base with the latest information, research findings, news, and trends, ensuring that the knowledge they provide is current and relevant.
- **Dynamic learning from interaction:** LLMs will learn from each interaction, refining their models to improve accuracy and relevance for future responses. This could include learning from user corrections, feedback, and engagement metrics.
- **Cross-disciplinary synthesis:** By integrating knowledge from various disciplines, LLMs will be able to synthesize information, providing more nuanced and comprehensive answers that draw from multiple fields of expertise.
- **Personalized knowledge paths:** LLMs will create personalized knowledge paths for users, guiding their learning based on previous interactions, stated goals, and demonstrated interests.
- **Predictive learning:** Anticipating users' needs based on past behaviors, LLMs will preemptively learn and present information that aligns with their predicted queries or tasks.
- **Contextual understanding:** The ability to understand context deeply will allow LLMs to discern which pieces of their broad knowledge are most relevant to present in any given situation.
- **Semantic understanding and reasoning:** Future LLMs will exhibit a more profound semantic understanding, enabling them to reason through complex topics and provide explanations that are not only factually accurate but also contextually meaningful.
- **Collaborative learning:** LLMs may be able to learn collaboratively, both from other AI systems and humans, leveraging the collective intelligence to enhance their knowledge base.
- **Expertise in niche areas:** While having broad knowledge bases, LLMs will also specialize in niche areas, becoming experts in specific domains where depth of knowledge is crucial.
- **Learning from diverse data sources:** Incorporating diverse data sources will prevent knowledge silos and ensure a well-rounded perspective, making LLMs reliable for a wide range of topics.
- **Multilingual and cultural learning:** LLMs will be adept at understanding and learning from multilingual content, enabling them to serve a global user base with cultural awareness and sensitivity.

- **Adaptive learning mechanisms:** LLMs will employ advanced adaptive learning mechanisms to tailor their learning process to the most efficient strategies, based on the task at hand.

These advancements will position LLMs as powerful tools for information dissemination, education, and decision support, making them integral to businesses, educators, researchers, and everyday users.

## Ethical and bias mitigation

The integration of ethical considerations and bias mitigation is a pivotal aspect of the ongoing development of LLMs. Here's an expanded view of what this entails:

- **Fairness and equity:** Future LLMs will incorporate algorithms designed to ensure fairness, actively preventing discriminatory outcomes based on race, gender, age, or other sensitive attributes.
- **Bias detection and correction:** Advanced techniques will be developed to detect biases within the training data and model outputs. Once identified, mechanisms will be put in place to correct these biases, ensuring more balanced and fair responses.
- **Privacy preservation:** LLMs will be designed with privacy as a core feature, employing methods such as differential privacy and federated learning to protect user data. They'll handle personal information responsibly, adhering to global privacy standards and regulations.
- **Transparency in decision-making:** There will be an emphasis on explainability, with LLMs providing clear explanations for their decisions and suggestions, enabling users to understand the reasoning behind AI-generated content and actions.
- **Misinformation prevention:** Techniques to recognize and avoid the spread of misinformation will be integral to LLMs. They'll cross-reference facts and check against reputable sources before disseminating information.
- **Ethical training and guidelines:** Ethical training for developers and guidelines for LLMs will be established, ensuring that ethical considerations are embedded in the design and deployment of these models.
- **Inclusive and diverse training data:** To mitigate biases, the training data for LLMs will be curated to be as inclusive and diverse as possible, representing a wide range of voices and perspectives.

- **Cultural sensitivity and localization:** LLMs will be attuned to cultural nuances and local contexts, avoiding generalizations that could lead to insensitive or incorrect responses.
- **User control over data:** Users will have more control over how their data is used by LLMs, including the ability to opt out of data collection or have their data deleted.
- **Robust content moderation:** LLMs will include robust content moderation systems to prevent the generation or amplification of harmful content.
- **Ongoing monitoring and auditing:** Continuous monitoring and regular auditing of LLMs will ensure that ethical standards are maintained and biases are addressed promptly.
- **Stakeholder engagement:** A broader range of stakeholders, including ethicists, sociologists, and representatives from affected communities, will be involved in the development and oversight of LLMs.
- **Legal and ethical compliance:** LLMs will be designed to comply with both the letter and the spirit of laws and ethical norms across jurisdictions, adjusting their behavior to align with local regulations and ethical expectations.

In summary, as LLMs become more sophisticated and widespread, addressing ethical issues and biases is crucial to creating trustworthy, fair, and ethically aligned models that benefit society equitably.

## Improved interaction with other AI systems

As we progress toward more integrated AI ecosystems, future LLMs are expected to serve as sophisticated intermediaries between humans and various specialized AI systems. Here's a deeper look into what this might involve:

- **Seamless integration:** LLMs will be designed to seamlessly integrate with a variety of AI systems, enabling smooth data exchange and interoperability without the need for extensive customization.
- **Translation of human requests:** They will be capable of translating complex human requests into the specific formats required by other AI services, acting as a user-friendly interface for more technical or specialized systems.
- **Central coordination role:** LLMs may assume a central coordination role within AI frameworks, directing tasks to the most suitable AI service and ensuring that outputs are combined to provide coherent and comprehensive responses.

- **Inter-AI communication:** Future LLMs will enable different AI systems to communicate with each other, even if they were developed independently, by standardizing the communication protocols and data formats.
- **Contextual relay:** When relaying information between systems, LLMs will provide context to ensure that each AI component understands the relevance of the data it receives or processes.
- **Real-time data synthesis:** LLMs will synthesize real-time data from multiple AI systems to provide up-to-date information and insights, critical for applications such as financial analysis or emergency response.
- **Human-in-the-loop interfaces:** They will facilitate human-in-the-loop interfaces, allowing human operators to step in or review decisions when necessary, thus ensuring that the integration of AI systems remains under human oversight.
- **Dynamic service selection:** By assessing the capabilities and current loads of various AI systems, LLMs will dynamically select the most appropriate service for a given task, optimizing the use of resources.
- **Error handling and diagnostics:** LLMs will assist in identifying and diagnosing errors or inconsistencies across interconnected AI systems, aiding in maintaining system integrity and performance.
- **Automated learning and updating:** They will enable automated learning and updating processes across different AI systems, sharing insights and new data to collectively improve performance.
- **Multi-agent collaboration:** LLMs will facilitate collaboration in multi-agent systems, where several AI agents work together on complex tasks, ensuring that each agent's contributions align with the overall objective.
- **Modular AI development:** The integration capabilities of LLMs will encourage more modular development of AI systems, where individual components can be developed independently but are designed to work together cohesively.
- **Customizable user experiences:** By interacting with different AI services, LLMs will be able to create highly customizable user experiences, combining services in unique ways to meet individual user needs.

In essence, LLMs will evolve into the connective tissue of AI, enabling various services to collaborate more effectively and enhancing AI's capability to perform complex tasks across diverse applications.

## More robust data privacy and security

As LLMs become more pervasive in both personal and professional settings, the importance of data privacy and security is paramount. Future developments in this area are expected to include the following:

- **Advanced encryption:** Encryption standards will evolve, with LLMs utilizing more sophisticated encryption methods to secure data both at rest and in transit, ensuring that sensitive information remains confidential
- **Differential privacy:** The implementation of differential privacy techniques will ensure that LLMs can learn from data without compromising the privacy of individuals within a dataset
- **Federated learning:** LLMs may use federated learning to train on decentralized data, allowing a model to learn from user data without that data ever leaving a user's device
- **Secure multi-party computation:** By maintaining the confidentiality of private inputs, **secure multi-party computation (SMPC)** facilitates the collaborative computation of a function among multiple entities, thereby enhancing the security of collaborative learning environments
- **Data anonymization and pseudonymization:** Enhanced methods of anonymization and pseudonymization will make it difficult to reverse-engineer personal data from model outputs, or from the model itself
- **Access control and authentication:** Robust access control mechanisms and authentication protocols will be in place to ensure that only authorized individuals can access sensitive data and model functionalities
- **Auditing and compliance:** LLMs will feature comprehensive auditing capabilities to track data usage and access, facilitating compliance with global data protection regulations such as GDPR and CCPA
- **Ethical data use frameworks:** Ethical frameworks for data use will guide the collection, storage, and processing of data within LLMs, ensuring that ethical considerations are at the forefront of data handling practices
- **Decentralized data storage:** Decentralized data storage solutions, such as blockchain, could be employed to enhance security and provide an immutable record of data transactions
- **Consent management:** Improved consent management systems will allow users to have granular control over what data they share and how it is used by LLMs

- **Continuous security monitoring:** LLMs will be monitored continuously for potential security breaches or vulnerabilities, with automated systems in place to detect and respond to threats in real time
- **AI-specific security protocols:** Given the unique nature of AI systems, specialized security protocols will be developed to protect against AI-specific threats, such as model inversion attacks or adversarial inputs
- **Data minimization principles:** Adhering to the principle of data minimization, LLMs will collect and process only the data that is absolutely necessary for the task at hand

These measures will collectively contribute to a more secure and privacy-respecting AI ecosystem. By embedding privacy and security into the fabric of LLMs, we can ensure that these powerful tools can be leveraged safely and responsibly.

## Customizable and scalable deployment

The next generation of LLMs is expected to offer a high degree of customization and scalability that will enable businesses of all sizes to tailor AI solutions to their specific needs. Here's what to anticipate:

- **Modular design:** Future LLMs may be designed in a modular fashion, allowing businesses to plug and play different components based on their specific requirements
- **Task-specific customization:** LLMs will be highly customizable, enabling businesses to fine-tune models for specialized tasks, whether it's customer service, data analysis, or content creation
- **Scalability:** These models will be inherently scalable, designed to handle varying loads of work, from small datasets and user bases to vast streams of data and millions of users – without a drop in performance
- **On-demand resources:** Cloud-based deployment will allow for on-demand resource allocation, meaning that businesses can scale their LLM usage up or down as needed, optimizing costs and resources
- **Integration with existing systems:** LLMs will come with tools and APIs that facilitate easy integration with existing business systems and workflows, minimizing the need for extensive overhauls
- **Automated deployment:** Deployment processes will be increasingly automated, using AI itself to assist in the setup and tuning of LLMs for specific business environments

- **Self-optimizing models:** LLMs will have self-optimizing capabilities, continuously learning from their performance and user feedback to improve their accuracy and efficiency over time
- **Industry-specific solutions:** There will be a proliferation of industry-specific LLMs, pre-trained on domain-specific data, which can then be further customized by individual businesses
- **User-friendly interfaces:** Businesses will have access to more user-friendly interfaces for customizing and managing LLMs, reducing the barrier to entry for those without extensive technical expertise
- **Performance monitoring and analytics:** Advanced monitoring and analytics will be built in, providing insights into model performance and helping businesses make data-driven decisions about scaling and customization
- **Edge AI deployment:** Some LLMs will be deployable at the edge, closer to where data is generated, to reduce latency and bandwidth use, particularly important for time-sensitive applications
- **Containerization and microservices:** The use of containerization and microservices architectures will promote more agile deployment and scaling of LLMs
- **Compliance and governance:** Customization options will include compliance controls, ensuring that LLMs adhere to regional regulations and industry standards as they scale

These advancements will make LLMs more accessible and effective across industries, allowing businesses to drive innovation, improve services, and stay competitive with customizable and scalable AI deployment options.

## Regulatory compliance and transparency

The future development of LLMs will be heavily influenced by the need for regulatory compliance and transparency. As these models become more integral to various sectors, from healthcare to finance, their alignment with legal and ethical standards becomes critical. Here's what this may entail:

- **Alignment with legal frameworks:** LLMs will be designed to comply with existing and emerging legal frameworks, such as the GDPR in Europe and others globally that regulate data privacy and AI ethics
- **Transparency in AI decision-making:** There will be a greater emphasis on creating LLMs that can explain their decision-making processes in understandable terms, allowing users to grasp how conclusions are reached

- **Audit trails:** LLMs will generate comprehensive audit trails that document the decision-making process, which is essential for compliance purposes and for reviewing AI's actions if there are disputes
- **Bias and fairness assessments:** Regular assessments will be conducted to ensure that LLM outputs are free from bias and that the models operate fairly across different demographics
- **Ethical AI design:** Ethical considerations will be embedded into the design process of LLMs, ensuring that they operate within the accepted moral bounds of society
- **Data handling and consent:** Robust systems to manage user consent regarding data use will be implemented, ensuring that LLMs handle data in a manner that respects user preferences and privacy laws
- **User empowerment:** Users will have more control over what data is used and how it is used by LLMs, including the ability to opt out or correct data
- **Standardization of practices:** Industry-wide standards for the development, deployment, and management of LLMs will likely emerge, ensuring consistency and reliability
- **Risk assessment and management:** LLMs will incorporate mechanisms to assess and manage risks, particularly in areas where AI decisions could have significant impacts on individuals or businesses
- **Interoperability:** LLMs will be designed for interoperability with other systems and AI models, facilitating a seamless exchange of information within a regulated framework
- **Governance structures:** Clear governance structures will be established to oversee LLM operations, including the roles and responsibilities of all parties involved in the AI life cycle
- **Consumer protection:** Measures will be put in place to protect consumers from potential harm caused by LLMs, including incorrect or harmful content generation
- **Open standards and protocols:** The use of open standards and protocols will likely be encouraged to promote transparency and allow independent verification of LLM compliance and performance
- **Cross-border compliance:** LLMs will need to navigate the complexities of cross-border compliance, adhering to the laws and regulations of all the jurisdictions they operate in

By proactively addressing regulatory compliance and transparency, developers and users of LLMs can foster trust in AI technologies. These measures are not only about adhering to regulations but also about ensuring that LLMs are used responsibly and ethically, contributing positively to society and instilling confidence in their output.

## Accessible AI for smaller businesses

The trend toward making powerful LLMs more accessible to smaller businesses marks a significant democratization of AI technology. Here's an overview of how this could unfold:

- **Cost-effective solutions:** Innovations in AI will lead to more cost-effective LLM solutions, reducing the financial barriers that often prevent smaller businesses from leveraging advanced AI technologies
- **Simplified integration:** LLM providers will likely offer simplified integration options, with Plug and Play solutions that can be easily incorporated into existing business processes without the need for specialized expertise
- **Cloud-based services:** Cloud-based AI services will enable small businesses to use state-of-the-art LLMs without the need for significant hardware investments, paying only for the services they use
- **User-friendly platforms:** The rise of user-friendly AI platforms, with intuitive interfaces and guided workflows, will allow smaller businesses to implement and manage LLMs without the need for in-house AI specialists
- **Pre-trained models:** Smaller businesses will have access to pre-trained models that can be fine-tuned for their specific needs, avoiding the costs and complexities of training models from scratch
- **Scalable performance:** LLMs will be scalable in performance, ensuring that small businesses can start with modest AI implementations and scale up as their needs grow
- **Tailored business applications:** AI developers will create tailored applications, designed for the unique challenges and opportunities of small businesses across various industries
- **Educational resources and support:** An increase in educational resources and community support will empower smaller businesses to make informed decisions about AI and how to implement LLMs effectively
- **Subscription models:** Subscription-based models will provide smaller businesses, with the flexibility to use advanced LLMs without upfront capital investments

- **Marketplaces for AI services:** Online marketplaces for AI services will emerge, where businesses can find and deploy LLMs suited to their specific tasks and industries
- **API economy:** The expansion of the API economy will give smaller businesses the ability to integrate LLMs into their operations through simple API calls
- **Regulatory support:** Regulations may evolve to support smaller businesses in adopting AI, possibly through incentives or frameworks that lower the entry barrier
- **Community-driven development:** Open source projects and community-driven AI development will provide small businesses with access to high-quality, collaboratively created LLMs

These advancements will not only make AI more affordable and accessible but also enable smaller businesses to compete more effectively with larger corporations, fostering innovation and growth across the entire business landscape.

## Enhanced interdisciplinary applications

LLMs are set to become even more versatile, with enhanced applications across a broad range of fields. The interdisciplinary use of LLMs will be driven by their ability to understand and generate domain-specific content, analyze complex data, and interact with users in contextually relevant ways. Here's what we can expect in various sectors:

- **Healthcare:** In healthcare, LLMs will be able to digest medical literature and patient data to assist with diagnosis, treatment planning, and patient education. They could help to parse complex medical records, providing summaries for healthcare providers, and even engage in patient monitoring by analyzing notes and reports for signs of change in a patient's condition.
- **Education:** LLMs will revolutionize education by offering personalized learning experiences, automating administrative tasks, and providing tutoring in a range of subjects. They could adapt to individual students' learning styles and progress, recommend resources, and assess student work.
- **Legal industry:** In the legal field, LLMs will assist in researching case law, drafting documents, and even predicting litigation outcomes. They could provide support in understanding complex legal language for both professionals and the public.
- **Creative industries:** For creative industries, LLMs will aid in content creation, from writing scripts to generating artistic concepts. They could also function as design assistants, proposing ideas and refining creative works based on user input.

- **Customer service:** Customer service will be bolstered by LLMs capable of conducting sophisticated conversations, handling inquiries, and resolving issues, with a level of personalization and understanding that mirrors human agents.
- **Financial services:** In financial services, LLMs will analyze market reports, financial statements, and economic data to provide insights, forecast trends, and personalize financial advice for clients.
- **Scientific research:** LLMs will assist researchers by sifting through vast amounts of scientific publications to identify relevant studies, generate hypotheses, or even draft research papers.
- **Engineering:** Engineers could use LLMs to interpret technical specifications, generate CAD drawings from descriptions, or simulate how changes in design could affect performance.
- **Supply chain and logistics:** LLMs will optimize supply chain operations by predicting disruptions, automating communications, and providing real-time analysis of logistics data.
- **Environmental sciences:** LLMs could process environmental data to model climate change effects, suggest conservation strategies, or generate reports on biodiversity.
- **Public sector:** Governments and public sector organizations will employ LLMs to enhance citizen services, draft policy, and analyze public feedback for better governance.
- **Language translation and localization:** LLMs will provide advanced translation services and localization, making content accessible across linguistic and cultural boundaries with a high degree of accuracy.
- **Psychology and mental health:** In mental health, LLMs could support therapy sessions by providing conversational agents that help with stress relief, or they could analyze patient language to support diagnosis and treatment.
- **Agriculture:** LLMs will aid in agricultural planning and management by analyzing reports and data, providing farmers with actionable insights.

As LLMs advance, their interdisciplinary applications will drive innovation and efficiency across sectors, transforming professional practices by processing and generating specialized knowledge.

The evolution of LLMs represents a convergence of technological progress, user-centric design, and ethical AI governance. As these models become more advanced, they will present both new opportunities and challenges that will shape the future of human-AI interaction.

# Getting ready for GPT-5 – infrastructure and skillsets

Preparing for the arrival of GPT-5, or any advanced iteration of LLMs, involves several key areas of focus for businesses and individuals looking to leverage the technology. Here's an outline of what preparation could involve:

- **Infrastructure readiness:**
  - **Cloud services:** Ensuring access to scalable cloud services that can support the high computational demands of GPT-5
  - **Data storage:** Upgrading data storage solutions to handle the increased volume of data processing and interactions
  - **Security measures:** Implementing robust cybersecurity measures to protect the data that GPT-5 will process and secure AI's output
  - **High-speed connectivity:** Investing in high-speed internet connections to facilitate real-time interactions with cloud-based GPT-5 services
  - **API integration:** Developing or updating APIs for the smooth integration of GPT-5 capabilities into existing systems and applications.
  - **Hardware accelerators:** Utilizing hardware accelerators, such as GPUs or TPUs, to locally run intensive machine learning tasks where latency is a critical factor
- **Skillset enhancement:**
  - **AI literacy:** Building AI literacy across an organization to ensure that all levels of staff understand the capabilities and limitations of GPT-5
  - **Technical training:** Providing technical training for IT teams on managing and maintaining AI systems, including GPT-5
  - **Data science skills:** Investing in data science skills, including understanding how to work with large datasets, model training, and fine-tuning
  - **AI ethics and governance:** Understanding the ethical considerations and governance required when deploying AI at scale, including bias mitigation and data privacy
  - **Change management:** Preparing a workforce for change management, ensuring that the introduction of GPT-5 enhances workflows without causing disruption
  - **Creative problem-solving:** Encouraging skills in creative problem-solving and design thinking to fully utilize the generative aspects of GPT-5

- **Advanced prompt engineering:** Developing expertise in fine-tuning GPT-5 to deliver more relevant, accurate, and context-aware responses, aligned with business needs
- **Interdisciplinary collaboration:** Fostering a culture of interdisciplinary collaboration, as GPT-5's applications will span across various departments and industries
- **Organizational strategies:**
  - **AI-first approach:** Adopting an AI-first approach in strategic planning, considering how GPT-5 can be used to achieve business goals
  - **Innovation labs:** Establishing innovation labs or task forces to explore and experiment with GPT-5 applications relevant to the business
  - **Partnerships:** Forming partnerships with AI research institutions and technology providers to stay at the forefront of AI developments
  - **Pilot projects:** Running pilot projects to understand the impact of GPT-5 and identify best practices for wider deployment
  - **Feedback mechanisms:** Creating feedback mechanisms to continuously learn from GPT-5 interactions and improve user experience and outcomes
  - **Prepare for higher compute requirements:** Plan for the increased computational demands of deploying GPT-5 by investing in scalable cloud solutions, high-performance computing, and efficient data storage to support large-scale AI workloads

Preparing for GPT-5 is not just about technology and skills; it's also about cultivating a forward-thinking culture that is ready to embrace the transformative potential of AI while addressing the challenges and responsibilities it brings.

## Potential breakthroughs and challenges ahead

As the field of AI and, more specifically, LLMs continues to evolve, we are likely to witness several breakthroughs along with significant challenges. Here's a look at what the future may hold:

- **Potential breakthroughs:**
  - **Advanced cognitive understanding:** LLMs may develop a deeper understanding of context, sarcasm, and nuanced language, effectively managing tasks that require a high level of cognitive abilities

- **Multimodal capabilities:** The ability to process and generate multimodal content, integrating text with images, audio, and video, could revolutionize how we interact with AI
- **Personalized AI interactions:** Breakthroughs in personalization could lead to LLMs that adapt to individual user's preferences, learning styles, and needs in real time
- **Generalized AI:** We could move toward more generalized AI that can perform a variety of tasks without extensive retraining or fine-tuning
- **Quantum computing integration:** Integration with quantum computing may dramatically increase the speed and capacity of LLMs, enabling them to solve complex problems previously deemed unsolvable
- **Language and culture translation:** LLMs could become powerful tools for breaking down language and cultural barriers, providing accurate and context-aware translations in real time
- **AI-assisted research and development:** In fields such as pharmaceuticals and environmental science, LLMs might significantly speed up research and development cycles
- **Ethical AI governance:** The establishment of robust ethical AI governance frameworks could ensure that LLMs are developed and used responsibly
- **Challenges ahead:**
  - **Bias and fairness:** Despite improvements, ensuring that LLMs are free from bias and treat all users fairly remains a major challenge
  - **Data privacy:** Balancing the data needs of LLMs with the privacy rights of individuals will continue to be a complex issue, especially with varying global regulations
  - **Explainability:** As LLMs become more complex, making their decision-making processes transparent and understandable to non-experts is a significant challenge
  - **Misinformation control:** Preventing LLMs from generating or propagating misinformation requires sophisticated understanding and filtering mechanisms
  - **Security threats:** The risk of malicious use of LLMs, such as creating deepfakes or automated hacking tools, is a concern for cybersecurity
  - **Resource intensity:** The environmental impact of training and running large-scale LLMs, which require significant computational resources, is a growing concern

- **Intellectual property questions:** The ability of LLMs to generate content raises complex questions about copyright and intellectual property rights
- **Dependency and skill erosion:** Over-reliance on LLMs could lead to the erosion of human skills in areas such as writing, analysis, and decision-making
- **Regulatory compliance:** As AI regulations are developed and become more stringent, ensuring that LLMs comply with these regulations is a challenge
- **Interdisciplinary integration:** Effectively integrating LLMs into interdisciplinary fields, each with its own complexities and nuances, requires extensive domain-specific expertise
- **AI ethics:** Developing AI in an ethical way, particularly as it becomes more autonomous and capable, is a profound challenge
- **Access and equity:** Ensuring equitable access to the benefits of LLMs across different socioeconomic, geographic, and cultural groups remains a challenge

The road ahead for LLMs is filled with both exciting possibilities and formidable obstacles. The true potential of LLMs will be realized through a collaborative effort, involving technologists, ethicists, policymakers, and the broader community, who address these challenges and guide the development of these powerful systems for the greater good.

## Strategic planning for future LLMs

Strategic planning to integrate and utilize future LLMs in businesses and organizations involves several key steps and considerations to ensure that these advanced tools are harnessed effectively and responsibly. Here's an outline of what such strategic planning might involve:

- **Assessment of organizational needs and goals:**
  - **Identify opportunities:** Determine where LLMs can solve existing problems or create new opportunities
  - **Set objectives:** Define clear objectives for what an organization aims to achieve with LLMs
- **Resource allocation:**
  - **Budgeting:** Allocate budget for infrastructure, training, and ongoing costs associated with LLM deployment
  - **Talent acquisition:** Invest in hiring or training personnel with the expertise to manage and work with LLMs

- **Infrastructure preparation:**
  - **Technology investment:** Upgrade existing infrastructure to support the computational demands of LLMs
  - **Data management:** Establish robust data management practices to feed accurate data to the LLMs
- **Risk management:**
  - **Ethical considerations:** Plan for the ethical implications of using LLMs, including biases and decision-making impacts
  - **Data privacy:** Ensure that the use of LLMs complies with data privacy laws and regulations
- **Compliance and legal check:**
  - **Regulatory review:** Stay abreast of AI regulations that could affect the use of LLMs
  - **Intellectual property:** Address intellectual property concerns related to generated content
- **Technology partnerships:**
  - **Collaborate with AI leaders:** Partner with tech firms and AI research institutions for access to the latest developments
  - **Ecosystem engagement:** Engage with the broader AI ecosystem, including start-ups and academic entities
- **Employee training and development:**
  - **Upskilling programs:** Implement training programs to upskill employees in AI literacy
  - **Change management:** Prepare your workforce for changes in workflow and processes due to AI integration
- **Pilot testing:**
  - **Proof of concept:** Start with a proof of concept to test the value and integration of LLMs within your organization
  - **Iterative approach:** Use an iterative approach to gradually expand the scope of LLM applications, based on initial learnings
- **Knowledge management:**
  - **Documentation:** Keep thorough documentation of how LLMs are used and the knowledge they generate
  - **Knowledge sharing:** Facilitate knowledge sharing about LLM capabilities and best practices within your organization

- **Monitoring and evaluation:**
  - **Performance metrics:** Establish metrics to evaluate the performance and impact of LLMs
  - **Feedback loops:** Create mechanisms to gather feedback from users and adjust strategies accordingly
- **Future-proofing:**
  - **Scalability:** Ensure that LLM solutions are scalable to grow with your organization
  - **Flexibility:** Maintain flexibility in AI strategies to adapt to the rapidly evolving AI landscape
- **Ethical AI framework:**
  - **Develop ethical frameworks:** Create guidelines to ensure that AI is used ethically throughout your organization
  - **Transparency:** Plan for transparent AI operations, where stakeholders understand how and why AI makes decisions
- **Long-term vision:**
  - **Strategic AI vision:** Develop a long-term vision for how LLMs can transform the organization
  - **Innovation culture:** Foster a culture of innovation that embraces AI as a tool for enhancement, not replacement

Strategic planning for LLMs is an ongoing process that must be revisited regularly as technology evolves. It requires a multidisciplinary approach, involving stakeholders from various departments, to ensure that LLMs are implemented in a way that aligns with an organization's values and objectives.

Chapter

# 35

# LLM Integration for Applications

This chapter kickstarts the hands-on portions, focusing on how we can **leverage large language models (LLMs)** to build powerful AI applications. In fact, LLMs have introduced a whole new paradigm in software development, paving the way for new families of applications that have the peculiarity of making the communication between the user and the machine smooth and conversational. Plus, those models enhanced existing applications, such as chatbots and recommendation systems, with their unique reasoning capabilities.

Developing LLM-powered applications is becoming a key factor for enterprises to keep themselves competitive in the market, and this leads to the spreading of new libraries and frameworks that make it easier to embed LLMs within applications. Some examples are Semantic Kernel, Haystack, LlamaIndex, and LangChain.

## A brief note about LangChain

Just as generative AI has evolved so rapidly over the last year, so has LangChain.

You can look at the architecture in greater detail at [https://python.langchain.com/docs/get\\_started/introduction](https://python.langchain.com/docs/get_started/introduction).

There are three packages you can install to start using LangChain:

- `langchain-core`: This contains the base abstractions and runtime for the whole LangChain ecosystem.
- `langchain-experimental`: This holds experimental LangChain code, intended for research and experimental uses.
- `langchain-community`: This contains all third-party integrations.

On top of that, there are three additional packages, yet can be leveraged to monitor and maintain your LangChain applications:

- `langserve`: LangServe is a tool that lets you deploy **LangChain runnables and chains** as a REST API, making it easier to integrate LangChain applications into production environments.

- **langsmith:** Think of LangSmith as an **innovative testing framework** for evaluating language models and AI applications. It helps visualize inputs and outputs at each step in the chain, aiding understanding and intuition during development.
- **langchain-cli:** The **official command-line interface** for LangChain, it facilitates interactions with LangChain projects, including template usage and quickstarts.

Last but not least, LangChain introduced the **LangChain Expression Language (LCEL)** to enhance the efficiency and flexibility of text processing tasks.

Key features of LCEL include:

- **Streaming asynchronous support:** This allows for the efficient handling of data streams.
- **Batch support:** This enables processing data in batches.
- **Parallel execution:** This enhances performance by executing tasks concurrently.
- **Retries and fallbacks:** This ensures robustness by handling failures gracefully.
- **Dynamically routing logic:** This allows logic flow based on input and output.
- **Message history:** This keeps track of interactions for context-aware processing.

We are not going to cover LCEL here; however, all the code samples can be converted into LCEL if you want to speed up your development and leverage its native integration with the end-to-end LangChain development stack.



#### Important note

Before we start working with LangChain, it is important to note that all packages are versioned slightly differently, yet all releases are cut with high frequency by a maintainer with a clearer communication strategy for breaking changes.

Starting from the next section, we are going to cover the backbone concepts – such as memory, VectorDB, and agents – that remain solid in the LangChain framework and, more generally, in the landscape of LLM development.

## Getting started with LangChain

As introduced earlier, LangChain is a lightweight framework meant to make it easier to integrate and orchestrate LLMs and their components within applications. It is mainly Python based, yet it recently extended its support to JavaScript and TypeScript.

In addition to LLM integration (which we will cover in an upcoming dedicated section), we saw that LangChain offers the following main components:

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

These components are illustrated in the following diagram:

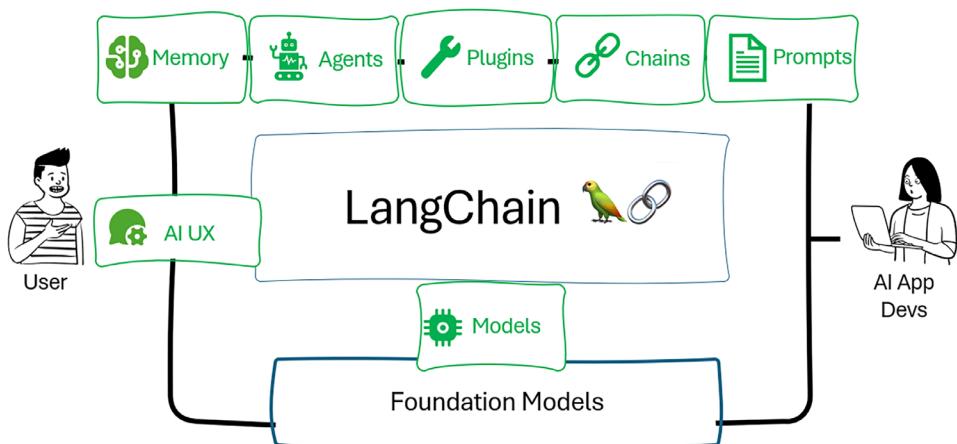


Figure 35.1: LangChain's components

The next sections will take a deep dive into each of these components.

## Models and prompts

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

Just to provide an example, let's see how easy it is to consume the OpenAI GPT-3 model (you can retrieve your OpenAI API key at <https://platform.openai.com/account/api-keys>):

```
from langchain.llms import OpenAI
llm = OpenAI(openai_api_key="your-api-key")
print(llm('tell me a joke'))
```

Here is the corresponding output:

```
Q: What did one plate say to the other plate?
A: Dinner's on me!
```

### Note



While running examples with LLMs, the output will vary at each run, due to the stochasticity of the models themselves. If you want to reduce the margin of variations in your output, you can make your model more “deterministic” by tuning the temperature hyperparameter. This parameter ranges from 0 (deterministic) to 1 (stochastic).

By default, the **OpenAI** module uses the `gpt-3.5-turbo-instruct` as a model. You can specify the model you want to use by passing the model’s name as a parameter.

As said previously, we will dive deeper into LLMs in the next section; so, for now, let’s focus on prompts. There are two main components related to LLM prompts and prompts design/engineering:

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

For example, suppose you want to use a language model to generate a translation from one language to another. You can use a prompt template like this:

```
Sentence: {sentence}
```

```
Translation in {language}:
```

{sentence} is a variable that will be replaced by the actual text. Translation in {language}: is a prefix that indicates the task and the expected output format.

You can easily implement this template as follows:

```
from langchain import PromptTemplate

template = """Sentence: {sentence}
Translation in {language}"""
```

```
prompt = PromptTemplate(template=template, input_
variables=[“sentence”, “language”])

print(prompt.format(sentence = “the cat is on the table”, language =
“spanish”))
```

Here is the output:

```
Sentence: the cat is on the table
Translation in spanish:
```

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

## Definition



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

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

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

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

```
{"prompt": “<prompt text>”, “completion”: “<ideal generated text>”}
```

LangChain offers the example selector class called `BaseExampleSelector` that you can import and modify as you wish. You can find the API reference at [https://python.langchain.com/docs/modules/model\\_io/prompts/example\\_selectors/](https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/).

## Data connections

Data connections refer to the building blocks needed to retrieve the additional non-parametric knowledge we want to provide the model with.

The idea is to cover the typical flow of incorporating user-specific data into applications that are made of five main blocks, as illustrated in the following figure:

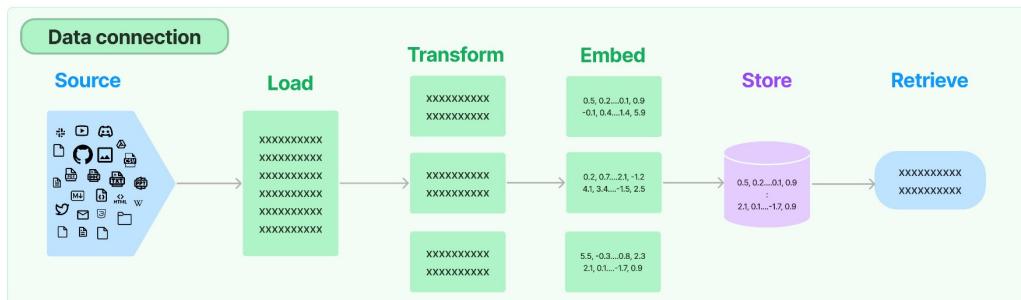


Figure 35.2: Incorporating user-specific knowledge into LLMs (source: [https://python.langchain.com/docs/modules/data\\_connection/](https://python.langchain.com/docs/modules/data_connection/))

Those blocks are addressed with the following LangChain tools:

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

For example, let's consider a sample CSV file to be loaded:

```

from langchain.document_loaders.csv_loader import CSVLoader

loader = CSVLoader(file_path='sample.csv')
data = loader.load()
print(data)

```

Here is the output:

```

[Document(page_content='Name: John\nAge: 25\nCity: New York',
metadata={'source': 'sample.csv', 'row': 0}), Document(page_
content='Name: Emily\nAge: 28\nCity: Los Angeles',
metadata={'source': 'sample.csv', 'row': 1}), Document(page_
content='Name: Michael\nAge: 22\nCity: Chicago', metadata={'source':
'sample.csv', 'row': 2})]

```

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

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

For example, let's split a document using the `RecursiveCharacterTextSplitter` module, which operates at a character level.

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

from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(

 chunk_size = 100, #number of characters for each chunk
 chunk_overlap = 20,#number of characters overlapping between a
preceding and following chunk
 length_function = len #function used to measure the number of
characters
)

texts = text_splitter.create_documents([mountain])
print(texts[0])
print(texts[1])
print(texts[2])
```

Here, `chunk_size` refers to the number of characters in each chunk while `chunk_overlap` represents the number of characters overlapping between successive chunks. Here is the output:

```
page_content="Amidst the serene landscape, towering mountains stand
as majestic guardians of nature's beauty." metadata={}
page_content='The crisp mountain air carries whispers of
tranquility, while the rustling leaves compose a' metadata={}
```

- **Text embedding models:** Earlier, we introduced the concept of embedding as a way to represent words, subwords, or characters in a continuous vector space.

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

To get started with embedding, you will need an embedding model.

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

For example, let's consider the embeddings using the **OpenAI** embedding model `text-embedding-ada-002` (for more details about OpenAI embedding models, you can refer to the official documentation at <https://platform.openai.com/docs/guides/embeddings/what-are-embeddings>):

```
from langchain.embeddings import OpenAIEMBEDDINGS

from dotenv import load_dotenv

load_dotenv()

os.environ["OPENAI_API_KEY"]

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

embeddings = embeddings_model.embed_documents(
 [
 "Good morning!",
 "Oh, hello!",
 "I want to report an accident",
 "Sorry to hear that. May I ask your name?",
 "Sure, Mario Rossi."
]
)

print("Embed documents:")
```

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

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

print("Embed query:")
print(f"Dimension of the vector: {len(embedded_query)}")
print(f"Sample of the first 5 elements of the vector: {embedded_
query[:5]}")
```

Here is the output:

```
Embed documents:
Number of vector: 5; Dimension of each vector: 1536
Embed query:
Dimension of the vector: 1536
Sample of the first 5 elements of the vector: [0.00538721214979887,
-0.0005941778072156012, 0.03892524912953377, -0.002979141427204013,
-0.008912666700780392]
```

Once we have both documents and the query embedded, the next step will be to compute the similarity between the two elements and retrieve the most suitable information from the document embedding. We will see the details of this when talking about vector stores.

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

## Definition



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

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

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

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

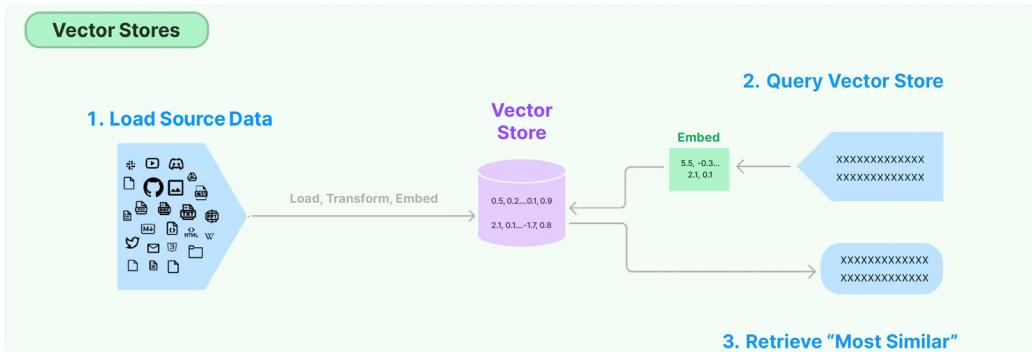


Figure 35.3: Sample architecture of a vector store (source: [https://python.langchain.com/docs/modules/data\\_connection/vectorstores/](https://python.langchain.com/docs/modules/data_connection/vectorstores/))

LangChain offers more than 40 integrations with third-party vector stores. Some examples are **Facebook AI Similarity Search (FAISS)**, Elasticsearch, MongoDB Atlas, and Azure Search. For an exhaustive list and descriptions of all the integrations, you can check the official documentation at <https://python.langchain.com/docs/integrations/vectorstores/>.

As an example, let's leverage the FAISS vector store, which has been developed by Meta AI research for efficient similarity search and clustering of dense vectors. We are going to leverage the same `dialogue.txt` file saved in the previous section:

```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEMBEDDINGS
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS

from dotenv import load_dotenv

load_dotenv()

os.environ["OPENAI_API_KEY"]

Load the document, split it into chunks, embed each chunk and Load
it into the vector store.
```

```
raw_documents = TextLoader('dialogue.txt').load()
text_splitter = CharacterTextSplitter(chunk_size=50, chunk_
overlap=0, separator = "\n",)
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEmbeddings())
```

Now that we've embedded and saved the non-parametric knowledge, let's also embed a user's query so that it can be used to search the most similar text chunk using cosine similarity as a measure:

```
query = "What is the reason for calling?"
docs = db.similarity_search(query)
print(docs[0].page_content)
```

The following is the output:

```
I want to report an accident
```

As you can see, the output is the piece of text that is more likely to contain the answer to the question. In an end-to-end scenario, it will be used as context to the LLM to generate a conversational response.

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

The difference between a retriever and a vector store is that a retriever is more general and flexible than a vector store. A retriever can use any method to find relevant documents, while a vector store relies on embeddings and similarity metrics. A retriever can also use different sources of documents, such as web pages, databases, or files, while a vector store needs to store the data itself.

However, a vector store can also be used as the backbone of a retriever if the data is embedded and indexed by a vector store. In that case, the retriever can use the vector store to perform a similarity search over the embedded data and return the most relevant documents. This is one of the main types of retrievers in LangChain, and it is called a vector store retriever.

For example, let's consider the FAISS vector store we previously initialized and "mount" a retriever on top of that:

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI

retriever = db.as_retriever()

qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff",
retriever=retriever)

query = "What was the reason of the call?"
qa.run(query)
```

Here is the output:

```
'The reason for the call was to report an accident.'
```

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

## Memory

In the context of LLM-powered applications, memory allows the application to keep references to user interactions, both in the short and long term. For example, let's consider the well-known ChatGPT. While interacting with the application, you have the possibility to ask follow-up questions referencing previous interactions without explicitly telling the model.

Plus, all conversations are saved into threads, so that, if you want to follow up on a previous conversation, you can re-open the thread without providing ChatGPT with all the contexts. This is made possible thanks to ChatGPT's ability to store users' interactions into a memory variable and use this memory as context while addressing follow-up questions.

LangChain offers several modules for designing your memory system within your applications, enabling it with both reading and writing skills.

The first step to do with your memory system is to actually store your human interactions somewhere. To do so, you can leverage numerous built-in memory integrations with third-party providers, including Redis, Cassandra, and Postgres.

Then, when it comes to defining how to query your memory system, there are various memory types you can leverage:

- **Conversation buffer memory:** This is the “plain vanilla” memory type available in LangChain. It allows you to store your chat messages and extract them in a variable.
- **Conversation buffer window memory:** It is identical to the previous one, with the only difference being allowing a sliding window over only  $K$  interactions so that you can manage longer chat history over time.
- **Entity memory:** Entity memory is a feature of LangChain that allows the language model to remember given facts about specific entities in a conversation. An entity is a person, place, thing, or concept that can be identified and distinguished from others. For example, in the sentence “Deven and Sam are working on a hackathon in Italy,” Deven and Sam are entities (person), as well as hackathon (thing) and Italy (place).

Entity memory works by extracting information on entities from the input text using an LLM. It then builds up its knowledge about that entity over time by storing the extracted facts in a memory store. The memory store can be accessed and updated by the language model whenever it needs to recall or learn new information about an entity.

- **Conversation knowledge graph memory:** This type of memory uses a knowledge graph to recreate memory.



### Definition

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

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

You can use this type of memory to save the input and output of each conversation turn as knowledge triplets (such as subject, predicate, and object) and then use them to generate relevant and consistent responses based on the current context. You can also query the knowledge graph to get the current entities or the history of the conversation.

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

LangChain provides specific modules for each of those memory types. Let's consider an example with the conversation summary memory, where we will also need an LLM to generate the summary of the interactions:

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

memory = ConversationSummaryMemory(llm=OpenAI(temperature=0))
memory.save_context({"input": "hi, I'm looking for some ideas to write an
essay in AI"}, {"output": "hello, what about writing on LLMs?"})

memory.load_memory_variables({})
```

Here is the output:

```
{'history': '\nThe human asked for ideas to write an essay in AI and the
AI suggested writing on LLMs.'}
```

As you can see, the memory summarized the conversation, leveraging the OpenAI LLM we initialized.

There is no recipe to define which memory to use within your applications; however, there are some scenarios that might be particularly suitable for specific memories. For example, a knowledge graph memory is useful for applications that need to access information from a large and diverse corpus of data and generate responses based on semantic relationships, while a conversation summary buffer memory could be suitable for creating conversational agents that can maintain a coherent and consistent context over multiple turns, while also being able to compress and summarize the previous dialogue history.

## Chains

Chains are predetermined sequences of actions and calls to LLMs that make it easier to build complex applications that require combining LLMs with each other or with other components.

LangChain offers four main types of chain to get started with:

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

### Definition



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

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

For example, let's retrieve the prompt template we built in the previous section:

```
from langchain import PromptTemplate

template = """Sentence: {sentence}
Translation in {language}:"""
prompt = PromptTemplate(template=template, input_variables=["sentence", "language"])
```

Now, let's put it into an LLMChain:

```
from langchain import OpenAI, LLMChain

llm = OpenAI(temperature=0)

llm_chain = LLMChain(prompt=prompt, llm=llm)

llm_chain.predict(sentence="the cat is on the table",
language="spanish")
```

Here is the output:

```
' El gato está en la mesa.'
```

- **RouterChain:** This is a type of chain that allows you to route the input variables to different chains based on some conditions. You can specify the conditions as functions or expressions that return a Boolean value. You can also specify the default chain to use if none of the conditions are met.

For example, you can use this chain to create a chatbot that can handle different types of requests, such as planning an itinerary or booking a restaurant reservation. To achieve this goal, you might want to differentiate two different prompts, depending on the type of query the user will make:

```
itinerary_template = """You are a vacation itinerary assistant. \
You help customers finding the best destinations and itinerary. \
You help customer screating an optimized itinerary based on their \
preferences.
```

Here is a question:

```
{input}"""
```

```
restaurant_template = """You are a restaurant booking assistant. \
You check with customers number of guests and food preferences. \
You pay attention whether there are special conditions to take into \
account.
```

Here is a question:

```
{input}"""
```

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

```
print(chain.run("I'm planning a trip from Milan to Venice by car.
What can I visit in between?"))
```

Here is the output:

```
> Entering new MultiPromptChain chain...
itinerary: {'input': "I'm planning a trip from Milan to Venice by
car. What attractions can I visit in between?"}
> Finished chain.
```

Answer:

There are many attractions that you can visit while traveling from Milan to Venice by car. Some of the most popular attractions include Lake Como, Verona, the Dolomites, and the picturesque towns of Bergamo and Brescia. You can also visit the stunning UNESCO World Heritage Sites in Mantua and Ferrara. Additionally, you can explore some of the local wineries and sample some of the wines of the region.

Here it is with a second query:

```
print(chain.run("I want to book a table for tonight"))
```

Here is the output:

```
> Entering new MultiPromptChain chain...
restaurant: {'input': 'I want to book a table for tonight'}
> Finished chain.
. How many people are in your party?
```

Hi there! How many people are in your party for tonight's reservation?

- **SequentialChain:** This is a type of chain that allows you to execute multiple chains in a sequence. You can specify the order of the chains and how they pass their outputs to the next chain. The simplest module of a sequential chain, takes by default the output of one chain as the input of the next chain.

However, you can also use a more complex module to have more flexibility to set input and output among chains.

As an example, let's consider an AI system that is meant to first generate a joke on a given topic, and then translate it in to another language. To do so, we will first create two chains:

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

llm = OpenAI(temperature=.7)
template = """You are a comedian. Generate a joke on the following
{topic}
Joke:"""
prompt_template = PromptTemplate(input_variables=[“topic”],
template=template)
joke_chain = LLMChain(llm=llm, prompt=prompt_template)

template = """You are translator. Given a text input, translate it
to {language}
Translation:"""
.prompt_template = PromptTemplate(input_variables=[“language”],
template=template)
translator_chain = LLMChain(llm=llm, prompt=prompt_template)
```

Now, let's combine them using the `SimpleSequentialChain` module:

```
This is the overall chain where we run these two chains in
sequence.
from langchain.chains import SimpleSequentialChain
overall_chain = SimpleSequentialChain(chains=[joke_chain,
translator_chain], verbose=True)
translated_joke = overall_chain.run("Cats and Dogs")
```

Here is the output:

```
> Entering new SimpleSequentialChain chain...
```

```
Why did the cat cross the road? To prove to the dog that it could be
done!
```

¿Por qué cruzó el gato la carretera? ¡Para demostrarle al perro que se podía hacer!

> Finished chain.

- **TransformationChain:** This is a type of chain that allows you to transform the input variables or the output of another chain using some functions or expressions. You can specify the transformation as a function that takes the input or output as an argument and returns a new value, as well as specify the output format of the chain.

For example, let's say we want to summarize a text, but before that, we want to rename one of the protagonists of the story (a cat) as "Silvester the Cat." As a sample text, I asked Bing Chat to generate a story about cats and dogs:

```
from langchain.chains import TransformChain, LLMChain,
SimpleSequentialChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
transform_chain = TransformChain(
 input_variables=[“text”], output_variables=[“output_text”],
 transform=rename_cat
)

template = “””Summarize this text:

{output_text}

Summary:”””
prompt = PromptTemplate(input_variables=[“output_text”],
template=template)
llm_chain = LLMChain(llm=OpenAI(), prompt=prompt)

sequential_chain = SimpleSequentialChain(chains=[transform_chain,
llm_chain])

sequential_chain.run(cats_and_dogs)
```

As you can see, we've combined a simple sequential chain with a transformation chain, where we set as a transformation function the `rename_cat` function (you can see the whole code in the GitHub repository).

The output is the following:

```
“ Silvester the Cat and a dog lived together but did not get along.
Silvester the Cat played a prank on the dog which made him angry.
When their owner found them fighting, she scolded them and made them
apologize. After that, they became friends and learned to respect
each other’s differences and appreciate each other’s strengths.”
```

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

## Agents

Agents are entities that drive decision-making within LLM-powered applications. They have access to a suite of tools and can decide which tool to call based on the user input and the context. Agents are dynamic and adaptive, meaning that they can change or adjust their actions based on the situation or the goal: in fact, while in a chain, the sequence of actions is hardcoded, in agents, the LLM is used as the reasoning engine with the goal of planning and executing the right actions in the right order.

A core concept while talking about agents is that of tools. In fact, an agent might be good at planning all the right actions to fulfill a user's query, but what if it cannot actually execute them, since it is missing information or executive power? For example, imagine I want to build an agent that is capable of answering my questions by searching the web. By itself, the agent has no access to the web, so I need to provide it with this tool. I will do so by using SerpAPI (the Google Search API) integration provided by LangChain (you can retrieve your API key at <https://serpapi.com/dashboard>).

Let's see it in Python:

```
from langchain import SerpAPIWrapper
from langchain.agents import AgentType, initialize_agent
from langchain.llms import OpenAI
from langchain.tools import BaseTool, StructuredTool, Tool, tool

import os
from dotenv import load_dotenv
```

```
load_dotenv()

os.environ["SERPAPI_API_KEY"]

search = SerpAPIWrapper()
tools = [Tool.from_function(
 func=search.run,
 name="Search",
 description="useful for when you need to answer questions about
current events"
)]

agent = initialize_agent(tools, llm = OpenAI(), agent=AgentType.ZERO_SHOT_
REACT_DESCRIPTION, verbose=True)

agent.run("When was Avatar 2 released?")
```

The following is the output:

```
> Entering new AgentExecutor chain...
I need to find out when Avatar 2 was released.
Action: Search
Action Input: "Avatar 2 release date"
Observation: December 16, 2022
Thought: I now know the final answer.
Final Answer: Avatar 2 was released on December 16, 2022.

> Finished chain.
'Avatar 2 was released on December 16, 2022.'
```

Note that, while initializing my agent, I set the agent type as ZERO\_SHOT\_REACT\_DESCRIPTION. This is one of the configurations we can pick and, specifically, it configures the agent to decide which tool to pick based solely on the tool's description with a ReAct approach:



## Definition

The ReAct approach is a way of using LLMs to solve various language reasoning and decision-making tasks. It was introduced in the paper *ReAct: Synergizing Reasoning and Acting in Language Models* by Shunyu Yao et al., back in October 2022. The ReAct approach prompts LLMs to generate both verbal reasoning traces and text actions in an interleaved manner, allowing for greater synergy between the two. Reasoning traces help the model to plan, track, and update its actions, as well as handle exceptions. Actions allow the model to interact with external sources, such as knowledge bases or environments, to gather additional information.

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

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

LangChain agents are pivotal whenever you want to let your LLMs interact with the external world. Plus, it is interesting to see how agents leverage LLMs not only to retrieve and generate responses, but also as reasoning engines to plan an optimized sequence of actions.

## Working with LLMs via the Hugging Face Hub

Now that we are familiar with LangChain components, it is time to start using our LLMs. If you want to use open-source LLMs, leveraging the Hugging Face Hub integration is extremely versatile. In fact, with just one access token you can leverage all the open-source LLMs available in Hugging Face's repositories.

As it is a non-production scenario, I will be using the free Inference API; however, if you are meant to build production-ready applications, you can easily scale to the Inference Endpoint, which grants you a dedicated and fully managed infrastructure to host and consume your LLMs.

So, let's see how to start integrating LangChain with the Hugging Face Hub.

### Create a Hugging Face user access token

To access the free Inference API, you will need a user access token, the credential that allows you to run the service. The following are the steps to activate the user access token:

1. **Create a Hugging Face account:** You can create a Hugging Face account for free at <https://huggingface.co/join>.
2. **Retrieve your user access token:** Once you have your account, go to the upper-right corner of your profile and go to **Settings | Access Tokens**. From that tab, you will be able to copy your secret token and use it to access Hugging Face models.

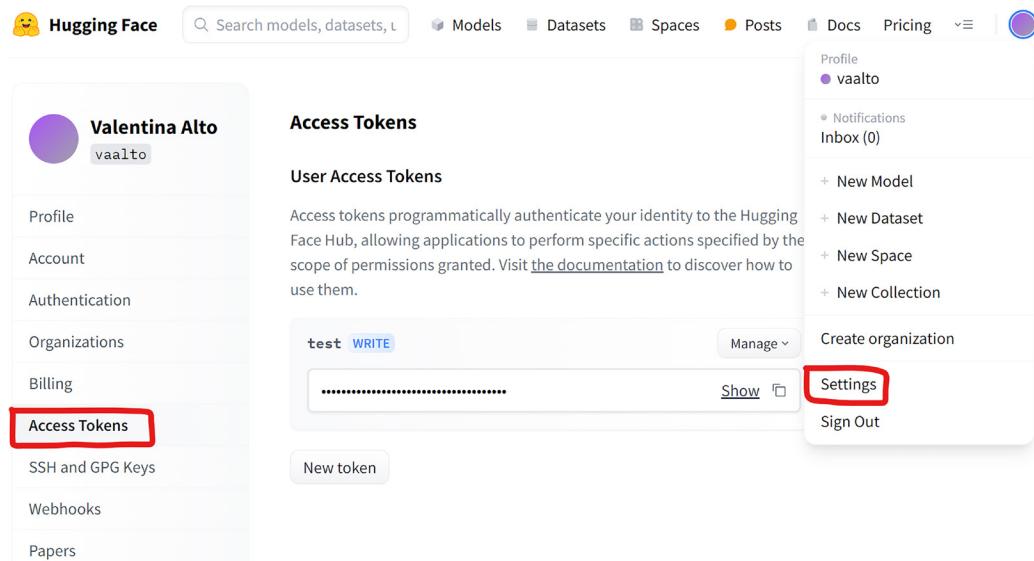


Figure 35.4: Retrieving access tokens from the Hugging Face account (source: <https://huggingface.co/settings/tokens>)

3. **Set permissions:** Access tokens enable users, applications, and notebooks to perform specific actions based on their assigned roles. There are two available roles:

- **Read:** This allows tokens to provide read access to repositories you have permission to read. This includes public and private repositories owned by you or your organization. This role is suitable for tasks like downloading private models or inference.
- **Write:** In addition to read access, tokens with this role grant write access to repositories where you have writing privileges. This token is useful for activities like training models or updating model cards.

In our series of use cases, we will keep a write permission on our token.

4. **Managing your user access token:** Within your profile, you can create and manage multiple access tokens, so that you can also differentiate permissions. To create a new token, you can click on the **New token** button:

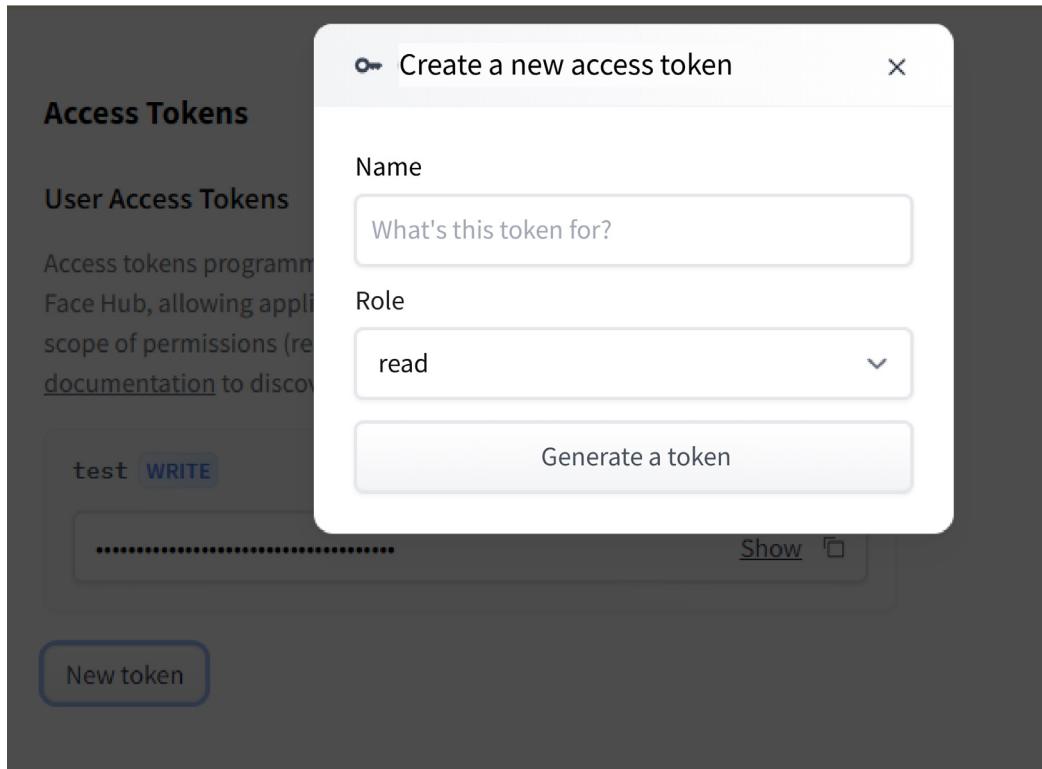


Figure 35.5: Creating a new token

- Finally, at any time, you can delete or refresh your token under the **Manage** button:

## Access Tokens

### User Access Tokens

Access tokens programmatically authenticate your identity to the Hugging Face Hub, allowing applications to perform specific actions specified by the scope of permissions (read, write, or admin) granted. Visit [the documentation](#) to discover how to use them.

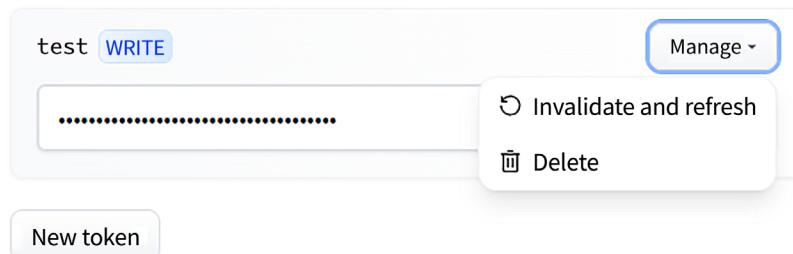


Figure 35.6: Managing tokens

It is important not to leak your token, and a good practice is to periodically regenerate it.

## Storing your secrets in an .env file

With our user access token generated in the previous section, we have the first secret to be managed.

### Definition



Secrets are data that needs to be protected from unauthorized access, such as passwords, tokens, keys, and credentials. Secrets are used to authenticate and authorize requests to API endpoints, as well as to encrypt and decrypt sensitive data.

Storing Python secrets in an `.env` file is a common practice to enhance security and maintainability in projects. To do this, create a file named `.env` in your project directory and list your sensitive information as key-value pairs: in our scenario, we will have `HUGGINGFACEHUB_API_TOKEN="your_user_access_token"`. This file should be added to your project's `.gitignore` to prevent accidental exposure.

To access these secrets in your Python code, use the `python-dotenv` library to load the `.env` file's values as environment variables. You can easily install it in your terminal via `pip install python-dotenv`.

This approach keeps sensitive data separate from your code base and helps ensure that confidential information remains confidential throughout the development and deployment processes.

Here, you can see an example of how to retrieve your access token and set it as an environmental variable:

```
import os
from dotenv import load_dotenv

load_dotenv()

os.environ["HUGGINGFACEHUB_API_TOKEN"]
```

Note that, by default, `load_dotenv` will look for the `.env` file in the current working directory; however, you can also specify the path to your secrets file:

```
from dotenv import load_dotenv
from pathlib import Path

dotenv_path = Path('path/to/.env')
load_dotenv(dotenv_path=dotenv_path)
```

Now that we have all the ingredients to start coding, it is time to try out some open-source LLMs.

## Start using open-source LLMs

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

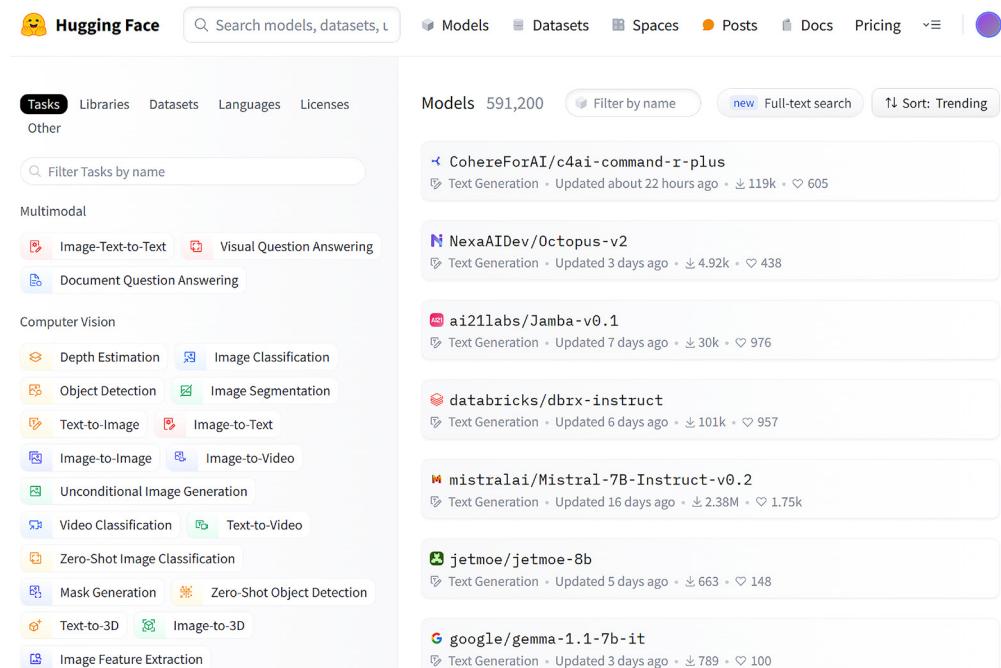


Figure 35.7: Home page of Hugging Face's model catalog

Since we are interested in LLMs, we will focus on the text generation category. For this first experiment, let's try Falcon LLM-7B:

```
from langchain import HuggingFaceHub

repo_id = "tiiuae/falcon-7b-instruct"
llm = HuggingFaceHub(
 repo_id=repo_id, model_kwargs={"temperature": 0.5, "max_length": 1000}
)
print(llm("what was the first disney movie?"))
```

Here is the corresponding output:

```
The first Disney movie was 'Snow White and the Seven Dwarfs'
```

As you can see, with just a few lines of code, we integrated an LLM from the Hugging Face Hub. With analogous code, you can test and consume all the LLMs available in the Hub.

Note that, here, we will be leveraging specific models for each application, both proprietary and open source. However, the idea is that you can use the model you prefer by simply initializing it as the main LLM and running the code as it is, simply changing the LangChain LLM integration. This is one of the main advantages of LLM-powered applications since you don't have to change the whole code to adapt to different LLMs.

Chapter

# 36

# Conversational AI with LLMs

## Getting started with conversational applications

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

Today, conversational applications are being taken to the next level thanks to LLMs. Let's look at some of the benefits that they provide:

- Not only do LLMs provide a new level of natural language interactions, but they can also enable applications to perform reasoning based on the best responses, given users' preferences.
- As we saw in previous chapters, LLMs can leverage their parametric knowledge, but are also enriched with non-parametric knowledge, thanks to embeddings and plug-ins.
- Finally, LLMs are also able to keep track of the conversation thanks to different types of memory.

The following image shows what the architecture of a conversational bot might look like:

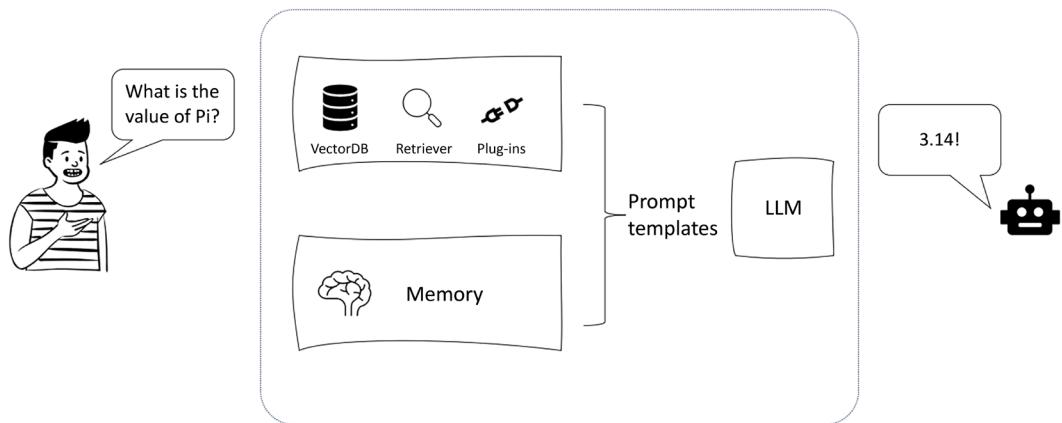


Figure 36.1: Sample architecture of a conversational bot

## Creating a plain vanilla bot

To start with, let's initialize our LLM and set the schema for our bot. The schema refers to the type of messages the bot is able to receive. In our case, we will have three types of messages:

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

Let's start with a simple configuration:

```
from langchain.schema import (
 AIMessage,
 HumanMessage,
 SystemMessage
)

from langchain.chains import LLMChain, ConversationChain
from langchain.chat_models import ChatOpenAI
```

```
from langchain.chat_models import ChatOpenAI

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

We can then save and print the output as follows:

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

Here is the output:

In Rome, there are many famous attractions to visit. Here's an optimized itinerary for your two-day trip:

Day 1:

1. Start your day by visiting the Colosseum, one of the world's most iconic ancient landmarks.
2. Next, explore the nearby Roman Forum, an ancient Roman marketplace.
3. Afterward, head to the Pantheon, a well-preserved Roman temple with a stunning dome.
4. Take a stroll through the historic district of Trastevere, known for its charming streets and authentic Roman atmosphere.
5. In the evening, visit the Trevi Fountain and toss a coin to ensure your return to Rome.

Day 2:

1. Begin your day at Vatican City, the smallest independent state in the world. Visit St. Peter's Basilica and admire Michelangelo's masterpiece, the Sistine Chapel.
2. Explore the Vatican Museums, home to an extensive collection of art and historical artifacts.
3. Enjoy a leisurely walk along the Tiber River and cross over to the picturesque neighborhood of Castel Sant'Angelo.
4. Visit the Spanish Steps, a popular meeting point with a beautiful view of the city.

5. End your day by exploring the charming neighborhood of Piazza Navona, known for its baroque architecture and lively atmosphere.

Remember to check the opening hours and availability of tickets for the attractions in advance. Enjoy your trip to Rome!

As you can see, the model was pretty good at generating an itinerary in Rome with only one piece of information from our side, the number of days.

However, we might want to keep interacting with the bot, so that we can further optimize the itinerary, providing more information about our preferences and habits. To achieve that, we need to add memory to our bot.

## Adding memory

As we're creating a conversational bot with relatively short messages, in this scenario, a `ConversationBufferMemory` could be suitable. To make the configuration easier, let's also initialize a `ConversationChain` to combine the LLM and the memory components.

Let's first initialize our memory and chain (I'm keeping `verbose = True` so that you can see the bot keeping track of previous messages):

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

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

Great, now let's have some interactions with our bot:

```
conversation.run("Hi there!")
```

The following is the output:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI
is talkative and provides lots of specific details from its context. If
the AI does not know the answer to a question, it truthfully says it does
not know.
```

```
Current conversation:
```

```
Human: Hi there!
```

```
AI:
```

```
> Finished chain.
```

```
'Hello! How can I assist you today?'
```

Next, we provide the following input:

```
conversation.run("what is the most iconic place in Rome?")
```

Here is the corresponding output:

```
> Entering new ConversationChain chain...
```

```
Prompt after formatting:
```

```
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.
```

```
Current conversation:
```

```
Human: Hi there!
```

```
AI: Hello! How can I assist you today?
```

```
Human: what is the most iconic place in Rome?
```

```
AI:
```

```
> Finished chain.
```

```
'The most iconic place in Rome is probably the Colosseum. It is a magnificent amphitheater that was built in the first century AD and is one of the most recognizable symbols of ancient Rome. The Colosseum was used for gladiatorial contests, public spectacles, and other events. Today, it is a major tourist attraction and a UNESCO World Heritage site.'
```

As you can see from the chain, it is keeping track of the previous interactions. Let's challenge it and ask something related to the previous context:

```
conversation.run("What kind of other events?")
```

The following is the output that we receive:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI
is talkative and provides lots of specific details from its context. If
the AI does not know the answer to a question, it truthfully says it does
not know.

Current conversation:
Human: Hi there!
AI: Hello! How can I assist you today?
Human: what is the most iconic place in Rome?
AI: The most iconic place in Rome is probably the Colosseum. It is a
magnificent amphitheater that was built in the first century AD and is one
of the most recognizable symbols of ancient Rome. The Colosseum was used
for gladiatorial contests, public spectacles, and other events. Today, it
is a major tourist attraction and a UNESCO World Heritage site.
Human: What kind of other events?
AI:

> Finished chain.
'Other events that took place at the Colosseum include mock sea battles,
animal hunts, and reenactments of famous battles. The Colosseum was also
used for executions and religious ceremonies. It was a versatile venue
that could accommodate a variety of events and entertainments.'
```

The bot was able to understand that our request was related to its previous answer. We can also retrieve the message history with the `memory.load_memory_variables()` method (you can see the full output in the GitHub repository). Here is a snippet of the output:

```
{'history': 'Human: Hi there!\nAI: Hello! How can I assist you today?\nHuman: what is the most iconic place in Rome?....'
```

Rather than running the `conversation.run` method at every interaction, I've coded a `while` cycle to make it interactive. The following is a snapshot of the whole conversation:

```
while True:
 query = input('you: ')
 if query == 'q':
 break
```

```
output = conversation({'input': query})
print('User: ', query)
print('AI system: ', output['response'])
```

The following is a truncated sample from the output:

```
User: hello
AI system: Hello! How can I assist you today?
User: I'm planning a 1-day trip in Venice. What should I visit?
AI system: That sounds like a wonderful plan! In Venice, there are
several must-visit attractions that you can explore in a day. Here's a
suggested itinerary for your 1-day trip in Venice:

1. St. Mark's Square (Piazza San Marco): [...] Enjoy your trip to Venice!

User: thanks! I'm planning to be around also the next day, and I love
hiking. Do you have any suggestion nearby?
AI system: Certainly! If you enjoy hiking and want to explore the natural
beauty around Venice, there are a few options you can consider for the
next day:

1. The Dolomites: [...]
User: which one is closer to Milan?
AI system: If you're looking for a hiking destination closer to Milan,
the best option would be the Lombardy region [...]
```

As you can see, now the AI assistant is capable of keeping track of the whole conversation. In the next section, we are going to add yet another layer of complexity: an external knowledge base.

## Adding non-parametric knowledge

Imagine that you also want your GlobeBotter to have access to exclusive documentation about itineraries that are not part of its parametric knowledge.

To do so, we can either embed the documentation in a VectorDB or directly use a retriever to do the job. In this case, we will use a vector-store-backed retriever using a particular chain, `ConversationalRetrievalChain`. This type of chain leverages a retriever over the provided knowledge base that has the chat history, which can be passed as a parameter using the desired type of memory previously seen.

With this goal in mind, we will use a sample Italy travel guide PDF downloaded from <https://www.minube.net/guides/italy>.

The following Python code shows how to initialize all the ingredients we need, which are:

- **Document Loader:** Since the document is in PDF format, we will use PyPDFLoader.
- **Text splitter:** We will use a RecursiveCharacterTextSplitter, which splits text by recursively looking at characters to find one that works.
- **Vector store:** We will use the FAISS VectorDB.
- **Memory:** We will use a ConversationBufferMemory.
- **LLMs:** We will use the gpt-3.5-turbo model for conversations.
- **Embeddings:** We will use the text-embedding-ada-002.

Let's take a look at the code:

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

text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1500,
 chunk_overlap=200
)

raw_documents = PyPDFLoader('italy_travel.pdf').load()
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEMBEDDINGS())
memory = ConversationBufferMemory()
```

```
 memory_key='chat_history',
 return_messages=True
)

llm = ChatOpenAI()
```

Let's now interact with the chain:

```
qa_chain = ConversationalRetrievalChain.from_llm(llm, retriever=db.as_
retriever(), memory=memory, verbose=True)
qa_chain.run({'question':'Give me some review about the Pantheon'})
```

The following is the output (I'm reporting a truncated version.):

```
> Entering new StuffDocumentsChain chain...

> Entering new LLMChain chain...
Prompt after formatting:
System: Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to
make up an answer.

cafes in the square. The most famous are the Quadri and
Florian.
Piazza San Marco,
Venice
4
Historical Monuments
Pantheon

Miskita:
“Angelic and non-human design,” was how
Michelangelo described the Pantheon 14 centuries after its
construction. The highlights are the gigantic dome, the upper
```

```
eye, the sheer size of the place, and the harmony of the
whole building. We visited with a Roman guide which is
...
> Finished chain.
'Miskita:\n"Angelic and non-human design," was how Michelangelo described
the Pantheon 14 centuries after its construction. The highlights
```

Note that, by default, the `ConversationalRetrievalChain` uses a prompt template called `CONDENSE_QUESTION_PROMPT`, which merges the last user's query with the chat history, so that it results as just one query to the retriever. If you want to pass a custom prompt, you can do so using the `condense_question_prompt` parameter in the `ConversationalRetrievalChain.from_llm` module.

Even though the bot was able to provide an answer based on the documentation, we still have a limitation. In fact, with such a configuration, our `GlobeBotter` will only look at the provided documentation, but what if we want it to also use its parametric knowledge? For example, we might want the bot to be able to understand whether it could integrate with the provided documentation or simply answer *freely*. To do so, we need to make our `GlobeBotter` *agentic*, meaning that we want to leverage the LLM's reasoning capabilities to orchestrate and invoke the available tools without a fixed order, but rather following the best approach given the user's query.

To do so, we will use two main components:

- `create_retriever_tool`: This method creates a custom tool that acts as a retriever for an agent. It will need a database to retrieve from, a name, and a short description, so that the model can understand when to use it.
- `create_conversational_retrieval_agent`: This method initializes a conversational agent that is configured to work with retrievers and chat models. It will need an LLM, a list of tools (in our case, the retriever), and a memory key to keep track of the previous chat history.

The following code illustrates how to initialize the agent:

```
from langchain.agents.agent_toolkits import create_retriever_tool

tool = create_retriever_tool(
 db.as_retriever(),
 "italy_travel",
 "Searches and returns documents regarding Italy."
)
```

```
tools = [tool]

memory = ConversationBufferMemory(
 memory_key='chat_history',
 return_messages=True
)

from langchain.agents.agent_toolkits import create_conversational_
retrieval_agent

from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(temperature = 0)

agent_executor = create_conversational_retrieval_agent(llm, tools, memory_
key='chat_history', verbose=True)
```

Great, now let's see the thought process of the agent with two different questions (I will report only the chain of thoughts and truncate the output, but you can find the whole code in the GitHub repo):

```
agent_executor({"input": "Tell me something about Pantheon"})
```

Here is the output:

```
> Entering new AgentExecutor chain...

Invoking: `italy_travel` with `Pantheon`

[Document(page_content='cafes in the square. The most famous are the
Quadri and\nFlorian. [...]

> Finished chain.
```

Let's now try with a question not related to the document:

```
output = agent_executor({"input": "what can I visit in India in 3 days?"})
```

The following is the output that we receive:

```
> Entering new AgentExecutor chain...
In India, there are numerous incredible places to visit, each with its
own unique attractions and cultural experiences. While three days is a
relatively short time to explore such a vast and diverse country, here are
a few suggestions for places you can visit:
1. Delhi: Start your trip in the capital city of India, Delhi. [...]
> Finished chain.
```

As you can see, when I asked the agent something about Italy, it immediately invoked the provided document, while this was not done in the last question.

The last thing we want to add to our GlobeBotter is the capability to navigate the web, since, as travelers, we want to have up-to-date information about the country we are traveling to. Let's implement it with LangChain's tools.

## Adding external tools

The tool we are going to add here is the Google SerpApi tool, so that our bot will be able to navigate the internet.



### Note

SerpApi is a real-time API designed to access Google search results. It simplifies the process of data scraping by handling complexities such as managing proxies, solving CAPTCHAs, and parsing structured data from search engine results pages. LangChain offers a pre-built tool that wraps SerpApi to make it easier to integrate it within your agents. To enable SerpApi, you need to sign in at [https://serpapi.com/users/sign\\_up](https://serpapi.com/users/sign_up), then go to the dashboard under the tab **API key**.

Since we don't want our GlobeBotter to be focused only on the web, we will add the SerpApi tool to the previous one, so that the agent will be able to pick the most useful tool to answer the question – or use no tool if not necessary.

Let's initialize our tools and agent:

```
from langchain import SerpAPIWrapper
import os
from dotenv import load_dotenv
```

```
load_dotenv()

os.environ["SERPAPI_API_KEY"]

search = SerpAPIWrapper()
tools = [
 Tool.from_function(
 func=search.run,
 name="Search",
 description="useful for when you need to answer questions about
current events"
),
 create_retriever_tool(
 db.as_retriever(),
 "italy_travel",
 "Searches and returns documents regarding Italy."
)
]

agent_executor = create_conversational_retrieval_agent(llm, tools, memory_
key='chat_history', verbose=True)
```

Great, now let's test it with three different questions (here, again, the output has been truncated):

- “What can I visit in India in 3 days?”

```
> Entering new AgentExecutor chain...
India is a vast and diverse country with numerous attractions to
explore. While it may be challenging to cover all the highlights in
just three days, here are some popular destinations that you can
consider visiting:
```

```
1. Delhi: Start your trip in the capital city of India, Delhi. [...]
```

```
> Finished chain.
```

In this case, the model doesn't need external knowledge to answer the question, hence it is responding without invoking any tool.

- “What is the weather currently in Delhi?”

```
> Entering new AgentExecutor chain...
```

```
Invoking: `Search` with `{'query': 'current weather in Delhi'}`
```

Current Weather • 95°F Mostly sunny • RealFeel® 105°. Very Hot. RealFeel Guide. Very Hot. 101° to 107°. Caution advised. Danger of dehydration, heat stroke, heat ...The current weather in Delhi is 95°F (35°C) with mostly sunny conditions. The RealFeel® temperature is 105°F (41°C), indicating that it feels very hot. Caution is advised as there is a danger of dehydration, heat stroke, and heat-related issues. It is important to stay hydrated and take necessary precautions if you are in Delhi or planning to visit.

```
> Finished chain.
```

Note how the agent is invoking the search tool; this is due to the reasoning capability of the underlying gpt-3.5-turbo model, which captures the user’s intent and dynamically understands which tool to use to accomplish the request.

- “I’m traveling to Italy. Can you give me some suggestions for the main attractions to visit?”

```
> Entering new AgentExecutor chain...
```

```
Invoking: `italy_travel` with `{'query': 'main attractions in Italy'}`
```

[Document(page\_content='ITALY\nMINUBE TRAVEL GUIDE\nThe best must-see places for your travels, [...]']

Here are some suggestions for main attractions in Italy:

1. Parco Sempione, Milan: This is one of the most important parks in Milan. It offers a green space in the city where you can relax, workout, or take a leisurely walk. [...]

```
> Finished chain.
```

Note how the agent is invoking the document retriever to provide the preceding output.

Overall, our GlobeBotter is now able to provide up-to-date information, as well as retrieving specific knowledge from curated documentation. The next step will be that of building a front-end. We will do so by building a web app using Streamlit.

## Developing the front-end with Streamlit

Streamlit is a Python library that allows you to create and share web apps. It is designed to be easy and fast to use, without requiring any front-end experience or knowledge. You can write your app in pure Python, using simple commands to add widgets, charts, tables, and other elements.

In addition to its native capabilities, in July 2023, Streamlit announced an initial integration and its future plans with LangChain. At the core of this initial integration, there is the ambition of making it easier to build a GUI for conversational applications, as well as showing all the steps LangChain's agents take before producing the final response.

To achieve this goal, the main module that Streamlit introduced is the Streamlit callback handler. This module provides a class called `StreamlitCallbackHandler` that implements the `BaseCallbackHandler` interface from LangChain. This class can handle various events that occur during the execution of a LangChain pipeline, such as tool start, tool end, tool error, LLM token, agent action, agent finish, etc.

The class can also create and update Streamlit elements, such as containers, expanders, text, progress bars, etc., to display the output of the pipeline in a user-friendly way. You can use the Streamlit callback handler to create Streamlit apps that showcase the capabilities of LangChain and interact with the user through natural language. For example, you can create an app that takes a user prompt and runs it through an agent that uses different tools and models to generate a response. You can use the Streamlit callback handler to show the agent's thought process and the results of each tool in real time.

To start building your application, you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `globebotter.py`.

The following are the main building blocks of the application:

1. Setting the configuration of the webpage:

```
import streamlit as st
```

- st.set\_page\_config(page\_title="GlobeBotter", page\_icon="🌐")
  - st.header('🌐 Welcome to Globebotter, your travel assistant with Internet access. What are you planning for your next trip?')
2. Initializing the LangChain backbone components we need. The code is the same as the one in the previous section, so I will share here only the initialization code, without all the preliminary steps:

```
search = SerpAPIWrapper()
text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1500,
 chunk_overlap=200
)

raw_documents = PyPDFLoader('italy_travel.pdf').load()
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEMBEDDINGS())

memory = ConversationBufferMemory(
 return_messages=True,
 memory_key="chat_history",
 output_key="output"
)

llm = ChatOpenAI()
tools = [
 Tool.from_function(
 func=search.run,
 name="Search",
 description="useful for when you need to answer questions about current events"
),
 create_retriever_tool(
 db.as_retriever(),
 "italy_travel",
 "Searches and returns documents regarding Italy."
)
]
```

```
agent = create_conversational_retrieval_agent(llm, tools, memory_
key='chat_history', verbose=True)
```

3. Setting the input box for the user with a placeholder question:

```
user_query = st.text_input(
 "***Where are you planning your next vacation?***",
 placeholder="Ask me anything!"
)
```

4. Setting Streamlit's session states. Session state is a way to share variables between reruns, for each user session. In addition to the ability to store and persist state, Streamlit also exposes the ability to manipulate state using callbacks. Session state also persists across apps inside a multipage app. You can use the session state API to initialize, read, update, and delete variables in the session state. In the case of our GlobeBotter, we want two main states: `messages` and `memory`:

```
if "messages" not in st.session_state:
 st.session_state["messages"] = [{"role": "assistant", "content": "How can I help you?"}]
if "memory" not in st.session_state:
 st.session_state['memory'] = memory
```

5. Making sure to display the whole conversation. To do so, I created a for loop that iterates over the list of messages stored in `st.session_state["messages"]`. For each message, it creates a Streamlit element called `st.chat_message` that displays a chat message in a nice format:

```
for msg in st.session_state["messages"]:
 st.chat_message(msg["role"]).write(msg["content"])
```

6. Configuring the AI assistant to respond when given a user's query. In this first example, we will keep the whole chain visible and printed to the screen:

```
if user_query:
 st.session_state.messages.append({"role": "user", "content": user_query})
 st.chat_message("user").write(user_query)
 with st.chat_message("assistant"):
 st_cb = StreamlitCallbackHandler(st.container())
 response = agent(user_query, callbacks=[st_cb])
```

```
 st.session_state.messages.append({"role": "assistant",
"content": response})
 st.write(response)
```

- Finally, adding a button to clear the history of the conversation and start from scratch:

```
if st.sidebar.button("Reset chat history"):
 st.session_state.messages = []
```

The final product looks as follows:

 **Welcome to Globebotter, your travel assistant with Internet access. What are you planning for your next trip?**

Where are you planning your next vacation?

What is the temperature today in Rome?

 How can I help you?

 What is the temperature today in Rome?



Search: temperature today in Rome



```
▼ {
 "input" : "What is the temperature today in Rome?"
 ▶ "chat_history" : [...]
 "output" : "The current temperature in Rome is 97 °F. It is sunny."
 ▶ "intermediate_steps" : [...]
}
```

Figure 36.2: Front-end of GlobeBotter with Streamlit

From the expander, we can see that the agent used the Search tool (provided with the SerpApi). We can also expand `chat_history` or `intermediate_steps` as follows:

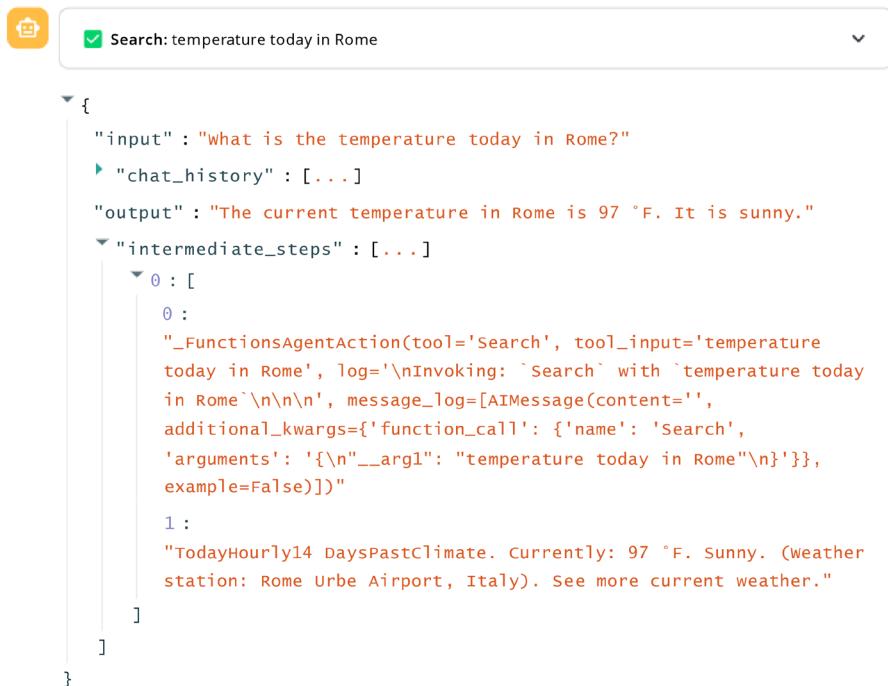


Figure 36.3: Example of Streamlit expander

Of course, we can also decide to only show the output rather than the whole chain of thoughts, by specifying in the code to return only `response['output']`.

Before we wrap up, let's discuss how you can give your users a streaming experience while interacting with your chatbot. You can leverage the `BaseCallbackHandler` class to create a custom callback handler in your Streamlit app:

```
from langchain.callbacks.base import BaseCallbackHandler
from langchain.schema import ChatMessage
from langchain_openai import ChatOpenAI
import streamlit as st
class StreamHandler(BaseCallbackHandler):
```

```
def __init__(self, container, initial_text=""):
 self.container = container
 self.text = initial_text

def on_llm_new_token(self, token: str, **kwargs) -> None:
 self.text += token
 self.container.markdown(self.text)
```

The `StreamHandler` is designed to capture and display streaming data, such as text or other content, in a designated container. Then, you can use it as follows in your Streamlit app, making sure to set `streaming=True` while initializing your OpenAI LLM.

```
with st.chat_message("assistant"):
 stream_handler = StreamHandler(st.empty())
 llm = ChatOpenAI(streaming=True, callbacks=[stream_handler])
 response = llm.invoke(st.session_state.messages)
 st.session_state.messages.append(ChatMessage(role="assistant",
 content=response.content))
```

You can refer to the original code on LangChain's GitHub repo at [https://github.com/langchain-ai/streamlit-agent/blob/main/streamlit\\_agent/basic\\_streaming.py](https://github.com/langchain-ai/streamlit-agent/blob/main/streamlit_agent/basic_streaming.py).

# **MODULE 8**

## **MASTERING RETRIEVAL-AUGMENTED GENERATION**

---

**CHAPTER 37:** Retrieval-Augmented Generation (RAG)

**CHAPTER 38:** Developing a Complete RAG Pipeline

**CHAPTER 39:** RAG in Practice: Use Cases and Solutions

**CHAPTER 40:** RAG Architecture

**CHAPTER 41:** Integrating RAG with AI Agents and LangGraph

Chapter

# 37

# Retrieval-Augmented Generation (RAG)

The field of **artificial intelligence (AI)** is rapidly evolving. At the center of it all is **generative AI**. At the center of generative AI is **retrieval-augmented generation (RAG)**. RAG is emerging as a significant addition to the generative AI toolkit, harnessing the intelligence and text generation capabilities of **large language models (LLMs)** and integrating them with a company's internal data. This offers a method to enhance organizational operations significantly.

## Understanding RAG – Basics and principles

Modern-day LLMs are impressive, but they have never seen your company's private data (hopefully!). This means the ability of an LLM to help your company fully utilize its data is very limited. This very large barrier has given rise to the concept of RAG, where you are using the power and capabilities of the LLM but combining it with the knowledge and data contained within your company's internal data repositories. This is the primary motivation for using RAG: to make new data available to the LLM and significantly increase the value you can extract from that data.

Beyond internal data, RAG is also useful in cases where the LLM has not been trained on the data, even if it is public, such as the most recent research papers or articles about a topic that is strategic to your company. In both cases, we are talking about data that was not present during the training of the LLM. You can have the latest LLM trained on the most tokens ever, but if that data was not present for training, then the LLM will be at a disadvantage in helping you reach your full productivity.

Ultimately, this highlights the fact that, for most organizations, it is a central need to connect new data to an LLM. RAG is the most popular paradigm for doing this.

## Advantages of RAG

Some of the potential advantages of using RAG include improved accuracy and relevance, customization, flexibility, and expanding the model's knowledge beyond the training data. Let's take a closer look:

- **Improved accuracy and relevance:** RAG can significantly enhance the accuracy and relevance of responses that are generated by LLMs. RAG fetches and incorporates specific information from a database or dataset, typically in real time, and ensures that the output is based on both the model's pre-existing knowledge and the most current and relevant data that you are providing directly.
- **Customization:** RAG allows you to customize and adapt the model's knowledge to your specific domain or use case. By pointing RAG to databases or datasets directly relevant to your application, you can tailor the model's outputs so that they align closely with the information and style that matters most for your specific needs. This customization enables the model to provide more targeted and useful responses.
- **Flexibility:** RAG provides flexibility in terms of the data sources that the model can access. You can apply RAG to various structured and unstructured data, including databases, web pages, documents, and more. This flexibility allows you to leverage diverse information sources and combine them in novel ways to enhance the model's capabilities. Additionally, you can update or swap out the data sources as needed, enabling the model to adapt to changing information landscapes.
- **Expanding model knowledge beyond training data:** LLMs are limited by the scope of their training data. RAG overcomes this limitation by enabling models to access and utilize information that was not included in their initial training sets. This effectively expands the knowledge base of the model without the need for retraining, making LLMs more versatile and adaptable to new domains or rapidly evolving topics.
- **Removing hallucinations:** The LLM is a key component within the RAG system. LLMs have the potential to provide wrong information, also known as hallucinations. These hallucinations can manifest in several ways, such as made-up facts, incorrect facts, or even nonsensical verbiage. Often, the hallucination is worded in a way that can be very convincing, causing it to be difficult to identify. A well-designed RAG application can remove hallucinations much more easily than when directly using an LLM.

## Challenges of RAG

There are some challenges to using RAG as well, which include dependency on the quality of the internal data, the need for data manipulation and cleaning, computational overhead, more complex integrations, and the potential for information overload. Let's review these challenges and gain a better understanding of how they impact RAG pipelines and what can be done about them:

- **Dependency on data quality:** When talking about how data can impact an AI model, the saying in data science circles is *garbage in, garbage out*. This means that if you give a model bad data, it will give you bad results. RAG is no different. The effectiveness of RAG is directly tied to the quality of the data it retrieves. If the underlying database or dataset contains outdated, biased, or inaccurate information, the outputs generated by RAG will likely suffer from the same issues.
- **Need for data manipulation and cleaning:** Data in the recesses of the company often has a lot of value to it, but it is not often in good, accessible shape. For example, data from PDF-based customer statements needs a lot of massaging so that it can be put into a format that can be useful to a RAG pipeline.
- **Computational overhead:** A RAG pipeline introduces a host of new computational steps into the response generation process, including data retrieval, processing, and integration. LLMs are getting faster every day, but even the fastest response can be more than a second, and some can take several seconds. If you combine that with other data processing steps, and possibly multiple LLM calls, the result can be a very significant increase in the time it takes to receive a response. This all leads to increased computational overhead, affecting the efficiency and scalability of the entire system. As with any other IT initiative, an organization must balance the benefits of enhanced accuracy and customization against the resource requirements and potential latency introduced by these additional processes.
- **Data storage explosion; complexity in integration and maintenance:** Traditionally, your data resides in a data source that's queried in various ways to be made available to your internal and external systems. But with RAG, your data resides in multiple forms and locations, such as vectors in a vector database, that represent the same data, but in a different format. Add in the complexity of connecting these various data sources to LLMs and relevant technical mechanisms such as vector searches and you have a significant increase in complexity. This increased complexity can be resource-intensive. Maintaining this integration over time, especially as data sources evolve or expand, adds even more complexity and cost. Organizations need to invest in technical expertise and infrastructure to leverage RAG capabilities effectively

while accounting for the rapid increase in complexities these systems bring with them.

- **Potential for information overload:** RAG-based systems can pull in too much information. It is just as important to implement mechanisms to address this issue as it is to handle times when not enough relevant information is found. Determining the relevance and importance of retrieved information to be included in the final output requires sophisticated filtering and ranking mechanisms. Without these, the quality of the generated content could be compromised by an excess of unnecessary or marginally relevant details.
- **Hallucinations:** While we listed removing hallucinations as an advantage of using RAG, hallucinations do pose one of the biggest challenges to RAG pipelines if they're not dealt with properly. A well-designed RAG application must take measures to identify and remove hallucinations and undergo significant testing before the final output text is provided to the end user.
- **High levels of complexity within RAG components:** A typical RAG application tends to have a high level of complexity, with many components that need to be optimized for the overall application to function properly. The components can interact with each other in several ways, often with many more steps than the basic RAG pipeline you start with. Every component within the pipeline needs significant amounts of trials and testing, including your prompt design and engineering, the LLMs you use and how you use them, the various algorithms and their parameters for retrieval, the interface you use to access your RAG application, and numerous other aspects that you will need to add over the course of your development.

## RAG vocabulary Check the sentence

Now is as good a time as any to review some vocabulary that should help you become familiar with the various concepts in RAG. In the following subsections, we will familiarize ourselves with some of this vocabulary, including LLMs, prompting concepts, inference, context windows, fine-tuning approaches, vector databases, and vectors/embeddings. This is not an exhaustive list, but understanding these core concepts should help you understand everything else we will teach you about RAG in a more effective way.

### LLM

LLMs are generative AI technologies that focus on generating text. We will keep things simple by concentrating on the type of model that most RAG pipelines use, the LLM.

---

However, we would like to clarify that while we will focus primarily on LLMs, RAG can also be applied to other types of generative models, such as those for images, audio, and videos.

Some popular examples of LLMs are the OpenAI ChatGPT models, the Meta Llama models, Google's Gemini models, and Anthropic's Claude models.

## Prompting, prompt design, and prompt engineering

These terms are sometimes used interchangeably, but technically, while they all have to do with prompting, they do have different meanings:

- **Prompting** is the act of sending a query or *prompt* to an LLM.
- **Prompt design** refers to the strategy you implement to *design* the prompt you will send to the LLM. Many different prompt design strategies work in different scenarios.
- **Prompt engineering** focuses more on the technical aspects surrounding the prompt that you use to improve the outputs from the LLM. For example, you may break up a complex query into two or three different LLM interactions, *engineering* it better to achieve superior results.

## LangChain and LlamaIndex

LangChain is an open source framework that supports not just RAG but any development that wants to use LLMs within a pipeline approach. With over 15 million monthly downloads, LangChain is the most popular generative AI development framework. It supports RAG particularly well, providing a modular and flexible set of tools that make RAG development significantly more efficient than not using a framework.

While LangChain is currently the most popular framework for developing RAG pipelines, LlamaIndex is a leading alternative to LangChain, with similar capabilities in general. LlamaIndex is known for its focus on search and retrieval tasks and may be a good option if you require advanced search or need to handle large datasets.

Many other options focus on various niches. Once you have gotten familiar with building RAG pipelines, be sure to look at some of the other options to see if there are frameworks that work for your particular project better.

## Inference

We will use the term **inference** from time to time. Generally, this refers to the process of the LLM generating outputs or predictions based on given inputs using a pre-trained language model. For example, when you ask ChatGPT a question, the steps it takes to provide you with a response is called inference.

## Context window

A context window, in the context of LLMs, refers to the maximum number of tokens (words, sub-words, or characters) that the model can process in a single pass. It determines the amount of text the model can *see* or *attend to* at once when making predictions or generating responses.

The context window size is a key parameter of the model architecture and is typically fixed during model training. It directly relates to the input size of the model as it sets an upper limit on the number of tokens that can be fed into the model at a time.

For example, if a model has a context window size of 4,096 tokens, it means that the model can process and generate sequences of up to 4,096 tokens. When processing longer texts, such as documents or conversations, the input needs to be divided into smaller segments that fit within the context window. This is often done using techniques such as sliding windows or truncation.

The size of the context window has implications for the model's ability to understand and maintain long-range dependencies and context. Models with larger context windows can capture and utilize more contextual information when generating responses, which can lead to more coherent and contextually relevant outputs. However, increasing the context window size also increases the computational resources required to train and run the model.

In the context of RAG, the context window size is essential because it determines how much information from the retrieved documents can be effectively utilized by the model when generating the final response. Recent advancements in language models have led to the development of models with significantly larger context windows, enabling them to process and retain more information from the retrieved sources. See *Table 37.1* to see the context windows of many popular LLMs, both closed and open sourced:

LLM	Context Window (Tokens)
ChatGPT-3.5 Turbo 0613 (OpenAI)	4,096
Llama 2 (Meta)	4,096
Llama 3 (Meta)	8,000
ChatGPT-4 (OpenAI)	8,192
ChatGPT-3.5 Turbo 0125 (OpenAI)	16,385
ChatGPT-4.0-32k (OpenAI)	32,000
Mistral (Mistral AI)	32,000
Mixtral (Mistral AI)	32,000
DBRX (Databricks)	32,000
Gemini 1.0 Pro (Google)	32,000
ChatGPT-4.0 Turbo (OpenAI)	128,000
ChatGPT-4o (OpenAI)	128,000
Claude 2.1 (Anthropic)	200,000
Claude 3 (Anthropic)	200,000
Gemini 1.5 Pro (Google)	1,000,000

Table 37.1 – Different context windows for LLMs

Figure 37.1, which is based on Table 37.1, shows that Gemini 1.5 Pro is far larger than the others.

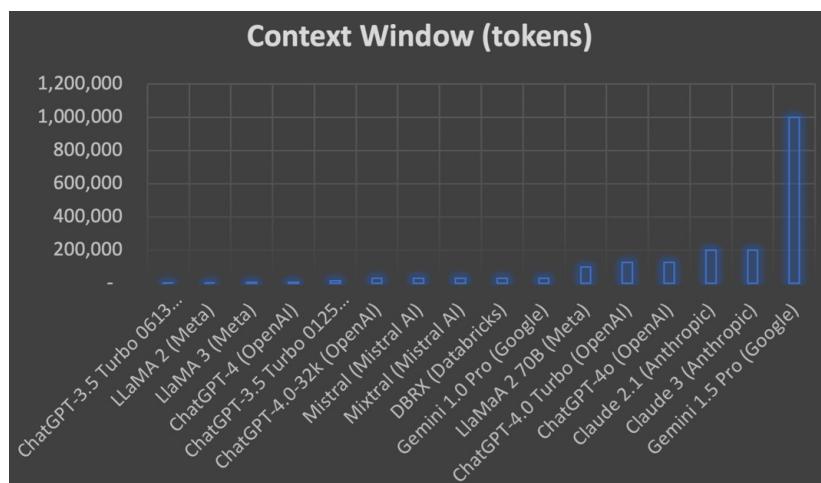


Figure 37.1 – Different context windows for LLMs

Note that *Figure 37.1* shows models that have generally aged from right to left, meaning the older models tended to have smaller context windows, with the newest models having larger context windows. This trend is likely to continue, pushing the typical context window larger as time progresses.

## Fine-tuning – full-model fine-tuning (FMFT) and parameter-efficient fine-tuning (PEFT)

FMFT is where you take a foundation model and train it further to gain new capabilities. You could simply give it new knowledge for a specific domain, or you could give it a skill, such as being a conversational chatbot. FMFT updates all the parameters and biases in the model.

PEFT, on the other hand, is a type of fine-tuning where you focus only on specific parts of the parameters or biases when you fine-tune the model, but with a similar goal as general fine-tuning. The latest research in this area shows that you can achieve similar results to FMFT with far less cost, time commitment, and data.

It is a very valid strategy to try to use a model fine-tuned with your data to give it more knowledge from your domain or to give it more of a *voice* from your domain. For example, you could train it to talk more like a scientist than a generic foundation model, if you're using this in a scientific field. Alternatively, if you are developing in a legal field, you may want it to sound more like a lawyer.

Fine-tuning also helps the LLM to understand your company's data better, making it better at generating an effective response during the RAG process. For example, if you have a scientific company, you might fine-tune a model with scientific information and use it for a RAG application that summarizes your research. This may improve your RAG application's output (the summaries of your research) because your fine-tuned model understands your data better and can provide a more effective summary.

## Vector store or vector database?

Both! All vector databases are vector stores, but not all vector stores are vector databases. OK, while you get out your chalkboard to draw a Venn diagram, I will continue to explain this statement.

There are ways to store vectors that are not full databases. They are simply storage devices for vectors. So, to encompass all possible ways to store vectors, LangChain calls them all **vector stores**. Let's do the same! Just know that not all the *vector stores* that LangChain connects with are officially considered vector databases, but in general, most of them are and many people refer to all of them as vector databases, even when

they are not technically full databases from a functionality standpoint. Phew – glad we cleared that up!

## Vectors, vectors, vectors!

A vector is a mathematical representation of your data. They are often referred to as embeddings when talking specifically about **natural language processing (NLP)** and LLMs. Vectors are one of the most important concepts to understand and there are many different parts of a RAG pipeline that utilize vectors.

## Vectors

It could be argued that understanding vectors and all the ways they are used in RAG is the most important part of this entire book. As mentioned previously, vectors are simply the mathematical representations of your external data, and they are often referred to as embeddings. These representations capture semantic information in a format that can be processed by algorithms, facilitating tasks such as similarity search, which is a crucial step in the RAG process.

Vectors typically have a specific dimension based on how many numbers are represented by them. For example, this is a four-dimensional vector:

```
[0.123, 0.321, 0.312, 0.231]
```

If you didn't know we were talking about vectors and you saw this in Python code, you might recognize this as a list of four floating points, and you aren't too far off. However, when working with vectors in Python, you want to recognize them as a NumPy array, rather than lists. NumPy arrays are generally more machine-learning-friendly because they are optimized to be processed much faster and more efficiently than Python lists, and they are more broadly recognized as the de facto representation of embeddings across machine learning packages such as SciPy, pandas, scikit-learn, TensorFlow, Keras, Pytorch, and many others. NumPy also enables you to perform vectorized math directly on the NumPy array, such as performing element-wise operations, without having to code in loops and other approaches you might have to use if you were using a different type of sequence.

When working with vectors for vectorization, there are often hundreds or thousands of dimensions, which refers to the number of floating points present in the vector. Higher dimensionality can capture more detailed semantic information, which is crucial for accurately matching query inputs with relevant documents or data in RAG applications.

Understanding vectors can be a crucial underlying concept to understand how to implement RAG, but how is RAG used in practical applications in the enterprise? We will discuss these practical AI applications of RAG in the next section.

## Implementing RAG in AI applications

RAG is rapidly becoming a cornerstone of generative AI platforms in the corporate world. RAG combines the power of retrieving internal or *new* data with generative language models to enhance the quality and relevance of the generated text. This technique can be particularly useful for companies across various industries to improve their products, services, and operational efficiencies. The following are some examples of how RAG can be used:

- **Customer support and chatbots:** These can exist without RAG, but when integrated with RAG, it can connect those chatbots with past customer interactions, FAQs, support documents, and anything else that was specific to that customer.
- **Technical support:** With better access to customer history and information, RAG-enhanced chatbots can provide a significant improvement to current technical support chatbots.
- **Automated reporting:** RAG can assist in creating initial drafts or summarizing existing articles, research papers, and other types of unstructured data into more digestible formats.
- **E-commerce support:** For e-commerce companies, RAG can help generate dynamic product descriptions and user content, as well as make better product recommendations.
- **Utilizing knowledge bases:** RAG improves the searchability and utility of both internal and general knowledge bases by generating summaries, providing direct answers to queries, and retrieving relevant information across various domains such as legal, compliance, research, medical, academia, patents, and technical documents.
- **Innovation scouting:** This is like searching general knowledge bases but with a focus on innovation. With this, companies can use RAG to scan and summarize information from quality sources to identify trends and potential areas for innovations that are relevant to that company's specialization.

- **Training and education:** RAG can be used by education organizations and corporate training programs to generate or customize learning materials based on specific needs and knowledge levels of the learners. With RAG, a much deeper level of internal knowledge from the organization can be incorporated into the educational curriculum in very customized ways to the individual or role.

You might be wondering, *"If I am using an LLM such as ChatGPT to answer my questions in my company, does that mean my company is using RAG already?"*

The answer is "No."

If you just log in to ChatGPT and ask questions, that is not the same as implementing RAG. Both ChatGPT and RAG are forms of generative AI, and they are sometimes used together, but they are two different concepts. In the next section, we will discuss the differences between generative AI and RAG.

## Comparing RAG with conventional generative AI

Conventional generative AI has already shown to be a revolutionary change for companies, helping their employees reach new levels of productivity. LLMs such as ChatGPT are assisting users with a rapidly growing list of applications that include writing business plans, writing and improving code, writing marketing copy, and even providing healthier recipes for a specific diet. Ultimately, much of what users are doing is getting done faster.

However, conventional generative AI does not know what it does not know. And that includes most of the internal data in your company. Can you imagine what you could do with all the benefits mentioned previously, but combined with all the data within your company – about everything your company has ever done, about your customers and all their interactions, or about all your products and services combined with a knowledge of what a specific customer's needs are? You do not have to imagine it – that is what RAG does!

Before RAG, most of the services you saw that connected customers or employees with the data resources of the company were just scratching the surface of what is

possible compared to if they could access *all* the data in the company. With the advent of RAG and generative AI in general, corporations are on the precipice of something really, really big.

Another area you might confuse RAG with is the concept of fine-tuning a model. Let's discuss what the differences are between these types of approaches.

## Comparing RAG with model fine-tuning

LLMs can be adapted to your data in two ways:

- **Fine-tuning:** With fine-tuning, you are adjusting the weights and/or biases that define the model's intelligence based on new training data. This directly impacts the model, permanently changing how it will interact with new inputs.
- **Input/prompts:** This is where you *use* the model, using the prompt/input to introduce new knowledge that the LLM can act upon.

Why not use fine-tuning in all situations? Once you have introduced the new knowledge, the LLM will always have it! It is also how the model was created – by being trained with data, right? That sounds right in theory, but in practice, fine-tuning has been more reliable in teaching a model specialized tasks (such as teaching a model how to converse in a certain way), and less reliable for factual recall.

The reason is complicated, but in general, a model's knowledge of facts is like a human's long-term memory. If you memorize a long passage from a speech or book and then try to recall it a few months later, you will likely still understand the context of the information, but you may forget specific details. On the other hand, adding knowledge through the input of the model is like our short-term memory, where the facts, details, and even the order of wording are all very fresh and available for recall. It is this latter scenario that lends itself better in a situation where you want successful factual recall. And given how much more expensive fine-tuning can be, this makes it that much more important to consider RAG.

There is a trade-off, though. While there are generally ways to feed all data you have to a model for fine-tuning, inputs are limited by the context window of the model. This is an area that is actively being addressed. For example, early versions of ChatGPT 3.5 had a 4,096 token context window, which is the equivalent of about five pages of text. When ChatGPT 4 was released, they expanded the context window to 8,192 tokens (10 pages) and there was a Chat 4-32k version that had a context window of 32,768 tokens.

(40 pages). This issue is so important that they included the context window size in the name of the model. That is a strong indicator of how important the context window is!

**Interesting fact!**

What about the latest Gemini 1.5 model? It has a 1 million token context window or over 1,000 pages!

As the context windows expand, another issue is created. Early models with expanded context windows were shown to lose a lot of the details, especially in the *middle* of the text. This issue is also being addressed. The Gemini 1.5 model with its 1 million token context window has performed well in tests called *needle in a haystack* tests for *remembering* all details well throughout the text it can take as input. Unfortunately, the model did not perform as well in the *multiple needles in a haystack* tests. Expect more effort in this area as these context windows become larger.

**Note**

It is important to note that token count differs from word count as tokens include punctuation, symbols, numbers, and other text representations. How a compound word such as *ice cream* is treated token-wise depends on the tokenization scheme and it can vary across LLMs. But most well-known LLMs (such as ChatGPT and Gemini) would consider *ice cream* as two tokens. Under certain circumstances in NLP, you may argue that it should be one token based on the concept that a token should represent a useful semantic unit for processing, but that is not the case for these models.

Fine-tuning can also be quite expensive, depending on the environment and resources you have available. In recent years, the costs for fine-tuning have come down substantially due to new techniques such as representative fine-tuning, LoRA-related techniques, and quantization. But in many RAG development efforts, fine-tuning is considered an additional cost to already expensive RAG efforts, so it is considered a more expensive addition to the efforts.

Ultimately, when deciding between RAG and fine-tuning, consider your specific use case and requirements. RAG is generally superior for retrieving factual information

that is not present in the LLM's training data or is private. It allows you to dynamically integrate external knowledge without modifying the model's weights. Fine-tuning, on the other hand, is more suitable for teaching the model specialized tasks or adapting it to a specific domain. Keep the limitations of context window sizes and the potential for overfitting in mind when fine-tuning a specific dataset.

Now that we have defined what RAG is, particularly when compared to other approaches that use generative AI, let's review the general architecture of RAG systems.

## The architecture of RAG systems

The following are the stages of a RAG process from a user's perspective:

1. A user enters a query/question.
2. The application thinks for a little while before checking the data it has access to so that it can see what is the most relevant.
3. The application provides a response that focuses on answering the user's question, but using data that has been provided to it through the RAG pipeline.

From a technical standpoint, this captures two of the stages you will code: the **retrieval** and **generation** stages. But there is one other stage, known as **indexing**, which can be and is often executed before the user enters the query. With indexing, you are turning supporting data into vectors, storing them in a vector database, and likely optimizing the search functionality so that the retrieval step is as fast and effective as possible.

Once the user passes their query into the system, the following steps occur:

1. The user query is vectorized.
2. The vectorized query is passed to a vector search to retrieve the most relevant data in a vector database representing your external data.
3. The vector search returns the most relevant results and unique keys referencing the original content those vectors represent.
4. The unique keys are used to pull out the original data associated with those vectors, often in a batch of multiple documents.
5. The original data might be filtered or post-processed but will typically then be passed to an LLM based on what you expect the RAG process to do.
6. The LLM is provided with a prompt that generally says something like "You are a helpful assistant for question-answering tasks. Take the following question (the user query) and use this helpful information (the

data retrieved in the similarity search) to answer it. If you don't know the answer based on the information provided, just say you don't know."

7. The LLM processes that prompt and provides a response based on the external data you provided.

Depending on the scope of the RAG system, these steps can be done in real time, or steps such as indexing can be done before the query so that it is ready to be searched when the time comes.

As mentioned previously, we can break these aspects down into three main stages (see *Figure 37.2*):

- Indexing
- Retrieval
- Generation

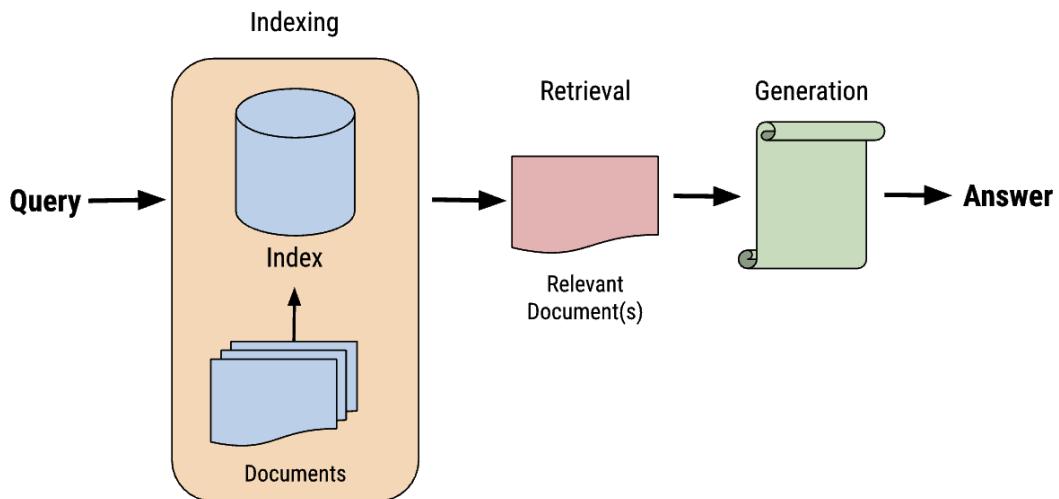


Figure 37.2 – The three stages of RAG

Chapter

# 38

# Developing a Complete RAG Pipeline

As we step through the code, you will gain a comprehensive understanding of each step in the RAG process programmatically by using tools such as LangChain, Chroma DB, and OpenAI's APIs.

You will need to run this chapter's code in an environment that's been set up to run Jupyter notebooks. Experience with Jupyter notebooks is a prerequisite for using this book, and it is too difficult to cover it in a short amount of text. There are numerous ways to set up a notebook environment. There are online versions, versions you can download, notebook environments that universities provide students, and different interfaces you can use. If you are doing this at a company, they will likely have an environment you will want to get familiar with. Each of these options takes very different instructions to set up, and those instructions change often. If you need to brush up on your knowledge about this type of environment, you can start on the Jupyter website: <https://docs.jupyter.org/en/latest/>. Start here, then ask your favorite LLM for more help to get your environment set up.

What do I use? When I use my Chromebook, often when I am traveling, I use a notebook set up in one of the cloud environments. I prefer Google Colab or their Colab Enterprise notebooks, which you can find in the Vertex AI section of Google Cloud Platform. But these environments cost money, often exceeding \$20 a month if you are active. If you are as active as me, it can exceed \$1,000 per month!

As a cost-effective alternative for when I am that active, I use Docker Desktop on my Mac, which hosts a Kubernetes cluster locally, and set up my notebook environment in the cluster. All these approaches have several environmental requirements that are often changing. It is best to do a little research and figure out what works best for your situation. There are similar solutions for Windows-based computers.

Ultimately, the primary requirement is to find an environment in which you can run a Jupyter notebook using Python 3. The code we will provide will indicate what other packages you will need to install.

**Note**

All of this code assumes you are working in a Jupyter notebook. You could do this directly in a Python file (.py), but you may have to change some of it. Running this in a notebook gives you the ability to step through it cell by cell and see what happens at each point to better understand the entire process.

## Setting up a large language model (LLM) account

For the general public, OpenAI's ChatGPT models are currently the most popular and well-known LLMs. However, there are many other LLMs available in the market that fit a myriad of purposes. You do not always need to use the most expensive, most powerful LLM. Some LLMs focus on one area, such as the Meditron LLMs, which are medical research-focused fine-tuned versions of Llama 2. If you are in the medical area, you may want to use that LLM instead as it may do better than a big general LLM in your domain. Often, LLMs can be used to double-check other LLMs, so you have to have more than one in those cases. I strongly encourage you to not just use the first LLM you have worked with and to look for the LLM that best suits your needs. But to keep things simpler this early, I am going to talk about setting up OpenAI's ChatGPT:

1. Go to the **API** section of the OpenAI website: <https://openai.com/api/>.
2. If you have not set up an account yet, do so now. The web page can change often, but look for where to sign up.
3. Once you've signed up, go to the documentation at <https://platform.openai.com/docs/quickstart> and follow the instructions to set up your first API key.
4. When creating an API key, give it a memorable name and select the type of permissions you want to implement (**All**, **Restricted**, or **Read Only**). If you do not know what option to select, it is best to go with **All** for now. However, be aware of the other options – you may want to share various responsibilities with other team members but restrict certain types of access:
  - A. **All**: This key will have read/write access to all of the OpenAI APIs.
  - B. **Restricted**: A list of available APIs will appear, providing you with granular control over which APIs the key has access to. You have the option of giving just read or write access to each API. Make sure you have at least enabled the models and embedding APIs you will use in these demos.

- C. **Read Only:** This option gives you read-only access to all APIs.
5. Copy the key provided. You will add this to your code shortly. In the meantime, keep in mind that if this key is shared with anyone else, whomever you provide this key can use it and you will be charged. So, this is a key that you want to consider top secret and take the proper precautions to prevent unauthorized use of it.
  6. The OpenAI API requires you to buy credits in advance to use the API. Buy what you are comfortable with, and then for more safety, make sure the **Enable auto recharge** option is off. This will ensure you are only spending what you intend to spend.

With that, you have set up the key component that will serve as the *brains* in your RAG pipeline: the LLM! Next, we will set up your development environment so that you can connect to the LLM.

## Installing the necessary packages

Make sure these packages are installed in your Python environment. Add the following lines of code in the first cell of your notebook:

```
%pip install langchain_community
%pip install langchain_experimental
%pip install langchain-openai
%pip install langchainhub
%pip install chromadb
%pip install langchain
%pip install beautifulsoup4
```

The preceding code installs several Python libraries using the pip package manager, something you will need to run the code I am providing. Here's a breakdown of each library:

- **langchain\_community:** This is a community-driven package for the LangChain library, which is an open source framework for building applications with LLMs. It provides a set of tools and components for working with LLMs and integrating them into various applications.
- **langchain\_experimental:** The langchain\_experimental library offers additional capabilities and tools beyond the core LangChain library that are not yet fully stable or production-ready but are still available for experimentation and exploration.

- `langchain-openai`: This package provides integration between LangChain and OpenAI's language models. It allows you to easily incorporate OpenAI's models, such as ChatGPT 4 or the OpenAI embeddings service, into your LangChain applications.
- `langchainhub`: This package provides a collection of pre-built components and templates for LangChain applications. It includes various agents, memory components, and utility functions that can be used to accelerate the development of LangChain-based applications.
- `chromadb`: This is the package name for Chroma DB, a high-performance embedding/vector database designed for efficient similarity search and retrieval.
- `langchain`: This is the core LangChain library itself. It provides a framework and a set of abstractions for building applications with LLMs. LangChain includes the components needed for an effective RAG pipeline, including prompting, memory management, agents, and other integrations with various external tools and services.

After running the preceding first line, you will need to restart your kernel to be able to access all of the new packages you just installed in the environment. Depending on what environment you are in, this can be done in a variety of ways. Typically, you will see a refresh button you can use or a **Restart kernel** option in the menu.

If you have trouble finding a way to restart the kernel, add this cell and run it:

```
import IPython
app = IPython.Application.instance();
app.kernel.do_shutdown(True)
```

This is a code version for performing a kernel restart in an IPython environment (notebooks). You shouldn't need it, but it is here for you just in case!

Once you have installed these packages and restarted your kernel, you are ready to start coding! Let's start with importing many of the packages you just installed in your environment.

## Imports

Now, let's import all of the libraries needed to perform the RAG-related tasks. I have provided comments at the top of each group of imports to indicate what area of RAG the imports are relevant to. This, combined with the description in the following list, provides a basic introduction to everything you need for your first RAG pipeline:

```
import os
from langchain_community.document_loaders import WebBaseLoader
import bs4
import openai
from langchain_openai import ChatOpenAI, OpenAIEMBEDDINGS
from langchain import hub
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
import chromadb
from langchain_community.vectorstores import Chroma
from langchain_experimental.text_splitter import SemanticChunker
```

Let's step through each of these imports:

- `import os`: This provides a way to interact with the operating system. It is useful for performing operations such as accessing environment variables and working with file paths.
- `from langchain_community.document_loaders import WebBaseLoader`: The `WebBaseLoader` class is a document loader that can fetch and load web pages as documents.
- `import bs4`: The `bs4` module, which stands for **Beautiful Soup 4**, is a popular library for web scraping and parsing HTML or XML documents. Since we will be working with a web page, this gives us a simple way to pull out the title, content, and headers separately.
- `import openai`: This provides an interface to interact with OpenAI's language models and APIs.
- `from langchain_openai import ChatOpenAI, OpenAIEMBEDDINGS`: This imports both `ChatOpenAI` (for the LLM) and `OpenAIEMBEDDINGS` (for the embeddings), which are specific implementations of language models and embeddings that use OpenAI's models that work directly with LangChain.
- `from langchain import hub`: The `hub` component provides access to various pre-built components and utilities for working with language models.
- `from langchain_core.output_parsers import StrOutputParser`: This component parses the output generated by the language model and extracts the relevant information. In this case, it assumes that the language model's output is a string and returns it as-is.
- `from langchain_core.runnables import RunnablePassthrough`: This component passes through the question or query without any modifications. It allows the question to be used as-is in the subsequent steps of the chain.

- `Import chromadb`: As mentioned previously, `chromadb` imports the Chroma DB vector store, a high-performance embedding/vector database designed for efficient similarity search and retrieval.
- `from langchain_community.vectorstores import Chroma`: This provides an interface to interact with the Chroma vector database using LangChain.
- `from langchain_experimental.text_splitter import SemanticChunker`: A text splitter is typically a function that we use to split the text into small chunks based on a specified chunk size and overlap. This splitter is called `SemanticChunker`, an experimental text-splitting utility provided by the `Langchain_experimental` library. The main purpose of `SemanticChunker` is to break down long text into more manageable pieces while preserving the semantic coherence and context of each chunk.

These imports provide the essential Python packages that will be needed to set up your RAG pipeline. Your next step will be to connect your environment to OpenAI's API.

## OpenAI connection

The following line of code is a very simple demonstration of how your API key will be ingested into the system. However, this is not a secure way to use an API key. There are many ways to do this more securely.

You are going to need replace `sk-#####` with your actual OpenAI API key:

```
os.environ['OPENAI_API_KEY'] = 'sk-#####'
openai.api_key = os.environ['OPENAI_API_KEY']
```

### Important

This is just a simple example; please use a secure approach to hide your API key!

You have probably guessed that this OpenAI API key will be used to connect to the ChatGPT LLM. But ChatGPT is not the only service we will use from OpenAI. This API key is also used to access the OpenAI embedding service. In the next section, which focuses on coding the indexing stage of the RAG process, we will utilize the OpenAI embedding service to convert your content into vector embeddings, a key aspect of the RAG pipeline.

## Indexing

The next few steps represent the *indexing* stage, where we obtain our target data, preprocess it, and vectorize it. These steps are often done *offline*, meaning they are done to prepare the application for usage later. But in some cases, it may make sense to do this all in real time, such as in rapidly changing data environments where the data that is used is relatively small. In this particular example, the steps are as follows:

1. Web loading and crawling.
2. Splitting the data into digestible chunks for the Chroma DB vectorizing algorithm.
3. Embedding and indexing those chunks.
4. Adding those chunks and embeddings to the Chroma DB vector store.

Let's start with the first step: web loading and crawling.

## Web loading and crawling

To start, we need to pull in our data. This could be anything of course, but we have to start somewhere!

For our example, I am providing a web page example based on some of the content from *Chapter RAG*. I have adopted the original structure from an example provided by LangChain at <https://lilianweng.github.io/posts/2023-06-23-agent/>.

You can try that web page as well if it is still available when you read this, but be sure to change the question you use to query the content to a question more suitable to the content on that page. You also need to restart your kernel if you change web pages; otherwise, it will include content from both web pages if you rerun the loader. That may be what you want, but I'm just letting you know!

I also encourage you to try this with other web pages and see what challenges these other pages present. This example involves a very clean piece of data compared to most web pages, which tend to be rife with ads and other content you do not want showing up. But maybe you can find a relatively clean blog post and pull that in? Maybe you can create your own? Try different web pages and see!

```
loader = WebBaseLoader(
 web_paths=("https://kbourne.github.io/chapter1.html",),
 bs_kwarg=dict(
 parse_only=bs4.SoupStrainer(
 class_=("post-content", "post-title",
```

```
 "post-header")
)
),
)
docs = loader.load()
```

The preceding code starts with using the `WebBaseLoader` class from the `langchain_community/document_loaders` module to load web pages as documents. Let's break it down:

1. Creating the `WebBaseLoader` instance: The `WebBaseLoader` class is instantiated with the following parameters:
  - `web_paths`: A tuple containing the URLs of the web pages to be loaded. In this case, it contains a single URL: `https://kbourne.github.io/chapter1.html`.
  - `bs_kwarg`s: A dictionary of keyword arguments to be passed to the `BeautifulSoup` parser.
  - `parse_only`: `Abs4.SoupStrainer` object specifies the HTML elements to parse. In this case, it is set to parse only the elements with the CSS classes, such as `post-content`, `post-title`, and `post-header`.
2. The `WebBaseLoader` instance initiates a series of steps that represent the loading of the document into your environment: The `load` method is called on `loader`, the `WebBaseLoader` instance that fetches and loads the specified web pages as documents. Internally, `loader` is doing a lot!

Here are the steps it performs just based on this small amount of code:

- I. Makes HTTP requests to the specified URLs to fetch the web pages.
- II. Parses the HTML content of the web pages using `BeautifulSoup`, considering only the elements specified by the `parse_only` parameter.
- III. Extracts the relevant text content from the parsed HTML elements.
- IV. Creates `Document` objects for each web page that contain the extracted text content, along with metadata such as the source URL.

The resulting `Document` objects are stored in the `docs` variable for further use in our code!

The classes that we are passing to `bs4` (`post-content`, `post-title`, and `post-header`) are CSS classes. If you are using an HTML page that does not have those CSS classes, this will not work. So, if you are using a different URL and are not getting data, take a look at what the CSS tags are in the HTML you are crawling. Many web pages do use this pattern, but not all! Crawling web pages presents many challenges like this.

Once you have collected the documents from your data source, you need to pre-process them. In this case, this involves splitting.

## Splitting

If you are using the provided URL, you will only parse the elements with the `post-content`, `post-title`, and `post-header` CSS classes. This will extract the text content from the main article body (usually identified by the `post-content` class), the title of the blog post (usually identified by the `post-title` class), and any header information (usually identified by the `post-header` class).

In case you were curious, this is what this document looks like on the web (*Figure 38.1*):

### Introduction to Retrieval Augmented Generation (RAG)

Date: March 10, 2024 | Estimated Reading Time: 15 min | Author: Keith Bourne

In the rapidly evolving field of artificial intelligence, Retrieval-Augmented Generation (RAG) is emerging as a significant addition to the Generative AI toolkit. RAG harnesses the strengths of Large Language Models (LLMs) and integrates them with internal data, offering a method to enhance organizational operations significantly. This book delves into the essential aspects of RAG, examining its role in augmenting the capabilities of LLMs and leveraging internal corporate data for strategic advantage.

As it progresses, the book outlines the potential of RAG in business, suggesting how it can make AI applications smarter, more responsive, and aligned with organizational objectives. RAG is positioned as a key facilitator of customized, efficient, and insightful AI solutions, bridging the gap between Generative AI's potential and specific business needs. This exploration of RAG encourages readers to unlock the full potential of their corporate data, paving the way for an era of AI-driven innovation.

### What You Can Expect to Learn

Expect to launch a comprehensive journey to understand and effectively incorporate Retrieval Augmented Generation (RAG) into AI systems. You'll explore a broad spectrum of essential topics, including vector databases, the vectorization process, vector search techniques, prompt engineering and design, and the use of AI agents for RAG applications, alongside methods for evaluating and visualizing RAG outcomes. Through practical, working code examples utilizing the latest tools and technologies like LangChain and Chroma's vector database, you'll gain hands-on experience in implementing RAG in your projects.

At the outset, you'll delve into the core principles of RAG, appreciating its significance in the broader landscape of Generative AI. This foundational knowledge equips you with the perspective needed to discern how RAG applications are designed and why they succeed, paving the way for innovative solution development and problem-solving in AI.

You'll discover the symbiosis between Large Language Models (LLMs) and internal data to bolster organizational operations. By learning about the intricacies of this integration, particularly the process of vectorization, including the creation and management of vector databases for efficient information retrieval, you'll gain crucial skills for navigating and harnessing vast data landscapes effectively in today's data-driven environments.

Gain expertise in vector search techniques, an essential skill set for identifying pertinent data within extensive datasets. Coupled with this, you'll learn strategies for prompt engineering and design, ensuring that you can craft queries that elicit precise and relevant AI responses.

Explore how AI agents play a pivotal role in RAG applications, facilitating sophisticated data interaction and retrieval tasks. You'll also learn methods for evaluating and visualizing RAG implementation outcomes, providing a framework for assessing performance and impact critically.

Throughout this journey, you'll engage in practical, hands-on learning, guided through the use of cutting-edge tools like LangChain and Chroma's vector database, supported by real, working code examples. These detailed coding demonstrations, grounded in current frameworks, offer a practical foray into implementing RAG in AI systems, providing a rich learning experience.

Case studies and coding exercises strategically interspersed throughout your learning path highlight the application of RAG in various real-world scenarios. These insights into addressing common and complex challenges prepare you to navigate the application of RAG across diverse settings with confidence. The code will build off the same starting use case provided in the next chapter. For each topic that relates to code, we will add code that shows how that topic impacts the RAG pipeline, giving you an in-depth understanding about how your coding choices can impact the capabilities of your RAG-based application.

You'll also explore optimization strategies for data retrieval and enhancing the interpretability of AI-generated content. These insights are pivotal for improving the usability and effectiveness of AI applications, ensuring they are more aligned with strategic business objectives and user needs.

As you progress, you'll gain a deeper understanding of how RAG can revolutionize AI applications, making them more intelligent, responsive, and tailored to specific requirements. The potential of RAG to facilitate personalized, efficient, and insightful AI solutions is thoroughly examined, bridging the theoretical and practical divides.

Throughout this learning experience, a spirit of exploration and experimentation is encouraged, aiming to unlock the full potential of data through RAG, fostering innovation, and advancing the domain of AI-driven solutions. By the end, you will have gained comprehensive knowledge and practical skills in RAG, equipping you to contribute to the evolution of AI technologies and applications in your business and beyond.

Figure 38.1 – A web page that we will process

It goes down many pages too! There is a lot of content here, too much for an LLM to process directly. So, we will need to split the document into digestible chunks:

```
text_splitter = SemanticChunker(OpenAIEmbeddings())
splits = text_splitter.split_documents(docs)
```

There are many text splitters available in LangChain, but I chose to start with an experimental, but very interesting, option called `SemanticChunker`. As I mentioned previously, when talking about the imports, `SemanticChunker` focuses on breaking down long text into more manageable pieces while preserving the semantic coherence and context of each chunk. Other text splitters typically take an arbitrary chunk length that is not context-aware, something that creates issues when important content gets split by the chunker.

It should also be noted that the `SemanticChunker` splitter you use in this code uses `OpenAIEmbeddings`, and it costs money to process the embeddings. The OpenAI embedding models currently cost between \$0.02 and \$0.13 per million tokens, depending on what model you use. At the time of writing, if do not designate an embedding model, OpenAI will use the `text-embedding-ada-002` model by default, which costs \$0.02 per million tokens. If you want to avoid the cost, fall back to `RecursiveCharacterTextSplitter`.

I encourage you to go ahead and try different splitters and see what happens! For example, do you think you get better results from `RecursiveCharacterTextSplitter` than from `SemanticChunker`, which we are using here? Maybe speed is more important than quality in your particular case – which one is faster?

Once you have chunked up your content, the next step is to convert it into the vector embeddings we have talked so much about!

## Embedding and indexing the chunks

The next few steps represent the retrieval and generation steps, where we will use Chroma DB as the vector database. As mentioned multiple times now, Chroma DB is a great vector store! I selected this vector store because it is easy to run locally and it works well for demos like this, but it is a fairly powerful vector store. As you may recall when we talked about vocabulary and the difference between vector stores and vector databases, Chroma DB is indeed both! Chroma is one of many options for your vector store though.

We are using OpenAI embeddings here as well, which will use your OpenAI key to send your chunks of data to the OpenAI API, convert them into embeddings, and then send them back in their mathematical form. Note that this *does* cost money! It is a fraction of a penny for each embedding, but it is worth noting. So, please use caution when using this code if you are doing this on a tight budget!

```
vectorstore = Chroma.from_documents(
 documents=splits,
 embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
```

First, we create the Chroma vector store with the `Chroma.from_documents` method, which is called to create a Chroma vector store from the split documents. This is one of many methods we can use to create a Chroma database. This typically depends on the source, but for this particular method, it takes the following parameters:

- `documents`: The list of split documents (`splits`) obtained from the previous code snippet
- `embedding`: An instance of the `OpenAIEmbeddings` class, which is used to generate embeddings for the documents

Internally, the method is doing a few things:

1. It iterates over each `Document` object in the `splits` list.
2. For each `Document` object, it uses the provided `OpenAIEmbeddings` instance to generate an embedding vector.
3. It stores the document text and its corresponding embedding vector in the Chroma vector database.

At this point, you now have a vector database called `vectorstore`, and it is full of embeddings, which are...? That's right – mathematical representations of all of the content from the web page you just crawled! So cool!

But what is this next part – a retriever? Is this of the canine variety? Nope. This is creating the mechanism that you will use to perform vector similarity searches on your new vector database. You call the `as_retriever` method right on the `vectorstore` instance to create the retriever. The retriever is an object that provides a convenient interface for performing these similarity searches and retrieving the relevant documents from the vector database based on those searches.

If you just want to perform the document retrieval process, you can. This is not officially part of the code, but if you want to test this out, add this in an extra cell and run it:

```
query = "How does RAG compare with fine-tuning?"
relevant_docs = retriever.get_relevant_documents(query)
relevant_docs
```

The output should be what I list later in this code when I indicate what is passed to the LLM, but it is essentially a list of the content stored in the `vectorstore` vector database that is most similar to the query.

Aren't you impressed? This is a simple example of course, but this is the foundation for much more powerful tools that you can use to access your data and supercharge generative AI applications for your organization!

However, at this point in the application, you have only created the receiver. You have not used it within the RAG pipeline yet. We will review how to do that next!

## Retrieval and generation

In the code, the retrieval and generation stages are combined within the chain we set up to represent the entire RAG process. This leverages pre-built components from the **LangChain Hub**, such as **prompt templates**, and integrates them with a selected LLM. We will also utilize the **LangChain Expression Language (LCEL)** to define a chain of operations that retrieves relevant documents based on an input question, formats the retrieved content, and feeds it into the LLM to generate a response. Overall, the steps we take in retrieval and generation are as follows:

1. Take in a user query.
2. Vectorize that user query.
3. Perform a similarity search of the vector store to find the closest vectors to the user query vector, as well as their associated content.
4. Pass the retrieved content into a prompt template, a process known as **hydrating**.
5. Pass that *hydrated* prompt to the LLM.
6. Once you receive a response from the LLM, present it to the user.

From a coding standpoint, we will start by defining the prompt template so that we have something to hydrate when we receive the user query. We will cover this in the next section.

## Prompt templates from the LangChain Hub

The LangChain Hub is a collection of pre-built components and templates that can be easily integrated into LangChain applications. It provides a centralized repository for sharing and discovering reusable components, such as prompts, agents, and utilities.

Here, we are calling a prompt template from the LangChain Hub and assigning it to `prompt`, a prompt template representing what we will pass to the LLM:

```
prompt = hub.pull("jclemens24/rag-prompt")
print(prompt)
```

This code retrieves a pre-built prompt template from the LangChain Hub using the `pull` method of the `hub` module. The prompt template is identified by the `jclemens24/rag-prompt` string. This identifier follows the *repository/component* convention, where *repository* represents the organization or user hosting the component, and *component* represents the specific component being pulled. The `rag-prompt` component indicates it is a prompt designed for RAG applications.

If you print out the prompt with `print(prompt)`, you can see what is used here, as well as what the inputs are:

```
input_variables=['context', 'question']
messages=[HumanMessagePromptTemplate(prompt=PromptTemplate(input_
variables=['context', 'question'], template="You are an assistant
for question-answering tasks. Use the following pieces of retrieved-
context to answer the question. If you don't know the answer, just
say that you don't know.\nQuestion: {question} \nContext: {context} \
nAnswer:"))]
```

This is the initial part of the prompt that gets passed to the LLM, which in this case, tells it this:

```
"You are an assistant for question-answering tasks. Use the following
pieces of retrieved-context to answer the question. If you don't know
the answer, just say that you don't know.

Question: {question}
Context: {context}
Answer:"
```

Later, you add the `question` and `context` variables to *hydrate* the prompt, but starting with this format optimizes it to work better for RAG applications.

### Note

The `jclemens24/rag-prompt` string is one version of the predefined starting prompts. Visit the LangChain Hub to find many more – you may even find one that better fits your needs: <https://smith.langchain.com/hub/search?q=rag-prompt>.

You can also use your own! I can count over 30 options at the time of writing!

The prompt template is a key part of the RAG pipeline as it represents how you communicate with the LLM to receive the response you are seeking. But in most RAG pipelines, getting the prompt into a format so that it can work with the prompt template is not as straightforward as just passing it a string. In this example, the context variable represents the content we get from the retriever and that is not in a string format yet! We will walk through how to convert our retrieved content into the proper string format we need next.

## Formatting a function so that it matches the next step's input

First, we will set up a function that takes the list of retrieved documents (docs) as input:

```
def format_docs(docs):
 return "\n\n".join(doc.page_content for doc in docs)
```

Inside this function, a generator expression, `(doc.page_content for doc in docs)`, is used to extract the `page_content` attribute from each document object. The `page_content` attribute represents the text content of each document.

### Note

In this case, a *document* is not the entire document that you crawled earlier. It is just one small section of it, but we generally call these documents.

The `join` method is called on the `\n\n` string to concatenate `page_content` of each document with two newline characters between each document's content. The formatted string is returned by the `format_docs` function to represent the `context` key in the dictionary that is piped into the `prompt` object.

The purpose of this function is to format the output of the retriever into the string format that it will need to be in for the next step in the chain, after the retriever step. We will explain this further in a moment, but short functions like this are often necessary for LangChain chains to match up inputs and outputs across the entire chain.

Next, we will review the last step before we can create our LangChain chain – that is, defining the LLM we will use in that chain.

## Defining your LLM

Let's set up the LLM model you will use:

```
llm = ChatOpenAI(model_name="gpt-4o-mini", temperature=0)
```

The preceding code creates an instance of the `ChatOpenAI` class from the `langchain_openai` module, which serves as an interface to OpenAI's language models, specifically the GPT-4o mini model. Even though this model is newer, it was released at a significant discount to the older models. Using this model will help keep your inference costs down while still allowing you to use a recent model! If you would like to try a different version of ChatGPT, such as `gpt-4`, you can just change the model name. Look up the newest models on the OpenAI API website – they add them often!

## Setting up a LangChain chain using LCEL

This *chain* is in a code format specific to LangChain called LCEL. You will see me using LCEL throughout the code from here on out. Not only does it make the code easier to read and more concise, but it opens up new techniques focused on improving the speed and efficiency of your LangChain code.

If you walk through this chain, you'll see it provides a great representation of the entire RAG process:

```
rag_chain = (
 {"context": retriever | format_docs,
 "question": RunnablePassthrough()}
 | prompt
 | llm
 | StrOutputParser()
)
```

All of these components have already been described, but to summarize, the `rag_chain` variable represents a chain of operations using the LangChain framework. Let's walk through each step of the chain, digging into what is happening at each point:

1. **Retrieval:** The first *link* in the chain can be called retrieval since that is what it handles. However, it has its own chain. So, let's break this step down even further:
  - A. When we call on the `rag_chain` variable in a moment, we will pass it a "question." As shown in the preceding code, the chain starts with a dictionary that defines two keys: "context" and "question". The question part is pretty straightforward, but where does the context come from? The "context" key assigned is the result of the `retriever | format_docs` operation.

- B. Does `format_docs` sound familiar? Yes! That's because we just set up that function previously. Here, we use that function alongside `retriever`. The `|` operator, called a pipe, between the retriever and `format_docs` indicates that we are chaining these operations together. So, in this case, the `retriever` object is *piped* into the `format_docs` function. We are running the `retriever` operation here, which is the vector similarity search. The similarity search should return a set of matches; that set of matches is what is passed to the function. Our `format_docs` function, as described earlier, is then used on the content provided by the retriever to format all the results of that retriever into a single string. That complete string is then assigned to the `context`, which as you may remember is a variable in our prompt. The expected input format of the next step is a dictionary with two keys – that is, “`context`” and “`question`”. The values that are assigned to these keys are expected to be strings. So, we can't just pass `retriever` output, which is a list of objects. This is why we use the `format_docs` function – to convert the retriever results into the string we need for the next step. Let's go back to the `question` that was passed into the chain, which is already in the string format we require. We don't need any formatting! So, we use the `RunnablePassthrough()` object to just let that input (the `question` provided) pass through as the string that it is already formatted as. That object takes the `question` we pass into the `rag_chain` variable and passes it through without any modification. We now have our first step in the chain, which is defining the two variables that the prompt in the next step accepts.
2. We can see another pipe (`|`) followed by the `prompt` object, and we *pipe* the variables (in a dictionary) into that `prompt` object. This is known as hydrating the prompt. As mentioned previously, the `prompt` object is a prompt template that defines what we will pass to the LLM, and it typically includes input variables (`context` and `question`) that are filled/hydrated first. The result of this second step is the full prompt text as a string, with the variables filling in the placeholders for `context` and `question`. Then, we have another pipe (`|`) and the `llm` object that we defined earlier. As we have seen already, this step in the chain takes the output from the previous step, which is the prompt string that includes all the information from previous steps. The `llm` object represents the language model we set up, which in this case is ChatGPT 40. The formatted prompt string is passed as input to the language model, which generates a response based on the provided context and question.

3. It almost seems like this would be enough, but when you use an LLM API, it is not just sending you the text you might see when you type something into ChatGPT. It is in a JSON format and has a lot of other data included with it. So, to keep things simple, we are going to *pipe* the LLM's output to the next step and use LangChain's `StrOutputParser()` object. Note that `StrOutputParser()` is a utility class in LangChain that parses the key output of the language model into a string format. Not only does it strip away all the information you did not want to deal with right now, but it ensures that the generated response is returned as a string.

Let's take a moment to appreciate everything we just did here. This *chain* we created using LangChain represents the core code for our entire RAG pipeline, and it is just a few strings long!

When the user uses your application, it will start with the user query. But from a coding standpoint, we set up everything else so that we can process the query properly. At this point, we are ready to accept the user query, so let's review this last step in our code.

## Submitting a question for RAG

So far, you have defined the chain, but you haven't run it. So, let's run the entire RAG pipeline in this one line, using a query you are feeding in:

```
rag_chain.invoke("What are the advantages of using RAG?")
```

As mentioned when stepping through what happens in the chain, "What are the advantages of using RAG?" is the string we are going to pass into the chain to begin with. The first step in the chain expects this string as the *question* we discussed in the previous section as one of the two expected variables. In some applications, this may not be in the proper format and will need an extra function to prepare it, but for this application, it is already in the string format we are expecting, so we pass it right into that `RunnablePassThrough()` object.

In the future, this prompt will include a query from a user interface, but for now, we will represent it as this variable string. Keep in mind that this is not the only text the LLM will see; you added a more robust prompt defined by `prompt` previously, hydrated by the "context" and "question" variables.

And that is it from a coding standpoint! But what happens when you run the code? Let's review the output you can expect from this RAG pipeline code.

## Final output

The final output will look something like this:

"The advantages of using Retrieval Augmented Generation (RAG) include:  
1. **Improved Accuracy and Relevance:** RAG enhances the accuracy and relevance of responses generated by large language models (LLMs) by fetching and incorporating specific information from databases or datasets in real time. This ensures outputs are based on both the model's pre-existing knowledge and the most current and relevant data provided.  
2. **Customization and Flexibility:** RAG allows for the customization of responses based on domain-specific needs by integrating a company's internal databases into the model's response generation process. This level of customization is invaluable for creating personalized experiences and for applications requiring high specificity and detail.  
3. **Expanding Model Knowledge Beyond Training Data:** RAG overcomes the limitations of LLMs, which are bound by the scope of their training data. By enabling models to access and utilize information not included in their initial training sets, RAG effectively expands the knowledge base of the model without the need for retraining. This makes LLMs more versatile and adaptable to new domains or rapidly evolving topics."

This has some basic formatting in it, so when it's displayed, it will look like this (including the bullets and bolded text):

The advantages of using Retrieval Augmented Generation (RAG) include:

- Improved Accuracy and Relevance: RAG enhances the accuracy and relevance of responses generated by large language models (LLMs) by fetching and incorporating specific information from databases or datasets in real time. This ensures outputs are based on both the model's pre-existing knowledge and the most current and relevant data provided.
- Customization and Flexibility: RAG allows for the customization of responses based on domain-specific needs by integrating a company's internal databases into the model's response generation process. This level of customization is invaluable for creating personalized experiences and for applications requiring high specificity and detail.
- Expanding Model Knowledge Beyond Training Data: RAG overcomes the limitations of LLMs, which are bound by the scope of their training data. By enabling models to access and utilize information not included in their initial training sets, RAG effectively expands the knowledge base of the model without the need for retraining. This makes LLMs more versatile and adaptable to new domains or rapidly evolving topics.

In your use cases, you will need to make decisions by asking questions such as, could a less expensive model do a good enough job at a significantly reduced cost? Or do I need to spend the extra money to get more robust responses? Your prompt may have said to keep it very brief and you end up with the same shorter response as a less expensive model anyway, so why spend the extra money? This is a common consideration when using these models, and in many cases, the largest, most expensive models are not always what is needed to meet the requirements of the application.

Here's what the LLM will see when you combine this with the RAG-focused prompt from earlier:

"You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know.

Question: What are the Advantages of using RAG?

Context: Can you imagine what you could do with all of the benefits mentioned above, but combined with all of the data within your company, about everything your company has ever done, about your customers and all of their interactions, or about all of your products and services combined with a knowledge of what a specific customer's needs are? You do not have to imagine it, that is what RAG does! Even smaller companies are not able to access much of their internal data resources very effectively. Larger companies are swimming in petabytes of data that are not readily accessible or are not being fully utilized. Before RAG, most of the services you saw that connected customers or employees with the data resources of the company were just scratching the surface of what is possible compared to if they could access ALL of the data in the company. With the advent of RAG and generative AI in general, corporations are on the precipice of something really, really big. Comparing RAG with Model Fine-Tuning\nEstablished Large Language Models (LLM), what we call the foundation models, can be learned in two ways:\nFine-tuning - With fine-tuning, you are adjusting the weights and/or biases that define the model's intelligence based

[TRUNCATED FOR BREVITY!]

Answer:"

As you can see, the context is quite large—it returns all of the most relevant information from the original document to help the LLM determine how to answer the new question. The context is what was returned by the vector similarity search.

## Complete code

Here is the code in its entirety:

```
%pip install langchain_community
%pip install langchain_experimental
%pip install langchain-openai
%pip install langchainhub
%pip install chromadb
%pip install langchain
%pip install beautifulsoup4
```

Restart the kernel before running the following code:

```
import os
from langchain_community.document_loaders import WebBaseLoader
import bs4
import openai
from langchain_openai import ChatOpenAI, OpenAIEMBEDDINGS
from langchain import hub
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
import chromadb
from langchain_community.vectorstores import Chroma
from langchain_experimental.text_splitter import SemanticChunker
os.environ['OPENAI_API_KEY'] = 'sk-#####'
openai.api_key = os.environ['OPENAI_API_KEY']
INDEXING ####
loader = WebBaseLoader(
 web_paths=("https://kbourne.github.io/chapter1.html",),
 bs_kwarg= dict(parse_only=bs4.SoupStrainer(
 class_= ("post-content",
 "post-title",
 "post-header"))
),
)
docs = loader.load()
text_splitter = SemanticChunker(OpenAIEMBEDDINGS())
splits = text_splitter.split_documents(docs)
vectorstore = Chroma.from_documents(
 documents=splits,
```

```
embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
RETRIEVAL and GENERATION
prompt = hub.pull("jclemens24/rag-prompt")
def format_docs(docs):
 return "\n\n".join(doc.page_content for doc in docs)
llm = ChatOpenAI(model_name="gpt-4o-mini")
rag_chain = (
 {"context": retriever | format_docs,
 "question": RunnablePassthrough()}
 | prompt
 | llm
 | StrOutputParser()
)
rag_chain.invoke("What are the Advantages of using RAG?")
```

Chapter

# 39

# RAG in Practice: Use Cases and Solutions

In previous *Chapter*, we listed several ways **retrieval-augmented generation (RAG)** is being implemented in AI applications, such as customer support with chatbots, automated reporting, product descriptions, searchability and utility of knowledge bases, innovation scouting, content personalization, product recommendations, and training and education.

The examples presented in this chapter are not meant to be exhaustive but rather to provide concrete, real-world scenarios that demonstrate the potential of RAG and stimulate your creativity in adapting this technology to your specific corporate environment. RAG's incredible flexibility allows it to be customized for a wide range of industries and use cases, with countless specific examples already in use across various organizations. By examining some of these examples, you will gain valuable insights into how RAG can be leveraged to enhance existing processes, improve efficiency, and drive innovation.

To conclude the chapter, we will present a code example that represents a common approach across many of these applications: an element of adding more relevant data to the response.

We start with discussing how chatbots can be significantly enhanced with the power of RAG and **generative AI (GenAI)**.

## Customer support and chatbots with RAG

**Chatbots** have evolved from simple scripted responses to the complex, RAG-driven conversational agents we see today. RAG has brought the next wave of innovation to chatbots, incorporating advanced Q&A systems into the capabilities of the chatbot in a way that is significantly more conversational and natural for the user. RAG combines the best of both worlds: the ability to retrieve information from vast datasets about your company and your customers and the capability to generate coherent, contextually

relevant responses. This has shown significant promise in customer support scenarios, where the ability to quickly access and leverage company-specific data, such as past customer interactions, FAQs, and support documents, has dramatically enhanced the quality of customer service.

RAG enables chatbots to provide personalized, efficient, and highly relevant responses to user queries in a way that far exceeds the performance of earlier models that relied solely on pre-programmed responses or basic **natural language processing (NLP)**.

Consider that this is not just a technical improvement. The level chatbots are reaching represents a transformative shift in how businesses can interact with their customers. GenAI has enabled companies to tap much deeper into all the data they have for each customer, such as reading through all your bank statements in PDF form, enabling true 1-to-1 service customization for each customer to be within reach. The RAG-enhanced chatbots can sift through vast amounts of this data to find the most relevant information, answering many more questions a user might have effectively, and ensuring that responses are accurate and tailored to the specific context of each interaction. Customers expect fast, relevant, and very personalized responses to their queries, and using RAG is easily the most effective way to deliver it.

Keep in mind that RAG-based systems are still in their infancy. In just a couple of years, expect RAG-based chatbots to push the boundaries much further. You may see conversational chatbots that can handle the most unique and specific inquiries without needing any human intervention. Their **natural language (NL)**, ability to handle most languages, and access to massive amounts of memory and computing power will revolutionize how companies can interface with every single one of their customers in very customized and personal ways.

The benefits of RAG in Q&A and chatbots extend across multiple sectors, such as technical support, financial services, healthcare, and e-commerce. We will briefly cover some of these popular examples, starting with technical support.

## Technical support

Consider a case where there is a recurring technical issue. For example, a large portion of inquiries to cable providers are related to the same technical issues. A RAG-enhanced chatbot could recognize the problem based on previous interactions and immediately provide a customized set of troubleshooting steps, acknowledging the customer's past attempts to resolve the issue and adjusting the current response. This not only demonstrates an understanding of the customer's experience but also builds trust and confidence in the support process.

## Financial services

Financial service RAG-enhanced chatbots can also assist with account inquiries, transaction issues, and personalized financial advice, drawing from a customer's transaction history and account details. When was the last time you tried to look up your interest rate on your credit card?

This seems like an easy question to answer, but for banks, it is often not as easy for a customer service agent to help with, as the data is buried in databases, PDF documents, and behind security measures. But once you have safely identified yourself online, a bank chatbot can use RAG to have access to all your financial documents and quickly answer 21.9%, while also providing related information, such as: You have only paid interest on your balance twice during your tenure as a customer. If you then need to talk about your mortgage account, you don't need to switch to a different agent, as the RAG-based chatbot can help you with that too! It can handle more complicated questions such as, How much more is the interest rate on my credit card than my mortgage? In addition, if you ask a follow-up question, it understands the context of the entire conversation and can converse about your financial accounts in a very human and natural way. This level of support can significantly enhance the customer experience, fostering loyalty and trust in the company's services much more than current systems can handle.

## Healthcare

In healthcare, RAG-powered chatbots can provide patient support by accessing medical records (with appropriate permissions) to offer personalized health advice or facilitate appointment scheduling. If a patient asks about managing their newly diagnosed diabetes, the chatbot can analyze their medical history, medications, and recent lab results to provide tailored advice on lifestyle changes, diet, and medication management. The chatbot can even schedule follow-up appointments with their healthcare provider and send reminders, creating a more comprehensive and engaging patient support experience. This level of personalized care can significantly improve patient outcomes, foster trust in the healthcare system, and reduce the burden on healthcare professionals.

RAG isn't just for chatbots, though. Let's discuss another area with significant RAG activity: automating reporting for data analysis.

## RAG for automated reporting

Companies that use RAG in combination with data analysis and reporting through its automated reporting capabilities are seeing significant improvements in their capabilities and the time it takes to perform the analysis. This innovative application of RAG serves as a bridge between the vast data lakes of unstructured data and the actionable insights that businesses need daily to drive key decisions and innovation. By utilizing RAG for automated reporting, companies can significantly streamline their reporting processes, enhance accuracy, and uncover valuable insights hidden within their data. Let's start with how it can be utilized in this environment.

### How RAG is utilized with automated reporting

While there are countless ways to set up this type of automated reporting, it is common to target areas where reports are relatively standard, making it easier to codify into a RAG environment. Often, you start with an automated report that is already in place from a code standpoint, but you can apply GenAI and/or RAG-like techniques to help make this automation even more effective. For example, a set of initial analysis questions can be fed to the RAG system without needing user input, with that content added to the initial report. This initial analysis can include not just charts and graphs, but commentary from a **large language model (LLM)** as well, all as an initial component in the RAG pipeline.

In many automated report scenarios, the automated report is often just the first step in helping the decision maker, the *user* of the automated report, understand the data. It is very common for the decision maker to ask for more analysis based on what they see in the automated report. Making your automated reports a part of a larger RAG system can essentially replace and/or accelerate the back-and-forth you typically have in these data analysis scenarios, helping important decision-makers gain access to key data analysis faster and make more effective decisions. You can make the report data, as well as the underlying data that it was generated from, available to the RAG system, which in turn allows non-technical staff to ask numerous and very wide-ranging questions of the data without having to wait for additional analysis to be performed by a data analyst. Additional data sources can be added to the sources available for the RAG system, enabling even more depth to the discussion and analysis, potentially far exceeding what could be provided in just a simple automated analysis.

Much of this data is hidden within unstructured data, which has proven to be a difficult type of data for companies to utilize. Let's talk about how RAG can help with unstructured data in support of automated reporting.

## Transforming unstructured data into actionable insights

The vast majority of data available to organizations today is unstructured, ranging from articles and research papers to social media feeds and web content. This unstructured nature makes it challenging to process and analyze through traditional data analysis means. RAG steps in as a powerful approach to parse through this data, extracting data from what has been traditionally difficult to reach data, creating initial drafts and summaries that can even be customized to the individual, and highlighting the most critical information for that person specifically based on their role or interests. RAG can serve as a much more sophisticated tool for users, compared to using an LLM directly, as it can replace many of the tasks they typically have to do manually when interacting with an LLM. This concept can also be applied within the world of data analysis and automated reporting, where many of the steps can be replicated in the RAG system, making them faster and more effective. This process not only saves time but also ensures that decision-makers can quickly grasp the essence of the data without getting bogged down by its volume.

It is important to note that when it comes to unstructured data, it is often a step in the indexing stage of RAG where this data is transformed into a new format that makes it more helpful to the overall RAG system. For example, a PDF might be extracted into a variety of different elements with different important levels. Titles and headers might be given more weight than paragraphs in their level of importance. Images might be detected as tables, which can then be extracted as a table summary, vectorized, and used later in the RAG system. It is these steps that make unstructured data more accessible to automated reporting.

When looking for areas in your organization to focus on for automated reporting, start with the areas where timely information is the most crucial. For instance, in market analysis, RAG can swiftly summarize news articles, financial reports, and competitor information, providing companies with a condensed view of market trends and dynamics. This rapid processing allows your users to react swiftly to market changes, capitalizing on opportunities or mitigating risks promptly for your company.

## Enhancing decision-making and strategic planning

The automated reporting and innovation scouting capabilities of RAG significantly enhance decision-making and strategic planning processes. By providing executives and strategists with concise, summarized reports and insights on industry trends, technological advancements, and competitive landscapes, RAG enables more informed

---

and strategic decision-making. By adding the RAG-oriented ability to ask additional questions about the data that generated these reports, the value for these users is taken much further.

RAG initiatives in this area hinge on their ability to quickly assimilate and analyze diverse datasets. This allows the companies using RAG for this purpose to adopt a more proactive approach to strategy formulation. Instead of being reactive to changes in the industry, businesses can anticipate shifts in the market or technology and adjust their strategies accordingly, ensuring they remain steps ahead of their competitors.

While the world of automated data analysis and reporting expands substantially due to RAG-based applications, another area of growth we are seeing is in generating product descriptions and recommendations dynamically on e-commerce websites, which we cover in the next section.

## E-commerce support

E-commerce is a key area that can benefit significantly from RAG applications. Let's review a couple of areas where RAG can be applied, starting with product descriptions.

### Dynamic online product descriptions

RAG's ability to generate personalized product descriptions is a game-changer for e-commerce businesses. By leveraging the power of RAG, companies can create highly targeted and persuasive product descriptions that resonate with individual customers, ultimately driving sales and fostering brand loyalty. RAG can produce personalized product descriptions or highlight features that are specifically tailored to the user's past behavior and preferences, taking into account RAG's ability to analyze vast amounts of customer data, including browsing history, past purchases, and even social media interactions.

For instance, let's say Rylee is a user who frequently purchases eco-friendly products. RAG can be used to emphasize the sustainability aspects of a product in its description when Rylee browses an e-commerce website. This results in an improved user experience for Rylee, as well as improving the likelihood of her purchasing that product. Now, let's say Rylee has previously purchased a product such as running shoes and frequently engages with content related to marathon training. When Rylee browses running shoes on an e-commerce site powered by RAG, the product descriptions can be dynamically generated to emphasize features such as cushioning, stability, and durability – all key factors for long-distance runners. The description might also include additional information relevant to Rylee's interests, such as how the shoes have been tested by marathon runners or have been designed to reduce the risk of common running injuries.

Moreover, RAG can be used to analyze customer reviews and feedback to identify the most frequently mentioned pros and cons of a product. This information can be incorporated into the product descriptions, providing potential buyers with a more balanced and informative overview of the item. By addressing common concerns or highlighting highly praised features, RAG-generated descriptions can help customers make more informed purchase decisions, reducing the likelihood of returns or negative reviews.

RAG can also be used to generate product descriptions in multiple languages, making it easier for businesses to expand into international markets. By training RAG on a diverse set of language data, companies can ensure that their product descriptions are culturally appropriate and resonate with customers in different regions.

Product descriptions are one area of e-commerce that is benefitting from RAG. Let's also talk about how recommendation engines are benefitting.

## Product recommendations for e-commerce sites

**Recommendation engines** are a significant use of AI in the business world. GenAI and RAG have the potential to make these recommendation engines even more effective. One of the key advantages of using RAG for product recommendations is its ability to analyze vast amounts of customer data, including browsing history, past purchases, search queries, and even social media interactions. By understanding a customer's unique interests, style preferences, and shopping habits, RAG can generate product recommendations that are not only relevant but also highly appealing to each individual. RAG can identify patterns and preferences at a much more in-depth level than previous methods to ultimately recommend products that the user is likely to find the most appealing. RAG allows us to go way beyond traditional recommendation engines, integrating a deeper understanding of individual user preferences, leading to more accurate and personalized recommendations.

For instance, let's say Aubri, one of our VIP customers, frequently purchases outdoor gear and has recently been browsing hiking boots on our e-commerce site. RAG can analyze Aubri's data and recommend not only the most suitable hiking boots based on her preferences and past purchases but also complementary products such as hiking socks, backpacks, and trekking poles. By presenting a curated selection of products that align with Aubri's interests, RAG can increase the likelihood of Aubri making a multi-item purchase and enhance the overall shopping experience.

Moreover, RAG can take product recommendations to the next level by considering factors beyond just a customer's purchase history. For example, RAG can analyze a customer's product reviews and ratings to gain insights into their satisfaction with

previous purchases. This information can be used to refine future recommendations, ensuring that customers are presented with products that not only match their interests but also meet their quality expectations.

The applications of RAG in personalizing product recommendations extend beyond the confines of e-commerce websites. By leveraging the power of RAG, companies can deliver highly targeted and engaging experiences across various touchpoints, including media, content platforms, and digital marketing campaigns.

As e-commerce continues to grow and evolve, the importance of personalized and compelling product descriptions cannot be overstated. By harnessing the power of RAG, businesses can create descriptions that not only inform but also persuade, ultimately driving sales and building long-lasting customer relationships. Much of this is powered by RAG's ability to search vast amounts of data. Next, let's discuss how these same search and data accessing capabilities can help a company to better utilize knowledge internally by its employees.

## Utilizing knowledge bases with RAG

RAG can access and utilize knowledge bases, whether they are internal or external. Let's start with **internal knowledge bases**.

### Searchability and utility of internal knowledge bases

Combining the concept of advanced information retrieval with the most advanced LLMs, it is no surprise that RAG is bringing significant advances to the area of internal search engines. The combination of advanced search and intelligence capabilities is transforming corporate operations by allowing us to access data in much more sophisticated ways, better leveraging the vast amounts of data that businesses accumulate. This transformative technology not only simplifies the retrieval of information but also enriches the quality of the data presented, making it a cornerstone for decision-making and operational efficiency across multiple sectors.

RAG significantly enhances the searchability and utility of internal knowledge bases, acting as a catalyst for improved information access and management. Internal knowledge bases, which include a wide range of documents in various unstructured formats (such as PDFs, Word, Google Docs, spreadsheets, slide decks, and more) are often underutilized due to their vastness and the difficulty in extracting them. RAG can address this challenge in several ways by extracting the data and processing it in many advanced ways. For example, generating concise summaries of documents

makes it easier for employees to grasp the essence of the content without having to go through the entire document. Moreover, RAG can provide direct answers to queries by analyzing the content of these resources and using the generative power of LLMs to provide coherent and accurate answers from content that can be buried in millions of pages of unstructured data. This direct answer capability is particularly useful for quickly resolving specific inquiries, having the potential to make significant reductions in the time employees spend searching for information.

The implementation of RAG in internal search engines also facilitates a more organized and efficient way of managing knowledge. By enabling better categorization and retrieval of information, employees can access relevant data more swiftly, leading to a more streamlined workflow. This is especially beneficial in fast-paced environments where time is of the essence and quick access to accurate information can significantly impact decision-making and project outcomes.

While internal knowledge is a key strategic advantage for most companies, external knowledge can also play a major role in maintaining a competitive edge in your industry. Let's review how RAG is enabling companies to better harness external data for use by their employees.

## Expanding and enhancing private data with general external knowledge bases

Beyond internal resources, RAG extends its benefits to general **external knowledge bases**, crucial in areas requiring up-to-date knowledge of laws, regulations, and industry standards, compliance, research and development, medical fields, academia, and patent-related industries. In these domains, the volume of information is not only vast but is constantly released and changed, making it challenging to stay current. RAG simplifies this task by retrieving and summarizing pertinent information from these extensive databases. For instance, in the legal and compliance sectors, RAG can quickly sift through thousands of documents to find relevant case laws, regulations, and compliance guidelines, significantly reducing the time legal professionals and compliance officers spend on research.

Similarly, in research and development, RAG can accelerate the process by providing researchers with quick access to existing and up-to-date studies, patents, and technical documents relevant to their work. This capability is invaluable for avoiding duplication of efforts and for sparking new ideas by building on existing knowledge. In the medical field, RAG's ability to pull up relevant case studies, research papers, and treatment guidelines can assist healthcare professionals in making informed decisions, helping to improve the care of patients.

One major issue with using an LLM such as ChatGPT to do your own research is that they can often give you fictitious, but convincing, information (called **hallucinations**). RAG is a great way to reduce these hallucinations significantly because it keeps the LLM component in your RAG pipeline grounded in real data, which reduces the time you have to spend verifying responses. You can add multiple calls to the LLM in your RAG pipeline that can not only help to answer your research questions but can verify that the response is highly relevant to the original question before providing the response back to you. You can also tailor the output to provide all supporting documentation and citations.

External data harnessing with RAG comes in other options than just accessing general knowledge bases. Let's discuss the concept of innovation scouting and how organizations are using RAG to further their innovating and more quickly detect trends in their industry.

## Innovation scouting and trend analysis

Scanning and summarizing information from various quality sources can also play an instrumental role in innovation scouting and trend analysis. RAG can help companies identify emerging trends and potential areas for innovation that align with their specialization. This is particularly relevant in fast-evolving industries where staying ahead of the curve is essential for maintaining competitive advantage.

In the technology sector, RAG can analyze patents, tech news, and research publications to identify emerging technologies and innovation patterns. This allows companies to direct their R&D efforts more effectively, focusing on areas with high potential for growth and market disruption.

RAG is being used to expedite the process of identifying new research findings and potential drug development opportunities by constantly analyzing the latest releases from medical journals, clinical trial reports, and patent databases. This is accelerating the pace of innovation and helping pharmaceutical companies to more efficiently allocate their research budgets and resources.

The applications of RAG extend beyond innovation scouting and trend analysis. Another area where RAG is making significant strides is in content personalization for media and content platforms. In today's digital landscape, where users are inundated with an overwhelming amount of content, RAG offers a powerful solution to cut through noise and deliver highly targeted, personalized experiences.

Next, let's discuss this holistic approach to personalization that can lead to increased customer loyalty, higher conversion rates, and, ultimately, a more successful business.

## RAG for personalized recommendations in marketing communications

RAG can also represent a significant advancement for companies that adopt it to enhance user engagement and satisfaction across media, content platforms, and digital marketing campaigns. RAG can be used to create personalized product bundles or collections based on a customer's preferences, which can be advertised and included in digital campaigns targeting each customer. By analyzing a customer's data and identifying complementary products, RAG can suggest pre-assembled bundles that offer convenience and value.

RAG can also enhance the effectiveness of email marketing campaigns by providing personalized product recommendations directly in marketing communications. By analyzing a customer's data and tailoring product suggestions to their interests, e-commerce sites can create highly targeted and compelling email content that drives **click-through rates (CTRs)** and conversions.

As marketing continues to evolve and businesses seek new ways to engage customers, RAG's potential extends beyond personalized product recommendations on websites. RAG can be leveraged to create tailored experiences across various digital touchpoints, enhancing user engagement and fostering long-term customer loyalty.

Next, let's review how RAG's capabilities can be applied to another critical area for businesses: employee training and education.

## Training and education

RAG can be used by educational organizations, such as universities and secondary education. It can also be used by internal corporate training programs to keep their employees up to speed on vast amounts of ever-changing concepts they need their employees to be aware of. RAG can help generate or customize learning materials based on the specific needs, knowledge levels, and functions of the learners.

RAG has shown incredible promise for significantly improving learning and development in corporate environments. It is often a challenging task for businesses to keep their employees updated with the latest industry knowledge and skills, especially given today's rapid pace of change across many industries. RAG helps address this challenge by offering personalized learning experiences that adapt to each employee's individual needs, learning style, and pace with their content.

RAG can analyze an employee's role, experience level, and learning history to curate a customized learning path that focuses on the most relevant and essential skills for their specific position. This personalized approach ensures that employees receive training

that directly contributes to their professional growth and aligns with the company's objectives.

Moreover, RAG can also be used to generate interactive learning materials, such as quizzes, case studies, and simulations, tailored to each employee's learning style and progress. This adaptive learning approach keeps employees engaged and motivated, as they receive content that challenges them at the right level and provides immediate feedback on their performance.

RAG's ability to quickly summarize and present relevant information from vast knowledge bases also makes it an invaluable tool for on-the-job learning and performance support. Employees can use the RAG system to quickly access the information they need to solve problems, make decisions, and complete tasks more efficiently. This **just-in-time (JIT)** learning approach reduces the need for lengthy, formal training sessions and empowers employees to take control of their learning and development.

In addition to personalized learning, RAG also enables more effective collaboration and knowledge sharing among employees. By analyzing the knowledge and skills of each individual, RAG can identify **subject matter experts (SMEs)** within the organization and facilitate connections between employees who can learn from one another. This internal knowledge flow promotes a culture of continuous learning and helps organizations retain and leverage their collective expertise.

As we've seen, RAG has immense potential to transform various aspects of a company's operations, from personalized learning and employee development to customer engagement and process optimization. However, the successful implementation of RAG requires a strategic approach that aligns with the company's goals and priorities.

One critical aspect of many RAG applications mentioned previously is the inclusion of relevant data and sources in generated responses. For example, when using RAG to crawl legal documents or scientific research papers, it's crucial to cite the sources to provide credibility and support for the information presented.

## Code lab 3.1 – Adding sources to your RAG

Many of the aforementioned applications mentioned include an element of adding more data to the response. For example, you are likely going to want to quote the sources of your response if you have a RAG pipeline that crawls legal documents or scientific research papers as a part of the efforts described in the *Expanding and enhancing private data with general external knowledge bases* or *Innovation scouting and trend analysis* sections.

Starting with the code from the previous *Chapter*, we need to introduce these elements to this code, which I will walk through and explain what is happening:

```
from langchain_core.runnables import RunnableParallel
```

This is a new import: the `RunnableParallel` object from LangChain runnables. This introduces the concept of running the retriever and question in parallel. This can improve performance by allowing the retriever to fetch the context while the question is being processed simultaneously:

```
rag_chain_from_docs = (
 RunnablePassthrough.assign(context=(
 lambda x: format_docs(x["context"])))
 | prompt
 | llm
 | StrOutputParser()
)

rag_chain_with_source = RunnableParallel(
 {"context": retriever,
 "question": RunnablePassthrough()}
).assign(answer=rag_chain_from_docs)
```

Compare this to our original `rag_chain` object:

```
rag_chain = (
 {"context": retriever | format_docs,
 "question": RunnablePassthrough()}
 | prompt
 | llm
 | StrOutputParser()
)
```

In the original code, `rag_chain` is constructed using a dictionary that combines the retriever and `format_docs` function for “context”, and `RunnablePassthrough()` for “question”. This dictionary is then piped (`|`) through `prompt`, `llm`, and `StrOutputParser()`.

In this new version, called `rag_chain_from_docs`, the construction of `rag_chain` is split into two parts:

- The `rag_chain_from_docs` chain is created using `RunnablePassthrough.assign()` to format the documents retrieved from the context. It then pipes the formatted context through `prompt`, `llm`, and `StrOutputParser()`.

- The `rag_chain_with_source` chain is created using `RunnableParallel()` to run the retriever and `RunnablePassthrough()` in parallel for “context” and “question”, respectively. The result is then assigned to “answer” using `rag_chain_from_docs`.

The main difference in functionality between these two approaches is that the new approach separates the retrieval of the context from the formatting and processing of the retrieved documents. This allows for more flexibility in handling the retrieved context before passing it through the prompt, LLM, and output parser.

Finally, we have to change the name of the chain that we pass the user query to match the new chain name, `rag_chain_with_source`. As we did in the past, we call the invoke method, `rag_chain_with_source.invoke()`, passing it the question, which triggers the parallel execution of the retriever and question, followed by the formatting and processing of the retrieved context using `rag_chain_from_docs` to generate the final answer:

```
rag_chain_with_source.invoke(
 "What are the Advantages of using RAG?")
```

The output will look like this:

```
{'context': [Document(page_content='Can you imagine what you could
do with all of the benefits mentioned above...', metadata={'source':
'https://kbourne.github.io/chapter1.html'}),
 Document(page_content='Maintaining this integration over time,
especially as data sources evolve or expand...', metadata={'source':
'https://kbourne.github.io/chapter1.html'})],...}
```

This is more code-looking than our previous final output, but it contains all of the information you would provide back to the user to indicate the source of the response you provided them. For many use cases, this sourcing of the material is very important in helping the user understand why the response was what it was, in fact-checking it, and in building off it if they have anything else they need to add.

Note the metadata source listed after each `page_content` instance, which is what you would provide as the source link. In situations where you have multiple documents in the results, this could be different across each individual document returned in the retrieval step, but we only use one document here.

When you're developing with **retrieval-augmented generation (RAG)**, it is essential to understand the intricacies of each component, how they can be integrated, and the technologies that empower these systems.

## Key component overview

This chapter delves into the intricate components that make up a RAG system. Let's start with an overview of the entire system.

Earlier, we introduced the three main stages of the RAG system from a technical standpoint (see *Figure 38.1*):

- **Indexing**
- **Retrieval**
- **Generation**

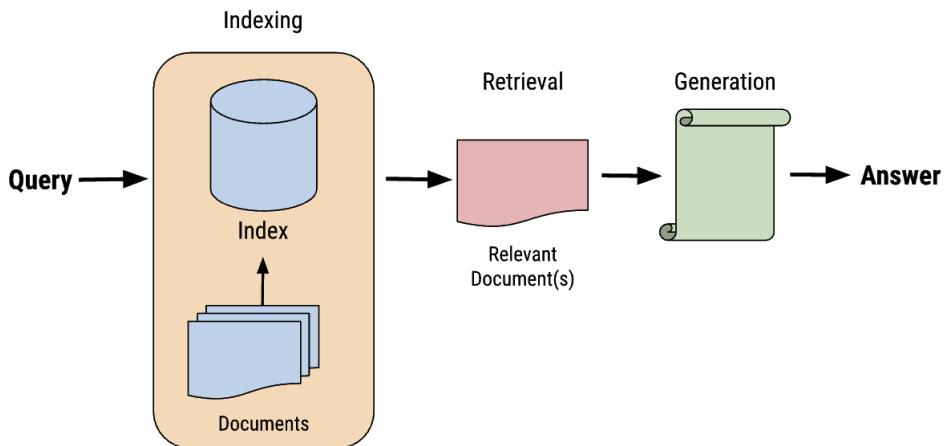


Figure 38.1 – The three stages of a RAG system

We will continue to build off this concept, but we will also introduce the practical aspects of development that are required for building an application. These include prompting, defining your **large language model (LLM)**, the UI, and an evaluation component. Later chapters will cover each of those areas even further. All of this will be done with code so that you can tie the conceptual framework we'll discuss directly with the implementation. Let's start with indexing.

## Indexing

The first stage in the RAG system we will examine more closely is indexing. Note that we are skipping the setup, where we install and import packages, as well as set up OpenAI and related accounts. That is a typical step in every generative artificial intelligence (AI) project, not just RAG systems.

Indexing occurs as the first main stage of RAG. As *Figure 38.2* indicates, it is the step after the user query:

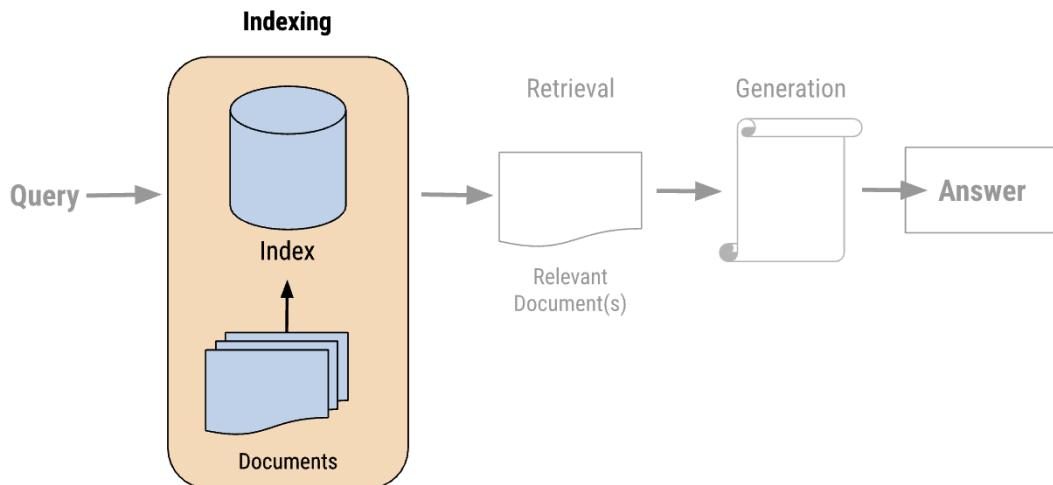


Figure 38.2 – The Indexing stage of RAG highlighted

In our code from the previous *Chapter*, *Indexing* is the first section of code you see. This is the step where the data you are introducing to the RAG system is processed. As you can see in the code, the *data* in this scenario is the web document that is being loaded by *WebBaseLoader*. This is the beginning of that document (*Figure 38.3*):

### Introduction to Retrieval Augmented Generation (RAG)

Date: March 10, 2024 | Estimated Reading Time: 15 min | Author: Keith Bourne

In the rapidly evolving field of artificial intelligence, Retrieval-Augmented Generation (RAG) is emerging as a significant addition to the Generative AI toolkit. RAG harnesses the strengths of Large Language Models (LLMs) and integrates them with internal data, offering a method to enhance organizational operations significantly. This book delves into the essential aspects of RAG, examining its role in AI-driven innovation and how it can be used to drive business success.

As it progresses, the book outlines the potential of RAG in business, suggesting how it can make AI applications smarter, more responsive, and aligned with organizational objectives. RAG is positioned as a key facilitator of customized, efficient, and insightful AI solutions, bridging the gap between Generative AI's potential and specific business needs. This exploration of RAG encourages readers to unlock the full potential of their corporate data, paving the way for an era of AI-driven innovation.

### What You Can Expect to Learn

Expect to launch a comprehensive journey to understand and effectively incorporate Retrieval-Augmented Generation (RAG) into AI systems. You'll explore a broad spectrum of essential topics, including the core principles of RAG, the integration of LLMs and internal data, vector search techniques, prompt engineering and design, and the use of AI agents for RAG applications. Alongside methods for evaluating and visualizing RAG outcomes. Through practical, working code examples utilizing the latest tools and technologies like LangChain and Chroma's vector database, you'll gain hands-on experience in implementing RAG in your project.

At the outset, you'll delve into the core principles of RAG, appreciating its significance in the broader landscape of Generative AI. This foundational knowledge equips you with the perspective needed to discern how RAG approaches are designed and why they succeed, paving the way for innovative solution development and problem-solving in AI.

You'll learn how to integrate Large Language Models (LLMs) and internal data to bolster organizational operations. By learning about the intricacies of this integration, particularly the process of vectorization, including the creation and management of vector databases for efficient information retrieval, you'll gain crucial skills for navigating and harnessing vast data landscapes effectively in today's data-driven environments.

Gain expertise in vector search techniques, an essential skill set for identifying pertinent data within extensive datasets. Coupled with this, you'll learn strategies for prompt engineering and design, ensuring that you can craft queries that elicit precise and relevant AI responses.

Explore how AI agents play a pivotal role in RAG applications, facilitating sophisticated data interaction and retrieval tasks. You'll also learn methods for evaluating and visualizing RAG implementation outcomes, providing a framework for assessing performance and impact critically.

Throughout the journey, you'll engage in practical, hands-on learning, guided through the use of cutting-edge tools like LangChain and Chroma's vector database, supported by real, working code examples. These exercises will challenge you to apply what you've learned, forcing you to implement RAG in AI systems, providing a rich learning experience.

Case studies and coding exercises strategically interspersed throughout your learning path highlight the application of RAG in various real-world scenarios. These insights into addressing common and complex challenges prepare you to navigate the application of RAG across diverse settings with confidence. The code will build off the same starting use case provided in the next chapter. For each topic that relates to code, we will add code that shows how that topic impacts the RAG pipeline, giving you an in-depth understanding about how your coding choices can impact the capabilities of your RAG system.

You'll also explore optimization strategies for data retrieval and enhancing the interoperability of AI-generated content. These insights are pivotal for improving the usability and effectiveness of AI applications, ensuring they are more aligned with strategic business objectives and user needs.

As you progress, you'll gain a deeper understanding of how RAG can revolutionize AI applications, making them more intelligent, responsive, and tailored to specific requirements. The potential of RAG to facilitate personalized, efficient, and insightful AI solutions is thoroughly examined, bridging the theoretical and practical divides.

Throughout this learning experience, a spirit of exploration and experimentation is encouraged, aiming to unlock the full potential of data through RAG, fostering innovation, and advancing the domain of AI-driven solutions. By the end, you will have gained comprehensive knowledge and practical skills in RAG, equipping you to contribute to the evolution of AI technologies and applications in your business and beyond.

Figure 38.3 – The web page that we process

In the previous *Chapter*, you may have noticed that the code in the latter stages, *Retrieval* and *Generation*, is used after the user query is passed to the chain. This is done in *real time*, meaning that it happens at the time that the user interacts with it. Indexing, on the other hand, typically happens well before the user interacts with the RAG application. This aspect of indexing makes it a very different step from the other two stages, with the flexibility of being run at a different time than when the application is used. This is called **pre-processing offline**, meaning that this step is done before the user has even opened the application. There are instances where indexing can be done in real time as well, but that is much less common. For now, we will focus on the much more common step of pre-processing offline.

The following code is our **document extraction**:

```
loader = WebBaseLoader(
 web_paths="https://kbourne.github.io/chapter1.html",
 bs_kwarg=dict(
 parse_only=bs4.SoupStrainer(
 class_=("post-content", "post-title",
 "post-header")
)
),
)
docs = loader.load()
```

In this extract, we are ingesting a web page. But imagine if this was pulling data in from a PDF or Word document, or other forms of unstructured data. As discussed earlier, unstructured data is a very popular data format in RAG applications. Historically, unstructured data has been very difficult for companies to access relative to structured data (from SQL databases and similar applications). But RAG has changed all of this, and companies are finally realizing how to significantly tap into this data.

Regardless of what type of data you are pulling in, it all goes through a similar process, as shown in *Figure 38.4*:

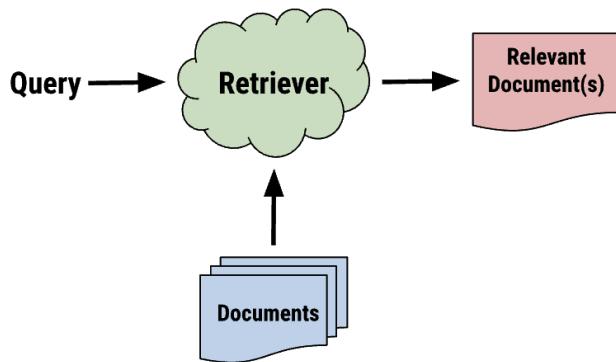


Figure 38.4 – Creating a retriever in the Indexing stage of the RAG process

The document loader from the code fills the **Documents** component of this so that they can be retrieved later using the user query. But in most RAG applications, you must turn that data into a more searchable format: vectors. We will talk more about vectors in a moment, but first, to get your data into vector format, you must apply **splitting**. In our code, that is this section:

```
text_splitter = SemanticChunker(OpenAIEmbeddings())
splits = text_splitter.split_documents(docs)
```

Splitting breaks your content into digestible chunks that can be vectorized. Different vectorization algorithms have different requirements for the maximum size of the content you can pass. In this case, we are using the `OpenAIEmbeddings()` vectorizer, which currently has a max input of 8191 tokens.

**Note**

In the OpenAI API, the text is tokenized using a byte-level **byte pair encoding (BPE)** vocabulary. This means the raw text is split into subword tokens rather than individual characters. The number of tokens that are consumed for a given input text depends on the specific content as common words and subwords are represented by single tokens while less common words may be split into multiple tokens. On average, one token is approximately four characters for English text. However, this is just a rough estimate and can vary significantly based on the specific text. For example, short words such as *a* or *the* would be a single token, while a long, uncommon word might be split into several tokens.

These digestible chunks need to be smaller than that 8191 token limit, and other embedding services have their token limits. If you're using a splitter that defines a chunk size and a chunk overlap, keep the chunk overlap in mind for that token limit as well. You have to add that overlap to the overall chunk size to be able to determine how large that chunk is. Here is an example of using a **RecursiveCharacterTextSplitter**, where the chunk size is 1000 and the chunk overlap is 200:

```
text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)
```

Expanding the chunk overlap is a common approach to ensuring that no context is lost between chunks. For example, if a chunk happens to cut an address in half in a legal document, it is unlikely you will find that address if you search for it. But with chunk overlap, you can account for issues like that.

The last part of the *Indexing* stage is defining the vector store and adding the embeddings, built from your data splits to that vector store. You see it here in this code:

```
vectorstore = Chroma.from_documents(
 documents=splits,
 embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
```

In this case, we use **Chroma DB** as a vector database (or store), pass it the splits, and apply OpenAI's embeddings algorithm. As with the other indexing steps, all of this is most often done *offline* before the application is accessed by the user. These vector-based embeddings are stored in this vector database for querying and retrieval in the future. Chroma DB is just one of many databases that can be used here. The OpenAIEmbeddings API is just one of many vectorizing algorithms that can be used here as well.

Going back to our diagram of the *Indexing* process, *Figure 38.5* is an even more accurate representation of what it looks like:

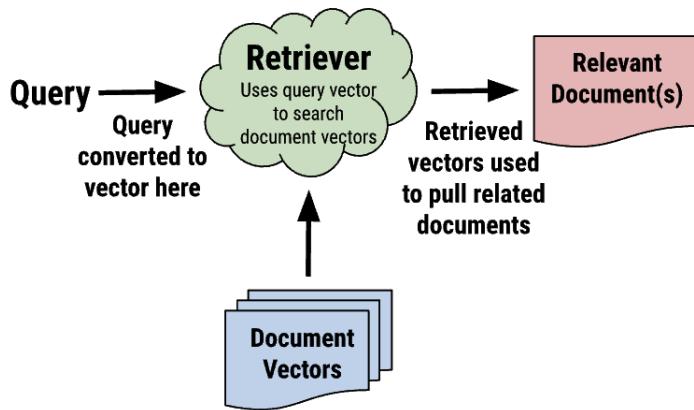


Figure 38.5 – Vectors during the Indexing stage of the RAG process

You might be wondering why we aren't calling the step where we define the *retriever* a part of the *Retrieval* step. This is because we are establishing this as the mechanism that we retrieve from, but we do not apply the retrieval until later during the retrieval step as a result of the user submitting their user query. The *Indexing* step focuses on building the infrastructure that the other two steps work from, and we are indeed indexing the data so that it can be retrieved later. At the end of this part of the code, you have a retriever ready and waiting to receive a user query when the process starts. Let's talk about the parts of the code that will use this retriever – the retrieval and generation steps!

## Retrieval and generation

In our RAG application code, we have combined the *Retrieval* and *Generation* stages. From a diagram standpoint, this looks like what's shown in *Figure 38.6*:

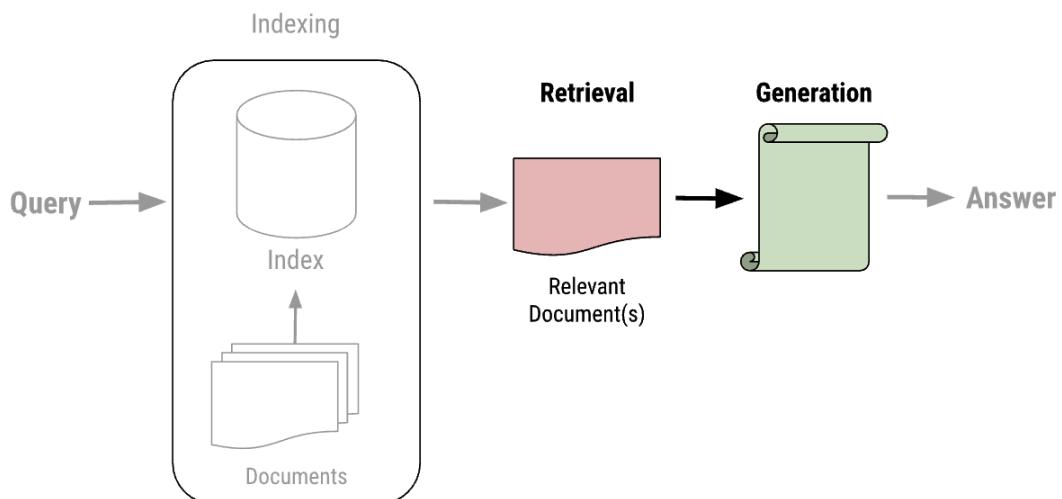


Figure 38.6 – Vectors during the Indexing stage of the RAG process

While retrieval and generation are two separate stages serving two important functions of the RAG application, they are combined in our code. When we invoke `rag_chain` as the last step, it is stepping through both of these stages, making them difficult to separate when talking about the code. But conceptually, we will separate them here, and then show how they pull them together to process the user query and provide an intelligent generative AI response. Let's start with the retrieval step.

## Retrieval focused steps

In the complete code (which can be found in the previous *Chapter*), there are only two areas in this code where actual retrieval takes place or is processed. This is the first:

```

Post-processing
def format_docs(docs):
 return "\n\n".join(doc.page_content for doc in docs)

```

The second can be found as the first step within the RAG chain:

```

{ "context": retriever | format_docs, "question":
RunnablePassthrough() }

```

When the code is initiated, it runs in this order:

```

rag_chain.invoke("What are the Advantages of using RAG?")

```

The chain is invoked with the user query and runs through the steps we defined in the chain here:

```
rag_chain = (
 {"context": retriever | format_docs,
 "question": RunnablePassthrough() }
 | prompt
 | llm
 | StrOutputParser()
)
```

With this chain, the user query is passed to the first link, which passes that user query into the retriever we defined earlier, where it performs a similarity search to match the user query against the other data in the vector store. At this point, we have a retrieved list of content strings that is contextually similar to the user query.

However, as shown in the previous *Chapter*, there is a bit of a glitch in our retrieval steps due to the formatting of the tools we are using. The {question} and {context} placeholders both expect strings, but the retrieval mechanism we use to fill in the context is a long list of separate content strings. We need a mechanism to turn that list of content pieces into the string format that the prompt in the next chain link is expecting.

So, if you look closely at the code for the retriever, you may notice that the retriever is actually in a mini-chain (retriever | format\_docs), indicated by the pipe (|) symbol, so the output of the retriever is passed right into the format\_docs function shown here:

```
def format_docs(docs):
 return "\n\n".join(doc.page_content for doc in docs)
```

Let's consider this a post-processing step in the *Retrieval* stage. The data has been retrieved, but it is not in the right format, so we are not done. The format\_docs function completes the task and returns our content in the proper format.

However, this only provides us with {context}, one of the input variable placeholders. The other placeholder we need to hydrate our prompt is the {question} placeholder. However, we do not have the same formatting problem with the *question* that we had with the *context* since the *question* is already a string. So, we can use a convenient object called RunnablePassThrough that, as its name suggests, passes the input (the *question*) through as-is.

If you take the entire first chain link in its entirety, this is essentially performing the retrieval step, formatting its output, and pulling it all together in the proper format to pass on to the next step:

```
{"context": retriever | format_docs, "question": RunnablePassthrough()}
```

But wait a minute. If you're doing a vector search, you need to convert the user query into a vector, right? Did we not say that we are taking the mathematical representation of the user query and measuring the distance to other vectors, finding which ones are closer? So, where does that happen? The retriever was created from a method of the vector store:

```
retriever = vectorstore.as_retriever()
```

The vector store that this was generated from is a Chroma vector database that was declared using the `OpenAIEmbeddings()` object as its embedding function:

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

That `.as_retriever()` method has all of the functionality built in to take that user query, convert it into an embedding that matches the embedding format of the other embeddings, and then run the retrieval process.

#### Note

Because this is using the `OpenAIEmbeddings()` object, it sends your embeddings to the OpenAI API and you will incur charges from this. In this case, it is just one embedding; with OpenAI, this currently costs \$0.10 per 1M tokens. So, for the `What are the Advantages of using RAG?` input, which is ten tokens according to OpenAI, this is going to cost a whopping \$0.000001. That may not seem like a lot, but we want to be completely transparent when there is any cost involved!

That concludes our *Retrieval* stage, with an output that is properly formatted for the next step – the prompt! Next, we'll discuss the *Generation* stage, where we utilize the LLM to take the final step of generating a response.

## Generation stage

The *Generation* stage is the final stage and is where you will use the LLM to generate the response to the user query based on the content you retrieved in the *Retrieval* stage. But before we can do this, we have to do a little bit of preparation work. Let's walk through this.

Overall, the *Generation* stage is represented by two parts of the code, starting with the prompt:

```
prompt = hub.pull("jclemens24/rag-prompt")
```

Then, we have the LLM:

```
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
```

With the prompt and LLM defined, these components are used in the RAG chain:

```
| prompt
| llm
```

Note that the question section was bolded in both the *Retrieval* and the *Generation* stages. We already noted how it is used in the *Retrieval* stage as the basis for what the similarity search is run against. Now, we will show how it is used again when integrating it into a prompt that is fed to the LLM for generation.

## Prompting

**Prompts** are a fundamental part of any generative AI application, not just RAG. When you start talking about prompts, particularly with RAG, you know LLMs are going to be involved soon after. But first, you must create and prepare a proper prompt for our LLM. In theory, you could write your prompt, but I wanted to take this chance to teach you this very common development pattern and get you used to using it when you need it. In this example, we'll pull the prompt from the **LangChain Hub**.

LangChain describes its Hub as a place to “*discover, share, and version control prompts*.” Other users of the hub have shared their polished prompts here, making it easier for you to build off common knowledge. It is a good way to start with prompts, pulling down pre-designed prompts and seeing how they are written. But you will eventually want to move on to writing your own, more customized prompts.

Let's talk about what the purpose of this prompt is in terms of the retrieval process. The “prompt” is the next link in the chain after the Retrieval stage we just discussed. You can see it here in `rag_chain`:

```
rag_chain = (
 {"context": retriever | format_docs,
 "question": RunnablePassthrough()
 | prompt
 | llm
 | StrOutputParser()
)
```

Staying true to the LangChain pattern, the inputs of the prompt are the outputs of the previous step. You can see these inputs at any time by printing them out like this:

```
prompt = hub.pull("jclemens24/rag-prompt")
prompt.input_variables
```

This results in the following output:

```
['context', 'question']
```

This matches what we defined in the previous step:

```
{"context": retriever | format_docs,
"question": RunnablePassthrough()}
```

Printing out the entire prompt object, using `print(prompt)`, shows that there is much more than just the text prompt and input variables:

```
input_variables=['context', 'question']
messages=[HumanMessagePromptTemplate(prompt=PromptTemplate(input_
variables=['context', 'question'], template="You are an assistant
for question-answering tasks. Use the following pieces of retrieved-
context to answer the question. If you don't know the answer, just
say that you don't know.\nQuestion: {question} \nContext: {context} \
nAnswer:"))]
```

Let's unravel this a bit further, starting with the input variables. These are the variables we just discussed, that this particular prompt takes as input. These can vary depending on the prompt. There is a `messages []` list, but in this case, there is only one message in the list. This message is an instance of `HumanMessagePromptTemplate`, which represents a specific type of message template. It is initialized with a `PromptTemplate` object. The `PromptTemplate` object is created with the specified `input_variables` and a template string. Again, `input_variables` are `context` and `question`, and you can see where they are placed in the template string:

```
template="You are an assistant for question-answering tasks. Use the
following pieces of retrieved-context to answer the question. If
you don't know the answer, just say that you don't know.\nQuestion:
{question} \nContext: {context} \nAnswer:"
```

The `{question}` and `{context}` placeholders will be replaced with the actual values of the `question` and `context` variables when the prompt is used in the chain. The output of this chain link is the string template that was filled in with `{question}` and `{context}` from the previous retrieval step.

The last part is simply `Answer:` with nothing after that. This prompts the LLM for an answer and is a prevalent pattern that works well for LLM interactions to elicit an answer.

In short, a prompt is an object that is plugged into your LangChain chain with inputs to fill a prompt template, generating the prompt that you will pass to an LLM for inference. This is essentially a preparation stage for the *Generation* stage of the RAG system.

In the next step, we will pull in the LLM, the brains behind the whole operation!

## Defining your LLM

With the prompt template selected, we can select an LLM, a central component for any RAG application. The following code shows the LLM model as the next chain link in `rag_chain`:

```
rag_chain = (
 {"context": retriever | format_docs,
 "question": RunnablePassthrough() }
 | prompt
 | llm
 | StrOutputParser()
)
```

As discussed previously, the output of the previous step, which was the `prompt` object, is going to be the input of the next step, the LLM. In this case, the prompt will *pipe* right into the LLM with the prompt we generated in the previous step.

Above `rag_chain`, we define the LLM we want to use:

```
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
```

This is creating an instance of the `ChatOpenAI` class from the `langchain_openai` module, which serves as an interface to OpenAI's language models, specifically the GPT-4o model. LLMs are typically fed a prompt using the `invoke` method, and you could call this directly in the code here by adding the following:

```
llm_only = llm.invoke("Answering in less than 100 words,
 what are the Advantages of using RAG?")
print(llm_only.content)
```

Doing it this way, you are asking the LLM directly for the answer.

If you run the preceding code, it will give you the response from GPT-4o, which will know about RAG. But for comparison, what if we changed it to GPT3.5? Here is the response I received when using ChatGPT 3.5:

RAG (Red, Amber, Green) status reporting allows for clear and straightforward communication of project progress or issues. It helps to quickly identify areas that need attention or improvement, enabling timely decision-making. RAG status also provides a visual representation of project health, making it easy for stakeholders to understand the current situation at a glance. Additionally, using RAG can help prioritize tasks and resources effectively, increasing overall project efficiency and success.'

Uh-oh! ChatGPT 3.5 doesn't know about RAG! At least not in the context we are talking about. This highlights the value of using RAG to add your data. The most recent cutoff date for ChatGPT 3.5 was January 2022. The generative AI-focused concept of RAG must not have been popular enough for it to instantly know what I was referring to with the RAG acronym.

Using RAG, we can augment its knowledge and utilize the LLM's other skills of summarizing and finding data to have a more successful result overall. But try changing this to the question answering in less than 100 words, what are the Advantages of using Retrieval Augmented Generation (RAG)? and see what results you can get. Try it with a newer model that likely has more information about RAG applications in its training data. You will likely get a better response because the data that the LLM was trained on has a more recent cutoff date!

But instead of calling the LLM directly, we pass it the prompt we have structured using the *Retrieval* stage and can get a much more informed answer. You could end the chain here and the output of your chain would be what's returned from the LLM. In most cases, this is not just the text you might see when you type something into ChatGPT – it is in JSON format and has a lot of other data included with it. So, if you want a nicely formatted string output reflecting the LLM's response, you have one more chain link to pipe the LLM response into: the `StrOutputParser()` object. The `StrOutputParser()` object is a utility class in LangChain that parses the key output of the language model into a string format. Not only does it strip away all the information you did not want to deal with right now, but it ensures that the generated response is returned as a string.

And of course, the last line of code is the line that kicks everything off:

```
rag_chain.invoke("What are the Advantages of using RAG?")
```

After the *Retrieval* stage, this user query is used a second time as one of the input variables for the prompt that is passed to the LLM. Here, `What are the advantages of using RAG?` is the string that's passed into the chain.

As we discussed in the previous *Chapter*, in the future, this prompt will include a query that comes from a UI. Let's discuss the UI as another important component of the RAG system.

## UI

At some point, to make this application more professional and usable, you must add a way for regular users who do not have your code to enter their queries directly and see the results. The UI serves as the primary point of interaction between the user and the system and therefore is a critical component when building a RAG application. Advanced interfaces might include **natural language understanding (NLU)** capabilities to interpret the user's intent more accurately, a form of **natural language processing (NLP)** that focuses on the understanding part of natural language. This component is crucial for ensuring that users can easily and effectively communicate their needs to the system.

This begins with replacing this last line with a UI:

```
rag_chain.invoke("What are the Advantages of using RAG?")
```

This line would be replaced with an entry field for the user to submit a text question, rather than a set string that we pass it in, as shown here. This also includes displaying the resulting response from the LLM in a more user-friendly interface, such as in a nicely designed screen.

When an application is loaded for a user, they will have some way to interact with it. This is typically facilitated through an interface that can range from simple text input fields on a web page to more complex voice recognition systems. The key is to accurately capture the intent of the user's query in a format that can be processed by the system. One obvious advantage of adding a UI is that it allows your users to test the results of other queries. A user could enter any query they want and see what the result is.

## Pre-processing

As we discussed, even though the user just enters a question such as `What is Task Decomposition?` in the UI, after that question is submitted, there is pre-processing that often occurs to make that query more LLM-friendly. This is primarily done in the prompt, which also gets help from many of the other functions. But all of this happens behind the scenes and not in the view of the user. All they will see in this scenario is the final output displayed in a user-friendly way.

## Post-processing

Even after the LLM has returned the response, this response is often post-processed before it is shown to the user.

Here's what an actual LLM output looks like:

```
AIMessage(content="The advantages of using RAG include improved accuracy and relevance of responses generated by large language models, customization and flexibility in responses tailored to specific needs, and expanding the model's knowledge beyond the initial training data.")
```

As a last step in the chain, we pass that through `StrOutput Parser()` to parse out just the string:

```
'The advantages of using RAG (Retrieval Augmented Generation) include improved accuracy and relevance, customization, flexibility, and expanding the model's knowledge beyond the training data. This means that RAG can significantly enhance the accuracy and relevance of responses generated by large language models, tailor responses to specific needs, and access and utilize information not included in initial training sets, making the models more versatile and adaptable.'
```

That is certainly better than the previous step's output, but this is still displaying in your notebook. In a more professional application, you will want to display this on a screen in a way that is friendly for the user. This will depend on the intentions of your application and will vary significantly across RAG systems.

## Output interface

For a full UI, this string will be passed to the interface that displays the message that's returned to the chain. This interface can be very simple, like what you can see with ChatGPT in *Figure 38.7*:

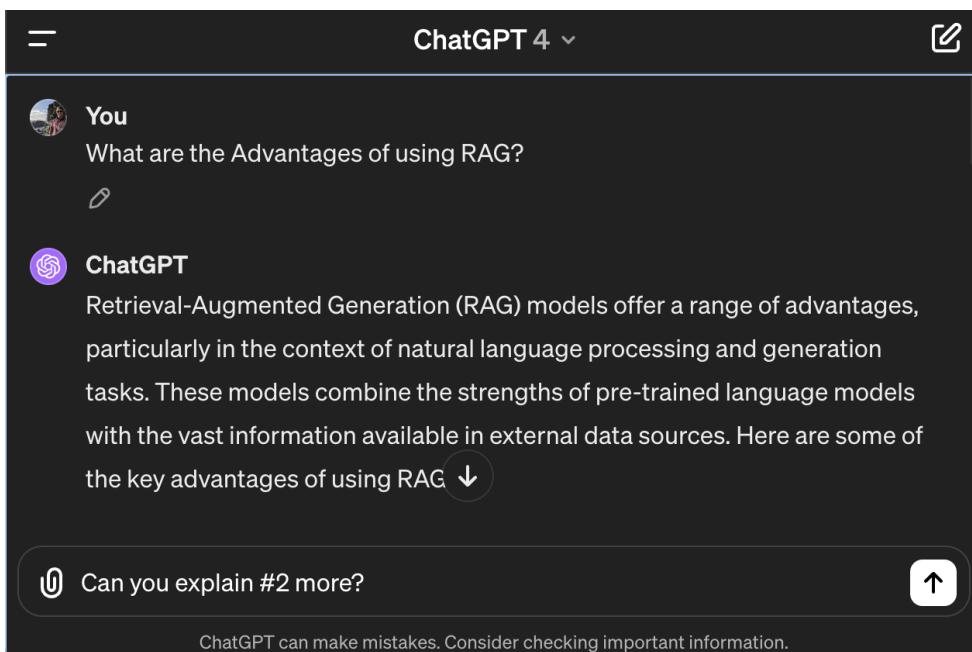


Figure 38.7 – The ChatGPT 4 interface

You could also build something more robust that is more suitable for your particular target user group. If it is meant to be more conversational, the interface should also be designed to facilitate further interaction. You could give users options to refine their queries, ask follow-up questions, or request additional information.

Another common feature in the UI is the collection of feedback on the usefulness and accuracy of the response. This can be used to continuously improve the system's performance. By analyzing user interactions and feedback, the system can learn to better understand user intent, refine the vector search process, and enhance the relevance and quality of the generated responses. This leads us to our last key component: evaluation.

## Evaluation

The evaluation component is essential for assessing and improving the RAG system's performance. While there are many common practices for evaluation, the most effective evaluation system will focus on what is most important for your users and provide an evaluation for improving those features and capabilities. Often, this involves analyzing the system's outputs using various metrics, such as accuracy, relevance, response

time, and user satisfaction. This feedback is used to identify areas of improvement, and guide adjustments in the system's design, data handling, and LLM integration. Continuous evaluation is crucial for maintaining high-quality responses and ensuring that the system meets users' needs effectively.

As mentioned previously, you can also collect user feedback in various ways, including qualitative data (entry forms with open-ended questions) or quantitative (true/false, ratings, or other numerical representations) on the usefulness and accuracy of the response. A thumbs up/down is often used to get a quick feedback response from the user and gauge the general effectiveness of the application among many users.

Chapter

# 41

# Integrating RAG with AI Agents and LangGraph

One call to an **large language model (LLM)** can be powerful, but put your logic in a loop with a goal toward achieving a more sophisticated task and you can take your **retrieval-augmented generation (RAG)** development to a whole new level. That is the concept behind **agents**. The past year of development for LangChain has focused significant energy on improving support for *agentic* workflows, adding functionality that enables more precise control over agent behavior and capabilities. Part of this progress has been in the emergence of **LangGraph**, another relatively new part of LangChain. Together, agents and LangGraph pair well as a powerful approach to improving RAG applications.

## Fundamentals of AI agents and RAG integration

When talking with new developers in generative AI, we have been told that the concept of an AI agent often tends to be one of the more challenging concepts to grasp. When experts talk about agents, they often talk about them in very abstract terms, focusing on all the things AI agents can be responsible for in a RAG application, but failing to really explain thoroughly what an AI agent is and how it works.

I find that it is easiest to dispel the mystery of the AI agent by explaining what it really is, which is actually a very simple concept. To build an AI agent in its most basic form, you are simply taking the same LLM concept you have already been working with throughout these chapters and adding a loop that terminates when the intended task is done. That's it! It's just a loop folks!

Figure 41.1 represents the **RAG agent loop** you will be working with in the code lab that you are about to dive into:

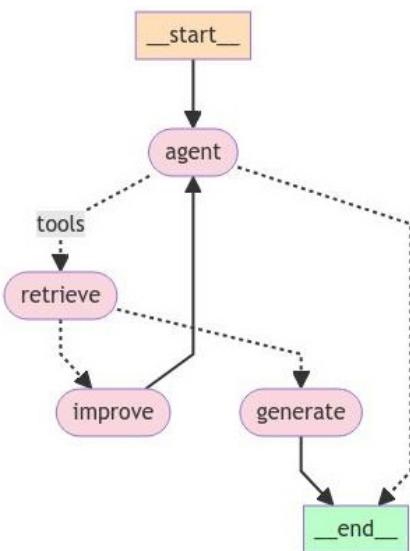


Figure 41.1 – Graph of the agent's control flow

This represents a relatively simple set of logic steps that loop through until the agent decides it has successfully completed the task you have given it. The oval boxes, such as *agent* and *retrieve*, are called **nodes** and the lines are called **edges**. The dotted lines are also edges, but they are a specific type called **conditional edges**, which are edges that are also decision points.

Despite the simplicity, the concept of adding a loop to your LLM calls does make it much more powerful than just using LLMs directly, because it takes more advantage of the LLM's ability to reason and break tasks down into simpler tasks. This improves the chances of success in whatever task you are pursuing and will come in especially handy with more complex multi-step RAG tasks.

While your LLM is looping through agent tasks, you also provide functions called *tools* to the agent, and the LLM will use its reasoning capabilities to determine which tool to use, how to use that tool, and what data to feed it. This is where it can get really complex very quickly. You can have multiple agents, numerous tools, integrated knowledge graphs that help guide your agents down a specific path, numerous frameworks that offer different *flavors* of agents, numerous approaches to agent architecture, and much more. But in this chapter, we are going to focus specifically on how an AI agent can help improve RAG applications. Once you see the power of using an AI agent though, I have no doubt you will want to use it in other generative AI applications, and you should!

## Living in an AI agent world

With all the excitement around agents, you might think LLMs are already going obsolete. But that couldn't be further from the truth. With AI agents, you are really tapping into an even more powerful version of an LLM, a version where the LLM serves as the "brain" of the agent, letting it reason and come up with multi-step solutions well beyond the one-off chat questions most people are using them for. The agent just provides a layer between the user and the LLM and pushes the LLM to accomplish a task that may take multiple queries of the LLM but will eventually, in theory, end up with a much better result.

If you think about it, this matches up more with how problems are solved in the real world, where even simple decisions can be complex. Most tasks we do are based on a long chain of observations, reasoning, and adjustments to new experiences. Very rarely do we interact with people, tasks, and things in the real world in the same way we interact with LLMs online. There is often this building of understanding, knowledge, and context that takes place and helps us find the best solutions. AI agents are better able to handle this type of approach to problem-solving.

Agents can make a big difference to your RAG efforts, but what about this concept of the LLMs being their brains? Let's dive into the concept further.

## LLMs as the agents' brains

If you consider the LLM as the brain of your AI agent, the next logical step is that you likely want the *smartest* LLM you can find to be that brain. The capabilities of the LLM are going to affect your AI agent's ability to reason and make decisions, which will certainly impact the results of the queries to your RAG application.

There is one major way this metaphor of an LLM brain breaks down though, but in a very good way. Unlike agents in the real world, the AI agent can always swap out their LLM brain for another LLM brain. We could even give it multiple LLM brains that can serve to check each other and make sure things are proceeding as planned. This gives us greater flexibility that will help us continually improve the capabilities of our agents.

So, how does LangGraph, or graphs in general, relate to AI agents? We will discuss that next.

## Graphs, AI agents, and LangGraph

LangChain introduced LangGraph in 2024, so it is still relatively new. It is an extension built on top of **LangChain Expression Language (LCEL)** to create composable and customizable agentic workloads. LangGraph leans heavily on graph theory concepts, such as nodes and edges (described earlier), but with a focus on using them to manage your AI agents. While an older way to manage agents, the `AgentExecutor` class, still exists, LangGraph is now the *recommended* way to build agents in LangChain.

LangGraph adds two important components for supporting agents:

- The ability to easily define cycles (cyclical graphs)
- Built-in memory

It provides a pre-built object equivalent to `AgentExecutor`, allowing developers to orchestrate agents using a graph-based approach.

Over the past couple of years, numerous papers, concepts, and approaches have emerged for building agents into RAG applications, such as orchestration agents, ReAct agents, self-refine agents, and multi-agent frameworks. A common theme among these approaches is the concept of a cyclical graph that represents the agent's control flow. While many of these approaches, from an implementation standpoint, are going obsolete, their concepts are still highly useful and are captured in the graph-based environment of LangGraph.

**LangGraph** has become a powerful tool for supporting agents and managing their flow and process in RAG applications. It enables developers to describe and represent single and multi-agent flows as graphs, providing extremely controlled *flows*. This controllability is crucial for avoiding the pitfalls encountered by developers when creating agents early on.

As an example, the popular ReAct approach was an early paradigm for building agents. **ReAct** stands for **r**eason + **a**ct. In this pattern, an LLM first thinks about what to do and then decides an action to take. That action is then executed in an environment and an observation is returned. With that observation, the LLM then repeats this process. It uses reasoning to think about what to do next, decides another action to take, and continues until it has been determined that the goal has been met. If you map this process out, it may look something like what you see here in *Figure 41.2*:

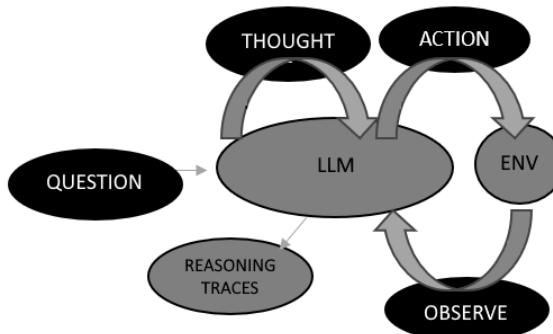


Figure 41.2 – ReAct cyclical graph representation

The set of loops in *Figure 41.2* can be represented by cyclical graphs in LangGraph, with each step represented by nodes and edges. Using this graphing paradigm, you can see how a tool such as LangGraph, a tool for building graphs in LangChain, can form the backbone of your agent framework. As we build our agent framework, we can represent these agent loops using LangGraph, which helps you describe and orchestrate the control flow. This focus on the control flow is critical to addressing some of the early challenges with agents, where a lack of control leads to rogue agents that can't complete their loops or focus on the wrong task.

Another key element that LangGraph has built into it is persistence. Persistence can be used to maintain the memory of the agent, giving it the information it needs to reflect on all of its actions so far, and representing the *OBSERVE* component presented in *Figure 41.2*. This is really helpful for having multiple conversations at the same time or remembering previous iterations and actions. This persistence also enables human-in-the-loop features that give you better control over what the agent is doing at key intervals during its actions.

The paper that introduced the ReAct approach to agent building can be found here: <https://arxiv.org/abs/2210.03629>

Let's dive right into the code lab for building our agent and walk through more key individual concepts as we encounter them in the code.

## Code lab 12.1 – adding a LangGraph agent to RAG

In this code lab, we will add an agent to our existing RAG pipeline that can make decisions about whether to retrieve from an index or use a web search. We will show the inner thoughts of the agent as it processes data that it retrieves toward the goal of providing you with a more thorough response to your question. As we add the code for our agent, we will see new components, such as tools, toolkits, graphs, nodes, edges, and, of course, the agent itself. For each component, we will go more in-depth into how that component interacts and supports your RAG application. We will also add code so that this functions more like a chat session, rather than a Q&A session:

1. First, we will install some new packages to support our agent development:

```
%pip install tiktoken
%pip install langgraph
```

In the first line, we install the `tiktoken` package, which is an OpenAI package used for tokenizing text data before feeding it into language models. Last, we pull in the `langgraph` package we have been discussing.

2. Next, we add a new LLM definition and update our existing one:

```
llm = ChatOpenAI(model_name="gpt-4o-mini",
 temperature=0, streaming=True)
agent_llm = ChatOpenAI(model_name="gpt-4o-mini",
 temperature=0, streaming=True)
```

The new `agent_llm` LLM instance will serve as our agent's brain, handling reasoning and execution of the agent tasks, whereas the original `llm` instance will still be present in our general LLM to do the same LLM tasks we have used it for in the past. While the two LLMs are defined with the same model and parameters in our example, you could and should experiment with using different LLMs for these different tasks, to see if there is a combination that works better for your RAG applications. You could even add additional LLMs to handle specific tasks, such as the `improve_documents` functions in this code, if you find an LLM better at those tasks or have trained or fine-tuned your own for these particular actions. For example, It is common for simple tasks to be handled by faster, lower-cost LLMs as long as they can perform the task successfully. There is a lot of flexibility built into this code that you can take advantage of! Also, note that we add `streaming=True` to the LLM definition. This turns on streaming data from the LLM, which is more conducive to an agent that may make several calls, sometimes in parallel, constantly interacting with the LLM.

Now, we are going to skip down to after the retriever definitions (`dense_retriever`, `sparse_retriever`, and `ensemble_retriever`) and add our first tool. A **tool** has a very specific and important meaning when it comes to agents; so, let's talk about that now.

## Tools and toolkits

In the following code, we are going to add a **web search** tool:

```
from langchain_community.tools.tavily_search import
TavilySearchResults
_ = load_dotenv(dotenv_path='env.txt')
os.environ['TAVILY_API_KEY'] = os.getenv('TAVILY_API_KEY')
!export TAVILY_API_KEY=os.environ['TAVILY_API_KEY']
web_search = TavilySearchResults(max_results=4)
web_search_name = web_search.name
```

You will need to get another API key and add it to the `env.txt` file we have used in the past for the OpenAI and Together APIs. Just like with those APIs, you will need to go to that website, set up your API key, and then copy that into your `env.txt` file. The Tavily website can be found at this URL: <https://tavily.com/>

We run the code again that loads the data from the `env.txt` file and then we set up the `TavilySearchResults` object with `max_results` of 4, meaning when we run it for search, we only want four search results maximum. We then assign the `web_search.name` variable to a variable called `web_search_name` so that we have that available later when we want to tell the agent about it. You can run this tool directly using this code:

```
web_search.invoke(user_query)
```

Running this tool code with `user_query` will give you a result like this (truncated for brevity):

```
[{'url': 'http://sustainability.google/','content': "Google Maps\nChoose the most fuel-efficient route\\nGoogle Shopping\nShop for more efficient appliances for your home\\nGoogle Flights\nFind a flight with lower per-traveler carbon emissions\\nGoogle Nest\\...[TRUNCATED HERE]"},
...
{'content': "2023 Environmental Report. Google's 2023 Environmental Report outlines how we're driving positive environmental outcomes throughout our business in three key ways: developing products and technology that empower individuals on their journey to a more
```

```
sustainable life, working together with partners and organizations
everywhere to transition to resilient, low-carbon systems, and
operating ..."}]
```

At this point, you have just established your first agent tool! This is a search engine tool that your agent can use to retrieve more information from the internet to help it achieve its goal of answering the question your user poses to it.

The *tool* concept in LangChain and when building agents comes from the idea that you want to make actions available to your agent so that it can carry out its tasks. Tools are the mechanism that allows this to happen. You define a tool like we just did for the web search, and then you later add it to a list of tools that the agent can use to accomplish its tasks. Before we set up that list though, we want to create another tool that is central for a RAG application: a retriever tool:

```
from langchain.tools.retriever import create_retriever_tool
retriever_tool = create_retriever_tool(
 ensemble_retriever,
 "retrieve_google_environmental_question_answers",
 "Extensive information about Google environmental
 efforts from 2023.",
)
retriever_tool_name = retriever_tool.name
```

Note that with the web search tool, we imported it from `langchain_community.tools.tavily_search`, whereas with this tool, we use `langchain.tools.retriever`. This reflects the fact that Tavily is a third-party tool, whereas the retriever tool we create here is part of the core LangChain functionality. After importing the `create_retriever_tool` function, we use it to create the `retriever_tool` tool for our agent. Again, like with `web_search_name`, we pull out the `retriever_tool.name` variable we can reference later when we want to refer to it for the agent. You may notice the name of the actual retriever this tool will use, the `ensemble_retriever` retriever, which we created earlier chapters.

You should also note that the name that we are giving this tool, as far as the agent is concerned, is found in the second field, and we are calling it `retrieve_google_environmental_question_answers`. When we name variables in code, we normally try to keep them smaller, but for tools that agents will use, it is helpful to provide more verbose names that will help the agent understand what can be used fully.

We now have two tools for our agent! However, we still need to tell the agent about them eventually; so, we package them up into a list that we can later share with the agent:

```
tools = [web_search, retriever_tool]
```

You see here the two tools we created previously, the `web_search` tool and the `retriever_tool` tool, getting added to the tools list. If we had other tools we wanted to make available to the agent, we could add those to the list as well. In the LangChain ecosystem, there are hundreds of tools available: <https://python.langchain.com/v0.2/docs/integrations/tools/>

You will want to make sure the LLM you are using is “good” at reasoning and using tools. In general, chat models tend to have been fine-tuned for tool calling and will be better at using tools. Non-chat-fine-tuned models may not be able to use tools, especially if the tools are complex or require multiple calls. Using well-written names and descriptions can play an important role in setting your agent LLM up for success as well.

In the agent we are building, we have all the tools we need, but you will also want to look at the toolkits, which are convenient groups of tools. LangChain provides a list of the current toolkits available on their website: <https://python.langchain.com/v0.2/docs/integrations/toolkits/>

For example, if you have a data infrastructure that uses pandas DataFrames, you could use the pandas DataFrame toolkit to offer your agent various tools to access those DataFrames in different ways. Drawing straight from the LangChain website, toolkits are described as follows: (<https://python.langchain.com/v0.1/docs/modules/agents/concepts/#toolkits>)

*For many common tasks, an agent will need a set of related tools. For this LangChain provides the concept of toolkits - groups of around 3-5 tools needed to accomplish specific objectives. For example, the GitHub toolkit has a tool for searching through GitHub issues, a tool for reading a file, a tool for commenting, etc.*

So, basically, if you are focusing on a set of common tasks for your agent or a popular integration partner with LangChain (such as a Salesforce integration), there is likely a toolkit that will give you access to all the tools you need all at once.

Now that we have the tools established, let’s start building the components of our agent, starting with the agent state.

## Agent state

The **agent state** is a key component of any agent you build with LangGraph. Using LangGraph, you create an `AgentState` class that establishes the “state” for your agent and tracks it over time. This state is a local mechanism to the agent that you make available to all parts of the graph and can be stored in a persistence layer.

Here, we set up this state for our RAG agent:

```
from typing import Annotated, Literal, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages
class AgentState(TypedDict):
 messages: Annotated[Sequence[BaseMessage], add_messages]
```

This imports relevant packages for setting up `AgentState`. For example, `BaseMessage` is a base class for representing messages in the conversation between the user and the AI agent. It will be used to define the structure and properties of messages in the state of the conversation. It then defines a graph and a “state” object that it passes around to each node. You can set the state to be a variety of types of objects that you can store different types of data, but for our RAG agent, we set up our state to be a list of “messages”.

We then need to import another round of packages to set up other parts of our agent:

```
from langchain_core.messages import HumanMessage
from langchain_core.pydantic_v1 import BaseModel, Field
from langgraph.prebuilt import tools_condition
```

In this code, we start with importing `HumanMessage`. `HumanMessage` is a specific type of message that represents a message sent by the human user. It will be used when constructing the prompt for the agent to generate a response. We also import `BaseModel` and `Field`. `BaseModel` is a class from the Pydantic library that is used to define data models and validate data. `Field` is a class from Pydantic that is used to define the properties and validation rules for fields in a data model. Last, we import `tools_condition`. The `tools_condition` function is a pre-built function provided by the `LangGraph` library. It is used to assess the agent’s decision on whether to use specific tools based on the current state of the conversation.

These imported classes and functions are used throughout the code to define the structure of messages, validate data, and control the flow of the conversation based on the agent’s decisions. They provide the necessary building blocks and utilities for constructing the language model application using the `LangGraph` library.

We then define our primary prompt (representing what the user would input) like this:

```
generation_prompt = PromptTemplate.from_template(
 """You are an assistant for question-answering tasks.
 Use the following pieces of retrieved context to answer
 the question. If you don't know the answer, just say
 that you don't know. Provide a thorough description to
 fully answer the question, utilizing any relevant
```

```
information you find.
```

```
Question: {question}
```

```
Context: {context}
```

```
Answer: """
```

```
)
```

This is a replacement for the code that we were using in the past code labs:

```
prompt = hub.pull("jclemens24/rag-prompt")
```

We alter the name to `generation_prompt` to make this prompt's use more clear.

Our graph usage is about to pick up in our code, but first, we need to cover some basic graph theory concepts.

## Core concepts of graph theory

To better understand how we are going to use LangGraph in the next few blocks of code, it is helpful to review some key concepts in **graph theory**. **Graphs** are mathematical structures that can be used to represent relationships between different objects. The objects are called **nodes** and the relationships between them, typically drawn with a line, are called **edges**. You have already seen these concepts in *Figure 41.1*, but it is important to understand how they relate to any graph and how that is used in LangGraph.

With LangGraph, there are also specific types of edges representing different types of these relationships. The “conditional edge” that we mentioned along with *Figure 41.1*, for example, represents when you need to make a decision about which node you should go to next; so, they represent the decisions. When talking about the ReAct paradigm, this has also been called the **action edge**, as it is where the action takes place, relating to the *reason + action* approach of ReAct. *Figure 41.3* shows a basic graph consisting of nodes and edges:

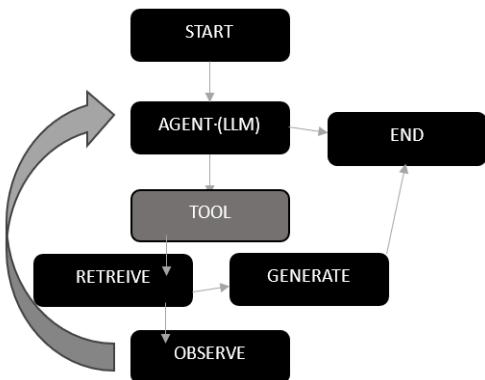


Figure 41.3 – Basic graph representing our RAG application

In this cyclical graph shown in *Figure 41.3*, you see nodes representing the start, agent, retrieve tool, generation, observation, and end. The key edges are where the LLM makes the decision of what tool to use (retrieve is the only one available here), observes if what is retrieved is sufficient, and then pushes to generation. If it is decided that the retrieved data is not sufficient, there is an edge that sends the observation back to the agent to decide if it wants to try again. These decision points are the *conditional edges* we discussed.

## Nodes and edges in our agent

OK, so let's review. We've mentioned that an agentic RAG graph has three key components: the *state* that we already talked about, the *nodes* that append to or update the state, and the *conditional edges* that decide which node to visit next. We are now to the point where we can step through each of these in code blocks, seeing how the three components interact with each other.

Given this background, the first thing we will add to the code is the conditional edge, where the decisions are made. In this case, we are going to define an edge that determines if the retrieved documents are relevant to the question. This is the function that will decide whether to move on to the generation stage or to go back and try again:

1. We will step through this code in multiple steps, but keep in mind that this is one large function, starting with the definition:

```
def score_documents(state) -> Literal[
 "generate", "improve"]:
```

This code starts by defining a function called `score_documents` that determines whether the retrieved documents are relevant to the given question. The function takes the state we've been discussing as a parameter, which is a set of messages that have been collected. This is how we make the state available to this conditional edge function.

2. Now, we build the data model:

```
class scoring(BaseModel):
 binary_score: str = Field(
 description="Relevance score 'yes' or 'no'")
```

This defines a data model class called `scoring` using Pydantic's `BaseModel`. The `scoring` class has a single field called `binary_score`, which is a string representing the relevance score as either yes or no.

3. Next, we add the LLM that will make this decision:

```
llm_with_tool = llm.with_structured_output(
 scoring)
```

This creates an instance of `llm_with_tool` by calling `llm.with_structured_output(scoring)`, combining the LLM with the scoring data model for structured output validation.

4. As we have seen in the past, we need to set up a `PromptTemplate` class to pass to the LLM. Here is that prompt:

```
prompt = PromptTemplate(
 template="""You are assessing relevance of a retrieved
 document to a user question with a binary grade. Here is the
 retrieved document:
 {context}
 Here is the user question: {question}
 If the document contains keyword(s) or semantic meaning related
 to the user question, grade it as relevant. Give a binary score
 'yes' or 'no' score to indicate whether the document is relevant
 to the question.""""
 input_variables=["context", "question"],
)
```

This defines a prompt using the `PromptTemplate` class, providing instructions to the LLM for applying a binary score for the relevance of the retrieved document based on the given question.

5. We can then use LCEL to build the chain that combines the prompt with the `llm_with_tool` tool we just set up:

```
chain = prompt | llm_with_tool
```

This chain represents the pipeline for scoring the documents. This defines the chain, but we haven't invoked it yet.

6. First, we want to pull in the state. Next, we pull the state ("messages") into the function so that we can use it, and we take the last message:

```
messages = state["messages"]
last_message = messages[-1]
question = messages[0].content
docs = last_message.content
```

This extracts the necessary information from the "state" parameter and then preps the state/message as the context we are going to pass to our agent brain (LLM). The specific components extracted here include the following:

- `messages`: The list of messages in the conversation
- `last_message`: The last message in the conversation
- `question`: The content of the first message, which is assumed to be the user's question
- `docs`: The content of the last message, which is assumed to be the retrieved documents

Then, finally, we invoke the chain with the prompt filled (if you remember, we call this **hydrating** the prompt) with `question` and context `docs` to get the scored result:

```
scored_result = chain.invoke({"question": question, "context": docs})
score = scored_result.binary_score
```

This extracts the `binary_score` variable from the `scored_result` object and assigns it to the `score` variable. The `llm_with_tool` step, which is the last step in the LangChain chain, aptly called `chain`, is going to return a string-based binary result based on the response from the scoring function:

```
if score == "yes":
 print("---DECISION: DOCS RELEVANT---")
 return "generate"
else:
 print("---DECISION: DOCS NOT RELEVANT---")
```

```
 print(score)
 return "improve"
```

This checks the value of the score. If the score value is yes, it prints a message indicating that the documents are relevant and returns generate as the final output from the `score_documents` function, suggesting that the next step is to generate a response. If the score value is no, or, technically, anything other than yes, it prints messages indicating that the documents are not relevant and returns `improve`, suggesting that the next step is to improve the query from the user.

Overall, this function acts as a decision point in the workflow, determining whether the retrieved documents are relevant to the question and directing the flow to either generate a response or rewrite the question based on the relevance score.

7. Now that we have our conditional edge defined, we are going to move on to defining our nodes, starting with the agent:

```
def agent(state):
 print("---CALL AGENT---")
 messages = state["messages"]
 llm = llm.bind_tools(tools)
 response = llm.invoke(messages)
 return {"messages": [response]}
```

This function represents the agent node on our graph and invokes the agent model to generate a response based on the current state. The `agent` function takes the current state ("state") as input, which contains the messages in the conversation, prints a message indicating that it is calling the agent, extracts the messages from the state dictionary, uses the `agent_llm` instance of the `ChatOpenAI` class we defined earlier, representing the agent *brain*, and then binds the tools to the model using the `bind_tools` method. We then invoke the agent's `llm` instance with the messages and assign the result to the `response` variable.

8. Our next node, `improve`, is responsible for transforming `user_query` to produce a better question if the agent determines this is needed:

```
def improve(state):
 print("---TRANSFORM QUERY---")
 messages = state["messages"]
 question = messages[0].content
 msg = [
```

```

HumanMessage(content=f"""\\n
Look at the input and try to reason about
the underlying semantic intent / meaning.
\\n
Here is the initial question:
\\n ----- \\n
{question}
\\n ----- \\n
Formulate an improved question:
""",)
]
response = llm.invoke(msg)
return {"messages": [response]}

```

This function, like all of our node and edge-related functions, takes the current state ("state") as input. The function returns a dictionary with the response appended to the messages list. The function prints a message indicating that it is transforming the query, extracts the messages from the state dictionary, retrieves the content of the first message (`messages[0].content`), which is assumed to be the initial question, and assigns it to the `question` variable. We then set up a message using the `HumanMessage` class, indicating that we want the `llm` instance to reason about the underlying semantic intent of the question and formulate an improved question. The result from the `llm` instance is assigned to the `response` variable. Finally, it returns a dictionary with the response appended to the messages list.

#### 9. Our next node function is the generate function:

```

def generate(state):
 print("----GENERATE---")
 messages = state["messages"]
 question = messages[0].content
 last_message = messages[-1]
 question = messages[0].content
 docs = last_message.content
 rag_chain = generation_prompt | llm |
 str_output_parser
 response = rag_chain.invoke({"context": docs,
 "question": question})
 return {"messages": [response]}

```

This function is similar to our generation step in the previous chapter's code labs but simplified to provide just the response. It generates an answer based on the retrieved documents and the question. The function takes the current state ("state") as input, which contains the messages in the conversation, prints a message indicating that it is generating an answer, extracts the messages from the state dictionary, retrieves the content of the first message (`messages[0].content`), which is assumed to be the question, and assigns it to the `question` variable.

The function then retrieves the last message (`messages[-1]`) and assigns it to the `last_message` variable. The `docs` variable is assigned the content of `last_message`, which is assumed to be the retrieved documents. At this point, we create a chain called `rag_chain` by combining the `generation_prompt`, `llm`, and `str_output_parser` variables using the `|` operator. As with other LLM prompting, we hydrate the predefined `generation_prompt` as the prompt for generating the answer, which returns a dictionary with the `response` variable appended to the `messages` list.

Next, we want to set up our cyclical graphs using LangGraph and assign our nodes and edges to them.

## Cyclical graph setup

The next big step in our code is setting up our graphs using LangGraph:

1. First, we import some important packages to get us started:

```
from langgraph.graph import END, StateGraph
from langgraph.prebuilt import ToolNode
```

This code imports the following necessary classes and functions from the `langgraph` library:

- `END`: A special node representing the end of the workflow
- `StateGraph`: A class for defining the state graph of the workflow
- `ToolNode`: A class for defining a node that represents a tool or action

2. We then pass `AgentState` as an argument to the `StateGraph` class we just imported for defining the state graph of the workflow:

```
workflow = StateGraph(AgentState)
```

This creates a new instance of `StateGraph` called `workflow` and defines a new graph for that `workflow` `StateGraph` instance.

3. Next, we define the nodes we will cycle between and assign our node functions to them:

```
workflow.add_node("agent", agent) # agent
retrieve = ToolNode(tools)
workflow.add_node("retrieve", retrieve)
retrieval from web and or retriever
workflow.add_node("improve", improve)
 # Improving the question for better retrieval
workflow.add_node("generate", generate) # Generating a response
after we know the documents are relevant
```

This code adds multiple nodes to the `workflow` instance using the `add_node` method:

- “agent”: This node represents the agent node, which invokes the `agent` function.
- “retrieve”: This node represents the retrieval node, which is a special `ToolNode` containing the tools list we defined early with the `web_search` and `retriever_tool` tools. In this code, to aid in readability, we explicitly break out the `ToolNode` class instance and define the `retrieve` variable with it, which indicates the “retrieve” focus of this node more explicitly. We then pass that `retrieve` variable into the `add_node` function.
- “improve”: This node represents the node for improving the question, which invokes the `improve` function.
- “generate”: This node represents the node for generating a response, which invokes the `generate` function.

4. Next, we need to define our starting point for our workflow:

```
workflow.set_entry_point("agent")
```

This sets the entry point of the `workflow` instance to the “agent” node using `workflow.set_entry_point("agent")`.

5. Next, we call the “agent” node to decide whether to retrieve or not:

```
workflow.add_conditional_edges("agent", tools_condition,
{
 "tools": "retrieve",
 END: END,
},
)
```

In this code, `tools_condition` is used as a conditional edge in the workflow graph. It determines whether the agent should proceed to the retrieval step ("tools": "retrieve") or end the conversation (END: END) based on the agent's decision. The retrieval step represents both of the tools that we made available for the agent to use where needed, and the other option, to end the conversation simply ends the workflow.

6. Here, we add more edges, which are used after the "action" node is called:

```
workflow.add_conditional_edges("retrieve",
 score_documents)
workflow.add_edge("generate", END)
workflow.add_edge("improve", "agent")
```

After the "retrieve" node is called, it adds conditional edges using `workflow.add_conditional_edges("retrieve", score_documents)`. This assesses the retrieved documents using the `score_documents` function and determines the next node based on the score. This also adds an edge from the "generate" node to the END node using `workflow.add_edge("generate", END)`. This indicates that, after generating a response, the workflow ends. Last, this adds an edge from the "improve" node back to the "agent" node using `workflow.add_edge("improve", "agent")`. This creates a loop where the improved question is sent back to the agent for further processing.

7. We are now ready to compile the graph:

```
graph = workflow.compile()
```

This line compiles the workflow graph using `workflow.compile` and assigns the compiled graph to the `graph` variable, which now represents a compiled version of the `StateGraph` graph instance we started with.

8. We have already shown you the visualization of what this graph looks like earlier in this chapter in *Figure 41.1*, but if you want to run the visualization yourself, you can use this code:

```
from IPython.display import Image, display
try:
 display(Image(graph.get_graph(
 xray=True).draw_mermaid_png())))
except:
 pass
```

We can use IPython to generate this visualization.

9. Last, we are going to finally put our agent to work:

```
import pprint
inputs = {
 "messages": [
 ("user", user_query),
]
}
```

This imports the `pprint` module, which provides a pretty-print function for formatting and printing data structures, allowing us to see a more human-readable version of our agent output. We then define a dictionary called `inputs` that represents the initial input to the workflow graph. The `inputs` dictionary contains a “`messages`” key with a list of tuples. In this case, it has a single tuple, `("user", user_query)`, where the “`user`” string represents the role of the message sender (`user`) and `user_query` is the user’s query or question.

10. We then initialize an empty string variable called `final_answer` to store the final answer generated by the workflow:

```
final_answer = ''
```

11. We then start our agent loop using the graph instance as the basis:

```
for output in graph.stream(inputs):
 for key, value in output.items():
 pprint.pprint(f"Output from node '{key}':")
 pprint.pprint("---")
 pprint.pprint(value, indent=2, width=80,
 depth=None)
 final_answer = value
```

This starts a double loop using the `output` in `graph.stream(inputs)`. This iterates over the outputs generated by the `graph` instance as it processes the inputs. The `graph.stream(inputs)` method streams the outputs from the `graph` instance execution.

Inside the outer loop, it starts another loop for two variables, `key` and `value`, representing the key-value pairs in the `output.items` variable. This iterates over each of those key-value pairs, where the `key` variable represents the node name and the `value` variable represents the output generated by that node.

This will print the node name using `pprint.pprint(f"Output from node '{key}':")` to indicate which node generated the output.

The code pretty-prints the value (`output`) using `pprint.pprint(value, indent=2, width=80, depth=None)`. The `indent` parameter specifies the indentation level, `width` specifies the maximum width of the output, and `depth` specifies the maximum depth of nested data structures to print (`None` means no limit).

It assigns the value (`output`) to the `final_answer` variable, overwriting it in each iteration. After the loop ends, `final_answer` will contain the output generated by the last node in the workflow.

A nice feature of this code is that it allows you to see the intermediate outputs generated by each node in the graph and track the progress of the query processing. These print outputs represent the agent's "thoughts" as it makes decisions within the loop. The pretty-printing helps in formatting the outputs for better readability.

When we start the agent and start seeing the output, we can see that a lot is going on!

I will truncate a lot of the printout, but this will give you an idea of what is provided:

```
---CALL AGENT---
"Output from node 'agent':"
'---'
{ 'messages': [AIMessage(content='', additional_kwargs={'tool_calls': [{ 'index': 0, 'id': 'call_46NqZuz3gN2F9IR5jq0MRdVm', 'function': { 'arguments': '{\"query\":\"Google\\\'s environmental initiatives\"}', 'name': 'retrieve_google_environmental_question_answers', 'type': 'function'}]}, response_metadata={'finish_reason': 'tool_calls'}, id='run-eba27f1e-1c32-4ffc-a161-55a32d645498-0', tool_calls=[{'name': 'retrieve_google_environmental_question_answers', 'args': { 'query': '\"Google\\\'s environmental initiatives\"'}, 'id': 'call_46NqZuz3gN2F9IR5jq0MRdVm'}]]}
'\n---\n'
```

This is the first part of our printout. Here, we see the agent is deciding to use the `retrieve_google_environmental_question_answers` tool. If you will recall, that is the text-based name we gave to the retriever tool when defining it. Good choice!

12. Next, the agent is going to determine if it thinks the documents retrieved are relevant:

```
---CHECK RELEVANCE---
---DECISION: DOCS RELEVANT---
```

The decision is that they are. Again, smart thinking, Mr. Agent.

13. Last, we see the output of what the agent is looking at, retrieved from the PDF document and the ensemble retriever we have been using (there was a lot of retrieved data here, so I truncated most of the actual content):

```
"Output from node 'retrieve':"
'---'
{ 'messages': [ToolMessage(content='iMasons Climate
AccordGoogle is a founding member and part of the governing
body of the iMasons Climate Accord, a coalition united on
carbon reduction in digital infrastructure.\nReFEDIn 2022,
to activate industry-wide change...[TRUNCATED]', tool_call_
id='call_46NqZuz3gN2F9IR5jq0MRdVm')] }
\n---\n'
```

When you look at the actual printout for this portion, you see that the retrieved data is concatenated together and provides substantial and in-depth data for our agent to work with.

14. At this point, just like our original RAG application was doing, the agent takes the question, retrieved data, and formulates a response based on the generation prompt we gave it:

```
---GENERATE---
"Output from node 'generate':"
'---'
{ 'messages': ['Google has a comprehensive and multifaceted
approach to '
'environmental sustainability, encompassing various '
'initiatives aimed at reducing carbon emissions, promoting'
'sustainable practices, and leveraging technology for '
'environmental benefits. Here are some key aspects of Google's "
'environmental initiatives:\n''\n'
'1. **Carbon Reduction and Renewable Energy**...'] }
\n---\n'
```

We included a mechanism here to print out the final message separately for readability:

```
final_answer['messages'][0]
```

This will print this out:

```
"Google has a comprehensive and multifaceted approach to environmental sustainability, encompassing various initiatives aimed at reducing carbon emissions, promoting sustainable practices, and leveraging technology for environmental benefits. Here are some key aspects of Google's environmental initiatives:\n\n- **Carbon Reduction and Renewable Energy**: Google is a founding member and part of the governing body of this coalition focused on reducing carbon emissions in digital infrastructure.\n- **Net-Zero Carbon**: Google is committed to operating sustainably with a focus on achieving net-zero carbon emissions. This includes investments in carbon-free energy and energy-efficient facilities, such as their all-electric, net water-positive Bay View campus..."
```

That is the full output of our agent!

# MODULE 9

## NAVIGATING ML: ARCHITECTURES, TOOLS, AND PLATFORMS

---

**CHAPTER 42** ML Systems: The Lifecycle and Architecture

**CHAPTER 43** Business Use Cases for Machine Learning

**CHAPTER 44** Breaking Down ML Algorithms

**CHAPTER 45:** Managing Data for Machine Learning Success

**CHAPTER 46:** Exploring the Best ML Libraries

**CHAPTER 47:** The Power of Open-Source ML Platforms

Chapter

# 42

# ML Systems: The Lifecycle and Architecture

The field of **artificial intelligence (AI)** and **machine learning (ML)** has had a long history. Over the last 70+ years, ML has evolved from checker game-playing computer programs in the 1950s to advanced AI capable of beating the human world champion in the game of *Go*. More recently, **Generative AI (GenAI)** technology such as ChatGPT has been taking the industry by storm, generating huge interest among company executives and consumers alike, promising new ways to transform businesses such as drug discovery, new media content, financial report analysis, and consumer product design. Along the way, the technology infrastructure for ML has also evolved from a single machine/server for small experiments and models to highly complex end-to-end ML platforms capable of training, managing, and deploying tens of thousands of ML models. The hyper-growth in the AI/ML field has resulted in the creation of many new professional roles, such as **MLOps engineering**, **AI/ML product management**, **ML software engineering**, **AI risk manager**, and **AI strategist** across a range of industries.

**Machine learning solutions architecture (ML solutions architecture)** is another relatively new discipline that is playing an increasingly critical role in the full end-to-end ML lifecycle as ML projects become increasingly complex in terms of *business impact*, *science sophistication*, and the *technology landscape*.

## ML versus traditional software

Before I started working in the field of AI/ML, I spent many years building computer software platforms for large financial services institutions. Some of the business problems I worked on had complex rules, such as identifying companies for comparable analysis for investment banking deals or creating a master database for all the different companies' identifiers from the different data providers. We had to implement hardcoded rules in database-stored procedures and application server backends to solve these problems. We often debated if certain rules made sense or not for the business problems we tried to solve.

As rules changed, we had to reimplement the rules and make sure the changes did not break anything. To test for new releases or changes, we often replied to human experts to exhaustively test and validate all the business logic implemented before the production release. It was a very time-consuming and error-prone process and required a significant amount of engineering, testing against the documented specification, and rigorous change management for deployment every time new rules were introduced, or existing rules needed to be changed. We often replied to users to report business logic issues in production, and when an issue was reported in production, we sometimes had to open up the source code to troubleshoot or explain the logic of how it worked. I remember I often asked myself if there were better ways to do this.

After I started working in the field of AI/ML, I started to solve many similar challenges using ML techniques. With ML, I did not need to come up with complex rules that often require deep data and domain expertise to create or maintain the complex rules for decision making. Instead, I focused on collecting high-quality data and used ML algorithms to learn the rules and patterns from the data directly. This new approach eliminated many of the challenging aspects of creating new rules (for example, a deep domain expertise requirement, or avoiding human bias) and maintaining existing rules. To validate the model before the production release, we could examine model performance metrics such as **accuracy**. While it still required data science expertise to interpret the model metrics against the nature of the business problems and dataset, it did not require exhaustive manual testing of all the different scenarios. When a model was deployed into production, we would monitor if the model performed as expected by monitoring any significant changes in production data versus the data we have collected for model training. We would collect new unseen data and labels for production data and test the model performance periodically to ensure that its predictive accuracy remains robust when faced with new, previously unseen production data. To explain why a model made a decision the way it did, we did not need to open up the source code to re-examine the hardcoded logic. Instead, we would rely on ML techniques to help explain the relative importance of different input features to understand what factors were most influential in the decision-making by the ML models.

The following figure shows a graphical view of the process differences between developing a piece of software and training an ML model:

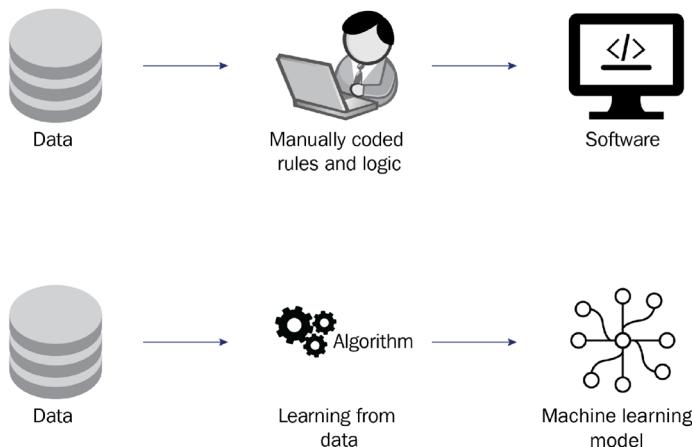


Figure 42.1: ML and computer software

Now that you know the difference between ML and traditional software, it is time to dive deep into understanding the different stages in an ML lifecycle.

## ML lifecycle

One of the early ML projects that I worked on was a fascinating yet daunting sports predictive analytics problem for a major league brand. I was given a list of predictive analytics outcomes to think about to see if there were ML solutions for the problems. I was a casual viewer of the sport; I didn't know anything about the analytics to be generated, nor the rules of the games in the detail that was needed. I was provided with some sample data but had no idea what to do with it.

The first thing I started to work on was an immersion in the sport itself. I delved into the intricacies of the game, studying the different player positions and events that make up each game and play. Only after being armed with the newfound domain knowledge did the data start to make sense. Together with the stakeholder, we evaluated the impact of the different analytics outcomes and assessed the modeling feasibility based on the data we had. With a clear understanding of the data, we came up with a couple of top ML analytics with the most business impact to focus on. We also decided how they would be integrated into the existing business workflow, and how they would be measured on their impacts.

Subsequently, I delved deeper into the data to ascertain what information was available and what was lacking. The raw dataset had a lot of irrelevant data points that needed to be removed while the relevant data points needed to be transformed to provide the strongest signals for model training. I processed and prepared the dataset based on a

few of the ML algorithms I had considered and conducted experiments to determine the best approach. I lacked a tool to track the different experiment results, so I had to document what I had done manually. After some initial rounds of experimentation, it became evident that the existing data was not sufficient to train a high-performance model. Hence, I decided to build a custom deep learning model to incorporate data of different modalities as the data points had temporal dependencies and required additional spatial information for the modeling. The data owner was able to provide the additional datasets I required, and after more experiments with custom algorithms and significant data preparations and feature engineering, I eventually trained a model that met the business objectives.

After completing the model, another hard challenge began – deploying and operationalizing the model in production and integrating it into the existing business workflow and system architecture. We engaged in many architecture and engineering discussions and eventually built out a deployment architecture for the model.

As you can see from my personal experience, the journey from business idea to ML production deployment involved many steps. A typical lifecycle of an ML project follows a formal structure, which includes several essential stages like business understanding, data acquisition and understanding, data preparation, model building, model evaluation, and model deployment. Since a big component of the lifecycle is experimentation with different datasets, features, and algorithms, the whole process is highly iterative. Furthermore, it is essential to note that there is no guarantee of a successful outcome. Factors such as the availability and quality of data, feature engineering techniques (the process of using domain knowledge to extract useful features from raw data), and the capability of the learning algorithms, among others, can all affect the final results.

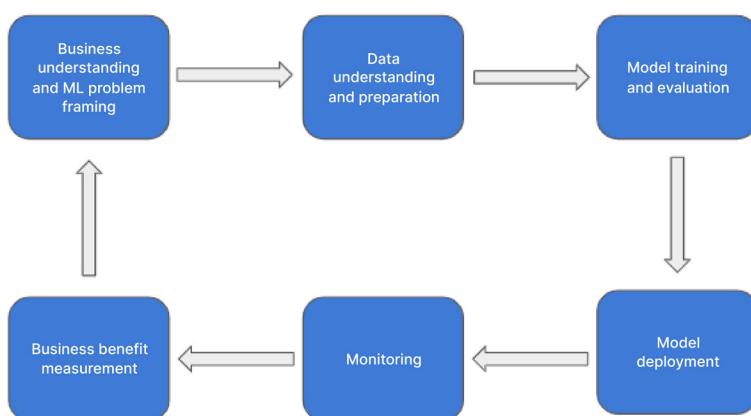


Figure 42.2: ML lifecycle

The preceding figure illustrates the key steps in ML projects, and in the subsequent sections, we will delve into each of these steps in greater detail.

## Business problem understanding and ML problem framing

The first stage in the lifecycle is **business understanding**. This stage involves the understanding of the business goals and defining business metrics that can measure the project's success. For example, the following are some examples of business goals:

- Cost reduction for operational processes, such as document processing.
- Mitigation of business or operational risks, such as fraud and compliance.
- Product or service revenue improvements, such as better target marketing, new insight generation for better decision making, and increased customer satisfaction.

To measure the success, you may use specific business metrics such as the number of hours reduced in a business process, an increased number of true positive frauds detected, a conversion rate improvement from target marketing, or the number of churn rate reductions. This is an essential step to get right to ensure there is sufficient justification for an ML project and that the outcome of the project can be successfully measured.

After you have defined the business goals and business metrics, you need to evaluate if there is an ML solution for the business problem. While ML has a wide scope of applications, it is not always an optimal solution for every business problem.

## Data understanding and data preparation

The saying that “data is the new oil” holds particularly true for ML. Without the required data, you cannot move forward with an ML project. That’s why the next step in the ML lifecycle is **data acquisition, understanding, and preparation**.

Based on the business problems and ML approach, you will need to gather and comprehend the available data to determine if you have the right data and data volume to solve the ML problem. For example, suppose the business problem to address is credit card fraud detection. In that case, you will need datasets such as historical credit card transaction data, customer demographics, account data, device usage data, and networking access data. Detailed data analysis is then necessary to determine if the dataset features and quality are sufficient for the modeling tasks. You

also need to decide if the data needs labeling, such as **fraud** or **not-fraud**. During this step, depending on the data quality, a significant amount of data wrangling might be performed to prepare and clean the data and to generate the dataset for model training and model evaluation, depending on the data quality.

## Model training and evaluation

Using the training and validation datasets established, a data scientist must run a number of experiments using different ML algorithms and dataset features for feature selection and model development. This is a highly iterative process and could require numerous runs of data processing and model development to find the right algorithm and dataset combination for optimal model performance. In addition to model performance, factors such as data bias and model explainability may need to be considered to comply with internal or regulatory requirements.

Prior to deployment into production, the model quality must be validated using the relevant technical metrics, such as the **accuracy score**. This is usually accomplished using a **holdout dataset**, also known as a **test dataset**, to gauge how the model performs on unseen data. It is crucial to understand which metrics are appropriate for model validation, as they vary depending on the ML problems and the dataset used. For example, model accuracy would be a suitable validation metric for a document classification use case if the number of document types is relatively balanced. However, model accuracy would not be a good metric to evaluate the model performance for a fraud detection use case – this is because the number of frauds is small and even if the model predicts **not-fraud** all the time, the model accuracy could still be very high.

## Model deployment

After the model is fully trained and validated to meet the expected performance metric, it can be deployed into production and the business workflow. There are two main deployment concepts here. The first involves the deployment of the model itself to be used by a client application to generate predictions. The second concept is to integrate this prediction workflow into a business workflow application. For example, deploying the credit fraud model would either host the model behind an API for real-time prediction or as a package that can be loaded dynamically to support batch predictions. Moreover, this prediction workflow also needs to be integrated into business workflow applications for fraud detection, which might include the fraud detection of real-time transactions, decision automation based on prediction output, and fraud detection analytics for detailed fraud analytics.

## Model monitoring

The ML lifecycle does not end with model deployment. Unlike software, whose behavior is highly deterministic since developers explicitly code its logic, an ML model could behave differently in production from its behavior in model training and validation. This could be caused by changes in the production data characteristics, data distribution, or the potential manipulation of request data. Therefore, model monitoring is an important post-deployment step for detecting model performance degradation (a.k.a model drift) or dataset distribution change in the production environment (a.k.a data drift).

## Business metric tracking

The actual business impact should be tracked and measured as an ongoing process to ensure the model delivers the expected business benefits. This may involve comparing the business metrics before and after the model deployment, or A/B testing where a business metric is compared between workflows with or without the ML model. If the model does not deliver the expected benefits, it should be re-evaluated for improvement opportunities. This could also mean framing the business problem as a different ML problem. For example, if churn prediction does not help improve customer satisfaction, then consider a personalized product/service offering to solve the problem.

## ML challenges

Over the years, I have worked on many real-world problems using ML solutions and encountered different challenges faced by different industries during ML adoptions.

I often get the same question when working on ML projects: *We have a lot of data – can you help us figure out what insights we can generate using ML?* I refer to companies with this question as having a *business use case challenge*. Not being able to identify business use cases for ML is a very big hurdle for many companies. Without a properly identified business problem and its value proposition and benefit, it becomes difficult to initiate an ML project.

In my conversations with different companies across their industries, data-related challenges emerge as a frequent issue. This includes data quality, data inventory, data accessibility, data governance, and data availability. This problem affects both data-poor and data-rich companies and is often exacerbated by data silos, data security, and industry regulations.

The shortage of data science and ML talent is another major challenge I have heard from many companies. Companies, in general, are having a tough time attracting and retaining top ML talents, which is a common problem across all industries. As ML platforms become more complex and the scope of ML projects increases, the need for other ML-related functions starts to surface. Nowadays, in addition to just data scientists, an organization would also need functional roles for ML product management, ML infrastructure engineering, and ML operations management.

Based on my experiences, I have observed that cultural acceptance of ML-based solutions is another significant challenge for broad adoption. There are individuals who perceive ML as a threat to their job functions, and their lack of knowledge in ML makes them hesitant to adopt these new methods in their business workflows.

The practice of ML solutions architecture aims to help solve some of the challenges in ML. In the next section, we will explore ML solutions architecture and its role in the ML lifecycle.

## ML solutions architecture

When I initially worked with companies as an ML solutions architect, the landscape was quite different from what it is now. The focus was mainly on data science and modeling, and the problems at hand were small in scope. Back then, most of the problems could be solved using simple ML techniques. The datasets were small, and the infrastructure required was not too demanding. The scope of the ML initiative at these companies was limited to a few data scientists or teams. As an ML architect at that time, I primarily needed to have solid data science skills and general cloud architecture knowledge to get the job done.

In more recent years, the landscape of ML initiatives has become more intricate and multifaceted, necessitating involvement from a broader range of functions and personas at companies. My engagement has expanded to include discussions with business executives about ML strategies and organizational design to facilitate the broad adoption of AI/ML throughout their enterprises. I have been tasked with designing more complex ML platforms, utilizing a diverse range of technologies for large enterprises to meet stringent security and compliance requirements. ML workflow orchestration and operations have become increasingly crucial topics of discussion, and more and more companies are looking to train large ML models with enormous amounts of training data. The number of ML models trained and deployed by some companies has skyrocketed to tens of thousands from a few dozen models in just a few years. Furthermore, sophisticated and security-sensitive customers have sought guidance on topics such as ML privacy, model explainability, and data and model bias. As an ML solutions architect, I've noticed that the skills and knowledge required to be successful in this role have evolved significantly.

Trying to navigate the complexities of a business, data, science, and technology landscape can be a daunting task. As an ML solutions architect, I have seen firsthand the challenges that companies face in bringing all these pieces together. In my view, ML solutions architecture is an essential discipline that serves as a bridge connecting the different components of an ML initiative. Drawing on my years of experience working with companies of all sizes and across diverse industries, I believe that an ML solutions architect plays a pivotal role in identifying business needs, developing ML solutions to address these needs, and designing the technology platforms necessary to run these solutions. By collaborating with various business and technology partners, an ML solutions architect can help companies unlock the full potential of their data and realize tangible benefits from their ML initiatives.

The following figure illustrates the core functional areas covered by the ML solutions architecture:

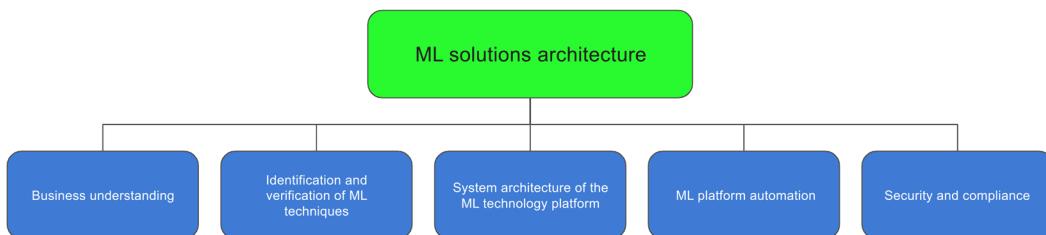


Figure 42.3: ML solutions architecture coverage

## Business understanding and ML transformation

The goal of the business workflow analysis is to identify inefficiencies in the workflows and determine if ML can be applied to help eliminate pain points, improve efficiency, or even create new revenue opportunities.

Picture this: you are tasked with improving a call center's operations. You know there are inefficiencies that need to be addressed, but you're not sure where to start. That's where business workflow analysis comes in. By analyzing the call center's workflows, you can identify pain points such as long customer wait times, knowledge gaps among agents, and the inability to extract customer insights from call recordings. Once you have identified these issues, you can determine what data is available and which business metrics need to be improved. This is where ML comes in. You can use ML to create virtual assistants for common customer inquiries, transcribe audio recordings to allow for text analysis, and detect customer intent for product cross-sell and up-sell. But sometimes, you need to modify the business process to incorporate ML

solutions. For example, if you want to use call recording analytics to generate insights for cross-selling or up-selling products, but there's no established process to act on those insights, you may need to introduce an automated target marketing process or a proactive outreach process by the sales team.

## Identification and verification of ML techniques

Once you have come up with a list of ML options, the next step is to determine if the assumption behind the ML approach is valid. This could involve conducting a simple **proof of concept (POC)** modeling to validate the available dataset and modeling approach, or technology POC using pre-built AI services, or testing of ML frameworks. For example, you might want to test the feasibility of text transcription from audio files using an existing text transcription service or build a customer propensity model for a new product conversion from a marketing campaign.

It is worth noting that ML solutions architecture does not focus on developing new machine algorithms, a job best suited for applied data scientists or research data scientists. Instead, ML solutions architecture focuses on identifying and applying ML algorithms to address a range of ML problems such as predictive analytics, computer vision, or natural language processing. Also, the goal of any modeling task here is not to build production-quality models but rather to validate the approach for further experimentations by full-time applied data scientists.

## System architecture design and implementation

The most important aspect of the ML solutions architect's role is the technical architecture design of the ML platform. The platform will need to provide the technical capability to support the different phases of the ML cycle and personas, such as data scientists and operations engineers. Specifically, an ML platform needs to have the following core functions:

- **Data explorations and experimentation:** Data scientists use ML platforms for data exploration, experimentation, model building, and model evaluation. ML platforms need to provide capabilities such as data science development tools for model authoring and experimentation, data wrangling tools for data exploration and wrangling, source code control for code management, and a package repository for library package management.
- **Data management and large-scale data processing:** Data scientists or data engineers will need the technical capability to ingest, store, access, and process large amounts of data for cleansing, transformation, and feature engineering.

- **Model training infrastructure management:** ML platforms will need to provide model training infrastructure for different modeling training using different types of computing resources, storage, and networking configurations. It also needs to support different types of ML libraries or frameworks, such as **scikit-learn**, **TensorFlow**, and **PyTorch**.
- **Model hosting/serving:** ML platforms will need to provide the technical capability to host and serve the model for prediction generations, for real-time, batch, or both.
- **Model management:** Trained ML models will need to be managed and tracked for easy access and lookup, with relevant metadata.
- **Feature management:** Common and reusable features will need to be managed and served for model training and model serving purposes.

## ML platform workflow automation

A key aspect of ML platform design is **workflow automation** and **continuous integration/continuous deployment (CI/CD)**, also known as MLOps. ML is a multi-step workflow – it needs to be automated, which includes data processing, model training, model validation, and model hosting. Infrastructure provisioning automation and self-service is another aspect of automation design. Key components of workflow automation include the following:

- **Pipeline design and management:** The ability to create different automation pipelines for various tasks, such as model training and model hosting.
- **Pipeline execution and monitoring:** The ability to run different pipelines and monitor the pipeline execution status for the entire pipeline and each of the steps in the ML cycle such as data processing and model training.
- **Model monitoring configuration:** The ability to monitor the model in production for various metrics, such as data drift (where the distribution of data used in production deviates from the distribution of data used for model training), model drift (where the performance of the model degrades in the production compared with training results), and bias detection (the ML model replicating or amplifying bias towards certain individuals).

## Security and compliance

Another important aspect of ML solutions architecture is the security and compliance consideration in a sensitive or enterprise setting:

- **Authentication and authorization:** The ML platform needs to provide authentication and authorization mechanisms to manage access to the platform and different resources and services.
- **Network security:** The ML platform needs to be configured for different network security controls such as a firewall and an IP address access allowlist to prevent unauthorized access.
- **Data encryption:** For security-sensitive organizations, data encryption is another important aspect of the design consideration for the ML platform.
- **Audit and compliance:** Audit and compliance staff need the information to help them understand how decisions are made by the predictive models if required, the lineage of a model from data to model artifacts, and any bias exhibited in the data and model. The ML platform will need to provide model explainability, bias detection, and model traceability across the various datastore and service components, among other capabilities.

Various industry technology providers have established best practices to guide the design and implementation of ML infrastructure, which is part of the ML solutions architect's practices. Amazon Web Services, for example, created *Machine Learning Lens* to provide architectural best practices across crucial domains like operational excellence, security, reliability, performance, cost optimization, and sustainability. Following these published guidelines can help practitioners implement robust and effective ML solutions.

Chapter

# 43

# Business Use Cases for Machine Learning

As an ML practitioner, it is essential for me to develop a deep understanding of different businesses to have effective conversations with business and technology leaders. This should not come as a surprise since the ultimate goal of any ML solutions architecture is to solve practical business problems with science and technology solutions. Therefore, one of the main areas of focus for ML solutions architecture is to develop a broad understanding of different business domains, workflows, and relevant data. Without this understanding, it would be challenging to make sense of the data and design and develop practical ML solutions for business problems.

## ML use cases in financial services

The **Financial Services Industry (FSI)** has always been at the forefront of technological innovation, and ML adoption is no exception. In recent years, we have seen a range of ML solutions being implemented across different business functions within financial services. For example, in capital markets, ML is being used across front, middle, and back offices to aid investment decisions, trade optimization, risk management, and transaction settlement processing. In insurance, companies are using ML to streamline underwriting, prevent fraud, and automate claim management. While in banking, banks are using it to improve customer experience, combat fraud, and facilitate loan approval decisions. In the following sections, we will explore different core business areas within financial services and how ML can be applied to overcome some of these business challenges.

### Capital market front office

In finance, the front office is the revenue-generating business area that includes customer-facing roles such as securities sales, traders, investment bankers, and financial advisors. Front office departments offer products and services such as **Merger and Acquisition (M&A)** and IPO advisory, wealth management, and the trading of financial assets such as equity (e.g., stocks), fixed income (e.g., bonds), commodities (e.g., oil), and currency products. Let's examine some specific business functions in the front office area.

## Sales trading and research

In sales trading, a firm's sales staff monitors investment news such as earnings reports or M&A activities to identify investment opportunities for their institutional clients. The trading staff then execute the trades for their clients, also known as agency trading. Additionally, trading staff can execute trades for their firm, which is known as **prop trading**. As trading staff often deal with large quantities of securities, it is crucial to optimize the trading strategy to acquire shares at favorable prices without driving up prices.

Research teams support sales and trading staff by analyzing equities and fixed-income assets and providing recommendations. Algorithmic trading is another type of trading that uses a computer to execute trades automatically based on predefined logic and market conditions.

The following diagram illustrates the business flow of a sales trading desk and how different players interact to complete a trading activity:

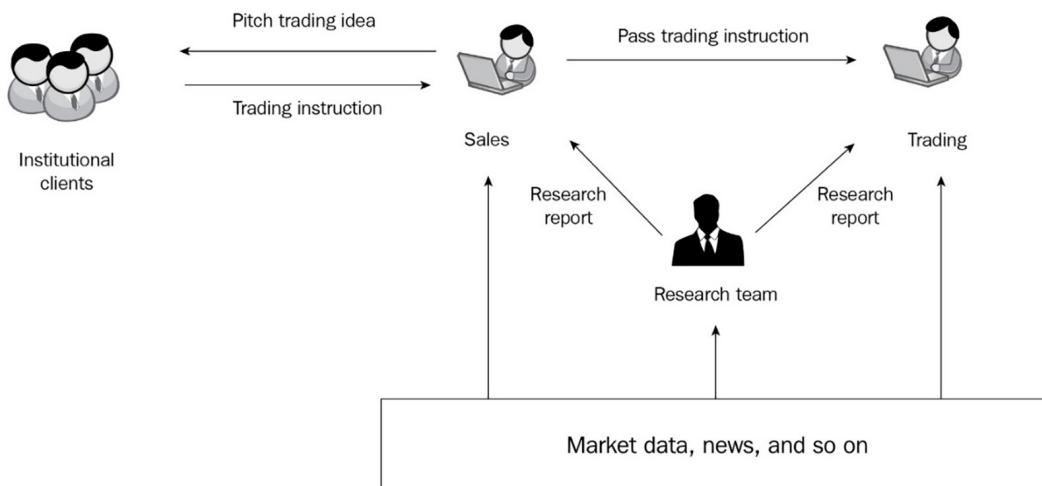


Figure 43.1: Sales, trading, and research

In the domain of sales trading and research, there are several core challenges that professionals in this industry face on a regular basis. These challenges revolve around generating accurate market insights, making informed investment decisions, and achieving optimal trading executions. The following are examples of these challenges:

- Tight timeline faced by research analysts to deliver research reports.
- Collecting and analyzing large amounts of market information to develop trading strategies and make trading decisions.

- Monitoring the markets continuously to adjust trading strategies.
- Achieving the optimal trading at the preferred price without driving the market up or down.

Sales trading and research offer numerous opportunities for ML. By leveraging **natural language processing (NLP)** and, increasingly, **large language models (LLMs)**, key entities such as people, events, organizations, and places can be automatically extracted from various data sources such as **Securities and Exchange Commission (SEC)** filing, news announcements, and earnings call transcripts.

NLP can also help discover relationships between entities and assess market sentiment toward a company and its stock by analyzing large amounts of news, research reports, and earnings calls to inform trading decisions. **Natural language generation (NLG)** powered by LLMs can assist with narrative writing and report generation, while computer vision has been used to help identify market signals from alternative data sources such as satellite images to understand business patterns such as retail traffic. In trading, ML models can sift through large amounts of data to discover patterns to inform trading strategies, such as pair trading, using data points such as company fundamentals, trading patterns, and technical indicators. In trade execution, ML models can help estimate trading costs and identify optimal trading execution strategies and execution paths to minimize costs and optimize profits. Financial services companies generate massive amounts of time series data, such as the prices of different financial instruments, which can be used to discover market signals and estimate market trends. As a result, ML has been adopted for use cases such as financial time series classification and forecasting financial instruments and economic indicators.

## Investment banking

When corporations, governments, and institutions need access to capital to fund business operations and growth, they engage investment bankers for capital raising (e.g., the selling of stocks or bonds) services. The following diagram illustrates the relationship between investment bankers and investors. In addition to capital raising, investment bankers also engage in M&A advisory to assist their clients in negotiating and structuring merger and acquisition deals from start to finish. Investment banking staff take on many activities, such as financial modeling, business valuation, pitch book generation, and transaction document preparation to complete and execute an investment banking deal. Additionally, they are responsible for general relationship management and business development management activities.

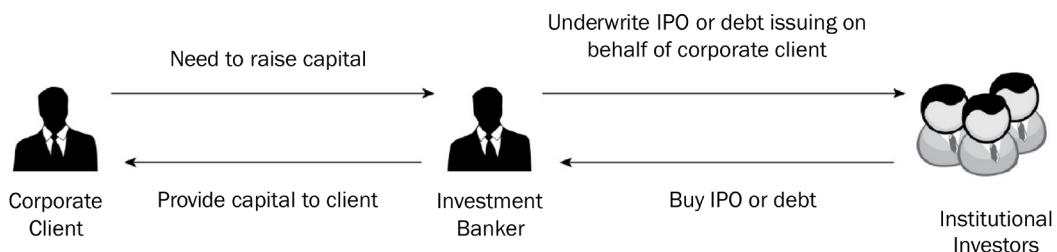


Figure 43.2: Investment banking workflow

The investment banking workflow poses a significant challenge in searching and analyzing large amounts of **structured** (earning, cashflow, estimates) and **unstructured** data (annual reports, filing, news, and internal documents). Typical junior bankers spend many hours searching for documents that might contain useful information and manually extracting information from the documents to prepare pitch books or perform financial modeling. To tackle this labor-intensive problem, investment banks have been exploring and adopting ML solutions. One such solution is using NLP to extract structured tabular data automatically from large amounts of PDF documents. Specifically, **named entity recognition (NER)** techniques can help with automatic entity extraction from documents. ML-based reading comprehension and question-answering technology can assist bankers in finding relevant information from large volumes of text quickly and accurately using natural human questions instead of simple text string matching. Documents can also be automatically tagged with metadata and classified using the ML technique to improve document management and information retrieval. Additionally, ML can help solve other challenges in investment banking, such as linking company identifiers from different data sources and resolving different variations of company names.

## Wealth management

The **wealth management (WM)** business involves advising clients on wealth planning and structuring to grow and preserve clients' wealth. Unlike investment advisory-focused brokerage firms, WM firms also offer tax planning, wealth preserving, and estate planning to meet their clients' more complex financial planning goals. WM firms engage clients to understand their life goals and spending patterns and design customized financial planning solutions for their clients. However, WM firms face various challenges in their operations, such as:

- WM clients are demanding more holistic and personalized financial planning strategies for their WM needs.
- WM clients are becoming increasingly tech-savvy, and many are demanding new channels of engagement in addition to direct client-advisor interactions.
- WM advisors need to cover increasingly more clients while maintaining the same personalized services and planning.
- WM advisors need to keep up with market trends, diverse client needs, and increasingly complex financial products and service portfolios to meet client needs.

WM firms are adopting ML-based solutions to offer more personalized services to their clients. By analyzing clients' transaction history, portfolio details, conversation logs, investment preferences, and life goals, ML models are built to recommend the most suitable investment products and services. These models take into account the clients' likelihood of accepting an offer and business metrics such as expected value to suggest the next best action. This enables wealth management firms to offer tailored financial planning solutions to their clients. The following diagram illustrates the concept of the **Next Best Action** method:

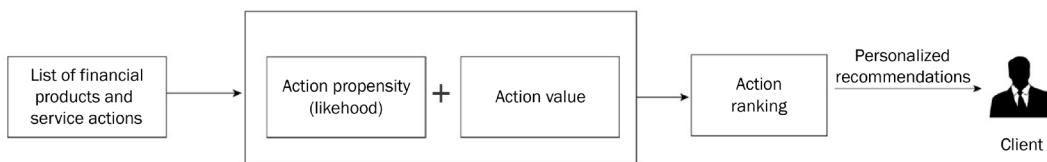


Figure 43.3: Next Best Action recommendation

WM firms are also increasingly leveraging AI and ML to enhance client engagement and experience, automate routine tasks, and equip **financial advisors (FAs)** with the right knowledge during client engagements. For instance, firms are building virtual assistants that provide personalized answers to client inquiries and automatically fulfill their requests. FAs are being equipped with AI-based solutions that can transcribe audio conversations to text for text analysis, assess clients' sentiment, and alert FAs to potential customer churn. Additionally, intelligent search and question-answering techniques are being adopted to enable FAs to quickly and accurately find relevant information during client engagements.

## Capital market back office operations

The back office is the backbone of financial services companies. While it may not be client-facing, it handles critical support activities such as trade settlement, record keeping, and regulatory compliance. As a result, it's an area that has been quick to

adopt ML. With the financial benefits and cost-saving it can bring, not to mention its ability to improve regulatory compliance and internal controls, it's no surprise that ML is transforming the back office. Let's explore some of the business processes where ML can make a significant impact.

## Net Asset Value review

Financial services companies that offer mutual funds and ETFs need to accurately reflect the values of the funds for trading and reporting purposes. They use a **Net Asset Value (NAV)** calculation, which is the value of an entity's assets minus its liability, to represent the value of the fund. NAV is the price at which an investor can buy and sell the fund. Every day, after the market closes, fund administrators must calculate the NAV price with 100% accuracy, which involves five critical steps:

1. Stock reconciliation
2. Reflection of any corporate actions
3. Pricing the instrument
4. Booking, calculating, and reconciling fees and interest accruals, as well as cash reconciliation
5. NAV/price validation

The NAV review process is depicted in the following diagram:



Figure 43.4: Net Asset Value review process

Step 5 is the most vital because if it is done incorrectly, the fund administrator could be liable, which can result in monetary compensation to investors. However, traditional methods of flagging exceptions using fixed thresholds often result in a high number of false positives, wasting analysts' time. Due to the large volumes of data involved in the investigation and review process, including instrument prices, fees, interest, assets, cash positions, and corporate actions data, efficient and accurate methods are essential.

The main objective of the NAV validation step is to detect pricing exceptions, which can be treated as an anomaly detection challenge. To identify potential pricing irregularities and flag them for further human investigation, financial services companies have implemented ML-based anomaly detection solutions. This approach has demonstrated a substantial reduction in false positives and saved a significant amount of time for human reviewers.

## Post-trade settlement failure prediction

After the front office executes a trade, several post-trade processes must be completed to finalize the trade, such as settlement and clearance. During post-trade settlement, buyers and sellers compare trade details, approve the transaction, update records of ownership, and arrange for securities and cash to be transferred. Although most trade settlements are handled automatically using **straight-through processing**, some trade settlements may fail due to various reasons, such as a seller's failure to deliver securities or a buyer's payment failure. When this occurs, brokers may need to use their reserves to complete the transaction. To ensure the stockpile is set at the correct level so that valuable capital can be used elsewhere, predicting settlement failure is critical.

The following diagram illustrates the trading workflow where buyers and sellers buy and sell their securities at an exchange through their respective brokerage firms:

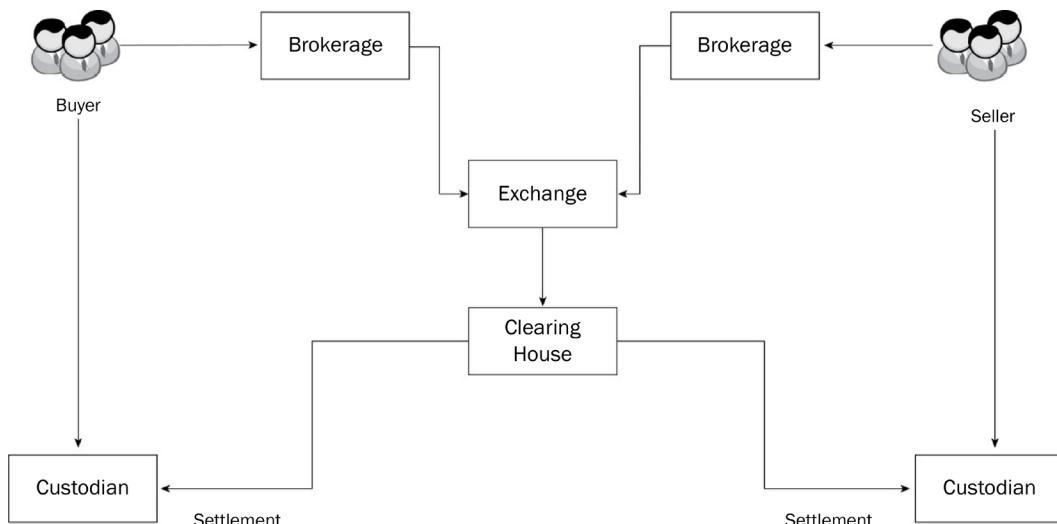


Figure 43.5: Trading workflow

After the trade is executed, a clearing house such as DTCC would handle the clearance and settlement of the trades with the respective custodians for the buyer and sellers.

Brokerage houses aim to optimize transaction rates and reduce capital expenditure costs by maintaining the right amount of stockpile reserve. To achieve this, ML models are utilized to predict trade failures early in the process. With these predictions, brokers can take preventive or corrective actions to prevent or resolve trade failures.

## Risk management and fraud

The middle office of financial services firms, including investment banks and commercial banks, encompasses risk management and fraud prevention. Due to their significant financial and regulatory implications, these areas are among the top areas for ML adoption in financial services. ML has many use cases in fraud prevention and risk management, such as detecting money laundering, monitoring trade activities, identifying credit card transaction fraud, and uncovering insurance claim fraud. In the following sections, we will examine some of these use cases in more detail.

### Anti-money laundering

Financial institutions are obligated to prevent money laundering by detecting activities that aid illegal money laundering. **Anti-money laundering (AML)** regulations require financial services companies to devote substantial resources to combat AML activities. Traditionally, rule-based systems have been used to detect AML activities, but they have a limited view and can only detect well-known frauds from the past. Furthermore, it is challenging to include a large number of features to be evaluated in a rule-based system, and it is difficult to keep the rules up to date with new changes. ML-based solutions have been leveraged in multiple areas of AML, such as:

- Network link analysis to reveal the complex social and business relationships among different entities and jurisdictions.
- Clustering analysis to find similar and dissimilar entities to spot trends in criminal activity patterns.
- Deep learning-based predictive analytics to identify criminal activity.
- NLP to gather as much information as possible for the vast number of entities from unstructured data sources.

The following diagram illustrates the data flow for AML analysis, the reporting requirements for regulators, and internal risk management and audit functions:

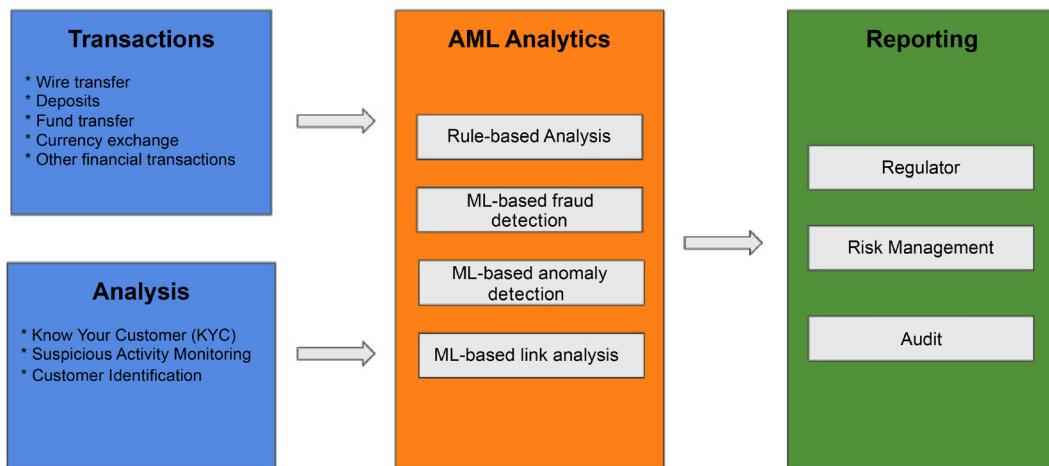


Figure 43.6: Anti-money laundering detection flow

An AML platform takes data from many different sources, including transaction data and internal analysis data such as **Know Your Customer (KYC)** and **Suspicious Activity** data. This data is processed and fed into different rule- and ML-based analytics engines to monitor fraudulent activities. The findings can then be sent to internal risk management and auditing, as well as regulators.

## Trade surveillance

Traders at financial firms are intermediaries who buy and sell securities and other financial instruments on behalf of their clients. They execute orders and advise clients on entering and exiting financial positions. To prevent market abuse by traders or financial institutions, **trade surveillance** is employed to identify and investigate potential market abuse. Examples of market abuse include market manipulation, such as the dissemination of false and misleading information, manipulating trading volumes through large amounts of wash trading, and insider trading through the disclosure of non-public information. Financial institutions are required to comply with market abuse regulations such as **Market Abuse Regulation (MAR)**, **Markets in Financial Instruments Directive II (MiFID II)**, and other internal compliance to protect themselves from reputational and financial damages. Enforcing trade surveillance can be challenging due to high noise/signal ratios and many false positives, which increases the cost of case processing and investigations. One typical approach to abuse detection is to build complex rule-based systems with different fixed thresholds for decision-making.

There are multiple ways to frame trade surveillance problems as ML problems, including:

- Framing the abuse detection of activities as a classification problem to replace rule-based systems.
- Framing data extraction information such as entities (for example, restricted stocks) from unstructured data sources (for example, emails and chats) as NLP entity extraction problems.
- Transforming entity relationship analysis (for example, trader-trader collaborations in market abuse) into ML-based network analysis problems.
- Treating abusive behaviors as anomalies and using unsupervised ML techniques for anomaly detection.

Many different datasets can be useful for building ML models for trade surveillance such as P&L information, positions, order book details, e-communications, linkage information among traders and their trades, market data, trading history, and details such as counterparty details, trade price, order type, and exchanges.

The following diagram illustrates the typical data flow and business workflow for trade surveillance management within a financial services company:

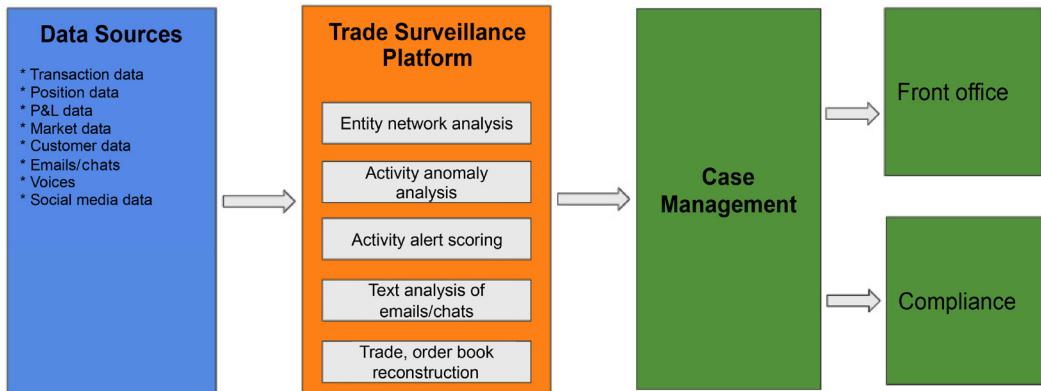


Figure 43.7: Trade surveillance workflow

A trade surveillance system monitors many different data sources, and feeds its findings to both the front office and compliance department for further investigation and enforcement.

## Credit risk

Banks face the potential risk of borrowers not being able to make required loan payments when issuing loans to businesses and individuals. This results in financial loss for banks, including both principal and interest from activities such as mortgage and credit card loans. To mitigate this default risk, banks utilize credit risk modeling to evaluate the risk of making a loan, focusing on two main aspects:

- The probability that the borrower will default on the loan.
- The impact on the lender's financial situation.

Traditional human-based reviews of loan applications are slow and error-prone, resulting in high loan processing costs and lost opportunities due to incorrect and slow loan approval processing. The following diagram depicts a typical business workflow for a credit risk assessment and the various decision points within the process:

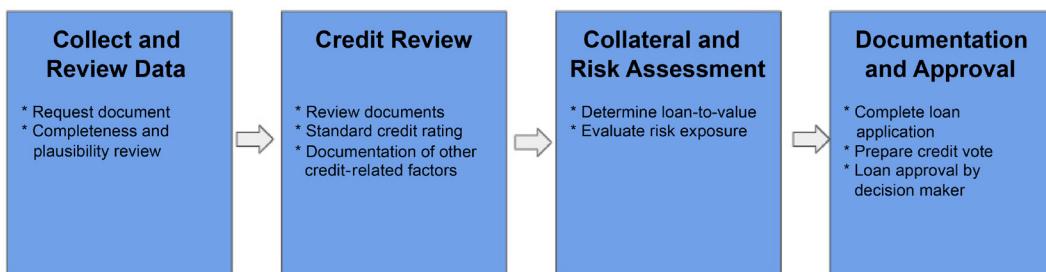


Figure 43.8: Credit risk approval workflow

To reduce the credit risk associated with loans, many banks have widely adopted ML techniques to predict loan default and associated risk scores more accurately and quickly. The credit risk management modeling process involves collecting financial information from borrowers, such as income, cash flow, debt, assets and collaterals, the utilization of credits, and other information such as loan type and loan payment behaviors. However, this process can involve analyzing large amounts of unstructured data from financial statements. To address this challenge, ML-based solutions such as **Optical Character Recognition (OCR)** and NLP information extraction and understanding have been widely adopted for automated intelligence document processing.

## Insurance

The insurance industry comprises various sub-sectors, each offering distinct insurance products, such as life insurance, property and casualty insurance, and accident and health insurance. Apart from insurance companies, insurance technology providers

also play a critical role in the industry. Most insurance companies have two primary business processes, namely, insurance underwriting and insurance claim management.

## Insurance underwriting

Insurance companies evaluate the risks of offering insurance coverage to individuals and assets through a process known as **insurance underwriting**. Using actuarial data and insurance software, insurance companies determine the appropriate insurance premium for the risks they are willing to undertake. The underwriting process varies depending on the insurance products offered. For instance, the steps involved in underwriting property insurance are typically as follows:

1. The customer files an insurance application through an agent or insurance company directly.
2. The underwriter at the insurance company assesses the application by considering different factors such as the applicant's loss and insurance history, and actuarial factors to determine whether the insurance company should take on the risk, and what the price and premium should be for the risk. Then, they make an additional adjustment to the policy, such as the coverage amount and deductibles.
3. If the application is accepted, then an insurance policy is issued.

During the underwriting process, a large amount of data needs to be collected and reviewed by an underwriter, who estimates the risk of a claim based on the data and personal experience to determine a justifiable premium. However, human underwriters are limited in their ability to review only a subset of data and can introduce personal bias into the decision-making process. In contrast, ML models can analyze vast amounts of data and make more accurate, data-driven decisions regarding risk factors such as claim probability and outcome, while also making faster decisions than human underwriters. Additionally, ML models can utilize large amounts of historical data and risk factors to generate recommended premiums for policies, reducing the amount of time needed for assessment.

## Insurance claim management

**Insurance claim management** involves the process of evaluating claims made by policyholders and providing compensation for the losses incurred, as specified in the policy agreement. The specific steps in the claim process can vary depending on the type of insurance. For example, in the case of property insurance, the following steps are usually followed:

1. The insured person submits a claim, along with supporting evidence like photographs of the damage and a police report (in the case of automobiles).
2. An adjuster is assigned by the insurance company to assess the extent of the damage.
3. The adjuster evaluates the damage, carries out fraud assessments, and sends the claim for payment approval.

Some of the main challenges that are faced in the insurance claim management process are as follows:

- Time-consuming manual effort is needed for the damaged/lost item inventory process and data entry.
- The need for speedy claim damage assessment and adjustment.
- Insurance fraud.

Insurance companies collect a lot of data during the insurance claim process, such as property details, details and photos of the damaged items, the insurance policy, the claims history, and historical fraud data.

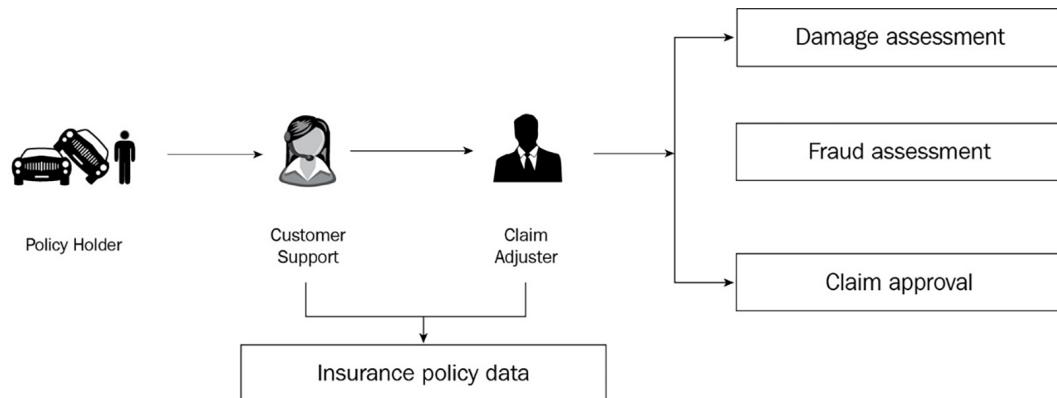


Figure 43.9: Insurance claim management workflow

ML can help automate manual processes, such as data extraction from documents and identification of insured objects from pictures, which can reduce the amount of manual effort required for data collection. For damage assessment, ML can be used to estimate the cost of repair and replacement, which can speed up the claim processing. Additionally, ML can be used to detect exceptions in insurance claims and predict potential fraud, which can help to identify cases for further investigation in the fight against insurance fraud.

## ML use cases in media and entertainment

The **media and entertainment** (M&E) industry encompasses the production and distribution of various forms of content, such as films, television, streaming content, music, games, and publishing. The industry has undergone significant changes due to the growing prevalence of streaming and **over-the-top** (OTT) content delivery over traditional broadcasting. M&E customers, having access to an ever-increasing selection of media content, are shifting their consumption habits and demanding more personalized and enhanced experiences across different devices, anytime, anywhere. The industry is also characterized by intense competition, and to remain competitive, M&E companies need to identify new monetization channels, improve user experience, and improve operational efficiency. The workflow for media production and distribution is illustrated in the following diagram:

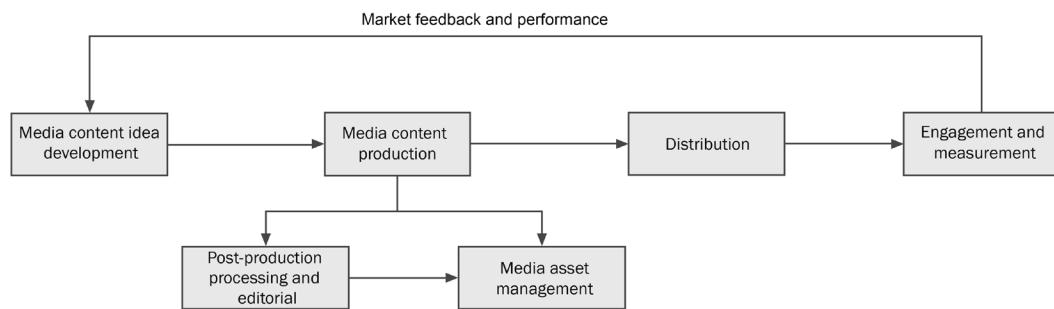


Figure 43.10: Media production and distribution workflow

In recent years, I have seen M&E companies increasingly adopting ML in the different stages of the media lifecycle, such as content generation and content distribution, to improve efficiency and spur business growth. For instance, ML has been used to enhance content management and search, develop new content development, optimize monetization, and enforce compliance and quality control.

## Content development and production

During the initial planning phase of the film production lifecycle, content producers need to make decisions on the next content based on factors such as estimated performance, revenue, and profitability. To aid this process, filmmakers have adopted ML-based predictive analytics models to help predict the popularity and profitability of new ideas by analyzing factors such as casts, scripts, the past performance of different films, and target audience. This allows producers to quickly eliminate ideas with limited market potential and focus their efforts on developing more promising and profitable ideas.

To support personalized content viewing needs, content producers often segment long video content into smaller micro-segments around certain events, scenes, or actors, so that they can be distributed individually or repackaged into something more personalized to individual preferences. This ML-based approach can be used to create video clips by detecting elements such as scenes, actors, and events for different target audiences with varying tastes and preferences.

## Content management and discovery

M&E companies with vast digital content assets need to curate their content to create new content for new monetization opportunities. To support this, these companies need rich metadata for the digital assets to enable different content to be searched and discovered. Consumers also need to search for content easily and accurately for different usages, such as for personal entertainment or research. Without metadata tagging, discovering relevant content is quite challenging. As part of the digital asset management workflow, many companies hire humans to review and tag this content with meaningful metadata for discovery. However, manual tagging is very costly and time-consuming, leading to insufficient metadata for effective content management and discovery.

Computer vision models can automatically tag image and video content for items such as objects, genres, people, places, or themes. ML models can also interpret the meaning of textual content such as topics, sentiment, entities, and sometimes video. Audio content can be transcribed into text using ML techniques for additional text analysis. ML-based text summarization can help you summarize long text as part of the content metadata generation. The following diagram illustrates where ML-based analysis solutions can be incorporated into the media asset management flow:

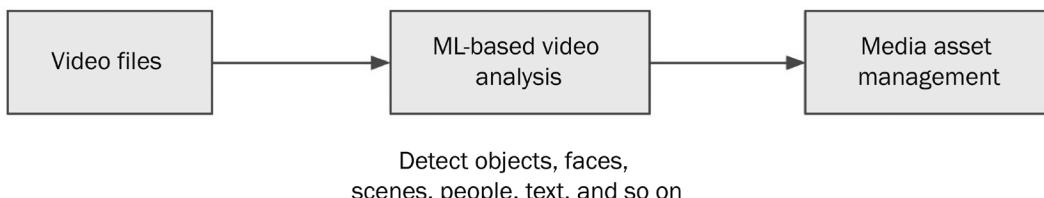


Figure 43.11: ML-based media analysis workflow

ML-based solutions are being increasingly adopted by M&E companies to streamline media asset management workflows. Overall, these solutions can lead to significant time and cost savings for M&E companies while improving the user experience for consumers.

## Content distribution and customer engagement

Nowadays, media content such as films and music is increasingly being distributed through digital **video on demand (VOD)** and live streaming on different devices, bypassing traditional media such as DVDs and broadcasting, providing consumers with a variety of media content choices. As a result, media companies are facing challenges in customer acquisition and retention. To keep users engaged and on their platforms, M&E companies are focusing on highly personalized product features and content. One effective way to achieve highly personalized engagement is through a content recommendation engine, which uses viewing and engagement behavior data to train ML models that target individuals based on their preferences and viewing patterns.

This allows for a diverse range of media content to be recommended to users, including videos, music, and games.

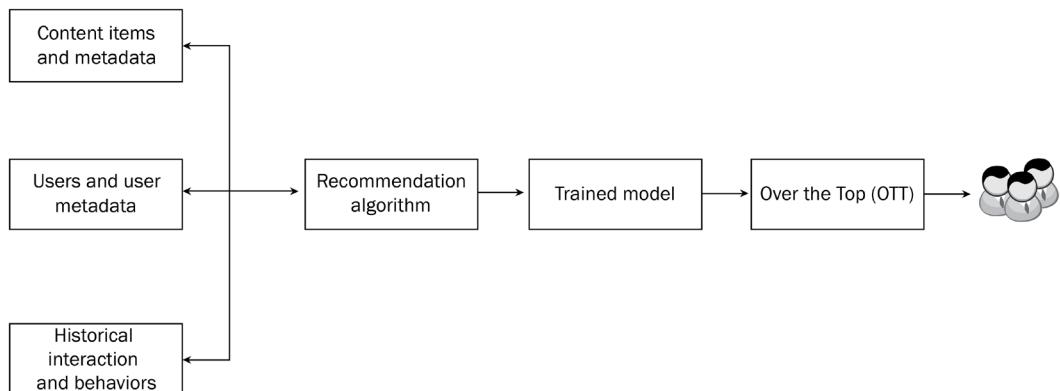


Figure 43.12: Recommendation ML model training

Recommendation technologies have been around for many years and have improved greatly over the years. Nowadays, recommendation engines can learn patterns using multiple data inputs, including historical interactions, sequential patterns, and the metadata associated with the users and content. Modern recommendation engines can also learn from the user's real-time behaviors/decisions and make dynamic recommendations based on them.

## ML use cases in healthcare and life sciences

The healthcare and life science industry is one of the largest and most important industries in the world, serving millions of people globally. The industry encompasses a wide range of sectors, each with its own unique set of challenges and opportunities. One of the most significant sectors within healthcare and life science is the drugs sector, which includes biotechnology firms, pharmaceutical companies, and manufacturers of genetic drugs. These companies are responsible for developing and producing medications to treat various illnesses and diseases, ranging from minor ailments to life-threatening conditions. They invest heavily in research and development to discover new drugs and therapies, often requiring significant financial resources and years of clinical trials before a product can be brought to market.

Another important sector within healthcare and life science is the medical equipment industry, which manufactures a wide range of products ranging from standard equipment such as syringes and bandages to hi-tech equipment such as MRI machines and surgical robots. These companies are at the forefront of innovation, constantly developing new technologies to improve patient outcomes and advance medical practices.

Managed healthcare is another critical sector within the healthcare and life science industry. These companies provide health insurance policies, covering medical costs for their policyholders. This sector faces many challenges, such as rising healthcare costs and changing regulations, and it requires careful management and planning to provide affordable and effective coverage to policyholders.

Health facilities such as hospitals, clinics, and labs are another significant sector within healthcare and life sciences. These facilities provide medical care and services to patients, ranging from routine check-ups to complex surgeries. They require significant resources to operate, such as skilled medical staff, state-of-the-art equipment, and advanced technologies.

Government agencies, such as the **Centers for Disease Control and Prevention (CDC)** and the **Food and Drug Administration (FDA)**, play a critical role in regulating and overseeing the healthcare and life sciences industry. They are responsible for ensuring the safety and efficacy of drugs and medical devices, monitoring public health issues, and developing policies to promote public health and safety.

In recent years, the healthcare and life sciences industry has seen a significant increase in the adoption of AI and ML. These technologies have been used to address complex challenges in the industry, such as improving patient outcomes, reducing costs, and accelerating drug discovery and development. With the availability of large amounts

of health data, including electronic health records, genomics data, and medical imaging, ML algorithms can extract meaningful insights and patterns to inform clinical decision-making, disease diagnosis, and treatment planning. In this way, ML is transforming the healthcare and life science industry, enabling practitioners and researchers to make more informed decisions and improve patient outcomes.

## Medical imaging analysis

**Medical imaging** is the process and technique of creating a visual representation of the human body for medical analysis. Medical professionals, such as radiologists and pathologists, use medical imaging to assist with medical condition assessments and prescribe medical treatments. However, the increasing demand for medical imaging analysis has led to a shortage of qualified professionals to review these images. This challenge has been partially addressed through the adoption of ML in medical imaging analysis.

One ML-based solution involves treating medical imaging analysis as a computer vision object detection and classification problem. For example, in the case of cancer cell detection, cancerous tissues can be identified and labeled in existing medical images as training data for computer vision algorithms.

Once trained, these models can be used to automate the screening of a large number of X-ray images, highlighting those that are important for the pathologists to review. This approach has the potential to improve the efficiency and accuracy of medical imaging analysis, reducing the workload of medical professionals and improving patient outcomes. The following diagram illustrates the process of training a computer vision model using labeled image data in medical imaging analysis:

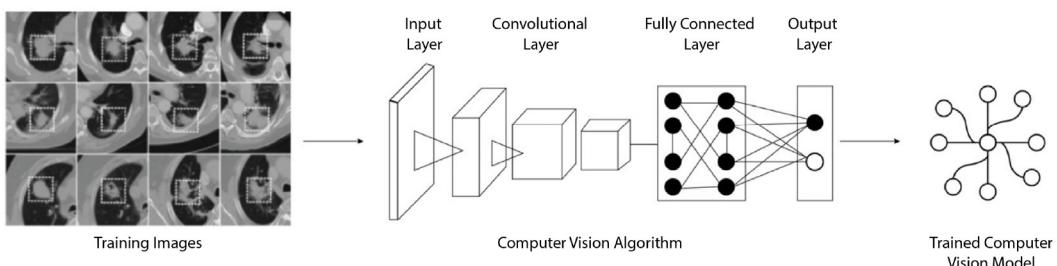


Figure 43.13: Using computer vision for cancer detection

Medical image analysis can be enhanced by combining image data with other clinical data, such as a patient's medical history, laboratory test results, and genetic data. This combination of data can improve the accuracy of medical diagnosis and enable early

detection of diseases. For example, in the case of breast cancer, medical imaging can be combined with other clinical data, such as age, family history, and hormonal status, to develop a more accurate prediction model for breast cancer risk.

The combined data can be fed into ML algorithms to train a joint model that takes into account all the available information to make more accurate predictions. The ML model can learn complex patterns and relationships between various features in the data, including the images, and predict the likelihood of the presence of a particular condition.

The use of non-image data in conjunction with medical images has the potential to provide a more comprehensive understanding of a patient's health status and allow for earlier and more accurate diagnosis of diseases. In addition, it can help medical professionals develop more effective treatment plans for patients based on their specific health conditions.

## Drug discovery

The process of drug discovery and development is a crucial aspect of the healthcare and life science industry. The first stage is discovery and development, which involves identifying a lead compound that can target a particular protein or gene to act as a drug candidate. This process typically involves basic research in fields such as molecular biology, biochemistry, and pharmacology.

Once a lead compound has been identified, it undergoes preclinical research to determine its efficacy and safety. This stage involves extensive laboratory testing and animal studies to understand the pharmacokinetics and pharmacodynamics of the drug. The ultimate goal is to identify the most promising drug candidates to move forward into clinical development.

Clinical development is the next stage, which involves clinical trials and volunteer studies to fine-tune the drug and optimize its dosage, safety, and efficacy. This stage is divided into three phases, with each phase becoming progressively larger and more expensive. The goal is to demonstrate that the drug is both safe and effective for its intended use.

After the clinical development stage, the drug undergoes an FDA review, where it is evaluated holistically to either approve or reject it. This involves a rigorous evaluation of the drug's safety, efficacy, and manufacturing processes.

Finally, post-market monitoring is carried out to ensure the safety of the drug once it has been approved and is available to the public. This involves ongoing monitoring of adverse reactions, side effects, and other safety concerns.

In the field of drug discovery and development, ML has emerged as a powerful tool in recent years. ML techniques can be used for various purposes, such as predicting the efficacy and toxicity of a drug candidate and identifying new drug targets. Additionally, ML can aid in one of the key challenges in drug discovery: understanding protein folding. Protein folding is the process by which a protein molecule assumes its functional three-dimensional shape. ML algorithms can analyze the complex interactions between protein molecules and predict their folding patterns. This can provide insights into the mechanisms of diseases and facilitate the discovery of new drugs targeting specific proteins. By leveraging large and complex datasets, ML can accelerate the drug development process, reduce costs, and improve the safety and efficacy of new drugs.

ML has been used to optimize clinical trials, such as identifying potential cohorts for clinical trials, an important step in the drug discovery process. By analyzing large amounts of patient data, ML models can help identify groups of patients that are most likely to benefit from a particular treatment. For example, in cancer research, ML has been used to analyze genetic and clinical data from patients to identify specific patient subgroups that may respond better to a particular drug.

ML models can also help optimize clinical trial design by predicting the likelihood of success of a particular trial. For instance, ML algorithms can be used to analyze historical clinical trial data to identify factors that are associated with successful trials, such as patient characteristics, dosage, and treatment duration. This information can then be used to design more effective trials in the future.

## Healthcare data management

Every day, the healthcare industry generates and collects a vast amount of patient healthcare data, which comes in various formats, such as handwritten notes, insurance claim data, recorded medical conversations, and medical images, such as X-rays. This data is crucial for developing a comprehensive view of patients or supporting medical coding for medical billing processes. However, extracting valuable insights from these sources often requires significant manual processing, which is both expensive and error-prone, often carried out by people with health domain expertise. Consequently, a substantial amount of patient healthcare data remains unutilized in its original form. ML-based approaches have been adopted to automate this process and improve the accuracy and efficiency of data processing. For instance, **natural language processing (NLP)** models can extract information from unstructured medical notes, while computer vision algorithms can analyze medical images to detect and diagnose diseases. This enables healthcare organizations to obtain valuable insights from patient healthcare data that was previously untapped.

The process of extracting information from unstructured data sources using ML is depicted in the following diagram, showcasing the flow of data and the integration of ML into different healthcare tasks, such as medical coding and clinical decision support.

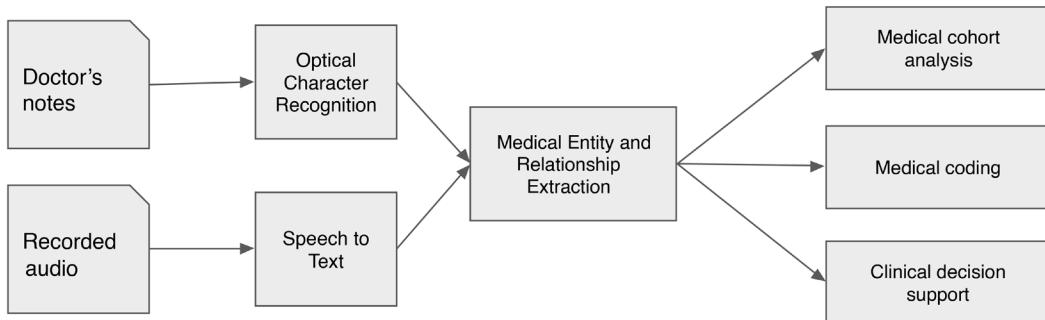


Figure 43.14: Medical data management

Overall, the adoption of ML-based solutions in healthcare is enabling healthcare organizations to unlock valuable insights from unstructured data sources, which can help improve patient outcomes, optimize resource utilization, and reduce costs.

## ML use cases in manufacturing

The manufacturing industry is a vast sector that is responsible for creating a wide range of physical products, such as consumer goods, electronics, automobiles, furniture, building materials, and more. Each sub-sector of manufacturing requires a specific set of tools, resources, and expertise to successfully produce the desired products.

The manufacturing process generally involves several stages, including product design, prototyping, production, and post-manufacturing service and support. During the design phase, manufacturers work on conceptualizing and planning the product. This includes defining the product's features, materials, and production requirements. In the prototyping stage, a small number of products are created to test their functionality and performance.

Once the product design has been finalized, manufacturing and assembling takes place. This is the stage where raw materials are transformed into finished products. Quality control is a critical aspect of the manufacturing process, as manufacturers need to ensure that each product meets the required standards and specifications. Finally, post-manufacturing service and support involves activities such as repair and maintenance, customer support, and product upgrades. The goal is to provide ongoing value to customers and to ensure the product continues to perform optimally.

The following diagram illustrates the typical business functions and flow in the manufacturing sector:

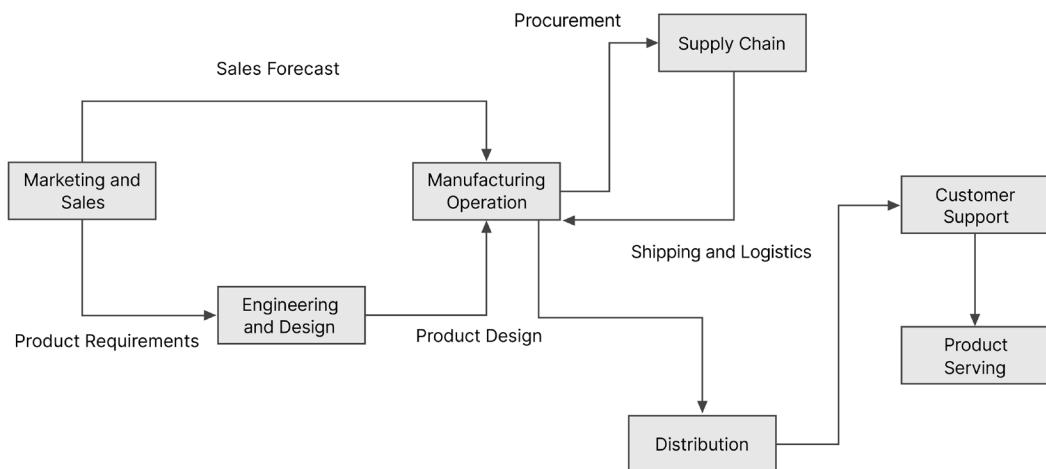


Figure 43.15: Manufacturing business process flow

AI and ML have become crucial tools in the manufacturing industry, driving significant improvements in various stages of the manufacturing process. For instance, ML algorithms are used to forecast sales, which allows companies to make informed decisions about production volume and material procurement. This, in turn, leads to more efficient inventory management, reduction of waste, and increased profitability.

Moreover, predictive machine maintenance is another area where AI and ML are making significant contributions. With the use of ML algorithms, manufacturers can analyze data from sensors and other sources to predict equipment failure before it occurs. This helps avoid unplanned downtime, reduces maintenance costs, and improves overall equipment effectiveness.

Quality control is another crucial area where AI and ML have made significant improvements. By analyzing data from sensors and cameras, ML algorithms can identify defective products or parts in real time, allowing for timely intervention to address issues in the manufacturing process.

In addition, AI and ML have been instrumental in automating various tasks in the manufacturing process. This includes the use of robots for assembling products, performing quality checks, and handling material movement. This not only improves manufacturing quality and yield but also helps ensure worker safety by reducing the risk of accidents in hazardous work environments.

Finally, AI and ML can also be used to optimize supply chain management, improving overall operational efficiency and reducing costs. ML algorithms can analyze data from multiple sources to identify inefficiencies and bottlenecks in the supply chain, enabling manufacturers to make data-driven decisions that improve production planning, inventory management, and distribution.

## Engineering and product design

Product design is a crucial aspect of the manufacturing process, where designers aim to create products that are both functional and appealing to consumers. During the design phase, designers need to strike a balance between their creative vision, the practical needs of the market, and the constraints of production. To achieve this, they may create multiple versions of a new product concept that cater to different needs and constraints. For instance, in the fashion industry, designers may analyze customer preferences in terms of color, texture, and style to develop new apparel designs and graphics that meet those demands.

The manufacturing industry has been leveraging generative design ML technology to assist with new product concept design. For example, generative AI, a type of machine learning, can be used in product design to generate a vast number of possible design variations that meet specific constraints and requirements. By inputting design constraints, such as cost, materials, and production capabilities, generative AI can produce thousands of design options that meet those criteria. This approach can significantly speed up the product design process and also enable designers to explore a broader range of design possibilities.

In addition to generative AI, ML techniques have proven to be invaluable in analyzing market requirements and estimating the potential of new products. By leveraging various data sources such as customer feedback, market trends, and competitor analysis, ML algorithms can predict the demand for new products accurately. Furthermore, ML models can analyze large amounts of data quickly, enabling businesses to stay ahead of the competition by identifying new market opportunities and trends. ML algorithms can also identify customer preferences, such as color, texture, style, and functionality, to guide the product development process. The ability of ML to analyze complex datasets can also provide insights into the underlying factors that influence consumer behavior and product preferences. This information can help businesses refine their product design and marketing strategies, leading to increased sales and revenue.

## Manufacturing operations – product quality and yield

In the manufacturing industry, quality control is crucial to ensure that the products meet the required standards and specifications. However, relying solely on human inspection can be time-consuming and expensive. That's why the adoption of computer vision-based technology has been a game-changer in the quality control process. Computer vision models can be trained using ML algorithms to identify defects and flaws in manufactured products. For instance, in the automotive industry, computer vision algorithms can detect even the slightest surface scratches, dents, or deformations that might affect the vehicle's performance. Moreover, computer vision-based technology can be applied to different stages of the manufacturing process, such as monitoring assembly lines, detecting defects in finished goods, and identifying problems with raw materials. The use of AI-powered systems in quality control not only enhances efficiency and reduces costs, but also ensures consistency and accuracy in the inspection process.

## Manufacturing operations – machine maintenance

Regular maintenance is crucial for industrial manufacturing equipment and machinery to ensure smooth operations and prevent unexpected failures. However, traditional maintenance practices that follow a regular maintenance schedule can be costly and may not always detect potential problems. Fortunately, ML-based predictive maintenance analytics have emerged as a solution to help manufacturers forecast potential problems in advance and reduce the risk of unforeseen equipment failures. By analyzing a variety of data, including telemetry data collected by **Internet of Things (IoT)** sensors, ML algorithms can predict whether a piece of equipment is likely to fail within a certain time window. The maintenance crew can then take proactive measures to prevent equipment failure and avoid costly repairs or replacements. This approach not only minimizes the risk of unplanned outages but also reduces overall maintenance costs and downtime.

## ML use cases in retail

The retail industry is a sector that sells consumer products directly to customers, either through physical retail stores or online platforms. Retailers acquire their merchandise from wholesale distributors or manufacturers directly. Over the years, the retail industry has undergone significant changes. The growth of e-commerce has

outpaced that of traditional retail businesses, compelling brick-and-mortar stores to adapt and innovate in-store shopping experiences to remain competitive. Retailers are exploring new approaches to enhance the shopping experience across both online and physical channels. Recent developments such as social commerce, augmented reality, virtual assistant shopping, smart stores, and 1:1 personalization have become key differentiators in the retail industry.

The retail industry is currently undergoing a transformation fueled by AI and ML technologies. Retailers are utilizing these technologies to optimize inventory, predict consumer demand, and deliver personalized and immersive shopping experiences. AI and ML algorithms can provide personalized product recommendations and enable virtual reality shopping, making it possible for shoppers to try on clothes virtually. Additionally, AI and ML technologies are being employed to enable cashier-less store shopping and to prevent fraudulent activities and shoplifting. Overall, the retail industry's adoption of AI and ML technologies is expected to enhance the shopping experience and enable retailers to meet the evolving needs and expectations of their customers.

## Product search and discovery

Online shopping has simplified the purchasing process for consumers, but searching for a product online can sometimes be difficult when you only have a picture and no information on the item's name or features. This is where deep learning-powered visual search technology comes in handy. This technology allows consumers to quickly identify similar-looking products by simply uploading a picture of the item they are looking for. Visual search technology works by creating a digital representation of the item's pictures, also known as encoding or embedding, and storing it in a high-performance item index. When a shopper needs to find a similar-looking item using a picture, the new picture is encoded into a digital representation and searched against the item index using an efficient distance-based comparison. The system then returns the items that are closest to the target item.

With visual search technology, consumers can easily find what they are looking for, even if they do not know the correct search terms. This technology has become increasingly popular among e-commerce retailers, and the architecture for building an ML-based image search capability is continually evolving to improve the accuracy and efficiency of visual search.

The following diagram illustrates an architecture for building an ML-based image search capability:

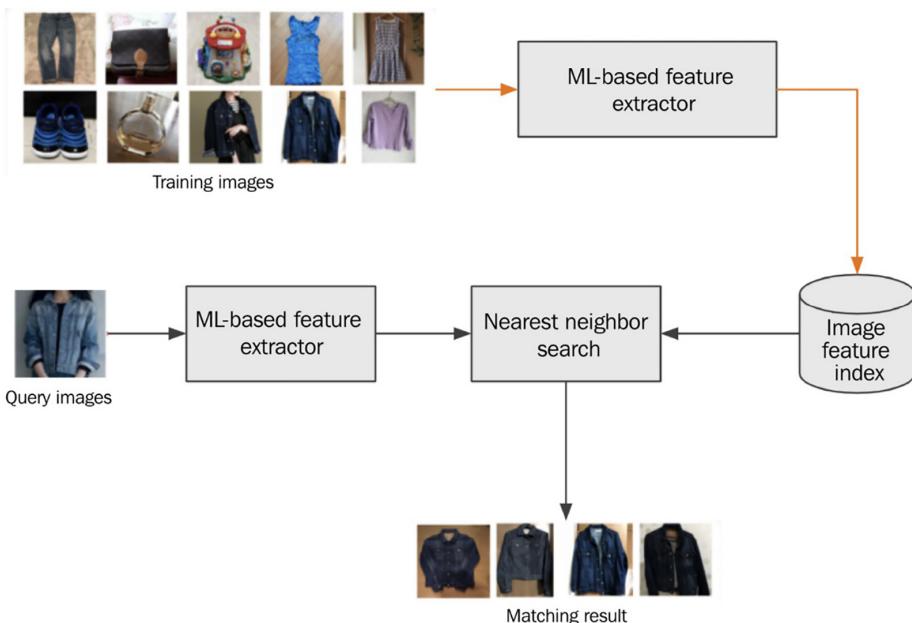


Figure 43.16: Image search architecture

Visual search-based recommendations have been adopted by many large e-commerce sites such as Amazon.com to enhance the shopping experience.

## Targeted marketing

Retailers employ various marketing campaigns and advertising tactics, including direct marketing emails or digital advertisements, to attract potential customers with incentives or discounts based on their demographics. The success of such campaigns relies heavily on effectively targeting the right customers to achieve a high conversion rate while minimizing advertising costs and reducing customer disturbances. ML models have been developed to optimize the effectiveness of marketing campaigns. These models use customer data and various demographic factors to identify potential customers who are most likely to convert, as well as to determine the most appropriate messaging and incentives for each customer segment. By leveraging ML techniques, retailers can improve the accuracy and efficiency of their marketing campaigns, resulting in a higher return on investment.

**Segmentation** is one traditional way to understand the different customer segments to help improve marketing campaigns' effectiveness. There are different ways to do segmentations with ML, such as unsupervised clustering of customers based on data such as basic demographic data. This allows you to group customers into several segments and create unique marketing campaigns for each segment.

A more effective targeted marketing approach is to use highly personalized **user-centric marketing campaigns**. They work by creating accurate individual profiles using large amounts of individual behavior data such as historical transaction data, response data to historical campaigns, and alternative textual data such as social media data. Highly personalized campaigns with customized marketing messages can be generated using these personal profiles for a higher conversion rate. The ML approach to user-centric targeted marketing predicts the conversion rate, such as the **click-through rate (CTR)**, for different users and sends ads to users with a high conversion rate. This can be a classification or regression problem by learning the relationship between the user features and the probability of conversion.

Contextual advertising is a targeted marketing technique that displays ads that are relevant to the content on a web page. It involves placing display or video ads on websites that match the advertisement's content, which can improve the ad's effectiveness. For instance, a cooking product advertisement may be placed on a cooking recipe website to reach a highly engaged audience. ML can assist with identifying the context of an ad to ensure it is placed appropriately. For instance, computer vision models can analyze video ads to detect objects, people, and themes to extract contextual information and match them to the website's content. By utilizing contextual advertising, marketers can increase the chances of their ads resonating with their target audience and achieving a higher click-through rate.

Generative AI presents a powerful opportunity for retailers to take their targeted marketing efforts to the next level through dynamically personalized content. By leveraging generative models, retailers can create customized images and text tailored to individual customers' preferences and interests. For example, an outdoor apparel company can generate customized ads featuring models wearing products suited to the local weather and climate.

## Sentiment analysis

Understanding consumer perception of their brand is crucial for retail businesses as it can have a significant impact on their success. With the rise of online platforms, consumers have become more vocal about their experiences and opinions, making it easier for retailers to monitor their brand reputation.

Retailers are adopting various techniques, including soliciting feedback from their shoppers and monitoring social media channels, to assess their customers' emotions and sentiments toward their brands and products. By effectively analyzing sentiment, retailers can identify areas for improvement, such as operational or product enhancements, as well as mitigating potentially malicious attacks on their brand reputation.

Sentiment analysis is a text classification problem that involves using labeled text data, such as product reviews, to determine whether the sentiment is positive, negative, or neutral. ML algorithms, including deep learning-based algorithms, can be used to train models that can detect sentiment in a piece of text. These models can then be used to automatically classify new text data, such as social media posts or customer feedback, to help retailers understand the overall sentiment toward their brand and products. With the recent advancements in generative AI, many LLMs provide pre-trained sentiment analysis capabilities without requiring you to train custom models with labeled data.

By leveraging sentiment analysis, retailers can gain valuable insights into customer preferences, identify areas for improvement, and make data-driven decisions to improve their overall customer experience.

## Product demand forecasting

Retail businesses rely on inventory planning and demand forecasting to manage inventory costs while maximizing revenue and avoiding out-of-stock situations. Traditional methods for demand forecasting, such as buyer surveys, expert opinions, and projections based on past demands, have limitations in accuracy and reliability.

To address these limitations, retailers are turning to statistical and ML techniques such as regression analysis and deep learning. These approaches can use historical demand and sales data, as well as other related data such as prices, holidays, special events, and product attributes, to create more accurate and data-driven demand forecasts.

Deep learning-based algorithms can be particularly effective in producing accurate demand forecasts by incorporating multiple data sources into the model. This approach involves training an ML model to recognize patterns and relationships in the data to generate highly accurate forecasts. The result is more reliable inventory planning that helps retailers optimize their inventory while maximizing revenue.

The following diagram illustrates the concept of building a deep learning model using multiple data sources to generate forecasting models:

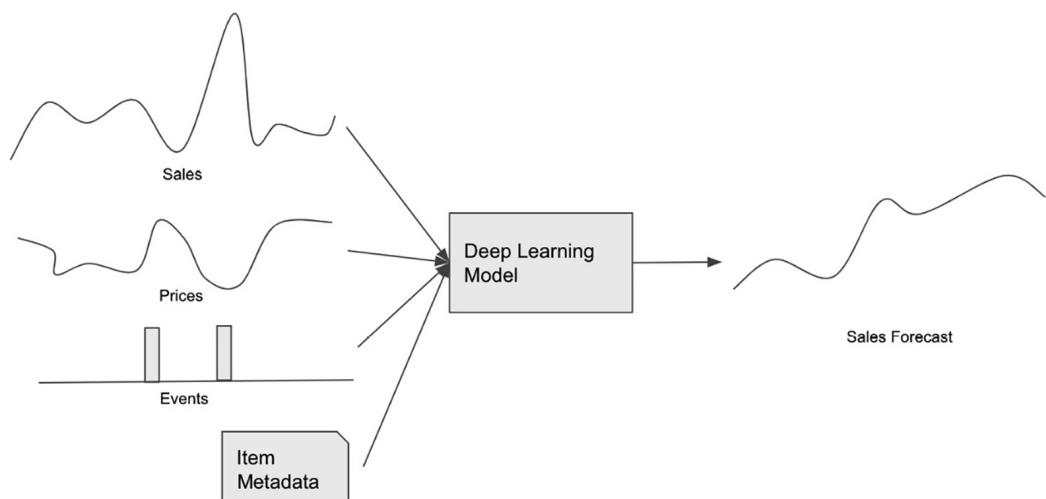


Figure 43.17: Deep learning-based forecasting model

ML-based forecasting models can generate both point forecasts (a number) or probabilistic forecasts (a forecast with a confidence score). Many retail businesses use ML to generate baseline forecasts, and then professional forecasters review them and make adjustments based on their expertise and other factors.

## ML use cases in the automotive industry

The automotive industry has undergone significant transformation in recent years, with technology playing a key role in shaping its evolution. AI and ML have emerged as powerful tools for automakers and suppliers to improve efficiency, safety, and customer experience. From production lines to connected cars, AI and ML are being used to automate processes, optimize operations, and enable new services and features.

### Autonomous vehicles

One of the most significant applications of AI and ML in the automotive industry is in autonomous driving. Automakers and tech companies are leveraging these technologies to build self-driving vehicles that can safely navigate roads and highways without human intervention. AI and ML algorithms are used to process data from sensors, cameras, and other inputs to make real-time decisions and actions, such as braking or changing lanes.

The system architecture of an **autonomous vehicle (AV)** consists of 3 main stages: 1) perception and localization, 2) decision and planning, and 3) control, as illustrated in the following diagram:

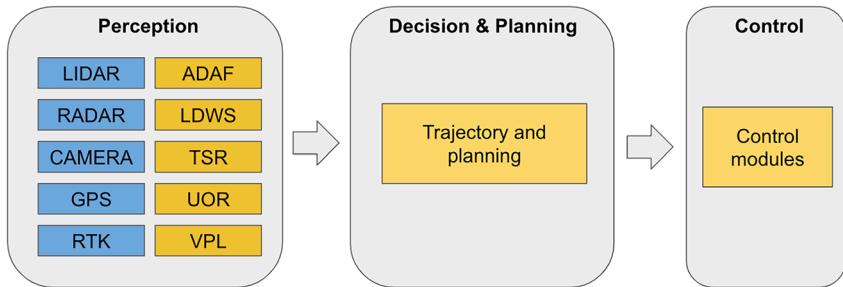


Figure 43.18: Autonomous vehicle system architecture

## Perception and localization

Perception is a crucial stage in autonomous driving, where the AV gathers information about its surroundings through a variety of sensors and determines its own position in relation to the environment. The AV employs sensors such as RADAR, LIDAR, cameras, and **real-time kinetic (RTK)** systems to capture data from the surrounding environment. This sensory data is then fed into recognition modules for further processing.

One of the key components in the perception stage is the **adaptive detection and recognition framework (ADAF)**, which utilizes ML algorithms to detect and classify objects such as pedestrians, vehicles, and obstacles in the AV's vicinity. Additionally, the AV incorporates modules such as **Lane Departure Warning System (LDWS)**, **Traffic Sign Recognition (TSR)**, **Unknown Obstacles Recognition (UOR)**, and **Vehicle Positioning and Localization (VPL)** to enhance its perception capabilities.

The perception stage forms a fundamental building block in the overall autonomous driving system. The accuracy and reliability of the perception module significantly impact the AV's ability to perceive and interpret its environment. Advances in ML algorithms, sensor technology, and sensor fusion techniques continue to improve the perception capabilities of autonomous vehicles, enabling them to operate safely and effectively in diverse and complex driving scenarios.

## Decision and planning

The decision and planning stage is a crucial aspect of autonomous driving, which controls the motion and behavior of the autonomous vehicle based on the data collected during the perception stage. AI and ML technologies play a vital role in this stage, which can be considered the brain of the AV. By analyzing data from sensors such as RADAR, LIDAR, and cameras, the decision and planning stage uses algorithms to determine the optimal path for the AV to follow.

AI/ML can help enhance the path planning process by taking into account various factors, such as real-time map information, traffic patterns, and user inputs, to make informed decisions. Through prediction and forecasting techniques, AVs can anticipate the actions of other road users and plan accordingly. AI/ML algorithms can also aid in obstacle avoidance by continuously monitoring the environment, detecting potential hazards, and making real-time adjustments to the vehicle's trajectory to avoid collisions.

The decision and planning stage serves as the intelligence behind the AV's actions, allowing it to make informed choices based on real-time and historical data. With advancements in AI and ML, decision-making algorithms have become increasingly sophisticated, enabling AVs to navigate complex scenarios and respond effectively to dynamic traffic conditions.

## Control

The control module in autonomous driving plays a vital role in translating the decisions made in the decision and planning stage into physical actions that control the **autonomous vehicle (AV)**. AI and ML techniques are applied in this module to enhance the control mechanisms and optimize the AV's performance.

One area where AI/ML can be applied in the control module is in adaptive control systems. By utilizing sensor data and real-time feedback, AI algorithms can dynamically adjust the control inputs to ensure the AV operates smoothly and safely. ML models can learn from past driving experiences and optimize control actions based on various driving conditions, such as different road surfaces, weather conditions, and traffic patterns.

Moreover, reinforcement learning techniques can be employed in the control module to enable the AV to learn optimal control policies through trial and error. By interacting with the environment and receiving feedback on the outcomes of its actions, the AV can iteratively improve its control strategies, leading to more efficient and effective driving behavior.

## Advanced driver assistance systems (ADAS)

In addition to autonomous driving, AI and ML are also being used to enhance the driving experience, with features such as **advanced driver assistance systems (ADAS)**.

ADAS leverages computer vision, sensor fusion, and AI techniques to detect and interpret the surrounding environment in real time. By analyzing data from cameras, radars, and other sensors, ADAS can identify potential hazards on the road, including pedestrians, cyclists, and other vehicles. This allows the system to issue warnings to the driver or even take autonomous corrective actions to mitigate risks. For example, lane departure warning systems alert drivers when they unintentionally deviate from their lane, while automatic emergency braking systems can autonomously apply the brakes to prevent or reduce the severity of a collision. ADAS technologies not only enhance safety but also contribute to reducing accidents and saving lives.

Chapter

# 44

# Breaking Down ML Algorithms

While ML algorithm design may not be the primary role of ML solutions architects, it is still essential for them to possess a comprehensive understanding of common real-world ML algorithms and their applications in solving business problems. This knowledge empowers ML solutions architects to identify suitable data science solutions and design the necessary technology infrastructure for deploying these algorithms effectively.

By familiarizing themselves with a range of ML algorithms, ML solutions architects can grasp the strengths, limitations, and specific use cases of each algorithm. This enables them to evaluate business requirements accurately and select the most appropriate algorithmic approach to address a given problem. Whether it's classification, regression, clustering, or recommendation systems, understanding the underlying algorithms equips architects with the knowledge required to make informed decisions.

## How machines learn

Earlier, we discussed the self-improvement capability of ML algorithms through data processing and parameter updates, leading to the generation of models akin to compiled binaries in computer source code. But how does an algorithm actually learn? In essence, ML algorithms learn by optimizing an objective function, also known as a loss function, which involves minimizing or maximizing it. An objective function can be seen as a business metric, such as the disparity between projected and actual product sales. The aim of optimization is to reduce this disparity. To achieve this, an ML algorithm iterates and processes extensive historical sales data (training data), adjusting its internal model parameters until the gaps between projected and actual values are minimized. This process of finding the optimal model parameters is referred to as optimization, with mathematical routines specifically designed for this purpose known as optimizers.

To illustrate the concept of optimization, let's consider a simple example of training an ML model to predict product sales based on its price. In this case, we can use a linear function as the ML algorithm, represented as follows:

$$\text{sales} = W * \text{price} + B$$

In this example, our objective is to minimize the disparity between the predicted and actual sales values. To achieve this, we employ the **mean square error (MSE)** as the loss function for optimization. The specific task is to determine the optimal values for the model parameters  $W$  and  $B$ , commonly referred to as weight and bias. The weight assigns a relative significance to each input variable, while the bias represents the average output value. Our aim is to identify the  $W$  and  $B$  values that yield the lowest MSE in order to enhance the accuracy of the sales predictions:

$$\text{Error} = \frac{1}{n} \sum_{i=1}^n (\text{predicted}_i - \text{actual}_i)^2$$

There are multiple techniques available for solving ML optimization problems. Among them, gradient descent and its variations are widely used for optimizing neural networks and various other ML algorithms. Gradient descent is an iterative approach that involves calculating the rate of error change (gradient) associated with each input variable. Based on this gradient, the model parameters ( $W$  and  $B$  in this example) are updated step by step to gradually reduce the error. The learning rate, a hyperparameter of the ML algorithm, controls the magnitude of parameter updates at each iteration. This allows for fine-tuning the optimization process. The following figure illustrates the optimization of the  $W$  value using gradient descent:

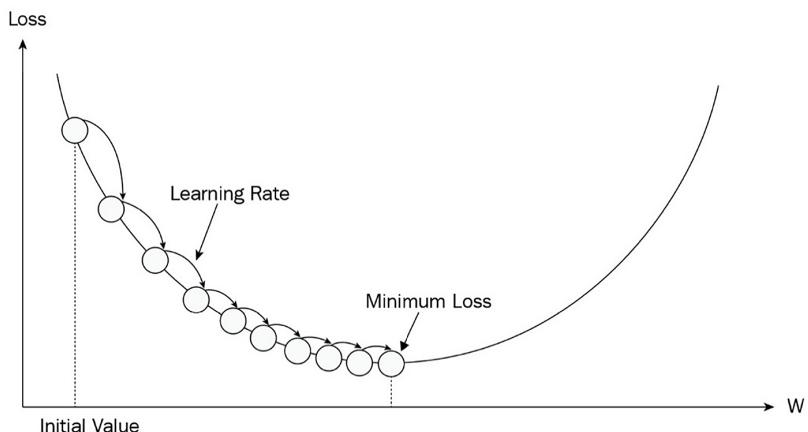


Figure 44.1: Gradient descent

The gradient descent optimization process involves several key steps:

1. Initialize the value of  $W$  randomly.
2. Calculate the error (loss) using the assigned value of  $W$ .

3. Compute the gradient (rate of change) of the error with respect to the loss function. The gradient can be positive, zero, or negative.
4. If the gradient is positive or negative, update the value of  $W$  in a direction that reduces the error in the next iteration. In this example, we move  $W$  to the right to increase its value.
5. Repeat steps 2 to 4 until the gradient becomes zero, indicating that the optimal value of  $W$  has been reached and convergence has been achieved.

In addition to gradient descent, alternative optimization techniques like the normal equation can be used to find optimal parameters for ML algorithms such as linear regression. Unlike the iterative approach of gradient descent, the normal equation offers a one-step analytical solution for calculating the coefficients of linear regression models. Other ML algorithms may also have algorithm-specific optimization methods for model training, which will be discussed in the next section.

## Overview of ML algorithms

With that brief overview of the fundamental concepts behind how machines learn, let's now explore various ML algorithms in more depth. The field of ML has seen the development of numerous algorithms, with ongoing research and innovation from both academia and industry. In this section, we will explore several well-known traditional and deep learning algorithms, examining their applications across various types of ML problems such as forecasting, recommendation, and natural language processing. Additionally, we will look at the strengths and weaknesses of different algorithms and discuss which situations each one is best suited for. This will help you build an understanding of the different capabilities of each algorithm and the types of problems they can be used to solve.

Before we delve into these algorithms, it's important to discuss the factors to consider when selecting an appropriate algorithm for a given task.

## Consideration for choosing ML algorithms

When choosing a ML algorithm, there are several key considerations to keep in mind:

- **Problem type:** Different algorithms are better suited for different types of problems. For example, classification algorithms are suitable for tasks where the goal is to categorize data into distinct classes, while regression algorithms are used for predicting continuous numerical values. Understanding the problem type is crucial in selecting the most appropriate algorithm.

- **Dataset size:** The size of your dataset can impact the choice of algorithm. Some algorithms perform well with small datasets, while others require large amounts of data to generalize effectively. If you have limited data, simpler algorithms with fewer parameters may be preferable to prevent overfitting. Overfitting is when a trained model that learns the training data too well but fails to generalize to new, unseen data.
- **Feature space:** Consider the number and nature of features in your dataset. Some algorithms can handle high-dimensional feature spaces, while others are more suitable for datasets with fewer features. Feature engineering and dimensionality reduction techniques can also be applied to enhance algorithm performance.
- **Computational efficiency:** The computational requirements of an algorithm should be taken into account, especially if you have large datasets or limited computational resources. Some algorithms are computationally expensive and may not be feasible for certain environments. Time complexity and space complexity are quantitative measures used to assess the efficiency of ML algorithms. Big  $O$  notation represents the upper bound estimation for time and space requirements. For example, linear search has a time complexity of  $O(N)$ , while binary search has  $O(\log N)$ . Understanding these complexities helps evaluate algorithm efficiency and scalability, aiding in algorithm selection for specific tasks.
- **Interpretability:** Depending on your application, the interpretability of the algorithm's results may be important. Some algorithms, such as decision trees or linear models, offer easily interpretable outcomes, while others, like deep neural networks, provide more complex and abstract representations.
- **Algorithm complexity and assumptions:** Different algorithms make different assumptions about the underlying data distribution. Consider whether these assumptions are valid for your dataset. Additionally, the complexity of the algorithm can impact its ease of implementation, training time, and ability to handle noisy or incomplete data.

By considering these factors, you can make an informed decision when selecting a ML algorithm that best suits your specific problem and available resources.

## Algorithms for classification and regression problems

The vast majority of ML problems today primarily involve classification and regression. Classification is a ML task that assigns categories or classes to data points, such as labeling a credit card transaction as fraudulent or not fraudulent. Regression, on the

---

other hand, is a ML technique used to predict continuous numeric values, such as predicting the price of a house.

In the upcoming section, we'll explore common algorithms used for classification and regression tasks. We will explain how each algorithm works, the types of problems each algorithm is suited for, and their limitations. This will help build intuition on when to select different algorithms for the different tasks.

## Linear regression algorithms

**Linear regression** algorithms are developed to solve regression problems by predicting continuous values based on independent inputs. They find wide applications in various practical scenarios, such as estimating product sales based on price or determining crop yield based on rainfall and fertilizer.

Linear regression utilizes a linear function of a set of coefficients and input variables to predict a scalar output. The formula for the linear regression is expressed as follows:

$$f(x) = W_1 * X_1 + W_2 * X_2 + \dots + W_n * X_n + \varepsilon$$

In the linear regression equation, the  $X_s$  represent the input variables,  $W_s$  denote the coefficients, and  $\varepsilon$  represents the error term. Linear regression aims to estimate the output value by calculating the weighted sum of the inputs, assuming a linear relationship between the output and inputs. The intuition behind linear regression is to find a line or hyperplane that can estimate the value for a set of input values. Linear regression can work efficiently with small datasets, offering interpretability through the coefficients' assessment of input and output variables. However, it may not perform well with complex, nonlinear datasets. Additionally, linear regression assumes independence among input features and struggles when there is co-linearity (the value of one feature influences the value of another feature), as it becomes challenging to assess the significance of correlated features.

## Logistic regression algorithms

**Logistic regression** is commonly employed for binary and multi-class classification tasks. It can predict the probability of an event occurring, such as whether a person will click on an advertisement or qualify for a loan. Logistic regression is a valuable tool in real-world scenarios where the outcome is binary and requires estimating the likelihood of a particular class. By utilizing a logistic function, this algorithm maps the input variables to a probability score, enabling effective classification decision-making.

Logistic regression is a statistical model used to estimate the probability of an event or outcome, such as transaction fraud or passing an exam. It is a linear model similar to linear regression, but with a different output transformation. The goal of logistic regression is to find a decision boundary, represented by a line or hyperplane, that effectively separates the two classes of data points. By applying a logistic function to the linear combination of input variables, logistic regression ensures that the predicted output falls within the range of 0 and 1, representing the probability of belonging to a particular class. The following formula is the function for the logistic regression, where X is a linear combination of input variables ( $b+wx$ ). Here, the w is the regression coefficient:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Like linear regression, logistic regression offers fast training speed and interpretability as its advantages. However, due to its linear nature, logistic regression is not suitable for solving problems with complex non-linear relationships.

## Decision tree algorithms

**Decision trees** find extensive application in various real-world ML scenarios, including heart disease prediction, target marketing, and loan default prediction. They are versatile and can be used for both classification and regression problems.

A decision tree is motivated by the idea that data can be divided hierarchically based on rules, leading to similar data points following the same decision path. It achieves this by splitting the input data using different features at different branches of the tree. For example, if age is a feature used for splitting at a branch, a conditional check like  $age > 50$  would be used to divide the data. The decision of which feature to use for splitting and where to split is made using algorithms such as the Gini purity index and information gain. The Gini index measures the probability of misclassification, while information gain quantifies the reduction in entropy resulting from the split.

Here, the general concept of decision tree involves experimenting with various split options and conditions, calculating metric values (e.g., information gain) for each split option, and selecting the option that yields the highest value. During prediction, input data traverses the tree based on the learned branching logic, and the final prediction is determined by the terminal node (leaf node). Refer to *Figure 44.2* for an example structure of a decision tree.

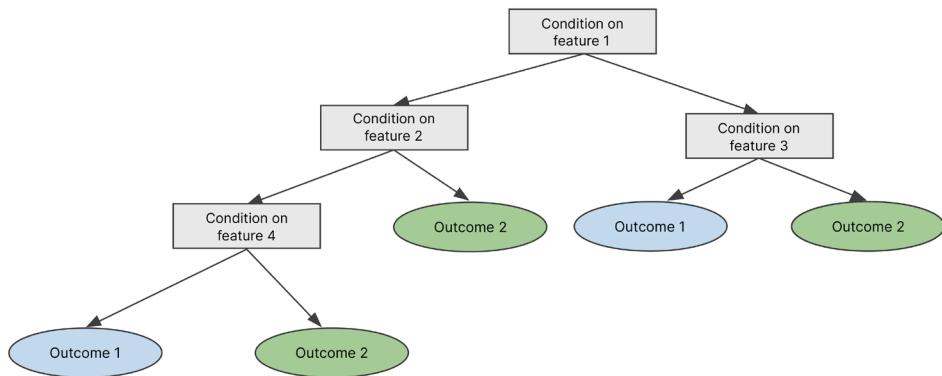


Figure 44.2: Decision tree

The main advantage of decision trees over linear regression and logistic regression is their ability to capture non-linear relationships and interactions between features. Decision trees can handle complex data patterns and are not limited to linear relationships between input variables and the output. They can represent decision boundaries that are more flexible and can handle both numerical and categorical features.

A decision tree is advantageous in that it can handle data with minimal preprocessing, accommodate both categorical and numerical features, and handle missing values and varying feature scales. It is also highly interpretable, allowing for easy visualization and analysis of decision paths. Furthermore, decision trees are computationally efficient. However, they can be sensitive to outliers and prone to **overfitting**, particularly when dealing with a large number of features and noisy data. Overfitting occurs when the model memorizes the training data but performs poorly on unseen data.

A notable limitation of decision trees and tree-based algorithms is their inability to extrapolate beyond the range of training inputs. For instance, if a housing price model is trained on square footage data ranging from 500 to 3,000 sq ft, a decision tree would be unable to make predictions beyond 3,000 sq ft. In contrast, a linear model would be capable of capturing the trend and making predictions beyond the observed range.

## Random forest algorithm

**Random forest** algorithm is widely employed in various real-world applications across e-commerce, healthcare, and finance sectors. They are particularly valuable for classification and regression tasks. Real-world examples of these tasks include insurance underwriting decisions, disease prediction, loan payment default prediction, and targeted marketing efforts. The versatility of random forest algorithms allows them to be applied in a wide range of industries to address diverse business challenges.

As discussed in the preceding decision tree section, a decision tree uses a single tree to make its decisions, and the root node of the tree (the first feature to split the tree) has the most influence on the final decision. The motivation behind this random forest is that combining the decisions of multiple trees can lead to improved overall performance. The way that a random forest works is to create multiple smaller **subtrees**, also called **weak learner trees**, where each subtree uses a random subset of all the features to come to a decision, and the final decision is made by either majority voting (for classification) or averaging (for regression). This process of combining the decision from multiple models is also referred to as **ensemble learning**. Random forest algorithms also allow you to introduce different degrees of randomness, such as **bootstrap sampling**, which involves using the same sample multiple times in a single tree. This helps make the model more generalized and less prone to overfitting. The following figure illustrates how the random forest algorithm processes input data instances using multiple subtrees and combines their outputs.

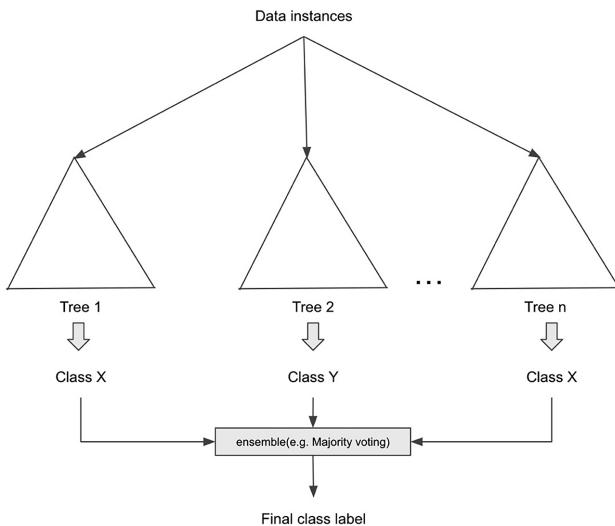


Figure 44.3: Random forest

Random forests have several advantages over decision trees. They provide improved accuracy by combining the predictions of multiple trees through majority voting or averaging. They also reduce overfitting by introducing randomness in the model and using diverse subsets of features. Random forests handle large feature sets better by focusing on different aspects of the data. They are robust to outliers and provide feature importance estimation. Additionally, random forests support parallel processing for training large datasets across multiple machines. The limitations of random forests include reduced interpretability compared to decision trees, longer training and prediction times, increased memory usage, and the need for hyperparameter tuning.

## Gradient boosting machine and XGBoost algorithms

Gradient boosting and XGBoost are also popular multi-tree-based ML algorithms used in various domains like credit scoring, fraud detection, and insurance claim prediction. Unlike random forests that combine results from weak learner trees at the end, gradient boosting sequentially aggregates results from different trees.

Random forests utilize parallel independent weak learners, while gradient boosting employs a sequential approach where each weak learner tree corrects the errors of the previous tree. Gradient boosting offers more hyperparameters to fine-tune and can achieve superior performance with proper tuning. It also allows for custom loss functions, providing flexibility in modeling real-world scenarios. Refer to the following figure for an illustration of how gradient boosting trees operate:

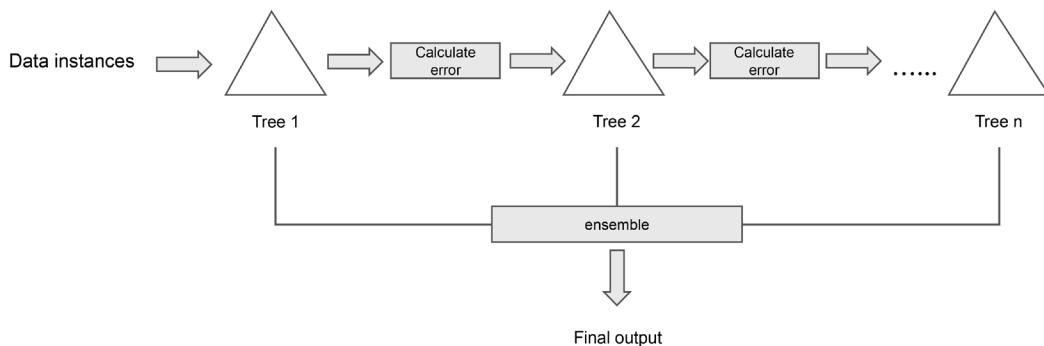


Figure 44.4: Gradient boosting

Gradient boosting offers several key advantages. Firstly, it excels in handling imbalanced datasets, making it highly suitable for tasks such as fraud detection and risk management. Secondly, it has the potential to achieve higher performance than other algorithms when properly tuned. Additionally, gradient boosting supports custom loss functions, providing flexibility in modeling real-world applications. Lastly, it can effectively capture complex relationships in the data and produce accurate predictions. Gradient boosting, despite its advantages, also has some limitations to consider. Firstly, due to its sequential nature, it lacks parallelization capabilities, making it slower in training compared to algorithms that can be parallelized. Secondly, gradient boosting is sensitive to noisy data, including outliers, which can lead to overfitting and reduced generalization performance. Lastly, the complexity of gradient boosting models can make them less interpretable compared to simpler algorithms like decision trees, making it challenging to understand the underlying relationships in the data.

XGBoost, a widely-used implementation of gradient boosting, has gained popularity for its success in Kaggle competitions. While it shares the same underlying concept as gradient boosting, XGBoost offers several improvements. It enables training a single tree across multiple cores and CPUs, leading to faster training times. XGBoost incorporates powerful regularization techniques to mitigate overfitting and reduce model complexity. It also excels in handling sparse datasets. In addition to XGBoost, other popular variations of gradient boosting trees include LightGBM and CatBoost.

## K-nearest neighbor algorithm

**K-nearest neighbor (K-NN)** is a versatile algorithm used for both classification and regression tasks. It is also employed in search systems and recommendation systems. The underlying assumption of K-NN is that similar items tend to have close proximity to each other in the feature space. To determine this proximity, distances between different data points are measured, often using metrics like Euclidean distance.

In the case of classification, K-NN starts by loading the training data along with their respective class labels. When a new data point needs to be classified, its distances to the existing data points are calculated, typically using Euclidean distance. The K nearest neighbors to the new data point are identified, and their class labels are retrieved. The class label for the new data point is then determined through majority voting, where the most frequent class among the K nearest neighbors is assigned to the new data point.

The following diagram is an illustration of using K-NN for classification:

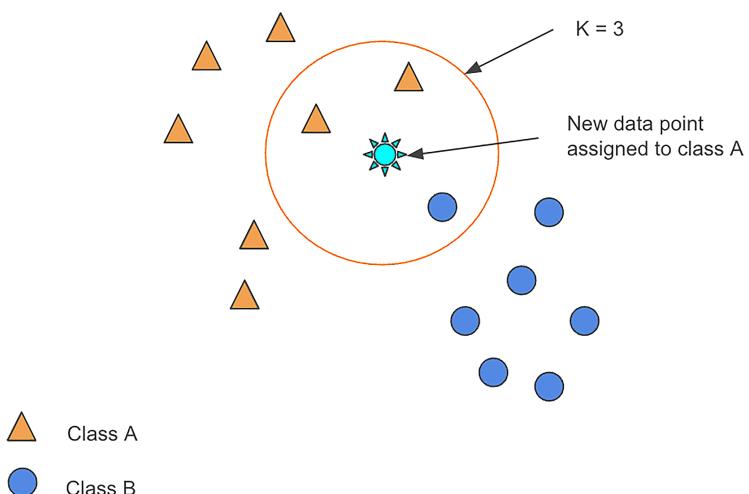


Figure 44.5: K-NN for classification

For regression tasks, K-NN follows a similar approach. The distances between the new data point and the existing data points are computed, and the K nearest neighbors are selected. The predicted scalar value for the new data point is obtained by averaging the values of the K closest data points.

One advantage of K-NN is its simplicity and lack of the need for training or tuning with hyperparameters, apart from selecting the value of K. The dataset is loaded directly into the model without the need to train a model. It is worth noting that the choice of K significantly impacts the performance of the K-NN mode. The optimal K is often found through an iterative trial-and-error process by evaluation of hold-out dataset. The results of K-NN are also easily explainable, as each prediction can be understood by examining the properties of the nearest neighbors. However, K-NN has some limitations.

As the number of data points increases, the complexity of the model grows, and predictions can become slower, especially with large datasets. K-NN is not suitable for high-dimensional datasets, as the concept of proximity becomes less meaningful in higher-dimensional spaces. The algorithm is also sensitive to noisy data and missing data, requiring outlier removal and data imputation techniques to handle such cases effectively.

## Multi-layer perceptron (MLP) networks

As mentioned earlier, an **artificial neural network** (ANN) emulates the learning process of the human brain. The brain comprises numerous interconnected neurons that process information. Each neuron in a network processes inputs (electrical impulses) from another neuron, processes and transforms the inputs, and sends the output to neurons in the network. Here is an illustration depicting a human neuron:

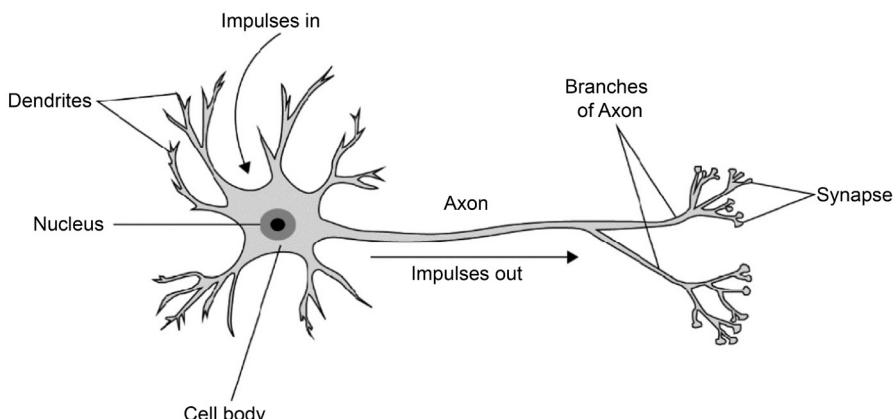


Figure 44.6: Human brain neuron

An artificial neuron operates in a similar manner. The following diagram illustrates an artificial neuron, which consists of a linear function combined with an activation function. The activation function modifies the output of the linear function, such as compressing it within a specific range, such as 0 to 1 (sigmoid activation), -1 to 1 (tanh activation), or maintaining values above 0 (ReLU). The activation function is employed to capture non-linear relationships between inputs and outputs. Alternatively, each neuron can be viewed as a linear classifier, akin to logistic regression.

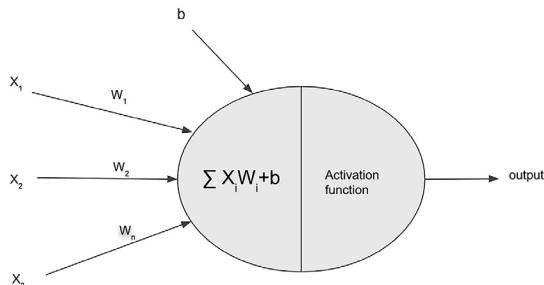


Figure 44.7: Artificial neuron

When you stack a large number of neurons into different layers (*input layer*, *hidden layers*, and *output layer*) and connect all of the neurons together between two adjacent layers, we have an ANN called **multi-layer perceptron** (MLP). Here, the term *perceptron* means *artificial neuron*, and it was originally invented by Frank Rosenblatt in 1957. The idea behind MLP is that each hidden layer will learn some higher-level representation (features) of the previous layer, and those higher-level features capture the more important information in the previous layer. When the output from the final hidden layer is used for prediction, the network has extracted the most important information from the raw inputs for training a classifier or regressor. The following figure shows the architecture of an MLP network:

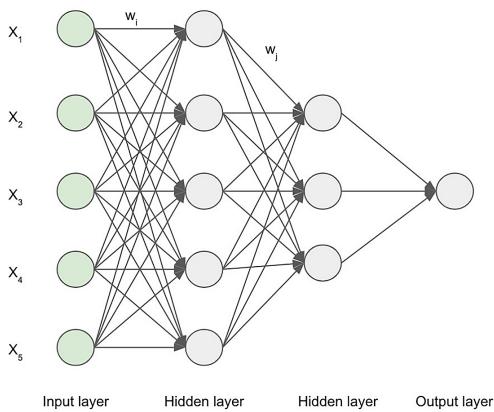


Figure 44.8: Multi-layer perceptron

During model training, the weights ( $W$ ) of each neuron in every layer are adjusted using gradient descent to optimize the training objective. This adjustment process is known as backpropagation. It involves propagating the total error back through the network, attributing a portion of the error to each neuron based on its contribution. This allows the fine-tuning of the weights in each neuron, ensuring that every neuron in every layer influences the final output to improve overall performance.

MLP is a versatile neural network suitable for both classification and regression tasks, similar to random forest and XGBoost. While commonly applied to tabular data, it can also handle diverse data formats like images and text. MLP excels in capturing intricate nonlinear patterns within the dataset and exhibits efficient computational processing, thanks to its parallelization capabilities. However, MLP typically demands a larger training dataset to achieve optimal performance compared to traditional ML algorithms.

## Algorithms for clustering

Clustering is a data mining method that involves grouping items together based on their shared attributes. One practical application of clustering is to create customer segments by analyzing demographics, transaction history, or behavior data. Other examples include social network analysis, document grouping, and anomaly detection. Various clustering algorithms exist, and we will focus on the K-means clustering algorithm in this section, which is one of the most widely used clustering algorithms due to its simplicity. Some other popular clustering algorithms are hierarchical clustering and DBSCAN.

### K-means algorithm

The K-means algorithm is widely employed in real-world applications, including customer segmentation analysis, document classification based on document attributes, and insurance fraud detection. It is a versatile algorithm that can effectively group data points in various domains for different purposes.

K-means aims to group similar data points together in clusters, and it is an unsupervised algorithm, meaning it doesn't rely on labeled data. The algorithm begins by randomly assigning K centroids, which represent the centers of the clusters. It then iteratively adjusts the assignment of data points to the nearest centroid and updates the centroids to the mean of the data points in each cluster. This process continues until convergence, resulting in well-defined clusters based on similarity.

K-means clustering offers several advantages, including its simplicity and ease of understanding, making it accessible to beginners. It is computationally efficient and can handle large datasets effectively. The resulting clusters are interpretable,

providing valuable insights into the underlying patterns in the data. K-means is versatile and applicable to various types of data, including numerical, categorical, and mixed attribute datasets. However, there are some drawbacks to consider. Selecting the optimal number of clusters ( $K$ ) can be subjective and challenging. The algorithm is sensitive to the initial placement of centroids, which can lead to different cluster formations. K-means assumes spherical clusters with equal variance, which may not hold true in all cases. It is also sensitive to outliers and struggles with non-linear data relationships.

## Algorithms for time series analysis

A time series consists of a sequence of data points recorded at successive time intervals. It is commonly used to analyze and predict trends in various domains, such as finance, retail, and sales. Time series analysis allows us to understand past patterns and make future predictions based on the relationship between current and past values. Forecasting in time series relies on the assumption that future values are influenced by previous observations at different time points.

Time series data exhibits several important characteristics, including trend, seasonality, and stationarity. **Trend** refers to the long-term direction of the data, whether it shows an overall increase or decrease over time. It helps to identify the underlying pattern and understand the general behavior of the time series. **Seasonality**, on the other hand, captures repeating patterns within a fixed interval, often occurring in cycles or seasons. It helps to identify regular fluctuations that repeat over specific time periods, such as daily, weekly, or yearly patterns. **Stationarity** refers to the property of a time series where statistical properties, such as mean and variance, remain constant over time. Stationarity is crucial because many forecasting techniques assume that the underlying data is stationary. Non-stationary time series can lead to inaccurate or unreliable forecasts. Therefore, it is important to assess and address the stationarity of a time series before applying forecasting techniques.

## ARIMA algorithm

The **autoregressive integrated moving average (ARIMA)** algorithm finds practical applications in various real-world scenarios, including budget forecasting, sales forecasting, patient visit forecasting, and customer support call volume forecasting. ARIMA is a powerful tool for analyzing and predicting time series data, allowing organizations to make informed decisions and optimize their operations in these areas. By leveraging historical patterns and trends in the data, ARIMA enables accurate forecasts and assists businesses in effectively managing their resources and planning for the future.

ARIMA operates on the premise that the value of a variable in a given period is influenced by its own previous values (autoregressive), the deviations from the mean follow a pattern based on previous deviations (moving average), and trend and seasonality can be eliminated by differencing (calculating the differences between consecutive data points). This differencing process aims to transform the time series into a stationary state, where statistical properties like mean and variance remain constant over time. These three components of ARIMA can be mathematically represented using the following formulas:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \dots + \phi_p y_{t-p} + \epsilon_t$$

Where the **autoregressive (AR)** component is expressed as a regression of previous values (also known as lags):

$$y_{t-1} \dots y_{t-p}$$

The constant  $C$  represents a drift:

$$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \dots + \theta_q \epsilon_{t-q}$$

The **moving average (MA)** component is expressed as a weighted average of forecasting errors for the previous time periods, where it represents a constant:

$$y'_t = y_t - y_{t-1}$$

The **integrated component** (time series differencing) of a time series can be expressed as the difference between the values in one period from the previous period.

ARIMA is a suitable choice for forecasting single time series (univariate) data as it doesn't rely on additional variables. It outperforms simpler forecasting techniques like simple moving average, exponential smoothing, or linear regression. Additionally, ARIMA provides interpretability, allowing for a clear understanding of the underlying patterns. However, due to its backward-looking nature, ARIMA may struggle to accurately forecast unexpected events. Furthermore, being a linear-based model, ARIMA may not effectively capture complex non-linear relationships in time series data.

## DeepAR algorithm

Deep learning-based forecasting algorithms offer solutions to the limitations of traditional models like ARIMA. They excel at capturing complex non-linear

---

relationships and can effectively utilize multivariate datasets. These models enable the training of a global model, allowing for a single model to handle multiple similar target time series. This eliminates the need for creating separate models for each individual time series, providing a more efficient and scalable approach.

**Deep Autoregressive (DeepAR)** is a state-of-the-art forecasting algorithm based on neural networks, designed to handle large datasets with multiple similar target time series. It has the capability to incorporate related time series, such as product prices or holiday schedules, to enhance the accuracy of its forecasting models. This feature proves particularly valuable when dealing with spiky events triggered by external variables, allowing for more precise and reliable predictions.

DeepAR utilizes a **recurrent neural network (RNN)** as its underlying model to capture patterns in the target time series. It goes beyond single-variable forecasting by incorporating multiple target time series and additional external supporting time series. Instead of considering individual values, the RNN takes input vectors representing the values of various variables at each time period. By jointly learning the patterns of these combined vectors over time, DeepAR can effectively capture the intrinsic non-linear relationships and shared patterns among the different time series. This approach enables DeepAR to train a single global model that can be used for forecasting across multiple similar target time series.

DeepAR excels in handling complex multivariate datasets; however, it performs best when trained with large amounts of data. It is particularly useful in real-world scenarios involving large-scale retail forecasting for numerous items, where external factors like marketing campaigns and holiday schedules need to be taken into account. By leveraging its capability to model multiple variables simultaneously, DeepAR can provide accurate predictions and insights in such practical use cases.

A significant drawback of DeepAR is the black-box nature of the deep learning model, which lacks interpretability and transparency. This makes the forecasts more difficult to explain and justify than simpler statistical methods. Another major disadvantage is the data-hungry nature of DeepAR, whereby it performs poorly when the dataset is small.

## Algorithms for recommendation

The recommender system is an essential ML technology that predicts a user's preference for items, primarily relying on user or item attribute similarities or user-item interactions. It has gained widespread adoption in various industries, including retail, media and entertainment, finance, and healthcare. Over the years, the field of recommendation algorithms has evolved significantly, from making recommendations based on preferences and behaviors of similar users, to a reinforcement-learning-based

---

approach where algorithms learn to make sequential decisions over time, taking into account user feedback and interaction. In the following section, we will explore some commonly used algorithms in the realm of recommender systems.

## Collaborative filtering algorithm

**Collaborative filtering** is a popular recommendation algorithm that leverages the notion that individuals with similar interests or preferences in one set of items are likely to have similar interests in other items as well. By analyzing the collective experiences and behaviors of different users, collaborative filtering can effectively recommend items to individual users based on the preferences of similar users. This approach taps into the experiences of the crowd to provide personalized and relevant recommendations.

The following figure illustrates an item-user interaction matrix in the context of movie ratings. As you can see, it is a **sparse matrix**. This means that there are many empty entries in the matrix, which is expected as it is unlikely for any individual to have watched every movie:

	User 1	User 2	User 3	User 4	....	User n
Movie 1	5			4		
Movie 2	4		2			4
...						4
Movie n			3			

Figure 44.9: User-item interaction matrix for collaborative filtering

One of the major benefits of collaborative filtering is that it can provide highly personalized recommendations matched to each user's unique interests. Unlike content-based systems, collaborative filtering models do not need to analyze and understand item features and content. Instead, they rely solely on behavioral patterns like ratings, purchases, clicks, and preferences across users to uncover correlations. This allows collaborative systems to get a nuanced profile of a user's likes and dislikes based on crowd wisdom. The algorithms can then generate recommendations tailored to that specific user, going beyond obvious suggestions. This level of personalization and ability to capture user preferences makes collaborative filtering a powerful approach, especially for large catalogs where analyzing content is infeasible.

Collaborative filtering also comes with some notable downsides. A major issue is the cold-start problem: collaborative models struggle when new users or items with no ratings are introduced. The algorithms rely heavily on crowd ratings, so they cannot effectively recommend to users new items that lack this historical data. Collaborative systems can also lead to limited diversity, creating filter bubbles and obvious recommendations rather than novel ones. They commonly face sparsity issues as the user-item matrix is often sparse, especially for large catalogs.

**Matrix factorization** is a technique commonly used in collaborative filtering for recommendation systems. It involves learning vector representations, or embeddings, for both users and items in the user-item interaction matrix. The goal is to approximate the original matrix by taking the product of the learned user and item embedding matrices.

This allows us to predict the missing entries in the matrix, which represent the likely ratings that a user would give to unseen items. To make predictions, we simply compute the dot product between the user embedding and the item embedding.



Embedding is a fundamental concept in ML that plays a crucial role in various domains. It involves creating numerical representations for entities, such as words or objects, in a manner that captures their semantic similarity. These representations are organized in a multi-dimensional space, where similar entities are positioned closer to each other. By using embeddings, we can uncover the underlying latent semantics of the objects, enabling more effective analysis and modeling. In the upcoming sections, we will delve deeper into embedding techniques and their applications in NLP algorithms.

Matrix factorization provides major scalability benefits so collaborative filtering can be applied to extremely large catalogs. However, the algorithm loses some transparency due to the latent factor modeling. Overall, matrix factorization extends collaborative filtering to much bigger datasets but sacrifices some interpretability.

## Multi-armed bandit/contextual bandit algorithm

Collaborative filtering-based recommender systems heavily rely on prior interaction data between identified users and items to make accurate recommendations. However, these systems face challenges when there is a lack of prior interactions or when the user is anonymous, resulting in a cold-start problem. To address this issue, one approach is to utilize a **multi-armed bandit (MAB)** based recommendation system. This approach draws inspiration from the concept of trial and error, similar to a gambler simultaneously playing multiple slot machines and observing which machine yields the best overall return. By employing reinforcement learning techniques, MAB-based

recommendation systems dynamically explore and exploit different recommendations to optimize the user experience, even in the absence of substantial prior interaction data.

MAB algorithms operate under the paradigm of online learning, where there is no pre-existing training data to train a model prior to deployment. Instead, the model incrementally learns and adapts as data becomes available. In the initial stages of MAB learning, the model recommends all available options (such as products on an e-commerce site) with equal probabilities to users. As users begin to interact with a subset of the items and provide feedback (rewards), the MAB model adjusts its strategy. It starts to offer items that have yielded higher rewards (e.g., more user interactions) more frequently, exploiting the knowledge of their positive performance.

However, the model also continues to allocate a smaller percentage of recommendations to new items, aiming to explore their potential for receiving interactions. This balance between exploration (offering new items) and exploitation (offering items with known rewards) is a fundamental tradeoff in MAB algorithms.

MAB algorithms face several limitations. Striking the right balance between exploration and exploitation can be challenging, leading to suboptimal solutions in certain environments. Handling high-dimensional contextual information poses a challenge as well, and the algorithms may be sensitive to noisy rewards. Additionally, the cold-start problem arises when there is limited historical data for new items or users.

## Algorithms for computer vision problems

Computer vision refers to the ability of computers to interpret and understand visual representations, such as images and videos, in order to perform tasks like object identification, image classification, text detection, face recognition, and activity detection. These tasks rely on pattern recognition, where images are labeled with object names and bounding boxes, and computer vision models are trained to recognize these patterns and make predictions on new images. Computer vision technology finds numerous applications in practical domains such as content management, security, augmented reality, self-driving cars, medical diagnosis, sports analytics, and quality inspection in manufacturing. In the following section, we will delve deeper into a few neural network architectures specifically designed for computer vision tasks.

## Convolutional neural networks

A **convolutional neural network (CNN)** is a deep learning architecture specifically designed for processing and analyzing image data. It takes inspiration from the functioning of the animal visual cortex. In the visual cortex, individual neurons

respond to visual stimuli within specific subregions of the visual field. These subregions, covered by different neurons, partially overlap to cover the entire visual field. Similarly, in a CNN, different filters are applied to interact with subregions of an image, capturing and responding to the information within that region. This allows the CNN to extract meaningful features and patterns from the image data.

A CNN architecture consists of multiple layers that repeat in a pattern. Each layer has different sublayers with specific functions. The convolutional layer plays a crucial role in feature extraction from input images. It utilizes convolutional filters, which are matrices defined by height and width, to extract relevant features. These convolutional layers process the input images by convolving them with the filters, producing feature maps that are passed to the next layer in the network.

The pooling layer, found after one or multiple convolutional layers, reduces the dimensionality of the extracted features. It combines multiple outputs into a single output, resulting in a more compact representation. Two commonly used pooling techniques are max pooling, which selects the maximum value from the outputs, and average pooling, which calculates the average value.

Following the convolutional and pooling layers, a fully connected layer is employed to combine and flatten the outputs from the previous layer. This layer aggregates the extracted features and feeds them into an output layer, typically used for tasks like image classification.

The architecture of a CNN is illustrated in the following figure, showcasing the flow of information through the various layers:

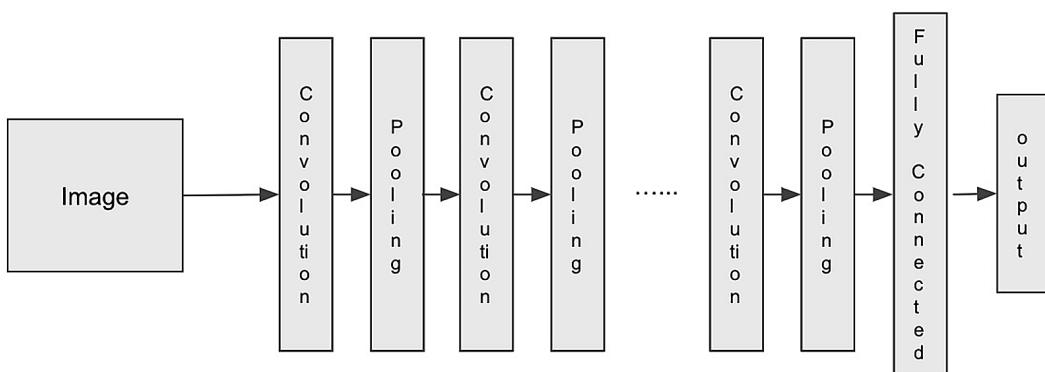


Figure 44.10: CNN architecture

CNN-based models offer efficient training due to their high degree of parallelism. This is particularly advantageous for tasks involving large-scale image data, where parallel processing can significantly accelerate training time. While CNNs are primarily

used for computer vision tasks, their success has led to their application in other domains as well, including natural language processing. By adapting the principles of convolution and hierarchical feature extraction, CNNs have shown promise in tasks such as text classification and sentiment analysis. This demonstrates the versatility and effectiveness of CNN-based models beyond their traditional application in computer vision.

CNNs have their limitations. CNNs lack interpretability due to their complex architecture, behaving like black boxes. This makes them unsuitable when model explainability is critical. In addition, CNNs require large training datasets to properly learn features and avoid overfitting. Their performance suffers significantly on smaller datasets.

## ResNet

As computer vision tasks grow in complexity, the addition of more layers in CNNs enhances their capability for image classification by enabling the learning of increasingly intricate features. However, as the number of layers increases in a CNN architecture, performance may deteriorate. This is commonly referred to as the **vanishing gradient** problem, where signals originating from the initial inputs, including crucial information, gradually diminish as they traverse through multiple layers of the CNN.

**Residual networks (ResNet)** address the vanishing gradient problem by implementing a layer-skipping technique. Rather than processing signals sequentially through each layer, ResNet introduces skip connections that allow signals to bypass certain layers. This can be visualized as a highway with fewer exits, enabling the signals from earlier layers to be preserved and carried forward without loss. The ResNet architecture is depicted in the following figure.

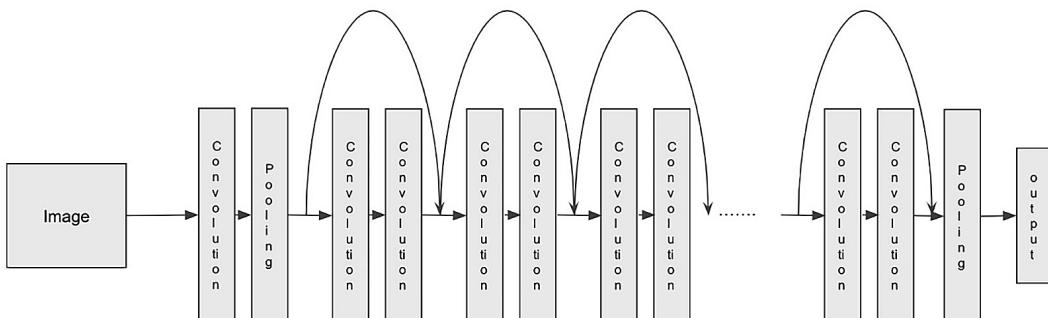


Figure 44.11: ResNet architecture

ResNet can be used for different computer vision tasks such as *image classification*, *object detection* (detecting all objects in a picture), and producing models with much higher accuracy than a vanilla CNN network. However, a potential disadvantage of ResNet is increased computational complexity due to the introduction of skip connections. The additional connections require more memory and computational resources, making training and inference more computationally expensive compared to shallower architectures.

## Algorithms for natural language processing (NLP) problems

NLP focuses on the relationship between computers and human language. It involves the processing and analysis of extensive amounts of natural language data with the objective of enabling computers to comprehend the meaning behind human language and extract valuable information from it. NLP encompasses a wide range of tasks within the field of data science. Some of these tasks include document classification, topic modeling, converting speech to text, generating speech from text, extracting entities from text, language translation, understanding and answering questions, reading comprehension, and language generation.

ML algorithms cannot process raw text data directly. To train NLP models effectively, it is necessary to convert the words within an input text into numerical representations within the context of other words, sentences, or documents. Before the advancement of embedding, there were two widely used methods for representing the relevance of words in a text: **bag-of-words (BOW)** and term **frequency-inverse document frequency (TF-IDF)**.

BOW is simply the count of a word appearing in a text (document). For example, if the input documents are I need to go to the bank to make a deposit and I am taking a walk along the river bank, and you count the number of appearances for each unique word in each input document, you will get 1 for the word *I*, and 3 for the word *to* in the first document, as an example. If we have a vocabulary for all the unique words in the two documents, the vector representation for the first document can be [1 1 3 1 1 1 1 1 0 0 0 0], where each position represents a unique word in the vocabulary (for example, the first position represents the word *I*, and the third position represents the word *to*). Now, this vector can be fed into an ML algorithm to train a model such as text classification. The main idea behind BOW is that a word that appears more frequently has stronger weights in a text.

TF-IDF has two components. The first component, *TF*, is the ratio of the number of times a vocabulary word appears in a document over the total number of words in the document. Using the preceding first document, the word *I* would have a TF value of

1/11 for the first sentence, and the word *walk* would have a TF value of 0/11, since *walk* does not appear in the first sentence. While TF measures the importance of a word in the context of one text, the IDF component measures the importance of a word across all the documents. Mathematically, it is the log of the ratio of the number of documents over the number of documents where a word appears. The final value of TF-IDF for a word would be the *TF* term multiplied by the *IDF* term. In general, TF-IDF works better than BOW.

Although BOW and TF-IDF are useful for NLP tasks, they lack the ability to capture the semantic meaning of words and often result in large and sparse input vectors. This is where the concept of embedding plays a crucial role.

Embedding is a technique used to generate low-dimensional representations (mathematical vectors) for words or sentences, which capture the semantic meaning of the text. The underlying idea is that words or sentences with similar semantic meanings tend to occur in similar contexts. In a multi-dimensional space, the mathematical representations of semantically similar entities are closer to each other than those with different meanings. For instance, if we consider sports-related words like soccer, tennis, and bike, their embeddings would be close to each other in the high-dimensional embedding space, measured by metrics like cosine similarity, which measures how similar two vectors are by calculating the cosine of the angle between them. The embedding vector represents the intrinsic meaning of the word, with each dimension representing a specific attribute associated with the word. Visualizing embeddings in a multidimensional space shows the proximity of related entities. The following diagram provides a visual depiction of the closeness in this multidimensional space:

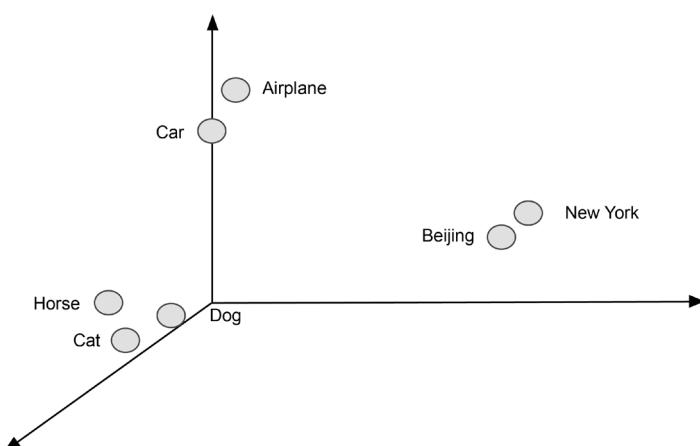


Figure 44.12: Embedding representation

Nowadays, embeddings have become a crucial component for achieving good results in most NLP tasks. Compared to other techniques like simple word counts, embeddings offer more meaningful representations of the underlying text. This has led to their widespread adoption in various ML algorithms designed for NLP. In this section, we will delve into several of these algorithms such as BERT and GPT, exploring their specific applications and benefits in the context of NLP tasks.

## Word2Vec

Thomas Mikolov created **Word2Vec** in 2013. It supports two different techniques for learning embedding: **continuous bag-of-words (CBOW)** and **continuous-skip-gram**. CBOW tries to predict a word for a given window of surrounding words, and continuous-skip-gram tries to predict surrounding words for a given word. The training dataset for Word2Vec could be any running text available, such as **Wikipedia**. The process of generating a training dataset for CBOW is to run a sliding window across running text (for example, a window of five words) and choose one of the words as the target and the rest as inputs (the order of words is not considered). In the case of continuous-skip-gram, the target and inputs are reversed. With the training dataset, the problem can be turned into a multi-class classification problem, where the model will learn to predict the classes (for example, words in the vocabulary) for the target word and assign each predicted word with a probability distribution.

Word2Vec embeddings can be trained using a straightforward one-hidden-layer MLP network. In this approach, the input to the MLP network is a matrix that represents the neighboring words, while the output is a probability distribution for the target words. During training, the weights of the hidden layer are optimized, and once the training process is complete, these weights serve as the actual embeddings for the words. The resulting embeddings capture the semantic relationships and contextual meanings of the words, enabling them to be effectively utilized in various natural language processing tasks.

As large-scale word embedding training can be expensive and time-consuming, Word2Vec embeddings are usually trained as a pre-training task so that they can be readily used for downstream tasks such as text classification or entity extraction. This approach of using embeddings as features for downstream tasks is called a **feature-based application**. There are pre-trained embeddings (for example, Tomas Mikolov' *Word2Vec* and Stanford's *GloVe*) in the public domain that can be used directly. The embeddings are a 1:1 mapping between each word and its vector representation.

## BERT

Word2Vec produces a fixed embedding representation for each word in the vocabulary, disregarding the contextual variations in meaning. However, words can have different meanings depending on the specific context in which they are used. For instance, the word “bank” can refer to a financial institution or the land alongside a body of water. To address this issue, contextualized word embeddings have been developed. These embeddings take into account the surrounding words or the overall context in which a word appears, allowing for a more nuanced and context-aware representation. By considering context, these embeddings capture the diverse meanings a word can have, enabling more accurate and context-specific analyses in downstream tasks.

**BERT**, which stands for **Bidirectional Encoder Representations from Transformers**, is a language model that takes context into consideration by the following:

- Predicting randomly masked words in sentences (the context) and taking the order of words into consideration. This is also known as **language modeling**.
- Predicting the next sentence from a given sentence.

Released in 2018, this context-aware embedding approach provides better representation for words and can significantly improve language tasks such as *reading comprehension*, *sentiment analysis*, and *named entity recognition*. Additionally, BERT generates embeddings at subword levels (a segment between a word and a character, for example, the word *embeddings* is broken up into *em*, *bed*, *ding*, and *s*). This allows it to handle the **out-of-vocabulary (OOV)** issue, another limitation of Word2Vec, which only generates embeddings on known words and will treat OOV words simply as unknown.

To obtain word embeddings with BERT, the process differs from the straightforward word-to-vector mapping used in Word2Vec. Instead, sentences are inputted into a pre-trained BERT model, and embeddings are extracted dynamically. This approach generates embeddings that are contextualized within the context of the given sentences. Besides word-level embeddings, BERT is also capable of producing embeddings for entire sentences. **Pre-training** is the term used to describe the process of learning embeddings using input tokens, and the following diagram illustrates the components involved in a BERT model for this purpose.

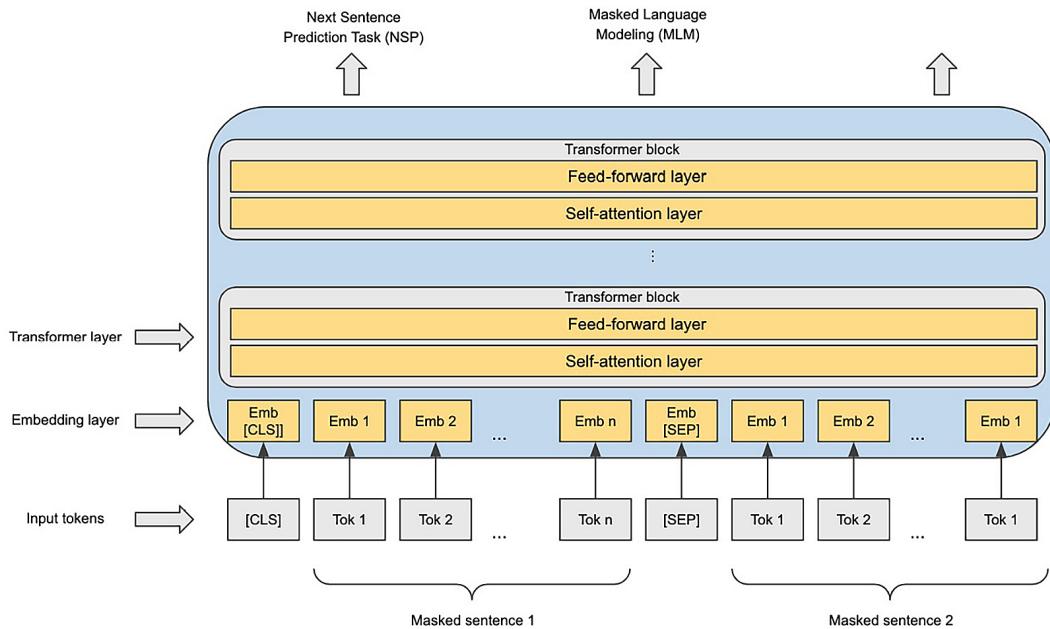


Figure 44.13: BERT model pre-training

Architecturally, BERT mainly uses a building block called a **transformer**. A transformer has a stack of encoders and a stack of decoders inside it, and it transforms one sequence of inputs into another sequence. Each encoder has two components:

- A self-attention layer mainly calculates the strength of the connection between one token (represented as a vector) and all other tokens in the input sentence, and this connection helps with the encoding of each token. One way to think about self-attention is which words in a sentence are more connected than other words in a sentence. For example, if the input sentence is *The dog crossed a busy street*, then we would say the words *dog* and *crossed* have stronger connections with the word *The* than the word *a* and *busy*, which would have strong connections with the word *street*. The output of the self-attention layer is a sequence of vectors; each vector represents the original input token as well as the importance it has with other words in the inputs.
- A feed-forward network layer (single hidden layer MLP) extracts higher-level representation from the output of the self-attention layer.

Inside the decoder, there is also a self-attention layer and feed-forward layer, plus an extra encoder-decoder layer that helps the decoder to focus on the right places in the inputs.

In the case of BERT, only the encoder part of the transformer is used. BERT can be used for a number of NLP tasks, including *question answering*, *text classification*, *named entity extraction*, and *text summarization*. It achieved state-of-the-art performance in many of the tasks when it was released. BERT pre-training has also been adopted for different domains, such as scientific text and biomedical text, to understand domain-specific languages. The following figure showcases how a pre-trained BERT model is used to train a model for a question-answering task using the fine-tuning technique:

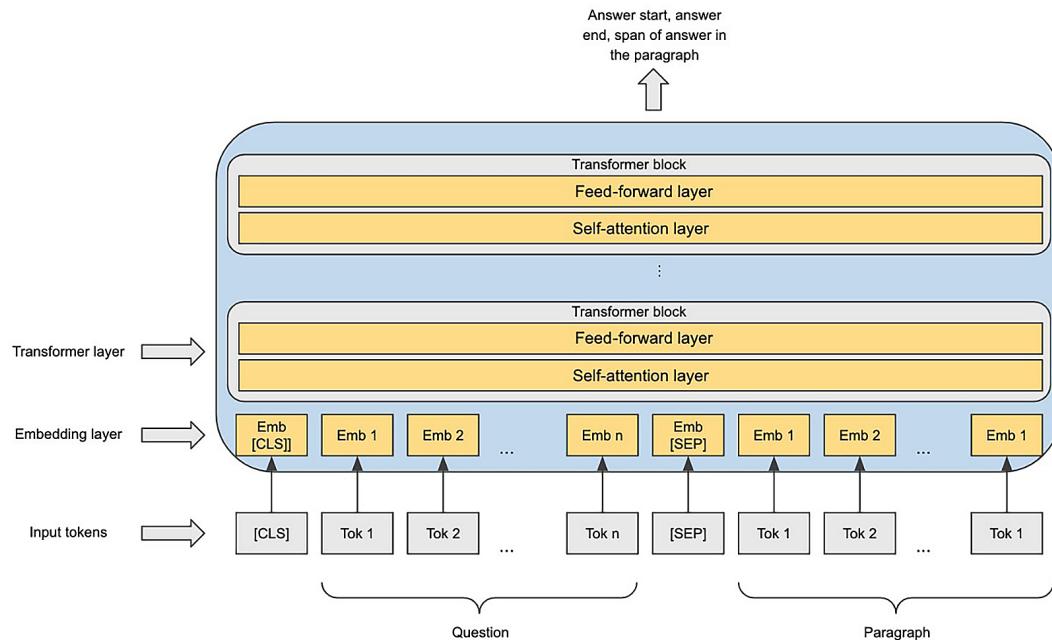


Figure 44.14: BERT fine-tuning

While BERT's pre-trained embeddings can be extracted for downstream tasks such as text classification and question answering, a more straightforward way to use its pre-trained embeddings is through a technique called **fine-tuning**. With fine-tuning, an additional output layer is added to the BERT network to perform a specific task, such as question answering or entity extraction. During fine-tuning, the pre-trained model is loaded, and you plug in the task-specific input (for example, question/passage pairs in question answering) and output (start/end and span for the answers in the passage) to fine-tune a task-specific model. With fine-tuning, the pre-trained model weights are updated.

Chapter

**45**

# Managing Data for Machine Learning Success

As an ML solutions architecture practitioner, I often receive requests for guidance on designing data management platforms for ML workloads. Although data management platform architecture is typically treated as a separate technical discipline, it plays a crucial role in ML workloads. To create a comprehensive ML platform, ML solutions architects must understand the essential data architecture considerations for ML and be familiar with the technical design of a data management platform that caters to the needs of data scientists and automated ML pipelines.

## Data management considerations for ML

Data management is a broad and complex topic. Many organizations have dedicated data management teams and organizations to manage and govern the various aspects of a data platform. Historically, data management primarily revolved around fulfilling the requirements of transactional systems and analytics systems. However, as ML solutions gain prominence, there are now additional business and technological factors to consider when it comes to data management platforms. The advent of ML introduces new requirements and challenges that necessitate an evolution in data management practices to effectively support these advanced solutions.

To understand where data management intersects with the ML workflow, let's bring back the ML lifecycle, as illustrated in the following figure:

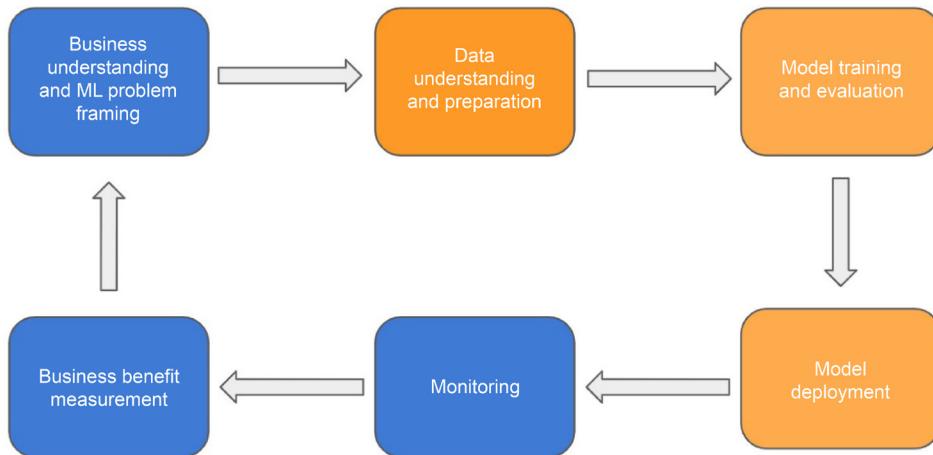


Figure 45.1: Intersection of data management and the ML lifecycle

At a high level, data management intersects with the ML lifecycle in three stages: *data understanding and preparation*, *model training and evaluation*, and *model deployment*.

During the *data understanding and preparation* stage, data scientists undertake several essential tasks. They begin by identifying relevant data sources that contain datasets suitable for their modeling tasks. Exploratory data analysis is then performed to gain insights into the dataset, including data statistics, correlations between features, and data sample distributions. Additionally, data preparation for model training and validation is crucial, involving a series of steps that typically include the following:

- **Data validation:** The data is checked for errors and anomalies to ensure its quality. This includes verifying the data range, distribution, and data types and identifying missing or null values.
- **Data cleaning:** Any identified data errors are fixed or corrected to ensure the accuracy and consistency of the dataset. This may involve removing duplicates, handling missing values, or resolving inconsistencies.
- **Data enrichment:** Additional value is derived from the data through techniques like joining different datasets or transforming the data. This helps generate new signals and insights that can enhance the modeling process.
- **Data labeling:** For supervised ML model training, training and testing datasets need to be labeled by human annotators or the ML model accurately. This critical step is necessary to guarantee the development and validation of high-quality models.

The data management capabilities needed during this stage encompass the following aspects:

- **Dataset discovery:** The capability to search and locate curated datasets using relevant metadata like dataset name, description, field name, and data owner.
- **Data access:** The ability to access both raw and processed datasets to perform exploratory data analysis. This ensures data scientists can explore and analyze the data effectively.
- **Querying and retrieval:** The capability to run queries against selected datasets to obtain details such as statistical information, data quality metrics, and data samples. Additionally, it includes the ability to retrieve data from the data management platform to a data science environment for further processing and feature engineering.
- **Scalable data processing:** The ability to execute data processing operations on large datasets efficiently. This ensures that data scientists can handle and process substantial amounts of data during model development and experimentation.

During the stage of model training and validation, data scientists are responsible for generating a training and validation dataset to conduct formal model training. To facilitate this process, the following data management capabilities are essential:

- **Data processing and automated workflows:** A data management platform should provide robust data processing capabilities along with automated workflows. This enables the conversion of raw or curated datasets into training and validation datasets in various formats suitable for model training.
- **Data repository and versioning:** An efficient data management platform should offer a dedicated data repository to store and manage the training and validation datasets. Additionally, it should support versioning, allowing data scientists to keep track of different iterations and modifications made to the datasets, along with the versions of the code and trained ML models.
- **Data labeling:** For supervised ML model training, training and testing datasets need to be labeled by human annotators or the ML model accurately. This critical step is necessary to guarantee the development and validation of high-quality models. This is a highly labor-intensive task, requiring purpose-built software tools to do it at scale.
- **ML features/embeddings generation and storage:** Some ML features/embeddings (e.g., averages, sums, and text embeddings) need to be pre-computed for one or more downstream model training tasks. These features/embeddings often need to be managed using purpose-built tools for efficient access and reuse.

- **Dataset provisioning for model training:** The platform should provide mechanisms to serve the training and validation datasets to the model training infrastructure. This ensures that the datasets are accessible by the training environment, allowing data scientists to train models effectively.

During the stage of model deployment, the focus shifts toward utilizing the trained models to serve predictions. To support this stage effectively, the following data management capabilities are crucial:

- **Serving data for feature processing:** The data management platform should be capable of serving the data required for feature processing as part of the input data when invoking the deployed models. This ensures that the models receive the relevant data inputs required for generating predictions.
- **Serving pre-computed features/embeddings:** In some cases, pre-computed features/embeddings are utilized as inputs when invoking the deployed models. The data management platform should have the capability to serve these pre-computed features seamlessly, allowing the models to incorporate them into the prediction process.

In contrast to traditional data access patterns for transactional or business intelligence solutions, where developers can utilize non-production data in lower environments for development purposes, data scientists typically require access to production data for model development.

Having explored the considerations for ML data management, we will now delve deeper into the data management architecture specifically designed for ML. It is important to understand that effective data management is crucial for success in applied ML. Organizations fail with ML not just due to poor algorithms or inaccurate models, but also due to problems with real-world data and production systems. Data management shortcomings can sink ML projects despite brilliant modeling.

## Data management architecture for ML

Depending on the scale of your ML initiatives, it is important to consider different data management architecture patterns to effectively support them. The right architecture depends on the scale and scope of the ML initiatives within an organization in order to balance the business needs with engineering efforts.

For *small-scale ML projects* characterized by limited data scope, a small team size, and minimal cross-functional dependencies, a purpose-built data pipeline tailored to meet specific project requirements can be a suitable approach. For instance, if your project involves working with structured data sourced from an existing data warehouse and a publicly available dataset, you can consider developing a straightforward data pipeline. This pipeline would extract the necessary data from the data warehouse and public domain and store it in a dedicated storage location owned by the project team. This data extraction process can be scheduled as needed to facilitate further analysis and processing. The following diagram illustrates a simplified data management flow designed to support a small-scale ML project:

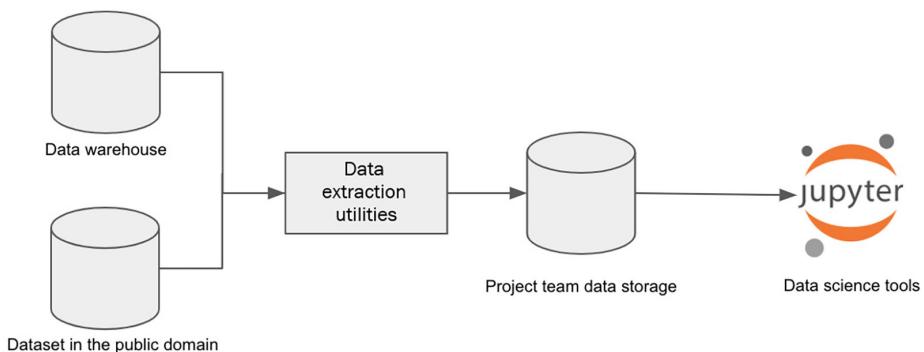


Figure 45.2: Data architecture for an ML project with limited scope

For *large-scale ML initiatives* at the enterprise level, the data management architecture closely resembles that of enterprise analytics. Both require robust support for data ingestion from diverse sources and centralized management of data for various processing and access requirements. While analytics data management primarily deals with structured data and often relies on an enterprise data warehouse as its core backend, ML data management needs to handle structured, semi-structured, and unstructured data for different ML tasks. Consequently, a data lake architecture is commonly adopted. ML data management is typically an integral part of the broader enterprise data management strategy, encompassing both analytics and ML initiatives.

The following diagram illustrates a logical enterprise data management architecture comprising key components such as data ingestion, data storage, data processing, data catalog, data security, and data access:

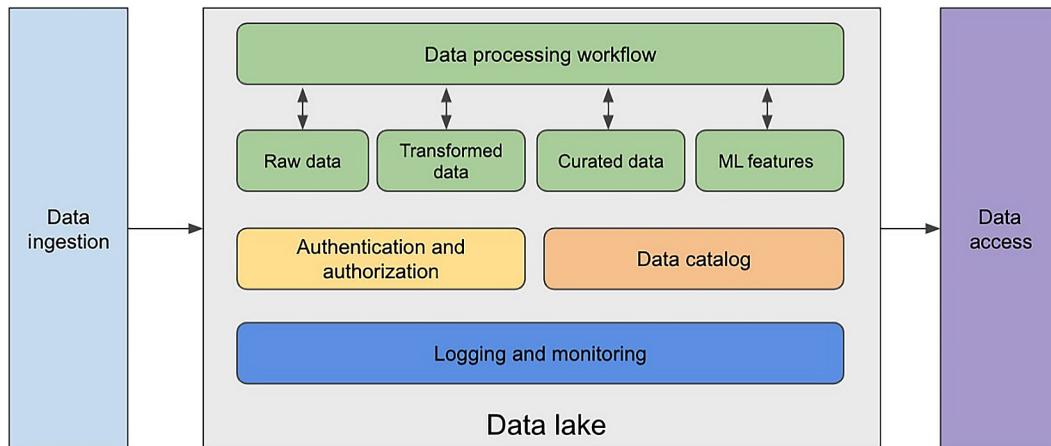


Figure 45.3: Enterprise data management

In the following sections, we will delve into a detailed analysis of each key component of enterprise data management, providing an in-depth understanding of their functionalities and implications within a data management architecture built using AWS native services in the cloud. By exploring the specific characteristics and capabilities of these components, we will gain valuable insights into the overall structure and mechanics of an AWS-based data management architecture.

## Data storage and management

Data storage and management is a fundamental component of the overall ML data management architecture. ML workloads often require data from diverse sources and in various formats, and the sheer volume of data can be substantial, particularly when dealing with unstructured data.

To address these requirements, cloud object data storage solutions like Amazon S3 are commonly employed as the underlying storage medium. Conceptually, cloud object storage can be likened to a file storage system that accommodates files of different formats. Moreover, the storage system allows for the organization of files using prefixes, which serve as virtual folders for enhanced object management. It is important to note

that these prefixes do not correspond to physical folder structures. The term “object storage” stems from the fact that each file is treated as an independent object, bundled with metadata, and assigned a unique identifier. Object storage boasts features such as virtually unlimited storage capacity, robust object analytics based on metadata, API-based access, and cost-effectiveness.

To efficiently handle the vast quantities of data stored in cloud object storage, it is advisable to implement a data lake architecture that leverages this storage medium. A data lake, tailored to encompass the entire enterprise or a specific line of business, acts as a centralized hub for data management and access. Designed to accommodate limitless data volumes, the data lake facilitates the organization of data across various lifecycle stages, including raw, transformed, curated, and ML feature data. Its primary purpose is to consolidate disparate data silos into a singular repository that enables centralized management and access for both analytics and ML requirements. Notably, a data lake can house diverse data formats, such as structured data from databases, unstructured data like documents, semi-structured data in JSON and XML formats, as well as binary formats encompassing images, videos, and audio files. This capability proves particularly invaluable for ML workloads, as ML often involves working with data in multiple formats.

The data lake should be organized into different zones. For example, a *landing zone* should be established as the target for the initial data ingestion from different sources. After data preprocessing and data quality management processing, the data can be moved to the raw data zone. Data in the *raw data zone* can be further transformed and processed to meet different business and downstream consumption needs. To further ensure the reliability of the dataset for usage, the data can be curated and stored in the *curated data zone*. For ML tasks, ML features often need to be pre-computed and stored in an ML feature zone for reuse purposes.

## AWS Lake Formation

AWS Lake Formation is a comprehensive data management service offered by AWS, which streamlines the process of building and maintaining a data lake on the AWS platform. The following figure illustrates the core components of AWS Lake Formation:

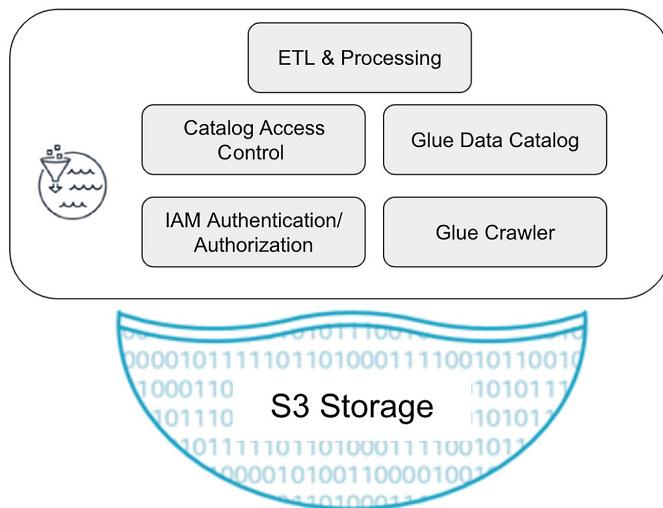


Figure 45.4: AWS Lake Formation

Overall, AWS Lake Formation offers four fundamental capabilities to enhance data lake management:

- **Data source crawler:** This functionality automatically examines data files within the data lake to infer their underlying structure, enabling efficient organization and categorization of the data.
- **Data catalog creation and maintenance:** AWS Lake Formation facilitates the creation and ongoing management of a data catalog, providing a centralized repository for metadata, enabling easy data discovery and exploration within the data lake.
- **Data transformation processing:** With built-in data transformation capabilities, the service allows for the processing and transformation of data stored in the data lake, enabling data scientists and analysts to work with refined and optimized datasets.
- **Data security and access control:** AWS Lake Formation ensures robust data security by providing comprehensive access control mechanisms and enabling fine-grained permissions management, ensuring that data is accessed only by authorized individuals and teams.

Lake Formation integrates with AWS Glue, a serverless **Extract, Transform, Load (ETL)** and data catalog service, to provide data catalog management and data ETL processing functionality. We will cover ETL and data catalog components separately in later sections.

Lake Formation provides a centralized data access management capability for managing data access permissions for databases, tables, or different registered S3 locations. For databases and tables, the permission can be granularly assigned to individual tables and columns and database functions, such as creating tables and inserting records.

## Data ingestion

Data ingestion is the bridge between data sources and data storage. It plays a crucial role in acquiring data from diverse sources, including structured, semi-structured, and unstructured formats, such as databases, knowledge graphs, social media, file storage, and IoT devices. Its primary responsibility is to store this data persistently in various storage solutions like object data storage (e.g., Amazon S3), data warehouses, or other data stores. Effective data ingestion patterns should incorporate both real-time streaming and batch ingestion mechanisms to cater to different types of data sources and ensure timely and efficient data acquisition.

Various data ingestion technologies and tools cater to different ingestion patterns. For streaming data ingestion, popular choices include Apache Kafka, Apache Spark Streaming, and Amazon Kinesis/Kinesis Firehose. These tools enable real-time data ingestion and processing. On the other hand, for batch-oriented data ingestion, tools like **Secure File Transfer Protocol (SFTP)** and AWS Glue are commonly used. AWS Glue, in particular, offers support for a wide range of data sources and targets, including Amazon RDS, MongoDB, Kafka, Amazon DocumentDB, S3, and any databases that support JDBC connections. This flexibility allows for seamless ingestion of data from various sources into the desired data storage or processing systems.

When making decisions on which tools to use for data ingestion, it is important to assess the tools and technologies based on practical needs. The following are some of the considerations when deciding on data ingestion tools:

- **Data format, size, and scalability:** Take into account the various data formats, data size, and scalability needs. ML projects could be using data from different sources and different formats (e.g., **CSV**, **Parquet**, JSON/XML, documents, or image/audio/video files). Determine whether the infrastructure can handle large data volumes efficiently when necessary and scale down to reduce costs during periods of low volume.
- **Ingestion patterns:** Consider the different data ingestion patterns that need to be supported. The tool or combination of several tools should support both batch ingestion patterns (transferring bulk data at specific time intervals) and real-time streaming (processing data such as sensor data or website clickstreams in real time).

- **Data preprocessing capability:** Evaluate whether the ingested data needs to be preprocessed before it is stored in the target data repository. Look for tools that offer built-in processing capability or seamless integration with external processing tools.
- **Security:** Ensure that the selected tools provide robust security mechanisms for authentication and authorization to protect sensitive data.
- **Reliability:** Verify that the tools offer failure recovery mechanisms to prevent critical data loss during the ingestion process. If recovery capability is lacking, ensure there is an option to rerun ingestion jobs from the source.
- **Support for different data sources and targets:** The chosen ingestion tools should be compatible with a wide range of data sources, including databases, files, and streaming sources. Additionally, they should provide an API for easy data ingestion.
- **Manageability:** Another important factor to consider is the level of manageability. Does the tool require self-management, or is it a fully managed solution? Consider the trade-offs between cost and operational complexity before making a decision.

AWS provides several services for data ingestion into a data lake on their platform. These services include Kinesis Data Streams, Kinesis Firehose, AWS Managed Streaming for Kafka, and AWS Glue Streaming, which cater to streaming data requirements. For batch ingestion, options such as AWS Glue, SFTP, and AWS **Data Migration Service (DMS)** are available. In the upcoming section, we will delve into the usage of Kinesis Firehose and AWS Glue to manage data ingestion processes for data lakes. We will also discuss AWS Lambda, a serverless compute service, for a simple and lightweight data ingestion alternative.

## Kinesis Firehose

Kinesis Firehose is a service that streamlines the process of loading streaming data into a data lake. It is a fully managed solution, meaning you don't have to worry about managing the underlying infrastructure. Instead, you can interact with the service's API to handle the ingestion, processing, and delivery of your data.

Kinesis Firehose provides comprehensive support for various scalable data ingestion requirements, including:

- Seamless integration with diverse data sources such as websites, IoT devices, and video cameras. This is achieved using an ingestion agent or ingestion API.
- Versatility in delivering data to multiple destinations, including Amazon S3, Amazon Redshift (an AWS data warehouse service), Amazon OpenSearch (a managed search engine), and Splunk (a log aggregation and analysis product).

- Seamless integration with AWS Lambda and Kinesis Data Analytics, offering advanced data processing capabilities. With AWS Lambda, you can leverage serverless computing to execute custom functions written in languages like Python, Java, Node.js, Go, C#, and Ruby. For more comprehensive information on the functionality of Lambda, please refer to the official AWS documentation.

The following figure illustrates the data flow with Kinesis Firehose:

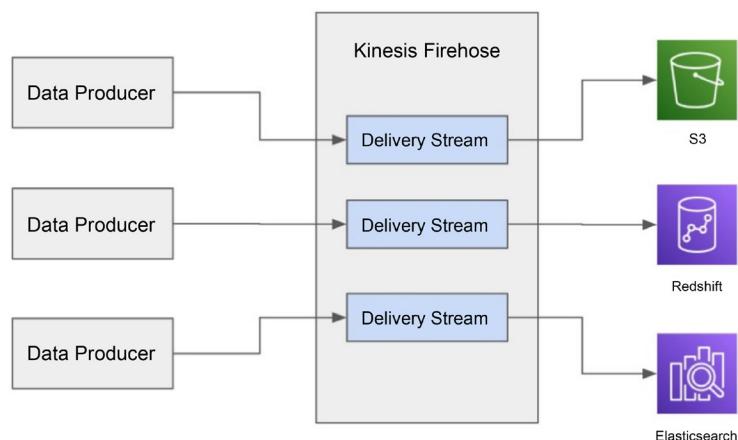


Figure 45.5: Kinesis Firehose data flow

Kinesis operates by establishing delivery streams, which are the foundational components in the Firehose architecture responsible for receiving streaming data from data producers. These delivery streams can be configured with various delivery destinations, such as S3 and Redshift. To accommodate the data volume generated by the producers, you can adjust the throughput of the data stream by specifying the number of shards. Each shard has the capacity to ingest 1 MB/sec of data and can support data reading at a rate of 2 MB/sec. Additionally, Kinesis Firehose offers APIs for increasing the number of shards and merging them when needed.

## AWS Glue

AWS Glue is a comprehensive serverless ETL service that helps manage the data integration and ingestion process for data lakes. It seamlessly connects with various data sources, including transactional databases, data warehouses, and NoSQL databases, facilitating the movement of data to different destinations, such as Amazon S3. This movement can be scheduled or triggered by events. Additionally, AWS Glue offers the capability to process and transform data before delivering it to the target. It provides a range of processing options, such as the Python shell for executing Python

scripts and Apache Spark for Spark-based data processing tasks. With AWS Glue, you can efficiently integrate and ingest data into your data lake, benefiting from its fully managed and serverless nature.

## AWS Lambda

AWS Lambda is AWS's serverless computing platform. It seamlessly integrates with various AWS services, including Amazon S3. By leveraging Lambda, you can trigger the execution of functions in response to events, such as the creation of a new file in S3. These Lambda functions can be developed to move data from different sources, such as copying data from a source S3 bucket to a target landing bucket in a data lake.

It's important to note that AWS Lambda is not specifically designed for large-scale data movement or processing tasks, due to limitations such as memory size and maximum execution time allowed. However, for simpler data ingestion and processing jobs, it proves to be a highly efficient tool.

## Data cataloging

A data catalog plays a crucial role in enabling data analysts and scientists to discover and access data stored in a central data storage. It becomes particularly important during the data understanding and exploration phase of the ML lifecycle when scientists need to search and comprehend available data for their ML projects. When evaluating a data catalog tool, consider the following key factors:

- **Metadata catalog:** The technology should support a central data catalog for effective management of data lake metadata. This involves handling metadata such as database names, table schemas, and table tags. The Hive metastore catalog is a popular standard for managing metadata catalogs.
- **Automated data cataloging:** The technology should have the capability to automatically discover and catalog datasets, as well as to infer data schemas from various data sources like Amazon S3, relational databases, NoSQL databases, and logs. Typically, this functionality is implemented through a crawler that scans data sources, identifies metadata elements (e.g., column names, data types), and adds them to the catalog.
- **Tagging flexibility:** The technology should have the ability to assign custom attributes or tags to metadata entities like databases, tables, and fields. This flexibility supports enhanced data search and discovery capabilities within the catalog.

- **Integration with other tools:** The technology should allow seamless integration of the data catalog with a wide range of data processing tools, enabling easy access to the underlying data. Additionally, native integration with data lake management platforms is advantageous.
- **Search functionality:** The technology should have a robust search capability across diverse metadata attributes within the catalog. This includes searching by database, table, and field names, custom tags or descriptions, and data types.

When it comes to building data catalogs, there are various technical options available. In this section, we first explore how AWS Glue can be utilized for data cataloging purposes. We will also discuss a **Do-It-Yourself (DIY)** option for a data catalog using standard AWS services such as Lambda and OpenSearch.

## AWS Glue Data Catalog

AWS Glue offers a comprehensive solution for data cataloging, integrating seamlessly with AWS Lake Formation and other AWS services. The AWS Glue Data Catalog can be a drop-in replacement for the Hive metastore catalog, so any Hive metastore-compatible applications can work with the AWS Glue Data Catalog. With AWS Glue, you can automatically discover, catalog, and organize your data assets, making them easily searchable and accessible to data analysts and scientists. Here are some key features and benefits of using AWS Glue for data cataloging:

- **Automated data discovery:** AWS Glue provides automated data discovery capabilities. By using data crawlers, Glue can scan and analyze data from diverse structured and semi-structured sources such as Amazon S3, relational databases, NoSQL databases, and more. It identifies metadata information, including table schemas, column names, and data types, that is stored in the AWS Glue Data Catalog.
- **Centralized metadata repository:** The AWS Glue Data Catalog serves as a centralized metadata repository for your data assets. It provides a unified view of your data, making it easier to search, query, and understand the available datasets.
- **Metadata management:** AWS Glue allows you to manage and maintain metadata associated with your data assets. You can define custom tags, add descriptions, and organize your data using databases, tables, and partitions within the Data Catalog.

The metadata hierarchy of the AWS Glue Data Catalog is organized using databases and tables. Databases serve as containers for tables, which hold the actual data. Like traditional databases, a single database can house multiple tables, which can be sourced from various data stores. However, each table is exclusively associated with a single database. To query these databases and tables, one can utilize Hive metastore-compatible tools such as Amazon Athena to execute SQL queries. When collaborating with AWS Lake Formation, access permissions to the catalog's databases and tables can be controlled through the Lake Formation entitlement layer.

## Custom data catalog solution

Another option for building a data catalog is to create your own with a set of AWS services. Consider this option when you have specific requirements that are not met by the purpose-built products. The architecture for this DIY approach involves leveraging services like DynamoDB and Lambda, as depicted in the accompanying diagram:

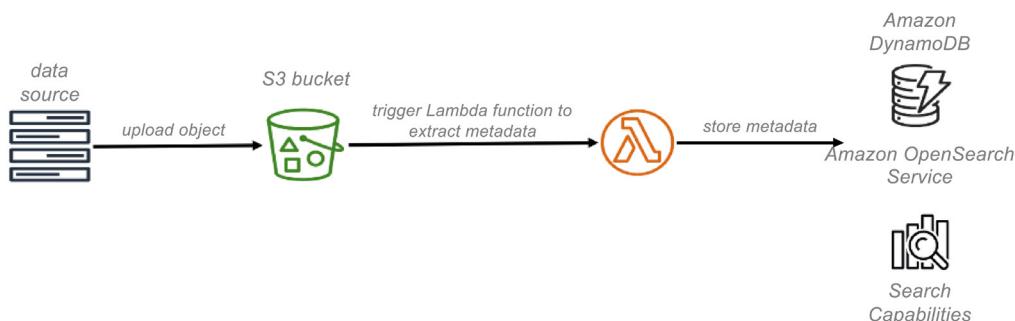


Figure 45.6: Custom data catalog solution

At a high level, AWS Lambda triggers are used to populate DynamoDB tables with object names and metadata when those objects are put into S3; Amazon OpenSearch Service is used to search for specific assets, related metadata, and data classifications.

## Data processing

The data processing functionality of a data lake encompasses the frameworks and compute resources necessary for various data processing tasks, such as data correction, transformation, merging, splitting, and ML feature engineering. This component is a key step in the ML lifecycle as it helps prepare the data for downstream model training and inference steps. Common data processing frameworks include Python shell scripts using libraries such as pandas, NumPy, and Apache Spark. The essential requirements for data processing technology are as follows:

- **Integration and compatibility with the underlying storage technology:** The ability to seamlessly work with the native storage system simplifies data access and movement between the storage and processing layers.
- **Integration with the data catalog:** The capability to interact with the data catalog's metastore to query databases and tables within the catalog.
- **Scalability:** The capacity to scale compute resources up or down to accommodate changing data volumes and processing velocity requirements.
- **Language and framework support:** Support for popular data processing libraries and frameworks, such as Python and Spark.
- **Batch and real-time processing capabilities:** The capability to handle both real-time data streams and bulk data processing in batch mode.

Now, let's examine a selection of AWS services that offer data processing capabilities within a data lake architecture:

- **AWS Glue ETL:** In addition to supporting data movement and data catalogs, the ETL features of AWS Glue can be used for ETL and general-purpose data processing. AWS Glue ETL provides several built-in functions for data transformation, such as dropping the `NULL` field (the `NULL` field represents new data) and data filtering. It also provides general processing frameworks for Python and Spark to run Python scripts and Spark jobs. Glue ETL works natively with the AWS Glue Data Catalog to access the databases and tables in the catalog. Glue ETL can also access the Amazon S3 storage directly.
- **Amazon Elastic MapReduce (EMR):** **Amazon EMR** is a fully managed big data processing platform on AWS. It is designed for large-scale data processing using the Spark framework and other Apache tools, such as **Apache Hive**, **Apache Hudi**, and **Presto**. It integrates with the Glue Data Catalog and Lake Formation natively to access databases and tables in Lake Formation.
- **AWS Lambda:** AWS Lambda can be used for lightweight data processing tasks or as part of a larger data processing pipeline within the data lake architecture. Lambda can be triggered by real-time events, so it is a good option for real-time data processing.

While efficient data processing prepares raw data for model training and consumption, robust data management must also ensure ML teams can track data provenance and access historical versions as needed through capabilities like data versioning.

## ML data versioning

To establish a lineage for model training across training data and ML models, it is crucial to implement version control for the training, validation, and testing datasets. Data

versioning control presents challenges as it necessitates the use of appropriate tools and adherence to best practices by individuals. During the model building process, it is common for data scientists to obtain a copy of a dataset, perform cleansing and transformations specific to their needs, and save the modified data as a new version. This poses significant challenges in terms of data management, including duplication and establishing links between the data and its various upstream and downstream tasks.

Data versioning for the entire data lake is out of the scope of this book. Instead, we will focus on discussing a few architectural options specifically related to versioning control for training datasets.

## S3 partitions

In this approach, each newly created or updated dataset is stored in a separate S3 partition with a unique prefix, typically derived from the name of the S3 folder. While this method can lead to data duplication, it offers a clear and simple approach to differentiate between different datasets intended for model training. To maintain data integrity, it is advisable to generate datasets through a controlled processing pipeline that enforces naming standards. The processing pipeline should also track data provenance and record the processing scripts used for data manipulation and feature engineering. Furthermore, the datasets should be configured as read-only for downstream applications, ensuring their immutability. The following example showcases an S3 partition structure, illustrating multiple versions of a training dataset:

```
s3://project1/<date>/<unique version id 1>/train_1.txt
s3://project1/<date>/<unique version id 1>/train_2.txt
s3://project1/<date>/<unique version id 2>/train_1.txt
s3://project1/<date>/<unique version id 2>/train_2.txt
```

In this instance, the two versions of the dataset are segregated using distinct S3 prefixes. To effectively track these training files, it is recommended to employ a database for storing metadata pertaining to these training files. When utilizing these files, it is crucial to establish links between the training datasets, ML training jobs, ML training scripts, and the resulting ML models to establish a comprehensive lineage.

## Versioned S3 buckets

Amazon S3 offers versioning support for S3 buckets, which can be leveraged to manage different versions of training datasets when enabled. With this approach, each newly created or updated dataset is assigned a unique version ID at the S3 object level.

Additionally, it is recommended to utilize a database to store all relevant metadata associated with each version of the training dataset. This enables the establishment of lineage, tracking the journey from data processing to ML model training. The metadata should capture essential information to facilitate comprehensive tracking and analysis.

## Purpose-built data version tools

Instead of developing custom solutions for data version control, there are purpose-built tools available for efficient data version management. For example, these tools can be used to track and store different versions of ML training and validation datasets, which are important for repeatable experimentations and model training tasks. Here are a few notable options:

- **Git LFS (Large File Storage):** Git LFS extends Git's capabilities to handle large files, including datasets. It stores these files outside the Git repository while retaining versioning information. Git LFS seamlessly integrates with Git and is commonly used to version large files in data-centric projects.
- **DataVersionControl (DVC):** DVC is an open-source tool designed specifically for data versioning and management. It integrates with Git and provides features for tracking and managing large datasets. DVC enables lightweight links to actual data files stored in remote storage, such as Amazon S3 or a shared file system. This approach maintains a history of changes and allows easy switching between different dataset versions, eliminating the need for data duplication.
- **Pachyderm:** Pachyderm is an open-source data versioning and data lineage tool. It offers version control for data pipelines, enabling tracking of changes to data, code, and configuration files. Pachyderm supports distributed data processing frameworks like Apache Spark and provides features like reproducibility, data lineage, and data lineage-based branching.

These purpose-built tools streamline the process of data versioning, ensuring efficient tracking and management of datasets.

## ML feature stores

In large enterprises, it is beneficial to centrally manage common reusable ML features like curated customer profile data and standardized product sales data. This practice helps reduce the ML project lifecycle, particularly during the data understanding and data preparation stages. To achieve this, many organizations have built central ML feature stores, an architectural component for storing common reusable ML features, as part of the ML development architecture to meet the downstream model development,

training, and model inference needs. Depending on the specific requirements, there are two main options for managing these reusable ML features.

Firstly, you can build custom feature stores that fulfill the fundamental requirements of inserting and looking up organized features for ML model training. These custom feature stores can be tailored to meet the specific needs of the organization.

Alternatively, you can opt for commercial-grade feature store products, such as Amazon SageMaker Feature Store, a ML service offered by AWS. It provides advanced capabilities such as online and offline functionality for training and inference, metadata tagging, feature versioning, and advanced search. These features enable efficient management and utilization of ML features in production-grade scenarios.

## Data serving for client consumption

The central data management platform should offer various methods, such as APIs or Hive metastore-based approaches, to facilitate online access to the data for downstream tasks such as data discovery and model training. Additionally, it is important to consider data transfer tools that support the movement of data from the central data management platform to other data-consuming environments, catering to different data consumption patterns such as local access to the data in the consuming environment. It is advantageous to explore tools that either have built-in data serving capabilities or can be seamlessly integrated with external data serving tools, as building custom data serving features could be a challenging engineering undertaking.

When supplying data to data science environments, there are multiple data serving patterns to consider. In the following discussion, we will explore two prominent data access patterns and their characteristics.

### Consumption via API

In this data serving pattern, consumption environments and applications have the capability to directly access data from the data lake. This can be achieved using Hive metastore-compliant tools or through direct access to S3, the underlying storage of the data lake. Amazon provides various services that facilitate this pattern, such as Amazon Athena, a powerful big data query tool, Amazon EMR, a robust big data processing tool, and Amazon Redshift Spectrum, a feature of Amazon Redshift.

By leveraging these services, data lake data indexed in Glue catalogs can be queried without the need to make a separate copy of the data. This pattern is particularly suitable when only a subset of the data is required for downstream data processing tasks. It offers the advantage of avoiding data duplication while enabling efficient selection and processing of specific data subsets as part of the overall data workflow.

## Consumption via data copy

In this data serving pattern, a specific portion of the data stored in the data lake is replicated or copied to the storage of the consumption environment. This replication allows for tailored processing and consumption based on specific needs. For instance, the latest or most relevant data can be loaded into a data analytics environment such as Amazon Redshift. Similarly, it can be delivered to S3 buckets owned by a data science environment, enabling efficient access and utilization for data science tasks. By replicating the required data subsets, this pattern provides flexibility and optimized performance for different processing and consumption requirements in various environments.

## Special databases for ML

Considering emerging ML paradigms like graph neural networks and generative AI, specialized databases have been developed to cater to ML-specific tasks such as link prediction, cluster classification, and retrieval-augmented generation. In the following section, we will delve into two types of databases—vector databases and graph databases—and examine how they are utilized in ML tasks. We will explore their unique characteristics and applications in the context of ML.

### Vector databases

Vector databases, also known as vector similarity search engines or vector stores, are specialized databases designed to efficiently store, index, and query high-dimensional vectors. Examples of high-dimensional vectors include numerical vectors' representation of images or text. These databases are particularly well suited for ML applications that rely on vector-based computations.

In ML, vectors are commonly used to represent data points, embeddings, or feature representations. These vectors capture essential information about the underlying data, enabling similarity search, clustering, classification, and other ML tasks. Vector databases provide powerful tools for handling these vector-based operations at scale.

One of the key features of vector databases is their ability to perform fast similarity searches, allowing efficient retrieval of vectors that are most similar to a given query vector. This capability is essential in various ML use cases, such as recommender systems, content-based search, and anomaly detection.

There are several vector database providers on the market, each offering its own unique features and capabilities. Some of the prominent ones include:

- **Facebook AI Similarity Search (FAISS)**: Developed by **Facebook AI Research (FAIR)**, FAISS is an open-source library for efficient similarity search and clustering of dense vectors. It provides highly optimized algorithms and data structures for fast and scalable vector search.
- **Milvus**: Milvus is an open-source vector database designed for managing and serving large-scale vector datasets. It offers efficient similarity search, supports multiple similarity metrics, and provides scalability through distributed computing.
- **Pinecone**: Pinecone is a cloud-native vector database service that specializes in high-performance similarity search and recommendation systems. It offers real-time indexing and retrieval of vectors with low latency and high throughput.
- **Elasticsearch**: Although primarily known as a full-text search and analytics engine, Elasticsearch also provides vector similarity search capabilities using plugins for efficient vector indexing and querying.
- **Weaviate**: Weaviate is an open-source vector database. It allows you to store data objects and vector embeddings from your favorite ML models, and scale seamlessly into billions of data objects.

These are just a few examples of vector database providers, and the landscape is continuously evolving with new solutions and advancements in the field. When choosing a vector database provider, it's important to consider factors such as performance, scalability, ease of integration, and the specific requirements of your ML use case.

## Graph databases

Graph databases are specialized databases designed to store, manage, and query graph-structured data. In a graph database, data is represented as nodes (entities) and edges (relationships) connecting these nodes, forming a graph-like structure. Graph databases excel at capturing and processing complex relationships and dependencies between entities, making them highly relevant for ML tasks.

Graph databases offer a powerful way to model and analyze data in domains where relationships play a crucial role, such as social networks, recommendation systems, fraud detection, knowledge graphs, and network analysis. They enable efficient traversal of the graph, allowing for queries that explore connections and patterns within the data.

In the context of ML, graph databases have multiple applications. One key use case is graph-based feature engineering, where graphs are used to represent relationships between entities, and the graph structure is leveraged to derive features that can enhance the performance of ML models. For example, in a recommendation system, a graph database can represent user-item interactions and graph-based features can be derived to capture user similarities, item similarities, or collaborative filtering patterns.

Graph databases also enable graph-based algorithms, such as **graph convolutional networks (GCNs)**, for tasks like node classification, link prediction, and graph clustering. These algorithms leverage the graph structure to propagate information across nodes and capture complex patterns in the data.

Furthermore, graph databases can be used to store and query graph embeddings, which are low-dimensional vector representations of nodes or edges. These embeddings capture the structural and semantic information of the graph and can be input to ML models for downstream tasks, such as node classification or recommendation.

Some of the notable graph databases include **Neo4j**, a popular and widely used graph database that allows for efficient storage, retrieval, and querying of graph-structured data, and Amazon Neptune, a fully managed graph database service provided by AWS.

## Data pipelines

Data pipelines streamline the flow of data by automating tasks such as data ingestion, validation, transformation, and feature engineering. These pipelines ensure data quality and facilitate the creation of training and validation datasets for ML models. Numerous workflow tools are available for constructing data pipelines, and many data management tools offer built-in capabilities for building and managing these pipelines:

- **AWS Glue workflows:** AWS Glue workflows provide a native workflow management feature within AWS Glue, enabling the orchestration of various Glue jobs like data ingestion, processing, and feature engineering. Comprised of trigger and node components, a Glue workflow incorporates schedule triggers, event triggers, and on-demand triggers. Nodes within the workflow can be either crawler jobs or ETL jobs. Triggers initiate workflow runs, while event triggers are emitted after the completion of crawler or ETL jobs. By structuring a series of triggers and jobs, workflows facilitate the seamless execution of data pipelines within AWS Glue.
- **AWS Step Functions:** AWS Step Functions is a powerful workflow orchestration tool that seamlessly integrates with various AWS data processing services like AWS Glue and Amazon EMR. It enables the creation of robust workflows

to execute diverse steps within a data pipeline, such as data ingestion, data processing, and feature engineering, ensuring smooth coordination and execution of these tasks.

- **AWS Managed Workflows for Apache Airflow:** AWS Managed Workflows for Apache Airflow (MWAA) is a fully managed service that simplifies the deployment, configuration, and management of Apache Airflow, an open-source platform for orchestrating and scheduling data workflows. This service offers scalability, reliability, and easy integration with other AWS services, making it an efficient solution for managing complex data workflows in the cloud.

Having explored the fundamental elements of ML data management architecture, the subsequent sections will delve into subjects related to security and governance.

## Authentication and authorization

Authentication and authorization are crucial for ensuring secure access to a data lake. Federated authentication, such as AWS Identity and Access Management (IAM), verifies user identities for administration and data consumption purposes. AWS Lake Formation combines the built-in Lake Formation access control with AWS IAM to govern access to data catalog resources and underlying data storage.

The built-in Lake Formation permission model utilizes commands like grant and revoke to control access to resources such as databases and tables, as well as actions like table creation. When a user requests access to a resource, both IAM policies and Lake Formation permissions are evaluated to verify and enforce access before granting it. This multi-layered approach enhances data lake security and governance.

There are several personas involved in the administration of the data lake and consumption of the data lake resources, including:

- **Lake Formation administrator:** A Lake Formation administrator has permission to manage all aspects of a Lake Formation data lake in an AWS account. Examples include granting/revoking permissions to access data lake resources for other users, registering data stores in S3, and creating/deleting databases. When setting up Lake Formation, you will need to register as an administrator. An administrator can be an AWS IAM user or IAM role. You can add more than one administrator to a Lake Formation data lake.
- **Lake Formation database creator:** A Lake Formation database creator is granted permission to create databases in Lake Formation. A database creator can be an IAM user or IAM role.

- **Lake Formation database user:** A Lake Formation database user can be granted permission to perform different actions against a database. Example permissions include create table, drop table, describe table, and alter table. A database user can be an IAM user or IAM role.
- **Lake Formation data user:** A Lake Formation data user can be granted permission to perform different actions against database tables and columns. Example permissions include insert, select, describe, delete, alter, and drop. A data user can be an IAM user or an IAM role.

Accessing and querying the database and tables in Lake Formation is facilitated through compatible AWS services like Amazon Athena and Amazon EMR. When performing queries using these services, Lake Formation verifies the principals (IAM users, groups, and roles) associated with them to ensure they have the necessary access permissions for the database, tables, and corresponding S3 data location. If access is granted, Lake Formation issues a temporary credential to the service, enabling it to execute the query securely and efficiently. This process ensures that only authorized services can interact with Lake Formation and perform queries on the data.

## Data governance

Having secure access to trustworthy data is essential to the success of an ML initiative. Data governance encompasses essential practices to ensure the reliability, security, and accountability of data assets. Trustworthy data is achieved through the identification and documentation of data flows, as well as the measurement and reporting of data quality. Data protection and security involve classifying data and applying appropriate access permissions to safeguard its confidentiality and integrity. To maintain visibility of data activities, monitoring and auditing mechanisms should be implemented, allowing organizations to track and analyze actions performed on data, ensuring transparency and accountability in data management.

A data catalog is one of the most important components of data governance. On AWS, the Glue Data Catalog is a fully managed service for data catalog management. You also have the option to build custom data catalogs using different foundational building blocks. For example, you can follow the reference architecture at <https://docs.aws.amazon.com/whitepapers/latest/enterprise-data-governance-catalog/implementation-reference-architecture-diagrams.html> for building a custom data catalog on AWS.

## Data lineage

To establish and document data lineage during the ingestion and processing of data across different zones, it is important to capture specific data points. When utilizing data ingestion and processing tools like AWS Glue, AWS EMR, or AWS Lambda in a

data pipeline, the following information can be captured to establish comprehensive data lineage:

- **Data source details:** Include the name of the data source, its location, and ownership information to identify the origin of the data.
- **Data processing job history:** Capture the history and details of the data processing jobs involved in the pipeline. This includes information such as the job name, unique **identifier (ID)**, associated processing script, and owner of the job.
- **Generated artifacts:** Document the artifacts generated because of the data processing jobs. For example, record the S3 URI or other storage location for the target data produced by the pipeline.
- **Data metrics:** Track relevant metrics at different stages of data processing. This can include the number of records, data size, data schema, and feature statistics to provide insights into the processed data.

To store and manage data lineage information and processing metrics, it is recommended to establish a central data operational data store. AWS DynamoDB, a fully managed NoSQL database, is an excellent technology choice for this purpose. With its capabilities optimized for low latency and high transaction access, DynamoDB provides efficient storage and retrieval of data lineage records and processing metrics. By capturing and documenting these data points, organizations can establish a comprehensive data lineage that provides a clear understanding of the data's journey from its source through various processing stages. This documentation enables traceability, auditability, and better management of the data as it moves through the pipeline.

## Other data governance measures

In addition to managing data lineage, there are several other important measures for effective data governance, including:

- **Data quality:** Automated data quality checks should be implemented at different stages, and quality metrics should be reported. For example, after the source data is ingested into the landing zone, an AWS Glue quality check job can run to check the data quality using tools such as the open-source Deequ library. Data quality metrics (such as counts, schema validation, missing data, the wrong data type, or statistical deviations from the baseline) and reports can be generated for reviews. Optionally, manual or automated operational data cleansing processes should be established to correct data quality issues.

- **Data cataloging:** Create a central data catalog and run Glue crawlers on datasets in the data lake to automatically create an inventory of data and populate the central data catalog. Enrich the catalogs with additional metadata to track other information to support discovery and data audits, such as the business owner, data classification, and data refresh date. For ML workloads, data science teams also generate new datasets (for example, new ML features) from the existing datasets in the data lake for model training purposes. These datasets should also be registered and tracked in a data catalog, and different versions of the data should be retained and archived for audit purposes.
- **Data access provisioning:** A formal process should be established for requesting and granting access to datasets and Lake Formation databases and tables. An external ticketing system can be used to manage the workflow for requesting access and granting access.
- **Monitoring and auditing:** Data access should be monitored, and access history should be maintained. Amazon S3 server access logging can be enabled to track access to all S3 objects directly. AWS Lake Formation also records all accesses to Lake Formation datasets in **AWS CloudTrail** (AWS CloudTrail provides event history in an AWS account to enable governance, compliance, and operational auditing). With Lake Formation auditing, you can get details such as event source, event name, SQL queries, and data output location.

By implementing these key data governance measures, organizations can establish a strong foundation for data management, security, and compliance, enabling them to maximize the value of their data assets while mitigating risks.

## Hands-on exercise – data management for ML

In this hands-on exercise, you will go through the process of constructing a simple data management platform for a fictional retail bank. This platform will serve as the foundation for an ML workflow, and we will leverage different AWS technologies to build it. If you don't have an AWS account, you can easily create one by following the instructions at <https://aws.amazon.com/console/>.

The data management platform we create will have the following key components:

- A data lake environment for data management using Lake Formation
- A data ingestion component for ingesting files to the data lake using Lambda
- A data catalog component using the Glue Data Catalog
- A data discovery and query component using the Glue Data Catalog and Athena

- A data processing component using Glue ETL
- A data pipeline component using a Glue pipeline

The following diagram shows the data management architecture we will build in this exercise:

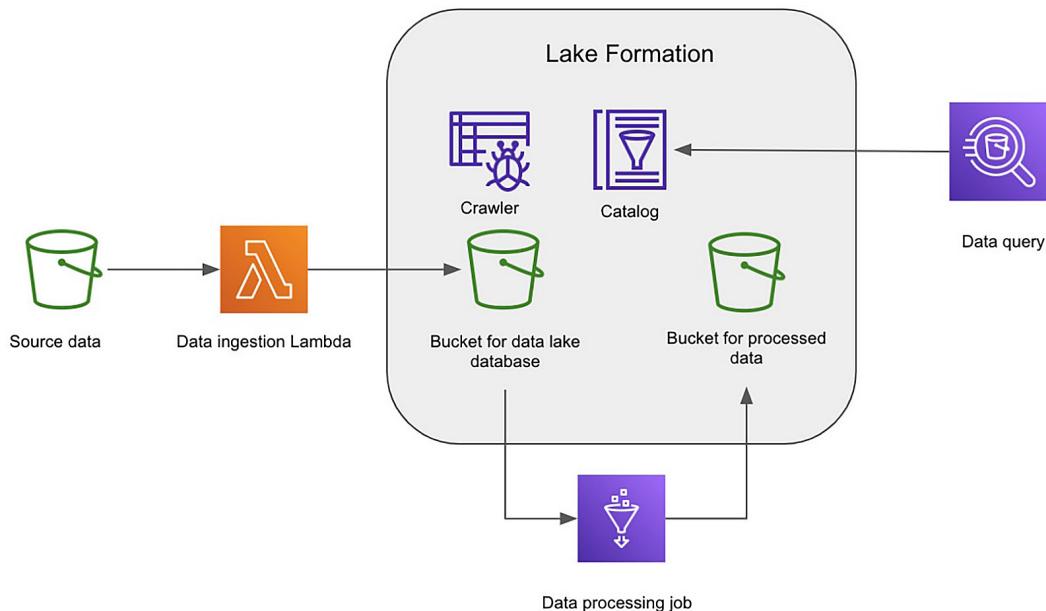


Figure 45.7: Data management architecture for the hands-on exercise

Let's get started with building out this architecture on AWS.

## Creating a data lake using Lake Formation

We will build the data lake architecture using AWS Lake Formation; it is the primary service for building data lakes on AWS. After you log on to the **AWS Management Console**, create an S3 bucket called `MLSA-DataLake-<your initials>`. We will use this bucket as the storage for the data lake. If you get a message that the bucket name is already in use, try adding some random characters to the name to make it unique. If you are not familiar with how to create S3 buckets, follow the instructions at the following link: <https://docs.aws.amazon.com/AmazonS3/latest/user-guide/create-bucket.html>

After the bucket is created, follow these steps to get started with creating a data lake:

1. **Register Lake Formation administrators:** We need to add Lake Formation administrators to the data lake. The administrators will have full permission to manage all aspects of the data lake. To do this, navigate to the Lake Formation management console, click on the **Administrative roles and tasks** link, and you should be prompted to add an administrator. Select **Add myself** and click on the **Get started** button.
2. **Register S3 storage:** Next, we need to register the S3 bucket (`MLSA-DataLake-<your initials>`) you created earlier in Lake Formation, so it will be managed and accessible through Lake Formation. To do this, click on the **Dashboard** link, expand **Data lake setup**, and then click on **Register Location**. Browse and select the bucket you created and click on **Register Location**. This S3 bucket will be used by Lake Formation to store data for the databases and manage its access permissions.
3. **Create database:** Now, we are ready to set up a database called `bank_customer_db` for managing retail customers. Before we register the database, let's first create a folder called the `bank_customer_db` folder under the `MLSA-DataLake-<your initials>` bucket. This folder will be used to store data files associated with the database. To do this, click on the **Create database** button on the Lake Formation dashboard and follow the instructions on the screen to create the database.

You have now successfully created a data lake powered by Lake Formation and created a database for data management. With this data lake created, we are now ready to build additional data management components.

## Creating a data ingestion pipeline

Now that the database is prepared, we can proceed to ingest data into this newly created database. As mentioned earlier, there are various data sources available, including databases like Amazon RDS, streaming platforms like social media feeds, and logs such as CloudTrail. Additionally, AWS offers a range of services for building data ingestion pipelines, such as AWS Glue, Amazon Kinesis, and AWS Lambda. In this phase of the exercise, we will focus on creating an AWS Lambda function job that will facilitate the ingestion of data from other S3 buckets into our target database. As mentioned earlier, Lambda functions can be used for lightweight data ingestion and processing tasks:

1. **Create a source S3 bucket and download data files:** Let's create another S3 bucket, called `customer-data-source`, to represent the data source where we will ingest the data from.
2. **Create a Lambda function:** Now, we will create the Lambda function that ingests data from the `customer-data-source` bucket to the `MLSA-DataLake-<your initials>` bucket:
  - i. To get started, navigate to the AWS Lambda management console, click on the **Functions** link in the left pane, and click on the **Create Function** button in the right pane. Choose **Author from scratch**, then enter `datalake-s3-ingest` for the function name, and select the latest Python version (e.g., 3.10) as the runtime. Keep the default for the execution role, which will create a new IAM role for this Lambda function. Click on **Create function** to continue.
  - ii. On the next screen, click on **Add trigger**, select **S3** as the trigger, and select the `customer-data-source` bucket as the source. For **Event Type**, choose the **Put** event and click on the **Add** button to complete the step. This trigger will allow the Lambda function to be invoked when there is an S3 bucket event, such as saving a file into the bucket.
  - iii. After you add the trigger, you will be brought back to the `Lambda->function-> datalake-s3-ingest` screen. Next, let's create the function by replacing the default function template with the following code block. Replace the `desBucket` variable with the name of the actual bucket:

```
import json
import boto3
def lambda_handler(event, context):
 s3 = boto3.resource('s3')
 for record in event['Records']:
 srcBucket = record['s3'][‘bucket’][‘name’]
 srckeyp = record[‘s3’][‘object’][‘key’]
 desBucket = “MLSA-DataLake-<your initials>”
 desFolder = srckeyp[0:srckeyp.find(‘.’)]
 desKey = “bank_customer_db/” + desFolder + “/” +
 srckeyp
 source= { ‘Bucket’ : srcBucket,’Key’:srckeyp}
 dest ={ ‘Bucket’ : desBucket,’Key’:desKey}
 s3.meta.client.copy(source, desBucket, desKey)
```

```
 return {
 'statusCode': 200,
 'body': json.dumps('files ingested')
 }
```

- iv. The new function will also need S3 permission to copy files (*objects*) from one bucket to another. For simplicity, just add the `AmazonS3FullAccess` policy to the **execution IAM role** associated with the function. You can find the IAM role by clicking on the **Permission** tab for the Lambda function.
3. **Trigger data ingestion:** Now, download the sample data files

Then, save the file to your local machine. Extract the archived files. There should be two files (`customer_data.csv` and `churn_list.csv`).

You can now trigger the data ingestion process by uploading the `customer_detail.csv` and `churn_list.csv` files to the `customer-data-source` bucket and verify the process completion by checking the `MLSA-DataLake-<your initials>/bank_customer_db` folder for the two files.

You have now successfully created an AWS Lambda-based data ingestion pipeline to automatically move data from a source S3 bucket to a target S3 bucket. With this simple ingestion pipeline created and data moved, we are now ready to implement components to support the discovery of these data files.

## Creating a Glue Data Catalog

To allow discovery and querying of the data in the `bank_customer_db` database, we need to create a data catalog. As discussed earlier, Glue Data Catalog is a managed data catalog on AWS. It comes with a utility called an AWS Glue crawler that can help discover data and populate the catalog.

Here, we will use an AWS Glue crawler to crawl the files in the `bank_customer_db` S3 folder and generate the catalog:

1. **Grant permission for Glue:**

- i. First, let's grant permission for AWS Glue to access the `bank_customer_db` database. We will create a new IAM role for the Glue service to assume on your behalf. To do this, create a new IAM service role called `AWSGlueServiceRole_data_lake`, and attach the `AWSGlueServiceRole` and `AmazonS3FullAccess` IAM-managed policies to it. Make sure you select **Glue** as the service when you create the role. If you are not familiar with

how to create a role and attach a policy, follow the instructions at the following link: <https://docs.aws.amazon.com/IAM/latest/UserGuide>

- ii. After the role is created, click on **Data lake permission** in the left pane of the Lake Formation management console and then click the **Grant** button in the right pane.

On the next screen, select **AWSGlueServiceRole\_data\_lake** for **IAM users and role** and **bank\_customer\_db** under **Named data catalog resources**, choose **Super** for both **Database permissions** and **Grantable permissions**, and finally click on **Grant**. The **Super** permission allows the service role to have access to create databases and grant permission as part of the automation. **AWSGlueServiceRole\_data\_lake** will be used later to configure the Glue crawler job.

## 2. Configure the Glue crawler job:

- i. Launch the Glue crawler by clicking on the **Crawler** link in the Lake Formation management console. A new browser tab for Glue will open. Click on the **Create Crawler** button to get started. Enter **bank\_customer\_db\_crawler** as the name of the crawler. Click on the **Add a data source** button, select **S3**, and enter **s3://MLSA-DataLake-<your initials>/bank\_customer\_db/churn\_list/** for the **include path** field.
- ii. Click on the **Add another data source** button again. This time, enter **s3://MLSA-DataLake-<your initials>/bank\_customer\_db/customer\_data/**.
3. On the next screen, **Configure security settings**, select **AWSGlueServiceRole\_data\_lake** for the existing IAM role, which you used earlier:
  - i. On the next **Set output and scheduling** screen, select **bank\_customer\_db** as the target database, and choose **on demand** as the frequency for the crawler schedule.
  - ii. On the next **Review and create** screen, select **Finish** on the final screen to complete the setup.
  - iii. On the **Crawler** screen, select the **bank\_customer\_db\_crawler** job you just created, click on **Run crawler**, and wait for the status to say **Ready**.
  - iv. Navigate back to the Lake Formation management console and click on the **Tables** link. You will now see two new tables created (**churn\_list** and **customer\_data**).
  - v. You have now successfully configured an AWS Glue crawler that automatically discovers table schemas from data files and creates data catalogs for the new data.

You have successfully created the Glue Data Catalog for the newly ingested data. With that, we now have the proper component to support data discovery and query. Next,

we will use Lake Formation and Athena to discover and query the data in the data lake.

## Discovering and querying data in the data lake

To facilitate the data discovery and data understanding phase of the ML workflow, it is essential to incorporate data discovery and data query capabilities within the data lake.

By default, Lake Formation already provides a list of tags, such as data type classification (for example, CSV), for searching tables in the database. Let's add a few more tags for each table to make it more discoverable:

1. Grant permission to edit the database tables by granting your current user ID **Super** permission for both the `customer_data` and `churn_list` tables.
2. Let's add some metadata to the table fields. Select the `customer_data` table, click on **Edit Schema**, select the `creditscore` field, click on **Edit** and **Add** to add a column property, and enter the following, where `description` is the key and the actual text is the value:

```
description: credit score is the FICO score for each customer
```

3. Follow the same previous steps and add the following column property for the `exited` field in the `churn_list` table:

```
description: churn flag
```

4. We are now ready to do some searches using metadata inside the Lake Formation management console. Try typing the following words separately in the text box for **Find table by properties** to search for tables and see what's returned:

- `FICO`
- `csv`
- `churn flag`
- `creditscore`
- `customerid`

Now that you have found the table you are looking for, let's query the table and see the actual data to learn how to query the data interactively, which is an important task performed by data scientists for data exploration and understanding. Select the table you want to query and click on the **View data** button in the **Actions** drop-down menu. This should bring you to the **Amazon Athena** screen. You should see a **Query**

tab already created, and the query already executed. The results are displayed at the bottom of the screen. If you get a warning message stating that you need to provide an output location, select the **Settings** tab, and then click on the **Manage** button to provide an S3 location as the output location. You can run any other SQL query to explore the data further, such as joining the `customer_data` and `churn_list` tables with the `customerid` field:

```
SELECT * FROM "bank_customer_db"."customer_data", " bank_customer_db"."churn_list" where "bank_customer_db"."customer_data"."customerid" = "bank_customer_db"."churn_list"."customerid" ;
```

You have now learned how to discover the data in Lake Formation and run queries against the data in a Lake Formation database and tables. Next, let's run a data processing job using the Amazon Glue ETL service to make the data ready for ML tasks.

## Creating an Amazon Glue ETL job to process data for ML

The `customer_data` and `churn_list` tables contain features that are useful for ML. However, they need to be joined and processed so they can be used to train ML models. One option is for the data scientists to download these datasets and process them in a Jupyter notebook for model training. Another option is to process the data using a separate processing engine so that the data scientists can work with the processed data directly. Here, we will set up an AWS Glue job to process the data in the `customer_data` and `churn_list` tables and transform them into new ML features that are ready for model training directly:

1. First, create a new S3 bucket called `MLSA-DataLake-Serving-<your initials>`. We will use this bucket to store the output training datasets from the Glue job.
2. Using the Lake Formation console, grant `AWSGlueService_Role` **Super** access to the `customer_data` and `churn_list` tables. We will use this role to run the Glue job.
3. To start creating the Glue job, go to the Glue console and click on the **ETL Jobs** link on the Glue console. Click on **Script editor** and then click on the **Create script** button.
4. On the script editor screen, change the job name from **Untitled job** to `customer_churn_process` for easy tracking.

5. On the **Job details** tab, select `AWSGlueService_Role` as the IAM role. Add a new Job parameter called `target_bucket` under `Advanced Properties` and enter the value of your target bucket for the output files.
6. On the **Script tab** screen, copy the following code blocks to the code section. Make sure to replace `default_bucket` with your own bucket in the code. The following code block first joins the `churn_list` and `customer_data` tables using the `customerid` column as the key, then transforms the `gender` and `geo` columns with an index, creates a new DataFrame with only the relevant columns, and finally saves the output file to an S3 location using the date and generated version ID as partitions. The code uses default values for the target bucket and prefix variables and generates a date partition and version partition for the S3 location. The job can also accept input arguments for these parameters.

The following code block sets up default configurations, such as `SparkContext` and a default bucket:

```
import sys
from awsglue.utils import getResolvedOptions
from awsglue.transforms import Join
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
import pandas as pd
from datetime import datetime
import uuid
from pyspark.ml.feature import StringIndexer
glueContext = GlueContext(SparkContext.getOrCreate())
logger = glueContext.get_logger()
current_date = datetime.now()
default_date_partition = f'{current_date.year}-{current_date.month}-'
{current_date.day}'
default_version_id = str(uuid.uuid4())
default_bucket = "<your default bucket name>"
default_prefix = "ml-customer-churn"
target_bucket = ""
prefix = ""
day_partition = ""
version_id = ""
```

```
try:
 args = getResolvedOptions(sys.argv,['JOB_NAME','target_
bucket','prefix','day_partition','version_id'])
 target_bucket = args['target_bucket']
 prefix = args['prefix']
 day_partition = args['day_partition']
 version_id = args['version_id']
except:
 logger.error("error occurred with getting arguments")
if target_bucket == "":
 target_bucket = default_bucket
if prefix == "":
 prefix = default_prefix
if day_partition == "":
 day_partition = default_date_partition
if version_id == "":
 version_id = default_version_id
```

The following code joins the `customer_data` and `churn_list` tables into a single table using the `customerid` column as the key:

```
catalog: database and table names
db_name = "bank_customer_db"
tbl_customer = "customer_data"
tbl_churn_list = "churn_list"
Create dynamic frames from the source tables
customer = glueContext.create_dynamic_frame.from_
catalog(database=db_name, table_name=tbl_customer)
churn = glueContext.create_dynamic_frame.from_catalog(database=db_
name, table_name=tbl_churn_list)
Join the frames to create customer churn dataframe
customer_churn = Join.apply(customer, churn, 'customerid',
'customerid')
customer_churn.printSchema()
```

The following code block transforms several data columns from string labels to label indices and writes the final file to an output location in S3:

```
---- Write out the combined file ----
current_date = datetime.now()
str_current_date = f'{current_date.year}-{current_date.month}-
{current_date.day}'
random_version_id = str(uuid.uuid4())
output_dir = f's3://{target_bucket}/{prefix}/{day_partition}/
{version_id}'
s_customer_churn = customer_churn.toDF()
gender_indexer = StringIndexer(inputCol="gender",
outputCol="genderindex")
s_customer_churn = gender_indexer.fit(s_customer_churn).transform(s_
customer_churn)
geo_indexer = StringIndexer(inputCol="geography",
outputCol="geographyindex")
s_customer_churn = geo_indexer.fit(s_customer_churn).transform(s_
customer_churn)
s_customer_churn = s_customer_churn.select('geographyindex',
'estimatedsalary','hascrcard','numofproducts', 'balance', 'age',
'genderindex', 'isactivemember', 'creditscore', 'tenure', 'Exited')
s_customer_churn = s_customer_churn.coalesce(1)
s_customer_churn.write.option("header","true").format("csv").
mode('Overwrite').save(output_dir)
logger.info("output_dir:" + output_dir)
```

7. Click on **Save** and then the **Run job** button to run the job. Check the job running status by clicking on the **ETL jobs** link in the Glue console, and then click on **Job run monitoring**.
8. After the job completes, check the `s3://MLSA-DataLake-Serving-<your initials>/ml-customer-churn/<date>/<guid>/` location in S3 and see whether a new CSV file was generated. Open the file and see whether you see the new processed dataset in the file.

You have now successfully built an AWS Glue job for data processing and feature engineering for ML. With this, you can automate data processing and feature engineering, which is critical to achieve reproducibility and governance. Try creating a crawler to crawl the newly processed data in the `MLSA-DataLake-Serving-<your`

initials> bucket to make it available in the Glue catalog and run some queries against it. You should see a new table created with multiple partitions (for example, m1-customer-churn, date, and GUID) for the different training datasets. You can query the data by using the GUID partition as a query condition.

## Building a data pipeline using Glue workflows

Next, we will construct a pipeline that executes a data ingestion job, followed by the creation of a database catalog for the data. Finally, a data processing job will be initiated to generate the training dataset. This pipeline will automate the flow of data from the source to the desired format, ensuring seamless and efficient data processing for ML model training:

1. To start, click on the **Workflows (orchestration)** link in the left pane of the Glue management console.
2. Click on **Add workflow** and enter a name for your workflow on the next screen. Then, click on the **Create workflow** button.
3. Select the workflow you just created and click on **Add trigger**. Select the **Add New** tab, and then enter a name for the trigger and select the on-demand trigger type.
4. On the workflow UI designer, you will see a new **Add Node** icon show up. Click on the **Add Node** icon, select the **Crawler** tab, and select `bank_customer_db_crawler`, then click on **Add**.
5. On the workflow UI designer, click on the **Crawler** icon, and you will see a new **Add Trigger** icon show up. Click on the **Add Trigger** icon, select the **Add new** tab, and select **Start after ANY event** as the trigger logic, and then click on **Add**.
6. On the workflow UI designer, click on the **Add Node** icon, select the **Jobs** tab, and select the `customer_churn_process` job.
7. On the workflow UI designer, the final workflow should look like the following diagram:

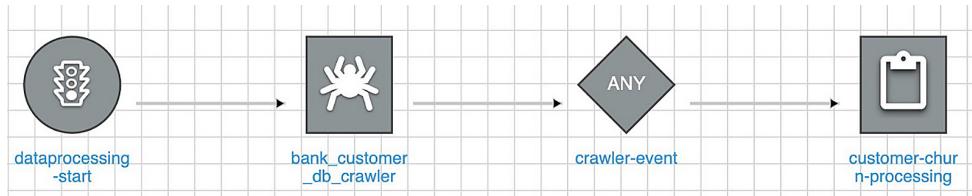


Figure 45.8: Glue data flow design

8. Now, you are ready to run the workflow. Select the workflow and select **Run** from the **Actions** dropdown. You can monitor the running status by selecting the **Run ID** and clicking on **View run details**. You should see something similar to the following screenshot:

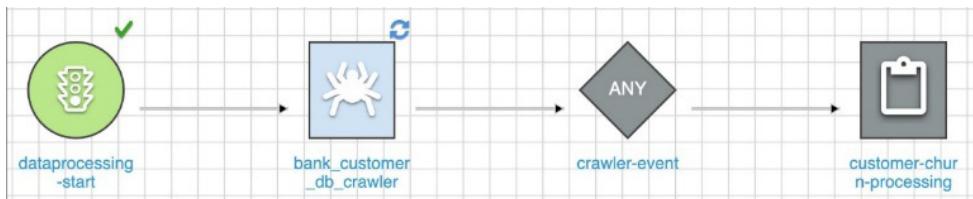


Figure 45.9: Glue workflow execution

9. Try deleting the `customer_data` and `churn_list` tables and re-run the workflow. See whether the new tables are created again. Check the `s3://MLSA-DataLake-Serving-<your initials>/ml-customer-churn/<date>/` S3 location to verify a new folder is created with a new dataset.

Congratulations! You have completed the hands-on lab and learned how to build a simple data lake and its supporting components to allow data cataloging, data querying, and data processing. You should now be able to apply some of the skills learned to real-world design and the implementation of a data management platform on AWS to support the ML development lifecycle.

Chapter

# 46

# Exploring the Best ML Libraries

There is a wide range of **machine learning (ML)** and data science technologies available, encompassing both open-source and commercial products. Different organizations have adopted different approaches when it comes to building their ML platforms. Some have opted for in-house teams that leverage open-source technology stacks, allowing for greater flexibility and customization. Others have chosen commercial products to focus on addressing specific business and data challenges. Additionally, some organizations have adopted a hybrid architecture, combining open-source and commercial tools to harness the benefits of both. As a practitioner in ML solutions architecture, it is crucial to be knowledgeable about the available open-source ML technologies and their applications in building robust ML solutions.

## Core features of open-source ML libraries

ML libraries are software libraries designed to facilitate the implementation of ML algorithms and techniques. While they share similarities with other software libraries, what sets them apart is their specialized support for various ML functionalities. These libraries typically offer a range of features through different sub-packages, including:

- **Data manipulation and processing:** This includes support for different data tasks such as loading data of different formats, data manipulation, data analysis, data visualization, data transformation, and feature extraction.
- **Model building and training:** This includes support for built-in ML algorithms as well as capabilities for building custom algorithms for a wide range of ML tasks. Most ML libraries also have built-in support for the commonly used loss functions (such as mean squared error or cross-entropy) and a list of optimizers (such as gradient descent, Adam, etc.) to choose from. Some libraries also provide advanced support for distributed model training across multiple CPU/GPU devices or compute nodes.
- **Model evaluation and validation:** This includes packages for evaluating the performance of trained models, such as model accuracy, precision, recall, or error rates.

- **Model saving and loading:** This includes support for saving the models to various formats for persistence and support for loading saved models into memory for predictions.
- **Model serving:** This includes model serving features to expose trained ML models behind an API, usually a RESTful API web service.
- **Interpretation:** This includes functionality for interpreting model predictions and feature importance.

ML libraries typically offer support for multiple programming languages, including popular options such as Python, Java, and Scala, catering to diverse user requirements. Python, in particular, has emerged as a prominent language in the field of ML, and many libraries provide extensive support for its interface. While the user-facing interface is often implemented in Python, the backend and underlying algorithms of these libraries are primarily written in compiled languages like C++ and Cython. This combination allows for efficient and optimized performance during model training and inference.

## Understanding the scikit-learn ML library

scikit-learn (<https://scikit-learn.org/>) is an open-source ML library for Python. Initially released in 2007, it is one of the most popular ML libraries for solving many ML tasks, such as classification, regression, clustering, and dimensionality reduction. scikit-learn is widely used by companies in different industries and academics for solving real-world business cases such as churn prediction, customer segmentation, recommendations, and fraud detection.

scikit-learn is built mainly on top of three foundational libraries: **NumPy**, **SciPy**, and **Matplotlib**:

- NumPy is a Python-based library for managing large, multidimensional arrays and matrices, with additional mathematical functions to operate on the arrays and matrices.
- SciPy provides scientific computing functionality, such as optimization, linear algebra, and Fourier transform.
- Matplotlib is used for plotting data for data visualization.

In all, scikit-learn is a sufficient and effective tool for a range of common data processing and model-building tasks.

## Core components of scikit-learn

The scikit-learn library provides a wide range of Python classes and functionalities for the various stages of the ML lifecycle. It consists of several main components, as depicted in the following diagram. By utilizing these components, you can construct ML pipelines and perform tasks such as classification, regression, and clustering.

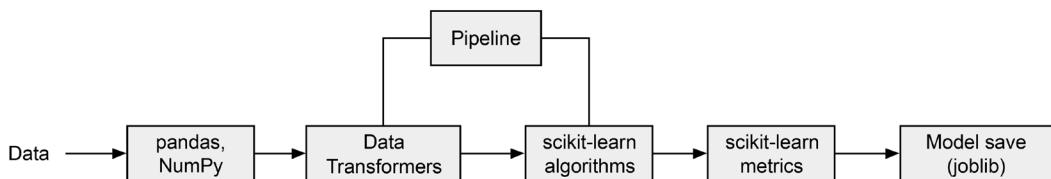


Figure 46.1: scikit-learn components

Now, let's delve deeper into how these components support the different stages of the ML lifecycle:

- **Preparing data:** For data manipulation and processing, the pandas library is commonly used. It provides core data loading and saving functions, as well as utilities for data manipulations such as data selection, data arrangement, and data statistical summaries. pandas is built on top of NumPy. The pandas library also comes with some visualization features such as pie charts, scatter plots, and box plots.

scikit-learn provides a list of transformers for data processing and transformation, such as imputing missing values, encoding categorical values, normalization, and feature extraction for text and images. You can find the full list of transformers at [https://scikit-learn.org/stable/data\\_transforms.html](https://scikit-learn.org/stable/data_transforms.html). Furthermore, you have the flexibility to create custom transformers.

- **Model training:** scikit-learn provides a long list of ML algorithms (also known as estimators) for classification and regression (for example, logistic regression, k-nearest neighbors, and random forest), as well as clustering (for example, k-means). You can find the full list of algorithms at <https://scikit-learn.org/stable/index.html>. The following sample code shows the syntax for using the `RandomForestClassifier` algorithm to train a model using a labeled training dataset:

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier (
```

```
 max_depth, max_features, n_estimators
)
model.fit(train_X, train_y)
```

- **Model evaluation:** scikit-learn has utilities for hyperparameter tuning and cross-validation, as well as `metrics` classes for model evaluations. You can find the full list of model selection and evaluation utilities at [https://scikit-learn.org/stable/model\\_selection.html](https://scikit-learn.org/stable/model_selection.html). The following sample code shows the `accuracy_score` class for evaluating the accuracy of classification models:

```
from sklearn.metrics import accuracy_score
acc = accuracy_score(true_label, predicted_label)
```



**Hyperparameter tuning** involves optimizing the configuration settings (hyperparameters) of an ML model to enhance its performance and achieve better results on a given task or dataset. Cross-validation is a statistical technique used to assess the performance and generalizability of an ML model by dividing the dataset into multiple subsets, training the model on different combinations, and evaluating its performance across each subset.

- **Model saving:** scikit-learn can save model artifacts using Python object serialization (`pickle` or `joblib`). The serialized `pickle` file can be loaded into memory for predictions. The following sample code shows the syntax for saving a model using the `joblib` class:

```
import joblib
joblib.dump(model, "saved_model_name.joblib")
```

- **Pipeline:** scikit-learn also provides a pipeline utility for stringing together different transformers and estimators as a single processing pipeline, and it can be reused as a single unit. This is especially useful when you need to preprocess data for modeling training and model prediction, as both require the data to be processed in the same way:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
pipe = Pipeline([('scaler', StandardScaler()), ('RF',
RandomForestClassifier())])
pipe.fit(X_train, y_train)
```

As demonstrated, getting started with scikit-learn for experimenting with and constructing ML models is straightforward. scikit-learn is particularly suitable for typical regression, classification, and clustering tasks performed on a single machine. However, if you're working with extensive datasets or require distributed training across multiple machines, scikit-learn may not be the optimal choice unless the algorithm supports incremental training, such as `SGDRegressor`. Therefore, moving on, let's explore alternative ML libraries that excel in large-scale model training scenarios.



**Incremental training** is an ML approach where a model is updated and refined continuously as new data becomes available, allowing the model to adapt to evolving patterns and improve its performance over time.

## Understanding the Apache Spark ML library

Apache Spark is an advanced framework for distributed data processing, designed to handle large-scale data processing tasks. With its distributed computing capabilities, Spark enables applications to efficiently load and process data across a cluster of machines by leveraging in-memory computing, thereby significantly reducing processing times.

Architecturally, a Spark cluster consists of a master node and worker nodes for running different Spark applications. Each application that runs in a Spark cluster has a driver program and its own set of processes, which are coordinated by the `SparkSession` object in the driver program. The `SparkSession` object in the driver program connects to a cluster manager (for example, Mesos, Yarn, Kubernetes, or Spark's standalone cluster manager), which is responsible for allocating resources in the cluster for the Spark application. Specifically, the cluster manager acquires resources on worker nodes called `executors` to run computations and store data for the Spark application. Executors are configured with resources such as the number of CPU cores and memory to meet task processing needs. Once the executors have been allocated, the cluster manager sends the application code (Java JAR or Python files) to the executors. Finally, `SparkContext` sends the tasks to the executors to run. The following diagram shows how a driver program interacts with a cluster manager and executor to run a task:

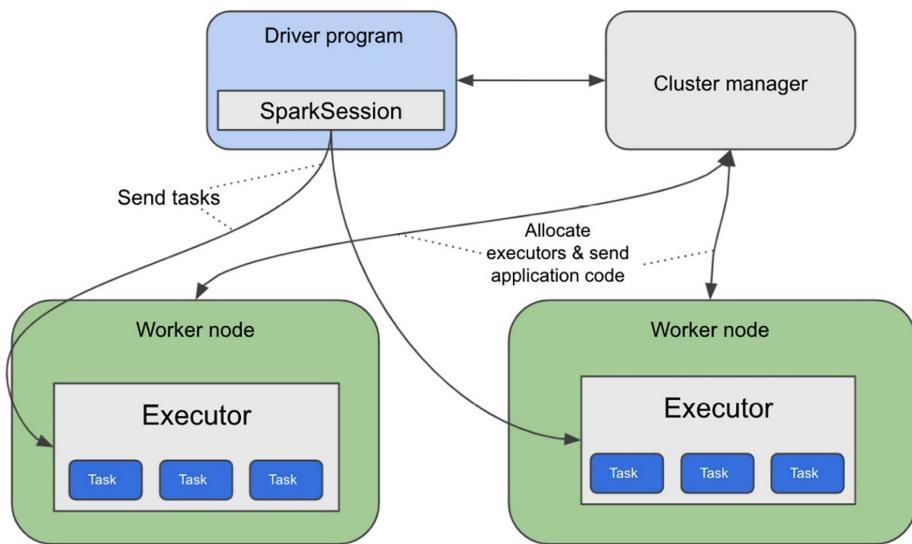


Figure 46.2: Running a Spark application on a Spark cluster

Each Spark application gets its own set of executors, which remain active for the duration of the application. The executors for different applications are isolated from each other, and they can only share data through external data storage.

The ML package for Spark is called MLLib, which runs on top of the distributed Spark architecture. It is capable of processing and training models with a large dataset that does not fit into the memory of a single machine. It provides APIs in different programming languages, including Python, Java, Scala, and R. From a structural perspective, it is very similar to that of the scikit-learn library in terms of core components and model development flow.

Spark is highly popular and adopted by companies of all sizes across different industries. Large companies such as **Netflix**, **Uber**, and **Pinterest** use Spark for large-scale data processing and transformation, as well as running ML models.

## Installing Spark ML

Spark ML libraries are included as part of the Spark installation. PySpark is the Python API for Spark, and it can be installed like a regular Python package using pip (`pip install pyspark`). Note that PySpark requires Java and Python to be installed on the machine before it can be installed. You can find Spark's installation instructions at <https://spark.apache.org/docs/latest/>.

## Core components of the Spark ML library

Similar to the scikit-learn library, Spark and Spark ML provide a full range of functionality for building ML models, from data preparation to model evaluation and model persistence. The following diagram shows the core components that are available in Spark for building ML models:

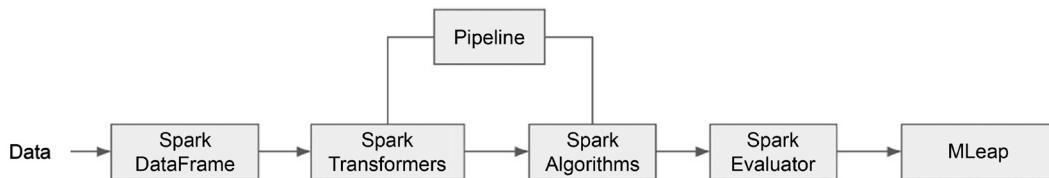


Figure 46.3: Core components of Spark ML

Let's take a closer look at the core functions supported by the Spark and Spark ML library packages:

- **Preparing data:** Spark supports Spark DataFrames, distributed data collections that can be used for tasks such as data joining, aggregation, filtering, and other data manipulation needs. Conceptually, a Spark DataFrame is equivalent to a table in a relational database. A Spark DataFrame can be distributed (that is, partitioned) across many machines, which allows fast data processing in parallel. Spark DataFrames also operate on a model called the lazy execution model. **Lazy execution** defines a set of transformations (for example, adding a column or filtering column) and the transformations are only executed when an action (such as calculating the min/max of a column) is needed. This allows an execution plan for the different transformations and actions to be generated to optimize the execution's performance.

To start using the Spark functionality, you need to create a Spark session. A Spark session creates a `SparkContext` object, which is the entry point to the Spark functionality. The following sample code shows how you can create a Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('appname').getOrCreate()
```

A Spark DataFrame can be constructed from many different sources, such as structured data files (for example, CSV or JSON) and external databases. The following code sample reads a CSV file into a Spark DataFrame:

```
dataFrame = spark.read.format('csv').load(file_path)
```

There are many transformers for data processing and transformation in Spark based on different data processing needs, such as `Tokenizer` (which breaks text down into individual words) and `StandardScalar` (which normalizes a feature into unit deviation and/or zero mean). You can find a list of supported transformers at <https://spark.apache.org/docs/2.1.0/ml-features.html>.

To use a transformer, first, you must initiate it with function parameters, such as `inputCol` and `outputCol`, then call the `fit()` function on the DataFrame that contains the data, and finally, call the `transform()` function to transfer the features in the DataFrame:

```
from pyspark.ml.feature import StandardScaler
scaler = StandardScaler(inputCol="features",
 outputCol="scaledFeatures", withStd=True, withMean=False)
scalerModel = scaler.fit(dataFrame)
scaledData = scalerModel.transform(dataFrame)
```

- **Model training:** Spark ML supports a wide range of ML algorithms for classification, regression, clustering, recommendation, and topic modeling. You can find a list of Spark ML algorithms at <https://spark.apache.org/docs/1.4.1/mllib-guide.html>. The following code sample shows how you can train a logistic regression model:

```
from pyspark.ml.classification import LogisticRegression
lr_algo = LogisticRegression(
 maxIter=regParam, elasticNetParam=elasticNetParam
)
lr_model = lr_algo.fit(dataFrame)
```

- **Model evaluation:** For model selection and evaluation, Spark ML provides utilities for cross-validation, hyperparameter tuning, and model evaluation metrics. You can find the list of evaluators at <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.MulticlassClassificationEvaluator.html>. The following code block illustrates using the `BinaryClassificationEvaluator` to evaluate a model with the `areaUnderPR` metric:

```
From pyspark.ml.evaluation import BinaryClassificationEvaluator
dataset = spark.createDataFrame(scoreAndLabels, ["raw", "label"])
evaluator = BinaryClassificationEvaluator()
evaluator.setRawPredictionCol("raw")
evaluator.evaluate(dataset)
evaluator.evaluate(dataset, {evaluator.metricName: "areaUnderPR"})
```

- **Pipeline:** Spark ML also has support for the pipeline concept, similar to that of scikit-learn. With the pipeline concept, you can sequence a series of transformation and model training steps as a unified repeatable step:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
lr_tokenizer = Tokenizer(inputCol, outputCol)
lr_hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol)
lr_algo = LogisticRegression(maxIter, regParam)
lr_pipeline = Pipeline(stages=[lr_tokenizer, lr_hashingTF, lr_algo])
lr_model = lr_pipeline.fit(training)
```

- **Model saving:** The Spark ML pipeline can be serialized into a serialization format called an Mleap bundle, which is an external library from Spark. A serialized Mleap bundle can be deserialized back into Spark for batch scoring or an Mleap runtime to run real-time APIs. You can find more details about Mleap at <https://combust.github.io/mleap-docs/>. The following code shows the syntax for serializing a Spark model into the Mleap format:

```
import mleap.pyspark
from pyspark.ml import Pipeline, PipelineModel
lr_model.serializeToBundle("saved_file_path", lr_model.
transform(dataframe))
```

Spark is a versatile framework that enables large-scale data processing and ML. While it excels in traditional ML tasks, it also offers limited support for neural network training, including the multilayer perceptron algorithm. However, for more comprehensive deep learning capabilities, we will explore dedicated ML libraries including TensorFlow and PyTorch in the upcoming sections.

# Understanding the TensorFlow deep learning library

Initially released in 2015, TensorFlow is a popular open-source ML library, primarily backed up by Google, which is mainly designed for deep learning. TensorFlow has been used by companies of all sizes for training and building state-of-the-art deep learning models for a range of use cases, including computer vision, speech recognition, question-answering, text summarization, forecasting, and robotics.

TensorFlow works based on the concept of the computational graph, where data flows through nodes that represent mathematical operations. The core idea is to construct a graph of operations and tensors, with tensors being  $n$ -dimensional arrays that carry data. An example of a tensor could be a scalar value (for example, 1.0), a one-dimensional vector (for example, [1.0, 2.0, 3.0]), a two-dimensional matrix (for example, [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), or even higher dimensional matrices. Operations are performed on these tensors, allowing for mathematical computations like addition or matrix multiplication. The following diagram shows a sample computational graph for performing a sequence of mathematical operations on tensors:

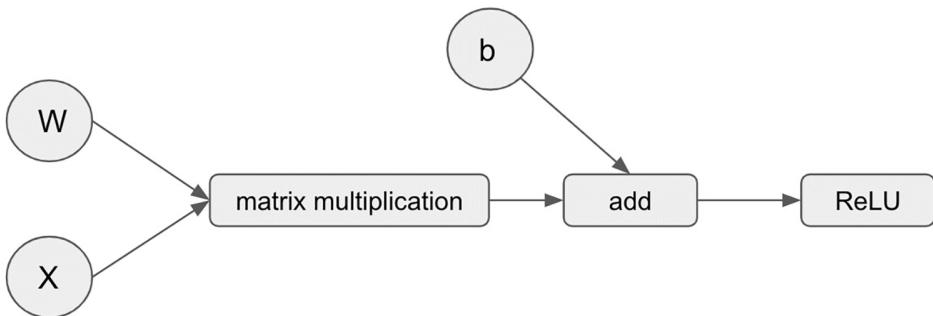


Figure 46.4: Data flow diagram

In the preceding computational diagram, the rectangular nodes are mathematical operations, while the circles represent tensors. This particular diagram shows a computational graph for performing an artificial neuron tensor operation, which is to perform a matrix multiplication of  $W$  and  $X$ , followed by the addition of  $b$ , and, lastly, apply a *ReLU* action function. The equivalent mathematical formula is as follows:

$$f(x) = \text{ReLU}(W_x + b)$$

TensorFlow allows users to define and manipulate the computation graph using its high-level API or by directly working with its lower-level components. This flexibility allows researchers and developers to create complex models and algorithms. Furthermore, TensorFlow supports distributed computing, allowing the graph to be

executed across multiple devices or machines, which is crucial for handling large-scale ML tasks. This distributed architecture enables TensorFlow to leverage the power of clusters or GPUs to accelerate the training and inference of deep learning models.

## Core components of TensorFlow

The TensorFlow library provides a rich set of features for different ML steps, from data preparation to model serving. The following diagram illustrates the core building blocks of the TensorFlow library:

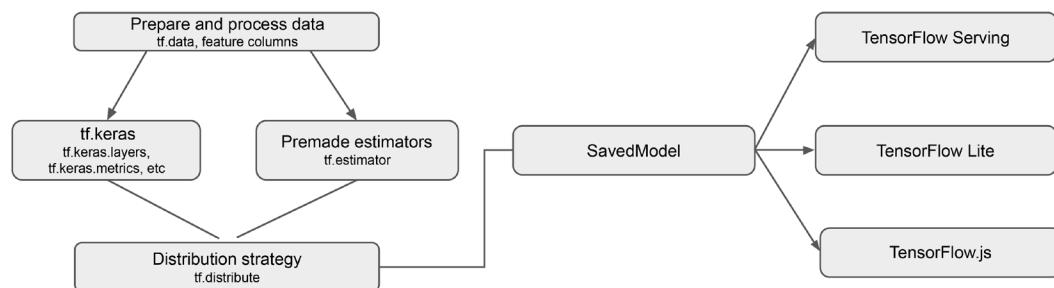


Figure 46.5: TensorFlow components

Training an ML model using TensorFlow 2.x involves the following main steps:

- Preparing the dataset:** TensorFlow 2.x provides a `tf.data` library for efficiently loading data from sources (such as files), transforming data (such as changing the values of the dataset), and setting up the dataset for training (such as configuring batch size or data prefetching). These data classes provide efficient ways to pass data to the training algorithms for optimized model training. The TensorFlow **Keras** API also provides a list of built-in classes (MNIST, CIFAR, IMDB, MNIST Fashion, and Reuters Newswire) for building simple deep learning models. You can also feed a NumPy array or Python generator (a function that behaves like an iterator) to a model in TensorFlow for model training, but `tf.data` is the recommended approach.
- Defining the neural network:** TensorFlow 2.x provides multiple ways to use or build a neural network for model training. You can use the premade estimators (the `tf.estimator` class) such as `DNNRegressor` and `DNNClassifier` to train models. Or, you can create custom neural networks using the `tf.keras` class, which provides a list of primitives such as `tf.keras.layers` for constructing neural network layers and `tf.keras.activation` such as `ReLU`, **Sigmoid**, and **Softmax** for building neural networks. Softmax is usually used as the last output of a neural network for a multiclass problem. It takes a vector

of real numbers (positive and negative) as input and normalizes the vector as a probability distribution to represent the probabilities of different class labels, such as the different types of hand-written digits. For binary classification problems, Sigmoid is normally used and it returns a value between 0 and 1.

3. **Defining the loss function:** TensorFlow 2.x provides a list of built-in loss functions such as **mean squared error (MSE)** and **mean absolute error (MAE)** for regression tasks and cross-entropy loss for classification tasks. You can find more details about MSE and MAE at [https://en.wikipedia.org/wiki/Mean\\_squared\\_error](https://en.wikipedia.org/wiki/Mean_squared_error) and [https://en.wikipedia.org/wiki/Mean\\_absolute\\_error](https://en.wikipedia.org/wiki/Mean_absolute_error). You can find a list of supported loss functions in the `tf.keras.losses` class. For more details about the different losses, refer to <https://keras.io/api/losses/>. There is also the flexibility to define custom loss functions if the built-in loss functions do not meet the needs.
4. **Selecting the optimizer:** TensorFlow 2.x provides a list of built-in optimizers for model training, such as the **Adam** optimizer and the **stochastic gradient descent (SGD)** optimizer for parameters optimization, with its `tf.keras.optimizers` class. You can find more details about the different supported optimizers at <https://keras.io/api/optimizers/>. Adam and SGD are two of the most commonly used optimizers.
5. **Selecting the evaluation metrics:** TensorFlow 2.x has a list of built-in model evaluation metrics (for example, accuracy and cross-entropy) for model training evaluations with its `tf.keras.metrics` class. You can also define custom metrics for model evaluation during training.
6. **Compiling the network into a model:** This step compiles the defined network, along with the defined loss function, optimizer, and evaluation metrics, into a computational graph that's ready for model training.
7. **Fitting the model:** This step kicks off the model training process by feeding the data to the computational graph through batches and multiple epochs to optimize the model parameters.
8. **Evaluating the trained model:** Once the model has been trained, you can evaluate the model using the `evaluate()` function against the test data.
9. **Saving the model:** The model can be saved in the TensorFlow **SavedModel** serialization format or **Hierarchical Data Format (HDF5)** format.
10. **Model serving:** TensorFlow comes with a model serving framework called TensorFlow Serving.

The TensorFlow library is designed for large-scale production-grade data processing and model training. As such, it provides capabilities for large-scale distributed data processing and model training on a cluster of servers against a large dataset.

To support the complete process of building and deploying ML pipelines, TensorFlow provides **TensorFlow Extended (TFX)**. TFX integrates multiple components and libraries from the TensorFlow ecosystem, creating a cohesive platform for tasks such as data ingestion, data validation, preprocessing, model training, model evaluation, and model deployment. Its architecture is designed to be modular and scalable, enabling users to tailor and expand the pipeline to meet their specific requirements. You can get more details about TFX at <https://www.tensorflow.org/tfx>.

TensorFlow offers an expanded ecosystem of libraries and extensions for solving a wide range of advanced ML problems, including federated learning (training a model using decentralized data), model optimization (optimizing a model for deployment and execution), and probabilistic reasoning (reasoning under uncertainty using probability theory). It also provides support for mobile and edge devices with this TensorFlow Lite component, and support for browsers through the TensorFlow.js library.

## Understanding the PyTorch deep learning library

PyTorch is an open-source ML library that was designed for deep learning using GPUs and CPUs. Initially released in 2016, it is a highly popular ML framework with a large following and many adoptions. Many technology companies, including tech giants such as **Facebook**, **Microsoft**, and **Airbnb**, all use PyTorch heavily for a wide range of deep learning use cases, such as computer vision and **natural language processing (NLP)**.

PyTorch strikes a good balance of performance (using a C++ backend) with ease of use with default support for dynamic computational graphs and interoperability with the rest of the Python ecosystem. For example, with PyTorch, you can easily convert between NumPy arrays and PyTorch tensors. To allow for easy backward propagation, PyTorch has built-in support for automatically computing gradients, a vital requirement for gradient-based model optimization.

The PyTorch library consists of several key modules, including tensors, **autograd**, **optimizer**, and **neural network**. Tensors are used to store and operate multidimensional arrays of numbers. You can perform various operations on tensors such as matrix multiplication, transpose, returning the max number, and dimensionality manipulation. PyTorch supports automatic gradient calculation with its Autograd module. When

performing a forward pass, the Autograd module simultaneously builds up a function that computes the gradient. The Optimizer module provides various algorithms such as SGD and Adam for updating model parameters. The Neural Network module provides modules that represent different layers of a neural network such as the linear layer, embedding layer, and dropout layer. It also provides a list of loss functions that are commonly used for training deep learning models.

## Core components of PyTorch

Similar to TensorFlow, PyTorch also supports the end-to-end ML workflow, from data preparation to model serving. The following diagram shows what different PyTorch modules are used to train and serve a PyTorch model:

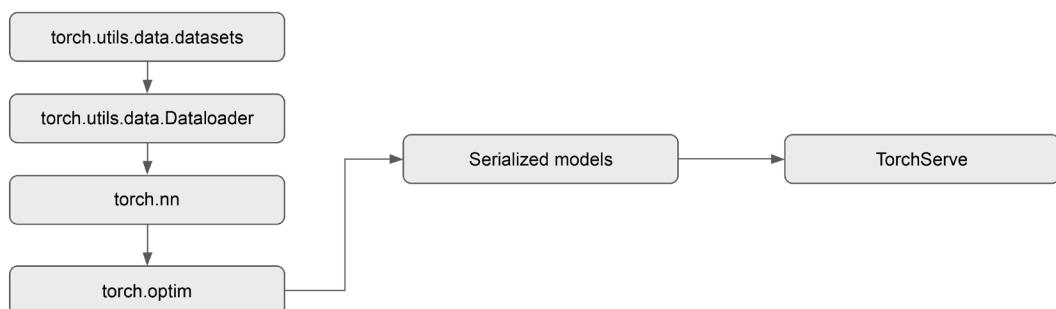


Figure 46.6: PyTorch modules for model training and serving

The steps involved in training a deep learning model are very similar to that of TensorFlow model training. We'll look at the PyTorch-specific details in the following steps:

1. **Preparing the dataset:** PyTorch provides two primitives for dataset and data loading management: `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`. `Dataset` stores data samples and their corresponding labels, while `DataLoader` wraps around the dataset and provides easy and efficient access to the data for model training. `DataLoader` provides functions such as `shuffle`, `batch_size`, and `prefetch_factor` to control how the data is loaded and fed to the training algorithm. Additionally, as the data in the dataset might need to be transformed before training is performed, `Dataset` allows you to use a user-defined function to transform the data.
2. **Defining the neural network:** PyTorch provides a high-level abstraction for building neural networks with its `torch.nn` class, which provides built-in support for different neural network layers such as linear layers and convolutional layers, as well as activation layers such as Sigmoid and ReLU. It

also has container classes such as `nn.Sequential` for packaging different layers into a complete network. Existing neural networks can also be loaded into PyTorch for training.

3. **Defining the loss function:** PyTorch provides several built-in loss functions in its `torch.nn` class, such as `nn.MSELoss` and `nn.CrossEntropyLoss`.
4. **Selecting the optimizer:** PyTorch provides several optimizers with its `nn.optim` classes. Examples of optimizers include `optim.SGD`, `optim.Adam`, and `optim.RMSProp`. All the optimizers have a `step()` function that updates model parameters with each forward pass. There's also a backward pass that calculates the gradients.
5. **Selecting the evaluation metrics:** The PyTorch `ignite.metrics` class provides several evaluation metrics such as precision, recall, and `RootMeanSquaredError` for evaluating model performances. You can learn more about precision and recall at [https://en.wikipedia.org/wiki/Precision\\_and\\_recall](https://en.wikipedia.org/wiki/Precision_and_recall). You can also use the scikit-learn metrics libraries to help evaluate models.
6. **Training the model:** Training a model in PyTorch involves three main steps in each training loop: forward pass the training data, backward pass the training data to calculate the gradient, and perform the optimizer step to update the gradient.
7. **Saving/loading the model:** The `torch.save()` function saves a model in a serialized pickle format. The `torch.load()` function loads a serialized model into memory for inference. A common convention is to save the files with the `.pth` or `.pt` extension. You can also save multiple models into a single file.
8. **Model serving:** PyTorch comes with a model serving library called TorchServe.

The PyTorch library supports large-scale distributed data processing and model training. Like TensorFlow, PyTorch also offers an ecosystem of library packages for a wide range of ML problems, including ML privacy, adversarial robustness, video understanding, and drug discovery.

## How to choose between TensorFlow and PyTorch

TensorFlow and PyTorch are the two most popular frameworks in the domain of deep learning. So, a pertinent question arises: How does one make an informed choice between the two? To help answer this question, let's do a quick comparative analysis of these frameworks:

- **Ease of use:** PyTorch is generally considered more user-friendly and Pythonic. The control flow feels closer to native Python and the dynamic computation graphs of PyTorch make it easier to debug and iterate compared to TensorFlow's static graphs. However, eager execution support in TensorFlow 2.0 helps close this gap. PyTorch is also considered more object-oriented than TensorFlow.
- **Community popularity:** Both frameworks enjoy robust community support and are highly popular. TensorFlow initially had a lead; however, PyTorch has caught up in popularity in recent years, according to the Google Trends report. PyTorch is more widely adopted within the research community and dominates the implementation of research papers.
- **Model availability:** TensorFlow has the TensorFlow Model Garden, which hosts a collection of models utilizing TensorFlow APIs, covering various ML tasks such as computer vision, NLP, and recommendation. It also features TensorFlow Hub, offering a collection of pre-trained models ready for deployment or fine-tuning across a wide range of ML tasks. Similarly, PyTorch has PyTorch Hub, a library integrated into PyTorch that provides easy access to a broad array of pre-trained models for computer vision, NLP, and more.
- **Deployment:** Both frameworks are suitable for the production deployment of ML models. TensorFlow is considered to have a more comprehensive model deployment stack with TensorFlow Serving, TensorFlow Lite for mobile and edge devices, and TensorFlow.js for browser deployment. TensorFlow Extended is an end-to-end model deployment platform encompassing model validation, monitoring, and explanation. PyTorch offers TorchServe, a model-serving framework for PyTorch models, and PyTorch Mobile for deploying models on iOS and Android devices. PyTorch relies more on third-party solutions for end-to-end integration in the deployment process.

To conclude, both frameworks provide comparable capabilities throughout the entire ML lifecycle, accommodating similar use cases. If your organization has already committed to either TensorFlow or PyTorch, it is advisable to proceed with that decision. However, for those embarking on the initial stages, PyTorch might offer a more accessible starting point owing to its ease of use.

\*\*\* END OF STUDY BOOK 3 \*\*\*

Chapter

# 47

# The Power of Open-Source ML Platforms

In the previous chapter, we covered how Kubernetes can be used as the foundational infrastructure for running ML tasks, such as running model training jobs or building data science environments such as **Jupyter Notebook** servers. However, to perform these tasks at scale and more efficiently for large organizations, you will need to build ML platforms with the capabilities to support the full data science lifecycle. These capabilities include scalable data science environments, model training services, model registries, and model deployment capabilities.

## Core components of an ML platform

An ML platform is a complex system encompassing multiple environments for running distinct tasks and orchestrating complex workflow processes. Furthermore, an ML platform needs to cater to a multitude of roles, including data scientists, ML engineers, infrastructure engineers, operations teams, and security and compliance stakeholders. To construct an ML platform, several components come into play.

These components include:

- **Data science environment:** The data science environment provides data analysis and ML tools, such as Jupyter notebooks, data sources and storage, code repositories, and ML frameworks. Data scientists and ML engineers use the data science environment to perform data analysis, run data science experiments, and build and tune models. The data science environment also provides collaboration capabilities, allowing data scientists to share and collaborate on code, data, experiments, and models.
- **Model training environment:** The model training environment provides a separate infrastructure tailored to meet specific model training requirements. While data scientists and ML engineers can execute small-scale model training tasks directly within their local Jupyter environment, they need a separate dedicated infrastructure for large-scale model training. By utilizing a dedicated training infrastructure, organizations can exercise greater control over model training process management and model lineage management processes.

- **Model registry:** Trained models need to be tracked and managed within a model registry. The model registry serves as a centralized repository for inventorying and managing models, ensuring effective lineage management, version control, model discovery, and comprehensive lifecycle management. This becomes particularly significant when dealing with a large number of models. Data scientists can register models directly in the registry as they perform experiments in their data science environment. Additionally, models can be registered as part of automated ML model pipeline executions, enabling streamlined and automated integration of models into the registry.
- **Model serving environment:** To serve predictions from trained ML models to client applications, it is necessary to host the models within a dedicated model serving infrastructure that operates behind an API endpoint in real time. This infrastructure should also provide support for batch transform capabilities, allowing predictions to be processed in large batches. Several types of model serving frameworks are available to fulfill these requirements.
- **ML pipeline development:** To effectively manage the various ML components and stages in the lifecycle, it is crucial to incorporate capabilities that enable pipeline development to orchestrate ML training and prediction workflows. These pipelines play an important role in coordinating different stages, such as data preparation, model training, and evaluation.
- **Model monitoring:** Robust model monitoring is crucial for maintaining the high performance of ML models in production. Continuous monitoring tracks metrics like prediction accuracy, data drift, latency, errors, and anomalies over time. Monitoring enables platform operators to detect production model degradation before it impacts users. When monitored metrics cross defined thresholds, alerts trigger investigative workflows and mitigation where needed. Effective monitoring also provides performance dashboards and visibility into all deployed models. This facilitates continuous improvement of models and allows replacing underperforming models proactively.
- **ML feature management:** Managing features is a key capability in the ML lifecycle. Feature management entails the ongoing curation, monitoring, and sharing of ML features to accelerate model development. This includes tools for discovery, lineage tracking, and governance of feature data. Centralized feature stores democratize access to high-quality features by teams across an organization. They provide a single source of truth, eliminating duplication of feature engineering efforts.
- **Continuous integration (CI)/continuous deployment (CD) and workflow automation:** Finally, to streamline the data processing, model training, and model deployment processes on an ML platform, it is crucial to establish CI/CD practices, along with workflow automation capabilities. These practices and tools significantly contribute to increasing the velocity, consistency, reproducibility, and observability of ML deployments.

In addition to these core components, there are several other platform architecture factors to consider when building an end-to-end ML platform. These factors include security and authentication, version control and reproducibility, and data management and governance. By integrating these additional architectural factors into the ML platform, organizations can enhance security, gain visibility into system operations, and enforce governance policies. In the following sections, we will explore various open-source technologies that can be used to build an end-to-end ML platform.

## Open-source technologies for building ML platforms

Managing ML tasks individually by deploying standalone ML containers in a Kubernetes cluster can become challenging when dealing with a large number of users and workloads. To address this complexity and enable efficient scaling, many open-source technologies have emerged as viable solutions. These technologies, including Kubeflow, MLflow, Seldon Core, GitHub, Feast, and Airflow, provide comprehensive support for building data science environments, model training services, model inference services, and ML workflow automation.

Before delving into the technical details, let's first explore why numerous organizations opt for open-source technologies to construct their ML platforms. For many, the appeal lies in the ability to tailor the platform to specific organizational needs and workflows, with open standards and interoperable components preventing vendor lock-in and allowing the flexibility to adopt new technologies over time. Leveraging popular open-source ML projects also taps into a rich talent pool, as many practitioners are already proficient with these technologies. Additionally, open-source allows complete control over the platform roadmap to internal teams, reducing dependence on a vendor's priorities. When executed efficiently, an open-source stack can lead to cost savings for organizations, as there are no licensing costs associated with the software.

Building an ML platform using open-source technology comes with notable advantages. However, it's also crucial to consider the potential drawbacks. Challenges may arise from integration complexities, a lack of comprehensive support, security vulnerabilities, and potential limitations in features compared to commercial solutions. Additionally, the resource-intensive nature of maintaining an open-source platform, coupled with a potential learning curve for the team, could impact efficiency and total cost of ownership. Concerns about documentation quality, the absence of standardization, and the responsibility for updates and maintenance further underscore the need for careful consideration. You must weigh these factors against the benefits, taking into account their specific requirements, resources, and expertise before opting for an open-source approach.

With these considerations in mind, let's explore the design of core ML platform components using open-source technologies.

## Implementing a data science environment

Kubeflow is an open-source ML platform built on top of Kubernetes. It offers a set of tools and frameworks specifically designed to simplify the deployment, orchestration, and management of ML workloads. Kubeflow provides features like Jupyter notebooks for interactive data exploration and experimentation, distributed training capabilities, and model serving infrastructure.

Core capabilities of Kubeflow include:

- A central UI dashboard
- A Jupyter Notebook server for code authoring and model building
- A Kubeflow pipeline for ML pipeline orchestration
- **KFServing** for model serving
- Training operators for model training support

The following figure illustrates how Kubeflow can provide the various components needed for a data science environment. Specifically, we will delve into its support for Jupyter Notebook servers as it is the main building block for a data science environment.

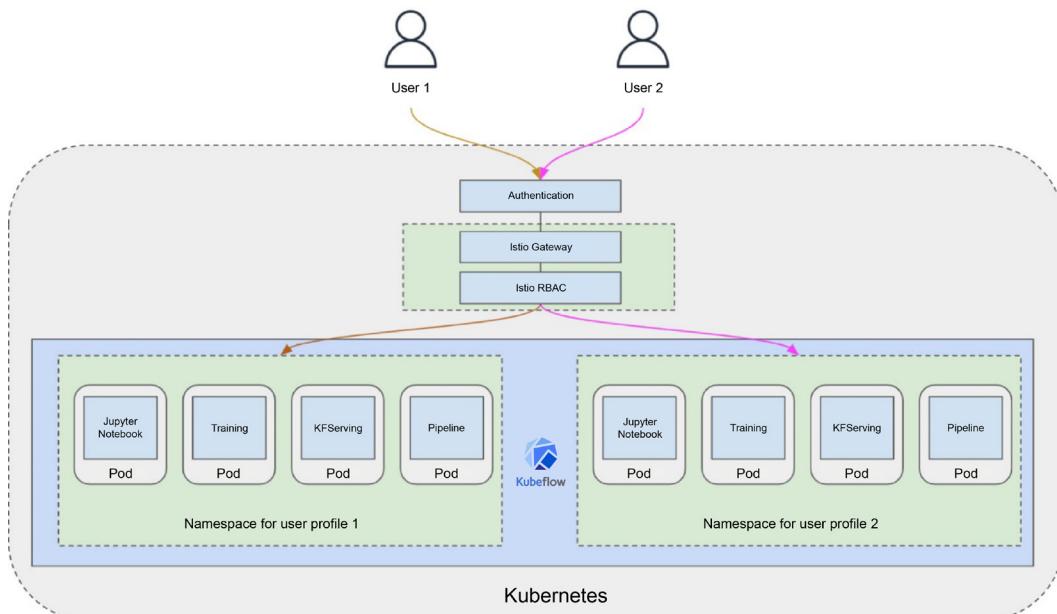


Figure 47.1: A Kubeflow-based data science environment

Kubeflow provides a multi-tenant Jupyter Notebook server environment with built-in authentication and authorization support. Let's discuss each of these core components in detail:

- **Jupyter Notebook:** As a data scientist, you can take advantage of the Kubeflow Jupyter Notebook server, which offers a platform to author and run your **Python** code to explore data and build models inside the Jupyter notebook. With Kubeflow, you can spawn multiple notebook servers, each server associated with a single Kubernetes namespace that corresponds to a team, project, or individual user. Each notebook server runs a container inside a **Kubernetes Pod**. By default, a Kubeflow notebook server provides a list of notebook container images hosted in public container image repositories to choose from. Alternatively, you can create custom notebook container images to tailor to your specific requirements. To ensure standards and consistency, Kubeflow administrators can provide a list of standard images for users to use. When creating a notebook server, you select the namespace to run the notebook server in. Additionally, you specify the **Universal Resource Identifier (URI)** of the container image for the notebook server. You also have the flexibility to specify the resource requirements, such as the number of CPUs/GPUs and memory size.
- **Authentication and authorization:** You access the notebook server through the Kubeflow UI dashboard, which provides an authentication service through the **Dex OpenID Connect (OIDC)** provider. Dex is an identity service that uses OIDC to provide authentication for other applications. Dex can federate with other authentication services such as the **Active Directory** service. Each notebook is associated with a default Kubernetes service account (`default-editor`) that can be used for entitlement purposes (such as granting the notebook permission to access various resources in the Kubernetes cluster). Kubeflow uses Istio **role-based access control (RBAC)** to control in-cluster traffic. The following **YAML** file grants the `default-editor` service account (which is associated with the Kubeflow notebook) access to the Kubeflow pipeline service by attaching the `ml-pipeline-services` service role to it:

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
 name: bind-ml-pipeline-nb-admin
 namespace: kubeflow
spec:
 roleRef:
 kind: ServiceRole
 name: ml-pipeline-services
```

```
subjects:
- properties:
 source.principal: cluster.local/ns/admin/sa/default-editor
```

- **Multi-tenancy:** Kubeflow offers the capability for multiple users to access a shared Kubeflow environment while ensuring resource isolation. This is achieved by creating individual namespaces for each user and leveraging Kubernetes RBAC and Istio RBAC to manage access control for these namespaces and their associated resources. For collaborative work within teams, the owner of a namespace has the ability to grant access to other users directly from the Kubeflow dashboard UI. Using the Manage Contributor function, the namespace owner can specify which users are granted access to the namespace and its resources.

In addition to the preceding core components, Kubeflow provides a mechanism for onboarding users to access different Kubeflow resources. To onboard a new Kubeflow user, you create a new user profile, which automatically generates a new namespace for the profile.

The following YAML file, once applied using `kubectl`, creates a new user profile called `test-user` with an email of `test-user@kubeflow.org`, and it also creates a new namespace called `test-user`:

```
apiVersion: kubeflow.org/v1beta1
kind: Profile
metadata:
 name: test-user
spec:
 owner:
 kind: User
 name: test-user@kubeflow.org
```

You can run the `kubectl get profiles` and `kubectl get namespaces` commands to verify that the profile and namespaces have been created.

After a user is created and added to the Kubeflow Dex authentication service, the new user can log in to the Kubeflow dashboard and access the Kubeflow resources (such as a Jupyter Notebook server) under the newly created namespace.

Utilizing Kubeflow for a data science environment poses several key challenges that one must be aware of before committing to its implementation. Installing Kubeflow on top of Kubernetes, whether on-premises or in the cloud on platforms like AWS, can be complicated, often requiring substantial configuration and debugging. The many

moving parts make installation non-trivial. Kubeflow consists of many loosely coupled components, each with its own version. Orchestrating these diverse components across different versions to work together seamlessly as an integrated platform can present difficulties. There is a lack of documentation on Kubeflow. Kubeflow documentation often points to older component versions. The out-of-date documentation makes adoption more difficult as new Kubeflow users battle mismatches between docs and platform versions. Despite these limitations, Kubeflow is a highly recommended technology for building ML platforms due to its support for end-to-end pipelines, rich support for different ML frameworks, and portability.

## Building a model training environment

As discussed earlier, within an ML platform, it is common to provide a dedicated model training service and infrastructure to support large-scale and automated model training in an ML pipeline.

This dedicated training service should be easily accessible from different components within the platform, such as the experimentation environment (such as a Jupyter notebook) as well as the ML automation pipeline.

In a Kubernetes-based environment, there are two main approaches for model training you can choose from depending on your training needs:

- Model training using **Kubernetes Jobs**
- Model training using **Kubeflow training operators**

Let's take a closer look at each one of these approaches in detail:

- **Model training using Kubernetes Jobs:** As we discussed earlier, a Kubernetes Job creates one or more containers and runs them through to completion. This pattern is well suited for running certain types of ML model training jobs, as an ML job runs a training loop to completion and does not run forever. For example, you can package a container with a Python training script and all the dependencies that train a model and use the Kubernetes Job to load the container and kick off the training script. When the script completes and exits, the Kubernetes Job also ends. The following sample YAML file kicks off a model training job if submitted with the `kubectl apply` command:

```
apiVersion: batch/v1
kind: Job
metadata:
 name: train-churn-job
spec:
 template:
```

```
spec:
 containers:
 - name: train-container
 imagePullPolicy: Always
 image: <model training uri>
 command: ["python", "train.py"]
 restartPolicy: Never
 backoffLimit: 4
```

To query the status of the job and see the detailed training logs, you can run the `kubectl get jobs` command and the `kubectl logs <pod name>` command, respectively.

- **Model training using Kubeflow training operators:** A Kubernetes Job can launch a model training container and run a training script inside the container to completion. Since the controller for a Kubernetes Job does not have application-specific knowledge about the training job, it can only handle generic Pod deployment and management for the running jobs, such as running the container in a Pod, monitoring the Pod, and handling generic Pod failure. However, some model training jobs, such as distributed training jobs in a cluster, require the special deployment, monitoring, and maintenance of stateful communications among various Pods. This is where the Kubernetes training operator pattern can be applied.

Kubeflow offers a list of pre-built training operators (such as the **TensorFlow**, **PyTorch**, and **XGBoost** operators) for complex model training jobs. Each Kubeflow training operator has a **custom resource (CR)** (for example, `TFJob` CR for TensorFlow jobs) that defines the training job's specific configurations, such as the type of Pod in the training job (for example, `master`, `worker`, or `parameter server`), or runs policies on how to clean up resources and for how long a job should run. The controller for the CR is responsible for configuring the training environment, monitoring the training job's specific status, and maintaining the desired training job's specific state. For instance, the controller can set environment variables to make the training cluster specifications (for example, types of Pods and indices) available to the training code running inside the containers. Additionally, the controller can inspect the exit code of a training process and fail the training job if the exit code indicates a permanent failure. The following YAML file sample template represents a specification for running training jobs using the TensorFlow operator (`tf-operator`):

```
apiVersion: "kubeflow.org/v1"
kind: "TFJob"
```

```
metadata:
 name: "distributed-tensorflow-job"
spec:
 tfReplicaSpecs:
 PS:
 replicas: 1
 restartPolicy: Never
 template:
 spec:
 containers:
 - name: tensorflow
 image: <model training image uri>
 command:
 Worker:
 replicas: 2
 restartPolicy: Never
 template:
 spec:
 containers:
 - name: tensorflow
 image: <model training image uri>
 command:
```

In this example template, the specification will create one copy of the parameter servers (which aggregate model parameters across different containers) and two copies of the workers (which run model training loops and communicate with the parameter servers). The operator will process the `TFJob` object according to the specification, keep the `TFJob` object stored in the system with the actual running services and Pods, and replace the actual state with the desired state. You can submit the training job using `kubectl apply -f <TFJob specs template>` and can get the status of the `TFJob` with the `kubectl get tfjob` command.

As a data scientist, you can submit Kubernetes training jobs or Kubeflow training jobs using the `kubectl` utility, or from your Jupyter Notebook environment using the **Python SDK**. For example, the `TFJob` object has a Python SDK called `kubernetes.tfjob`, and Kubernetes has a client SDK called `kubernetes.client` for interacting with the Kubernetes and Kubeflow environments from your Python code. You can also invoke training jobs using the `Kubeflow Pipeline` component, which we will cover later, in the *Kubeflow pipeline* section.

Using Kubernetes Jobs for ML training requires the installation and configuration of the necessary ML software components for training different models using different frameworks. You will also need to build logging and monitoring capabilities to monitor the training progress.

The adoption of Kubeflow training operators also presents its own set of challenges. Several operators, including the MPI training operator, are still in the maturing phase and are not yet suitable for production adoption. While operators provide certain logs and metrics, obtaining a comprehensive view of extensive training runs across Pods remains challenging and necessitates the integration of multiple dashboards. The existence of separate operators for various ML frameworks with fragmented capabilities and statuses complicates achieving a unified experience.

The learning curve for running training operators can be high, as it involves understanding many components, such as the development of training YAML files, distributed training job configuration, and training job monitoring.

## Registering models with a model registry

A **model registry** is an important component in model management and governance, and it is a key link between the model training stage and the model deployment stage, as models need to be properly stored in a managed repository for a governed model deployment. There are several open-source options for implementing a model registry in an ML platform. In this section, we will explore MLflow Model Registry for model management.

MLflow Model Registry is one of the leading model registry solutions with strong support in model artifacts lifecycle management, versioning support, and a range of deployment targets like Docker, Amazon SageMaker, and Azure ML. It is designed for managing all stages of the ML lifecycle, including experiment management, model management, reproducibility, and model deployment. It has the following four main components:

- **Experiment tracking:** During model development, data scientists run many training jobs as experiments using different datasets, algorithms, and configurations to find the best working model. Tracking the inputs and outputs of these experiments is critical to ensure efficient progress. Experiment tracking logs the parameters, code versions, metrics, and artifacts when running your ML code and for later visualizing of the results.
- **ML projects:** Projects package data science code in a format to reproduce runs on any platform.
- **Models:** Models provide a standard unit for packaging and deploying ML models.

- **Model Registry:** Model Registry stores, annotates, discovers, and manages models in a central repository.

The Model Registry component of MLflow provides a central model repository for saved models. It captures model details such as model lineage, model version, annotation, and description, and also captures model stage transitions from staging to production (so the status of the model state is clearly described).

To use MLflow Model Registry in a team environment, you need to set up an MLflow tracking server with a database as a backend and storage for the model artifacts. MLflow provides a UI and an API to interact with its core functionality, including Model Registry. Once the model is registered in Model Registry, you can add, modify, update, transition, or delete the model through the UI or the API.

The following figure shows an architecture setup for an MLflow tracking server and its associated Model Registry:

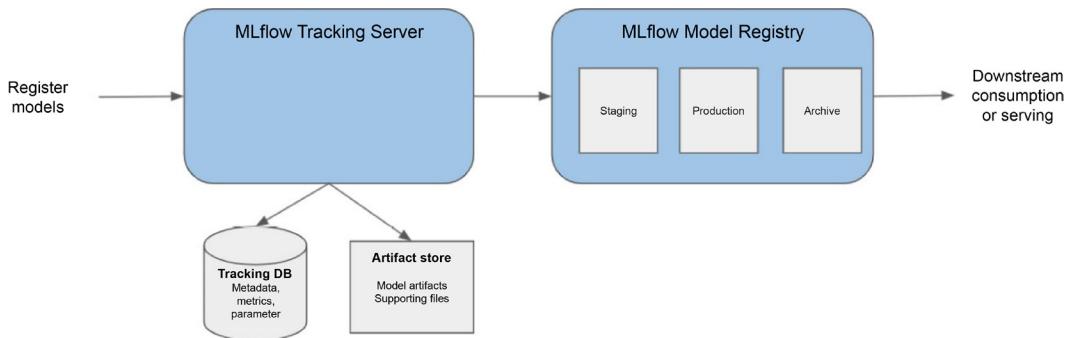


Figure 47.2: The MLflow tracking server and Model Registry

MLflow supports basic HTTP authentication to enable access control over experiments and registered models. The MLflow tracking server also supports basic authentication. However, these security capabilities might not be sufficient for enterprise requirements such as user group management and integration with third-party authentication providers. Organizations often need to implement separate security and authentication controls to manage access to resources.

## Serving models using model serving services

Once a model has been trained and saved, utilizing it to generate predictions is a matter of loading the saved model into an ML package and invoking the appropriate model prediction function provided by the package. However, for large-scale and complex

model serving requirements, you will need to consider implementing a dedicated model serving infrastructure to meet those needs.

In the subsequent sections, we will explore a variety of open-source model serving frameworks that can assist in addressing such needs.

## The Gunicorn and Flask inference engine

**Gunicorn** and **Flask** are often used for building custom model serving web frameworks. The following figure shows a typical architecture that uses Flask, Gunicorn, and Nginx as the building blocks for a model serving service.

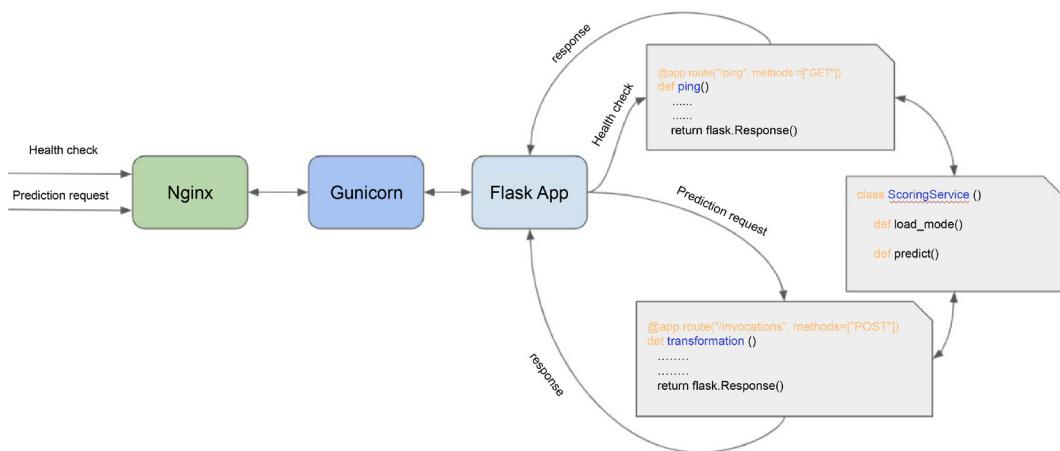


Figure 47.3: A model serving architecture using Flask and Gunicorn

Flask is a Python-based micro web framework for building web apps quickly. It is lightweight and has almost no dependencies on external libraries. With Flask, you can define different invocation routes and associate handler functions to handle different web calls (such as health check calls and model invocation calls). To handle model prediction requests, the Flask app would load the model into memory and call the predict function on the model to generate the prediction. Flask comes with a built-in web server, but it does not scale well as it can only support one request at a time.

This is where Gunicorn can help address the scalability gap. Gunicorn is a web server for hosting web apps, including Flask apps. It can handle multiple requests in parallel and distribute the traffic to the hosted web apps efficiently. When it receives a web

request, it will invoke the hosted Flask app to handle the request, such as invoking the function to generate the model prediction.

In addition to serving prediction requests as web requests, an enterprise inference engine also needs to handle secure web traffic (such as SSL/TLS traffic), as well as load balancing when there are multiple web servers. This is where Nginx can play an important role. Nginx can serve as a load balancer for multiple web servers and can handle termination for SSL/TLS traffic more efficiently, so web servers do not have to handle it.

A Flask/Gunicorn-based model serving architecture can be a good option for hosting simple model serving patterns. But for more complicated patterns such as serving different versions of models, A/B testing (showing two variants of a model to different user groups and comparing their responses), or large model serving, this architecture will have limitations. The Flask/Gunicorn architecture pattern also requires custom code (such as the Flask app) to work, as it does not provide built-in support for the different ML models.

Next, let's explore some purpose-built model serving frameworks and see how they are different from the custom Flask-based inference engine.

## The TensorFlow Serving framework

**TensorFlow Serving** is a production-grade, open-source model serving framework, and provides out-of-the-box support for serving TensorFlow models behind a RESTful endpoint. It manages the model lifecycle for model serving and provides access to versioned and multiple models behind a single endpoint. There is also built-in support for canary deployments. A **canary deployment** allows you to deploy a model to support a subset of traffic. In addition to the real-time inference support, there is also a batch scheduler feature that can batch multiple prediction requests and perform a single joint execution. With TensorFlow Serving, there is no need to write custom code to serve the model.

The following figure shows the architecture of TensorFlow Serving:

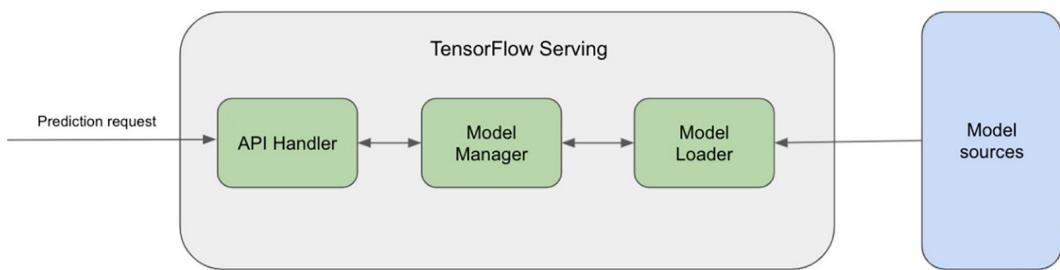


Figure 47.4: TensorFlow Serving architecture

Let's discuss each of the architecture components in more detail:

- The *API handler* provides APIs for TensorFlow Serving. It comes with a built-in, lightweight HTTP server to serve RESTful-based API requests. It also supports **gRPC** (a **remote procedure call** protocol) traffic. gRPC is a more efficient and fast networking protocol; however, it is more complicated to use than the REST protocol. TensorFlow Serving has a concept called a *servable*, which refers to the actual objects that handle a task, such as model inferences or lookup tables. For example, a trained model is represented as a *servable*, and it can contain one or more algorithms and lookup tables or embedding tables. The API handler uses the *servable* to fulfill client requests.
- The *model manager* manages the lifecycle of servables, including loading the servables, serving the servables, and unloading the servables. When a servable is needed to perform a task, the model manager provides the client with a handler to access the servable instances. The model manager can manage multiple versions of a servable, allowing gradual rollout of different versions of a model.
- The *model loader* is responsible for loading models from different sources, such as **Amazon S3**. When a new model is loaded, the model loader notifies the model manager about the availability of the new model, and the model manager will decide what the next step should be, such as unloading the previous version and loading the new version.

TensorFlow Serving can be extended to support non-TensorFlow models. For example, models trained in other frameworks can be converted to the **ONNX** format and served using TensorFlow Serving. ONNX is a common format for representing models to support interoperability across different ML frameworks.

## The TorchServe serving framework

**TorchServe** is an open-source framework for serving trained **PyTorch** models. Similar to TensorFlow Serving, TorchServe provides a REST API for serving models with its

built-in web server. With core features such as multi-model serving, model versioning, server-side request batching, and built-in monitoring, TorchServe can serve production workloads at scale. There is also no need to write custom code to host PyTorch models with TorchServe. In addition, TorchServe comes with a built-in web server for hosting the model.

The following figure illustrates the architecture components of the TorchServe framework:

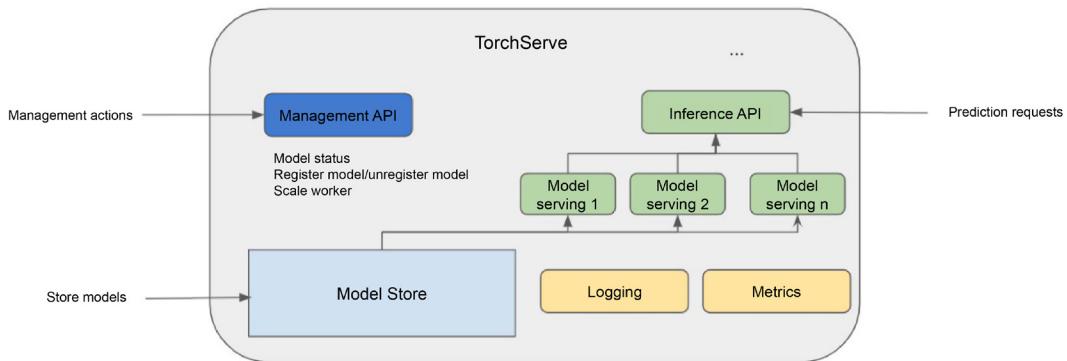


Figure 47.5: TorchServe architecture

The *inference API* is responsible for handling prediction requests from client applications using loaded PyTorch models. It supports the REST protocol and provides a prediction API, as well as other supporting APIs such as health check and model explanation APIs. The inference API can handle prediction requests for multiple models.

The model artifacts are packaged into a single archive file and stored in a model store within the TorchServe environment. You use a **command-line interface (CLI)** command called `torch-mode-archive` to package the model.

The TorchServe backend loads the archived models from the model store into different worker processes. These worker processes interact with the inference API to process requests and send back responses.

The management API is responsible for handling management tasks such as registering and unregistering PyTorch models, checking the model status, and scaling the worker process. The management API is normally used by system administrators.

TorchServe also provides built-in support for logging and metrics. The logging component logs both access logs and processing logs. The TorchServe metrics collect a list of system metrics, such as CPU/GPU utilization and custom model metrics.

## KFServing framework

TensorFlow Serving and TorchServe are standalone model serving frameworks for a specific deep learning framework. In contrast, **KFServing** is a general-purpose, multi-framework, model serving framework that supports different ML models. KFServing uses standalone model serving frameworks such as TensorFlow Serving and TorchServe as the backend model servers. It is part of the Kubeflow project and provides pluggable architecture for different model formats:

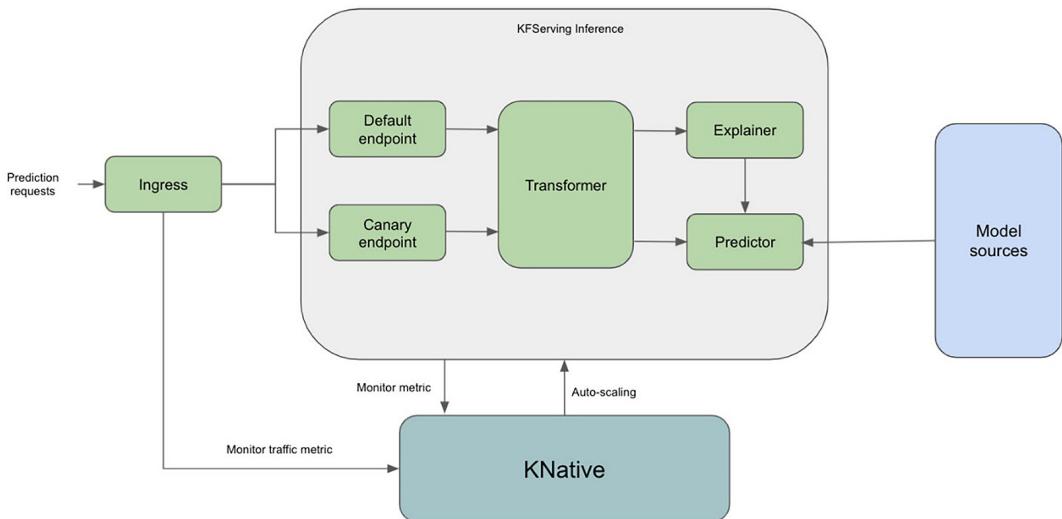


Figure 47.6: KFServing components

As a general-purpose, multi-framework model serving solution, KFServing provides several out-of-the-box model servers (also known as predictors) for different model types, including TensorFlow, PyTorch, XGBoost, scikit-learn, and ONNX. With KFServing, you can serve models using both REST and gRPC protocols. To deploy a supported model type, you simply need to define a YAML specification that points to the model artifact in a data store. Furthermore, you can build your own custom containers to serve models in KFServing. The container needs to provide a model serving implementation as well as a web server. The following code shows a sample YAML specification to deploy a tensorflow model using KFServing:

```
apiVersion: "serving.kubeflow.org/v1alpha2"
kind: "InferenceService"
metadata:
 name: "model-name"
```

```
spec:
 default:
 predictor:
 tensorflow:
 storageUri: <uri to model storage such as s3>
```

KFServing has a transformer component that allows the custom processing of the input payload before it is sent to the predictors, and also allows transforming the response from the predictor before it is sent back to the calling client. Sometimes, you need to provide an explanation for the model prediction, such as which features have a stronger influence on the prediction, which we will cover in more detail in a later chapter.

KFServing is designed for production deployment and provides a range of production deployment capabilities. Its auto-scaling feature allows the model server to scale up/down based on the amount of request traffic. With KFServing, you can deploy both the default model serving endpoint and the canary endpoint, split the traffic between the two, and specify model revisions behind the endpoint. For operational support, KFServing also has built-in functionality for monitoring (for example, monitoring request data and request latency).

## Seldon Core

**Seldon Core** is another multi-framework model serving framework for deploying models on Kubernetes. Compared to KFServing, Seldon Core provides richer model serving features, for example, model serving inference graphs for use cases such as A/B testing and model ensembles. The following figure shows the core components of the Seldon Core framework:

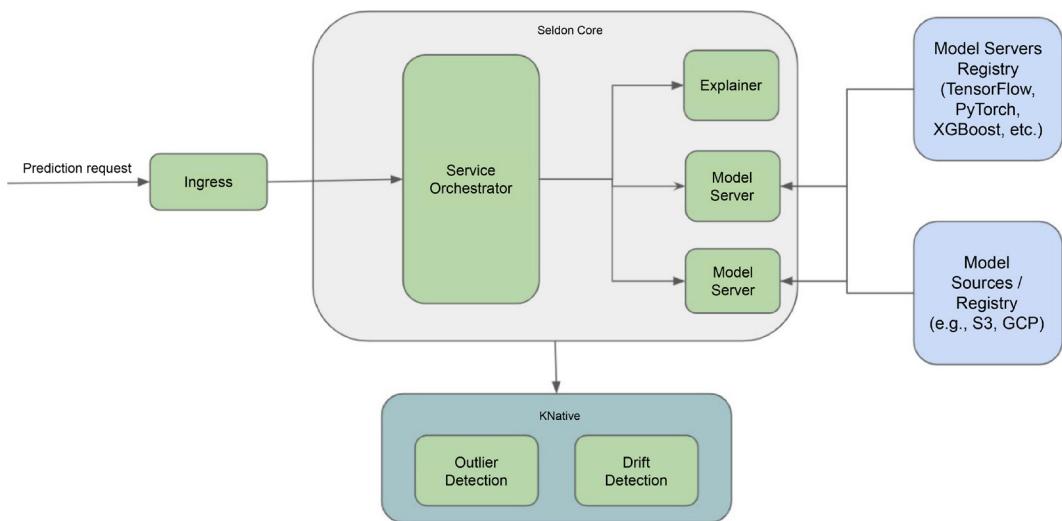


Figure 47.7: The Seldon Core model serving framework architecture

Seldon Core provides packaged model servers for some of the common ML libraries, including the `SKLearn` server for scikit-learn models, the `XGBoost` server for XGBoost models, `TensorFlow Serving` for TensorFlow models, and `MLflow` server-based model serving. You can also build your own custom serving container for specific model serving needs and host it using Seldon Core.

The following template shows how to deploy a model using the `SKLearn` server using Seldon Core. You simply need to change the `modelUri` path to point to a saved model on a cloud object storage provider such as **Google Cloud Storage**, **Amazon S3 storage**, or **Azure Blob storage**. To test with an example, you can change the following `modelUri` value to an example provided by Seldon Core – `gs://seldon-models/sklearn/iris`:

```
apiVersion: machinelearning.seldon.io/v1alpha2
kind: SeldonDeployment
metadata:
 name: sklearn
spec:
 name: sklearn-model
 predictors:
 - graph:
 children: []
 implementation: SKLEARN_SERVER
 modelUri: <model uri to model artifacts on the cloud storage>
```

```
name: classifier
name: default
replicas: 1
```

Seldon Core also supports an advanced workflow, known as an inference graph, for serving models. The *inference graph* feature allows you to have a graph with different models and other components in a single inference pipeline. An inference graph can consist of several components:

- One or more ML models for the different prediction tasks
- Traffic routing management for different usage patterns, such as traffic splitting to different models for A/B testing
- A component for combining results from multiple models, such as a model ensemble component
- Components for transforming the input requests (such as performing feature engineering) or output responses (for example, returning an array format as a JSON format)

To build inference graph specifications in YAML, you need the following key components in the `seldondeployment` YAML file:

- A list of predictors, with each predictor having its own `componentSpecs` section that specifies details such as container images
- A graph that describes how the components are linked together for each `componentSpecs` section

The following sample template shows the inference graph for a custom canary deployment to split the traffic into two different versions of a model – one with 75% of the traffic and another one with 25% of the traffic:

```
apiVersion: machinelearning.seldon.io/v1alpha2
kind: SeldonDeployment
metadata:
 name: canary-deployment
spec:
 name: canary-deployment
 predictors:
 - componentSpecs:
 - spec:
 containers:
 - name: classifier
 image: <container uri to model version 1>
```

```
graph:
 children: []
 endpoint:
 type: REST
 name: classifier
 type: MODEL
 name: main
 replicas: 1
 traffic: 75
 - componentSpecs:
 - spec:
 containers:
 - name: classifier
 image: <container uri to model version 2>
graph:
 children: []
 endpoint:
 type: REST
 name: classifier
 type: MODEL
 name: canary
 replicas: 1
 traffic: 25
```

Once a deployment manifest is applied, the Seldon Core operator is responsible for creating all the resources needed to serve an ML model. Specifically, the operator will create resources defined in the manifest, add orchestrators to the Pods to manage the orchestration of the inference graph, and configure the traffic using ingress gateways such as Istio.

## Triton Inference Server

Triton Inference Server is open-source software designed to streamline the process of AI inferencing. It offers a versatile solution for deploying AI models from various deep learning and ML frameworks, including TensorRT, TensorFlow, PyTorch, ONNX, OpenVINO, Python, and more. Triton is compatible with a wide range of devices, supporting inference across cloud environments, data centers, edge devices, and embedded systems. Compared to Seldon Core, Triton Inference Server is more focused on performance. It is designed to be highly scalable and efficient, making

it a good choice for high-traffic applications. The following figure depicts the core components of Triton Inference Server:

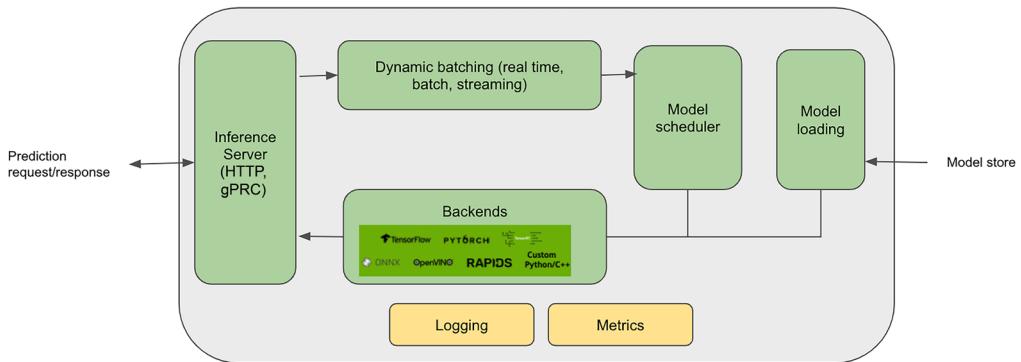


Figure 47.8: Triton Inference Server architecture

The Triton Inference Server architecture encompasses several components that work together to enable efficient and scalable inferencing. At its core is the backend, which represents specific deep learning or ML frameworks supported by Triton. Each backend handles the loading and execution of models trained with its corresponding framework. Triton Inference Server acts as the central hub, receiving and managing inference requests. It communicates with clients, such as web applications or services, and orchestrates the inferencing process. The model repository serves as the storage location for trained models. It contains serialized versions of models compatible with the supported backends. When requested by clients, the server accesses and loads the models into memory for inferencing.

Triton supports multiple inference protocols, including HTTP/REST and gRPC, allowing clients to communicate with the server and make inference requests. Clients can specify input data and desired output formats using these protocols. To monitor and optimize performance, Triton provides metrics and monitoring capabilities. These metrics include GPU utilization, server throughput, latency, and other relevant statistics. Monitoring these metrics helps administrators optimize resource utilization and identify potential bottlenecks.

Triton also offers dynamic batching capability. This feature allows for efficient processing of multiple inference requests by grouping them together into batches. This batching mechanism optimizes resource utilization and improves overall inferencing performance.

Overall, the architecture of Triton Inference Server is designed to facilitate the efficient deployment and execution of AI models across diverse frameworks and hardware platforms. It offers flexibility, scalability, and extensibility, enabling organizations to leverage their preferred frameworks while ensuring high-performance inferencing capabilities.

## Monitoring models in production

Model performance can deteriorate over time due to various factors such as changing data patterns, shifts in user behavior, or unforeseen scenarios. To ensure the ongoing effectiveness of deployed ML models, continuous monitoring of their performance and behavior in production is essential.

Model monitoring involves actively tracking and analyzing the performance of deployed ML models. This process includes collecting data on different metrics and indicators, comparing them to predefined thresholds or baselines, and identifying anomalies or deviations from expected behavior. Two critical aspects of model monitoring are data drift and model drift:

- **Data drift:** Data drift refers to the scenarios where the statistical properties of incoming data change over time. This can create a disconnect between the data used to train the model and the data it encounters in the production environment. Data drift significantly impacts the performance and reliability of ML models, as they may struggle to adapt to new and evolving patterns in the data.
- **Model drift:** Model drift refers to the degradation of an ML model's performance over time due to changes in underlying patterns or relationships in the data. When the assumptions made during model training no longer hold true in the production environment, model drift occurs. It can lead to decreased accuracy, increased errors, and suboptimal decision-making.

To support model monitoring efforts, there are several open-source and commercial products available in the market. These tools provide capabilities for monitoring model performance, detecting data drift, identifying model drift, and generating insights to help organizations take necessary corrective actions. Some popular examples include Evidently AI, Arize AI, Seldon Core, Fiddler, and Author AI.

## Managing ML features

As organizations increasingly adopt ML solutions, they recognize the need to standardize and share commonly used data and code throughout the ML lifecycle. One crucial element that organizations seek to manage centrally is ML features, which

are commonly used data attributes that serve as inputs to ML models. To enable standardization and reuse of these features, organizations often turn to a feature store.

A feature store acts as a centralized repository for storing and managing ML features. It provides a dedicated platform for organizing, validating, and sharing features across different ML projects and teams within an organization. By consolidating features in a single location, the feature store promotes consistency and facilitates collaboration among data scientists and ML practitioners.

The concept of a feature store has gained significant attention in the ML community due to its numerous benefits. Firstly, it enhances productivity by eliminating the need to recreate and engineer features for each ML project. Instead, data scientists can readily access precomputed and validated features from the store, saving time and effort. Additionally, a feature store improves model performance by ensuring the consistency and quality of features used in ML models. By centralizing feature management, organizations can enforce data governance practices, perform feature validation, and monitor feature quality, leading to more reliable and accurate ML models.

Several open-source feature store frameworks are available in the market, such as Feast and Hopsworks Feature Store, offering organizations flexible options for managing their ML features. Let's take a closer look at Feast as an example to get a deep understanding of how a feature store works.

Feast is an open-source feature store that enables organizations to manage, discover, and serve features for ML applications. Developed by Tecton, Feast is designed to handle large-scale, real-time feature data. It supports feature ingestion from various sources, including batch pipelines and streaming systems like Apache Kafka. Feast integrates well with popular ML frameworks such as TensorFlow and PyTorch, allowing seamless integration into ML workflows. With features like feature versioning, data validation, and online and offline serving capabilities, Feast provides a comprehensive solution for feature management.

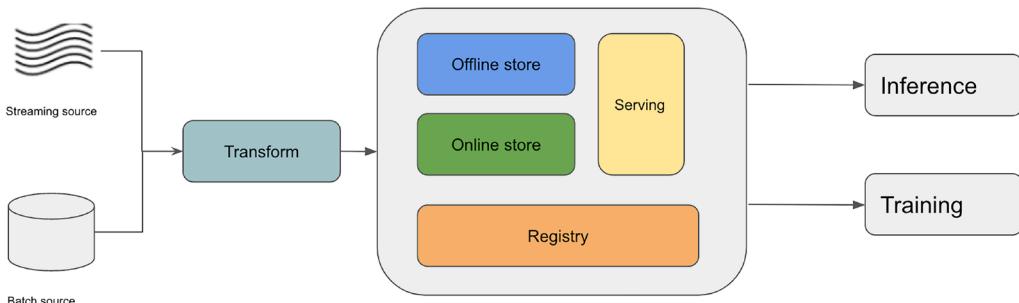


Figure 47.9: Feast feature store

At the core of the Feast architecture is the online and offline feature storage, which serves as the centralized storage for feature data. The feature repository stores feature data in a distributed storage system, allowing for scalable and efficient storage and retrieval of feature values.

Feast employs a decoupled architecture, where the ingestion of feature data and the serving of features are separated. The data ingestion component is responsible for extracting feature data from various sources, such as data warehouses, databases, and streaming platforms. It then transforms and loads the feature data into the feature storage, ensuring data quality and consistency.

The feature serving component is responsible for providing low-latency access to feature data for ML models during training and inference. The feature serving component also supports online and offline serving modes, allowing for real-time and batch feature serving.

To enable efficient data discovery, Feast employs a feature registry. The feature registry allows for the fast lookup and retrieval of feature values based on different feature combinations and time ranges.

Feast also integrates with popular ML frameworks, such as TensorFlow and PyTorch, through its SDKs and client libraries. These integrations enable seamless integration of Feast into ML pipelines and workflows, making it easy for data scientists and ML engineers to access and utilize feature data in their models.

Overall, the Feast feature store architecture provides a robust and scalable solution for managing and serving ML features. By centralizing feature data management, Feast enables organizations to enhance productivity, improve model performance, and promote collaboration in ML development.

## Automating ML pipeline workflows

To automate the core ML platform components we have discussed so far, we need to build pipelines that can orchestrate different steps using these components. Automation brings efficiency, productivity, and consistency while enabling reproducibility and minimizing human errors. There are several open-source technologies available to automate ML workflows, with Apache Airflow and Kubeflow Pipelines being prominent examples.

## Apache Airflow

**Apache Airflow** is an open-source software package for programmatically authoring, scheduling, and monitoring multi-step workflows. It is a general-purpose workflow orchestration tool that can be leveraged to define workflows for a wide range of tasks, including ML tasks. First, let's explore some core Airflow concepts:

- **Directed Acyclic Graph (DAG):** A DAG defines independent tasks that are executed independently in a pipeline. The sequences of the execution can be visualized like a graph.
- **Tasks:** Tasks are basic units of execution in Airflow. Tasks have dependencies between them during executions.
- **Operators:** Operators are DAG components that describe a single task in the pipeline. An operator implements the task execution logic. Airflow provides a list of operators for common tasks, such as a Python operator for running Python code, or an Amazon S3 operator to interact with the S3 service. Tasks are created when operators are instantiated.
- **Scheduling:** A DAG can run on demand or on a predetermined schedule.
- **Sensors:** Sensors are a special type of operator that are designed to wait for something to occur. They can then help trigger a downstream task to happen.

Airflow can run on a single machine or in a cluster. Additionally, it can be deployed on the Kubernetes infrastructure. The following figure shows a multi-node Airflow deployment:

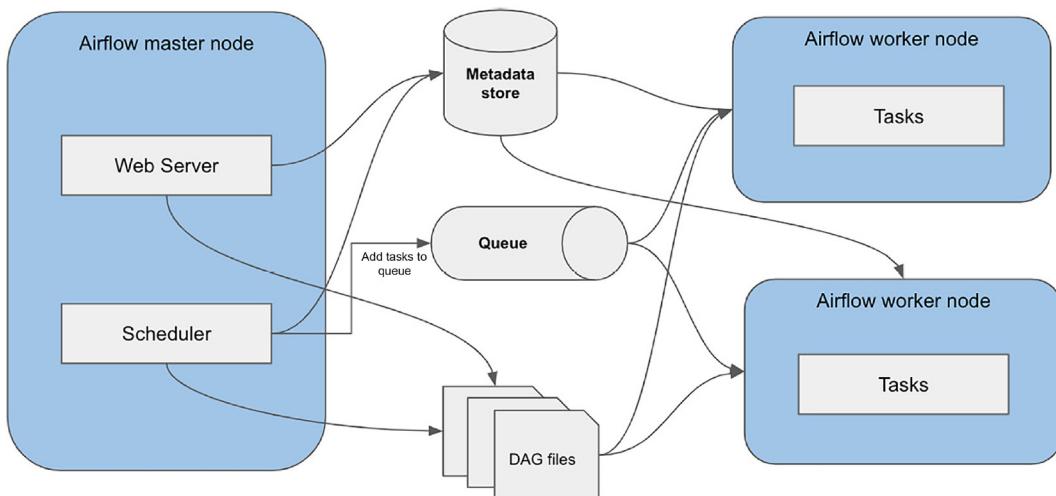


Figure 47.10: Apache Airflow architecture

The *master node* mainly runs the *web server* and *scheduler*. The scheduler is responsible for scheduling the execution of the DAGs. It sends tasks to a queue, and the worker nodes retrieve the tasks from the queue and run them. The metadata store is used to store the metadata of the Airflow cluster and processes, such as task instance details or user data.

You can author the Airflow DAGs using Python. The following sample code shows how to author a basic Airflow DAG in Python with two Bash operators in a sequence:

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta
default_args = {
 'owner': myname,
}
dag = DAG('test', default_args=default_args, schedule_
interval=timedelta(days=1))
t1 = BashOperator(
 task_id='print_date',
 bash_command='date',
 dag=dag)
t2 = BashOperator(
 task_id='sleep',
 bash_command='sleep 5',
 retries=3,
 dag=dag)
t2.set_upstream(t1)
```

Airflow can connect to many different sources and has built-in operators for many external services, such as **Amazon EMR** and **Amazon SageMaker**. It has been widely adopted by many organizations to run large-scale workflow orchestration jobs in production, such as coordinating ETL jobs and ML data processing jobs. AWS even has a managed Airflow offering to help reduce the operational overhead of running Airflow infrastructure.

Airflow also comes with some limitations. Airflow does not offer a UI designer for DAG development, which can be a challenge for users without Python programming skills to design workflows. The lack of versioning control with DAG pipelines also poses some challenges with managing and understanding the evolving variations of pipelines. Operating Airflow on Kubernetes can be complex, which is why many organizations opt for managed offerings. Despite these limitations, however, Airflow has emerged as a highly popular workflow orchestration tool due to its enterprise-ready capability, strong community support, and rich ecosystem.

## Kubeflow Pipelines

**Kubeflow Pipelines** is a Kubeflow component, and it is purpose-built for authoring and orchestrating end-to-end ML workflows on Kubernetes. First, let's review some core concepts of Kubeflow Pipelines:

- **Pipeline:** A pipeline describes an ML workflow, all the components in the workflow, and how the components are related to each other in the pipeline.
- **Pipeline components:** A pipeline component performs a task in the pipeline. An example of a pipeline component could be a data processing component or a model training component.
- **Experiment:** An experiment organizes different trial runs (model training) for an ML project so you can easily inspect and compare the different runs and their results.
- **Step:** The execution of one component in a pipeline is called a step.
- **Run trigger:** You use a run trigger to kick off the execution of a pipeline. A run trigger can be a periodic trigger (for example, to run every 2 hours), or a scheduled trigger (for example, run at a specific date and time).
- **Output artifacts:** Output artifacts are the outputs from the pipeline components. Examples of output artifacts could be model training metrics or visualizations of datasets.

Kubeflow Pipelines is installed as part of the Kubeflow installation. It comes with its own UI, which is part of the overall Kubeflow dashboard UI. The Pipelines service manages the pipelines and their run status and stores them in a metadata database. There is an orchestration and workflow controller that manages the actual execution of the pipelines and the components. The following figure illustrates the core architecture components in a Kubeflow pipeline:

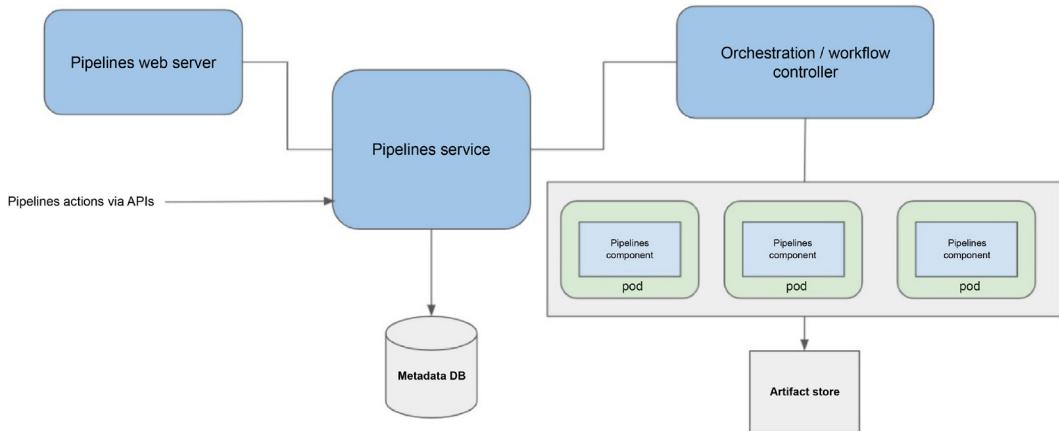


Figure 47.11: Kubeflow Pipelines architecture

You author the pipeline using the Pipeline SDK in Python. To create and run a pipeline, follow these steps:

1. Create a pipeline definition using the Kubeflow SDK. The pipeline definition specifies a list of components and how they are joined together in a graph.
2. Compile the definition into a static YAML specification to be executed by the Kubeflow Pipelines service.
3. Register the specification with the Kubeflow Pipelines service and call the pipeline to run from the static definition.
4. The Kubeflow Pipelines service calls the API server to create resources to run the pipeline.
5. Orchestration controllers execute various containers to complete the pipeline run.

It is important to note that running pipelines using Kubeflow Pipelines requires a high degree of competency with Kubernetes, which could be challenging for people without deep Kubernetes skills. Building the workflow in Python can be a complex task that involves writing a Dockerfile or YAML file for each component, and Python scripts for the workflow and execution. Kubeflow Pipelines mainly works within the Kubeflow environment, with very limited integration with external tools and services. It also lacks native pipeline versioning capability. Despite these challenges, Kubeflow Pipelines is still widely adopted due to its support for end-to-end ML management, workflow visualization, portability, and reproducibility.

Now that we have explored various open-source tools for building ML platforms, let's delve into end-to-end architecture using these open-source frameworks and components.

## Designing an end-to-end ML platform

After discussing several open-source technologies individually, let's now delve into their integration and see how these components come together. The architecture patterns and technology stack selection may vary based on specific needs and requirements. The following diagram presents the conceptual building blocks of an ML platform architecture:

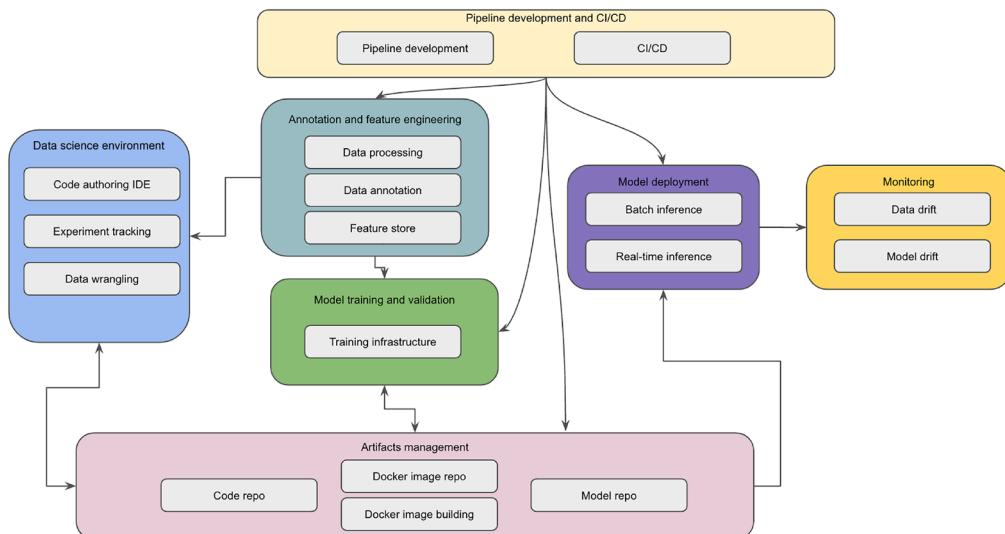


Figure 47.12: ML platform architecture

Next, let's delve into different strategies to implement this architecture concept with different combinations of open-source technologies.

## ML platform-based strategy

When designing an ML platform using open-source technologies, one effective strategy is to utilize an ML platform framework as a base platform and then integrate additional open-source components to address specific requirements. One such ML platform framework is Kubeflow, which provides a robust foundation with its built-in

building blocks for an ML platform. By leveraging Kubeflow, you can benefit from its core components while extending the platform's capabilities through the integration of complementary open-source tools.

This strategy allows for flexibility and customization by seamlessly integrating a range of open-source ML components into the platform. You would choose this approach if the base ML platform framework meets most of your requirements, or if you can work within the limitations of the framework. The following table outlines key ML platform components and their corresponding open-source frameworks and tools:

ML Platform Component	Open-Source Framework
Code repository	GitHub
Experimentation and model development	Kubeflow Jupyter Notebook
Experiment tracking	MLflow experiment tracker
Feature store	Feast feature store
Data annotation	Computer Vision Annotation Tool (CVAT)
Training	Kubeflow training operators
Data and model testing	Deepchecks
Model repository	MLflow Model Repository
ML pipeline development	Kubeflow Pipelines
Model inference	Kubeflow KFServing (Seldon Core, TFServing, Triton)
Docker image repository	Docker Hub
CI/CD	GitHub Actions
Drift monitoring	Deepchecks

Table 47.1: ML platform components and their corresponding open-source frameworks

By incorporating these frameworks and tools into the architectural conceptual diagram, we can visualize the resulting diagram as follows:

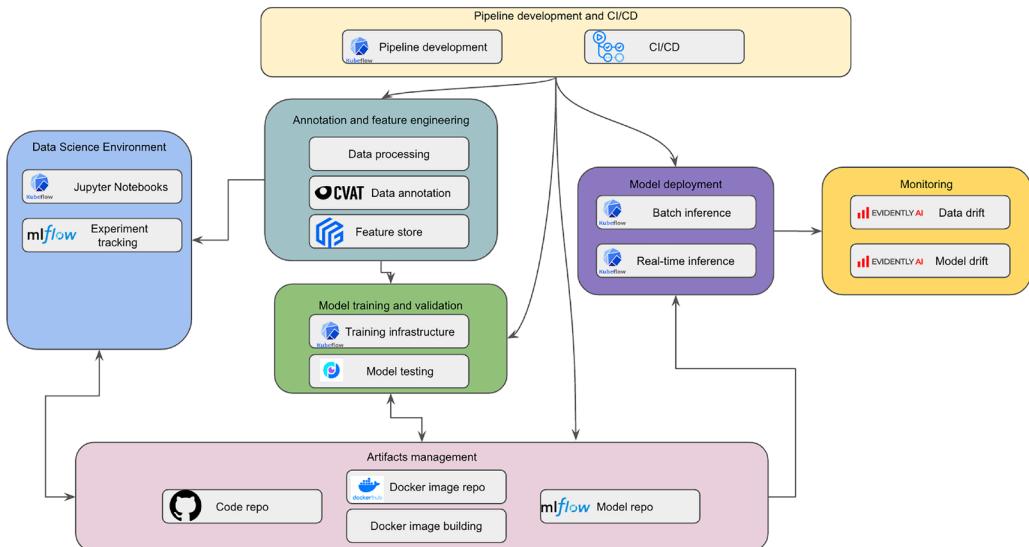


Figure 47.13: Kubeflow-based ML platform

Using this architecture, data scientists utilize Kubeflow Jupyter Notebook for conducting experiments and building models. Experiment runs and relevant details, such as data statistics, hyperparameters, and model metrics, are tracked and saved in the MLflow experiment tracking component.

Common ML features are stored in the Feast feature store. When there is a need for data annotation, data annotators can employ open-source data annotation tools like the **Computer Vision Annotation Tool (CVAT)** and Label Studio to label data for model training. Data scientists can utilize features from the feature store and the labeled dataset as part of their experimentation and model building.

GitHub serves as the code repository for data scientists. They save all source code, including training scripts, algorithm code, and data transformation scripts, in the code repository. Model training and inference Docker images are stored in Docker Hub. A Docker image build process can be deployed to create new Docker images for training and inference purposes.

For formal model training, the training script is pulled from the GitHub repository, and the training Docker image is pulled from Docker Hub into the Kubeflow training operator to initiate the model training, along with the training dataset. Deepchecks can be utilized to perform data validation and model performance checks. Once the model is trained, the model artifacts, along with any metadata such as model metrics and evaluation graphs, are stored in MLflow Model Registry.

When it is time to deploy the model, models are fetched from MLflow Model Registry and loaded into KFServing for model inference, along with the model inference Docker image and inference script. KFServing offers the flexibility to choose different inference servers, including Seldon Core, TServing, and Triton.

Prediction logs can be sent to the model monitoring component for detecting data drift and model drift. Open-source software tools like Evidently AI can be employed for data drift and model drift detection.

To orchestrate various tasks such as data processing, feature engineering, model training, and model validation, Kubeflow Pipelines can be developed. For **CI/CD (Continuous Integration/Continuous Deployment)**, GitHub Actions can be used as a triggering mechanism to initiate different pipelines.

Overall, this approach allows you to combine the benefits of a base ML platform framework with the flexibility provided by a wide range of open-source components.

## ML component-based strategy

An alternative approach is to build the ML platform using individual components rather than relying on a base ML platform framework. This strategy offers the advantage of selecting the best-in-class components for each aspect of the platform, allowing organizations to adhere to their existing open-source standards for core components like pipeline development or notebook IDEs. The following architectural pattern illustrates this approach.

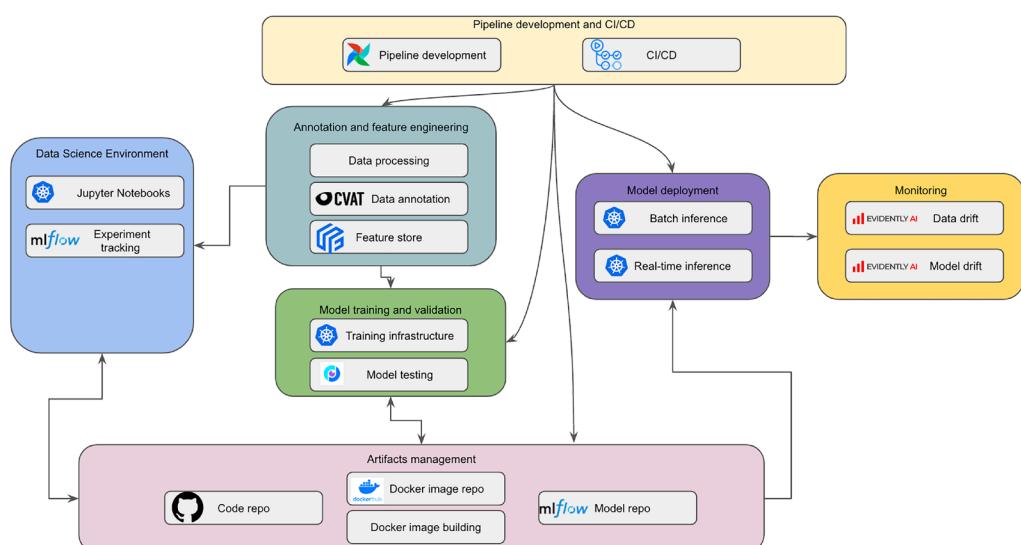


Figure 47.14: Component-based architecture

In this architecture pattern, alternative technologies and tools are utilized for pipeline development, notebook environments, and training infrastructure management. Additionally, Kubernetes serves as the infrastructure management framework. This approach allows organizations to leverage specific technologies and tools that align with their unique requirements and preferences.

One notable aspect is the use of Airflow as the standard orchestration and pipeline tool across various technical disciplines, including ML and data management. Airflow's widespread adoption within organizations enables it to serve as a unifying pipeline management tool, facilitating seamless integration between different components and workflows.

Moreover, this architecture pattern emphasizes the building of custom training and inference infrastructure on top of Kubernetes. By leveraging Kubernetes, organizations gain the flexibility to create customized training and inference environments tailored to their specific needs.

In addition to the availability of free open-source tools that meet ML platform requirements, it is important to consider the integration of commercial components into the open-source architecture. These commercial offerings can enhance specific aspects of the ML platform and provide additional capabilities.

For instance, when deploying this architecture pattern on AWS, it is advisable to explore the use of Amazon Elastic Container Registry (ECR) as the Docker image repository. Amazon ECR provides a managed and secure solution for storing and managing container images, integrating seamlessly with other AWS services.

When it comes to monitoring, there are commercial products like Fiddler and Author AI that can offer advanced features and insights. These tools can enhance the monitoring capabilities of the ML platform, providing in-depth analysis, model explainability, and visualization of model behavior and performance.

Overall, the advantages of this architecture pattern include the ability to choose alternative technologies and tools for different aspects of the ML platform, leveraging Airflow as a unifying pipeline management tool, and building custom training and inference infrastructure on Kubernetes. These choices enable organizations to create a tailored and optimized ML platform that aligns precisely with their requirements and allows for highly customized training and inference processes.

\*\*\* END OF STUDY BOOK 3 \*\*\*