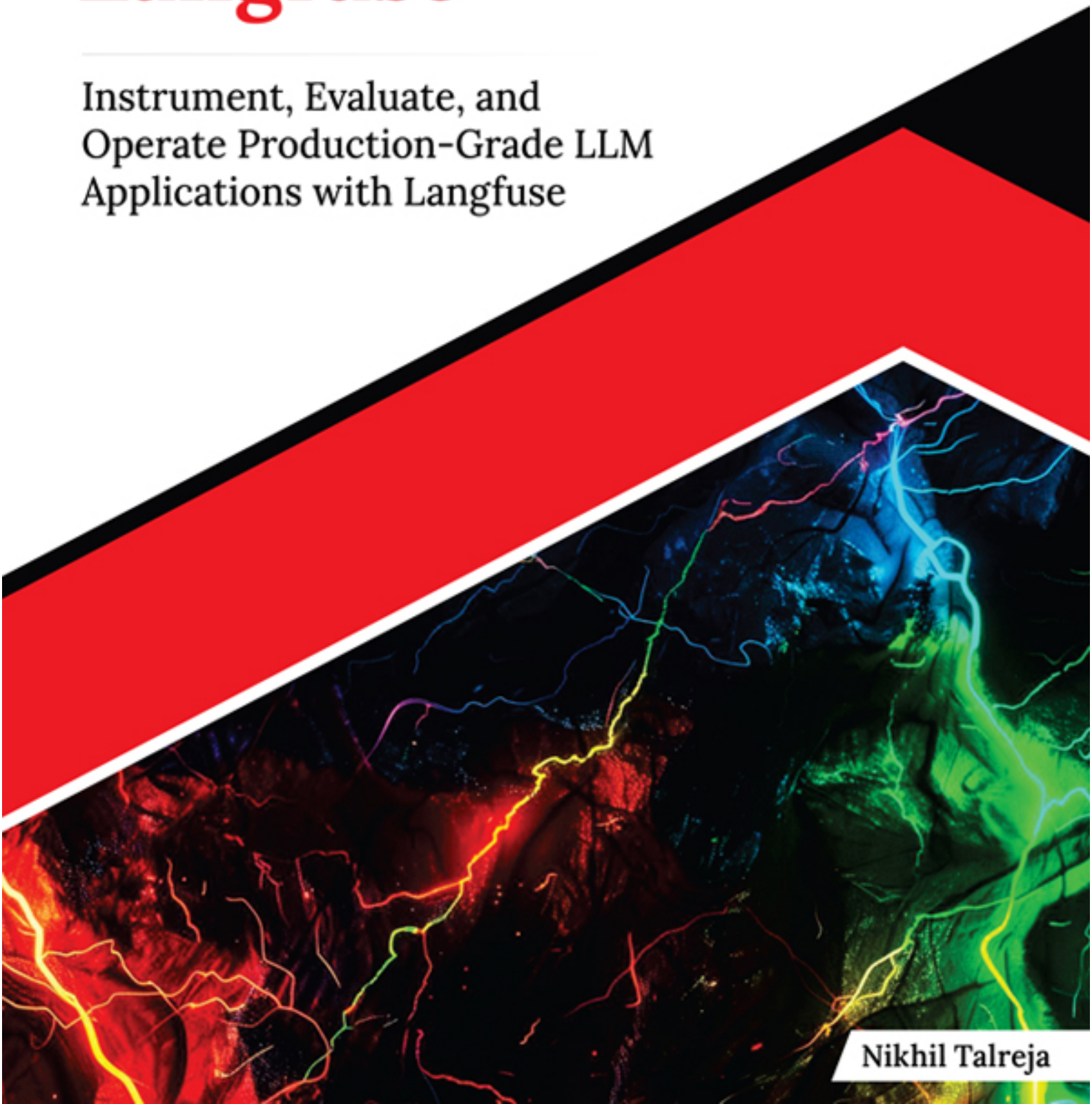
ULTIMATE

LLMOps with Langfuse

Instrument, Evaluate, and Operate Production-Grade LLM Applications with Langfuse



Nikhil Talreja

First Published: May 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,
N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-54-1

ISBN (E-BOOK): 978-93-49887-78-7

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

CHAPTER 1

Introduction to Large Language Models and Monitoring

Introduction

Large Language Models (LLMs) are transforming the world with their ability to understand and generate human-like content. This chapter introduces the fundamentals of LLMs—how they work, their core architectures, and where they are being applied today. We will explore the challenges that arise when deploying these models into real-world systems, including performance bottlenecks, hallucinations, escalating costs, and also the risk of bias or model drift. We will learn why monitoring is essential to ensure that LLMs remain accurate, efficient, and secure over time. The chapter further outlines the LLM lifecycle—from development and fine-tuning to deployment and continuous maintenance—and where monitoring fits in each stage. Tools and platforms such as OpenAI API, LangChain and Langfuse will be introduced, setting the stage for the more hands-on chapters ahead.

Structure

In this chapter, we will cover the following topics:

- Understanding Large Language Models (LLMs)
  - A Brief History of NLP
  - Architecture and Components of LLMs
- LLM Use Cases across Industries
  - Content Generation
  - Customer Support and Knowledge Management
  - Software Engineering

    - Finance
    - Healthcare
    - Retail, E-Commerce and Marketing
- Challenges in Deploying LLM Applications
    - Non-Deterministic Outputs
    - Infrastructure and Performance Requirements
    - Observability Requirements
    - Ethical, Safety, and Regulatory Concerns
- Key LLM Monitoring and Observability Metrics
    - Latency
    - Cost
    - Token Usage
    - Hallucination
    - Completeness
    - Relevance
    - Correctness
- Introduction to the LLM Application Lifecycle
    - Problem Definition and Requirements Gathering
    - Data Preparation
    - Prompt and System Design
    - Prototyping and Experimentation
    - Evaluation and Benchmarking
    - Deployment and Monitoring
- Exploring the LLM Toolkit for the Book
    - OpenAI SDK
    - LangChain
    - Langfuse

Understanding Large Language Models (LLMs)

A **Large Language Model** (LLM) is a type of artificial intelligence trained on massive collections of text to understand and generate human-like content. This content can be either text, audio, images or even videos. LLMs are built on the **Transformer Architecture** which is a type of **Artificial Neural Network** (ANN) architecture that allows them to capture context and relationships across long passages of text like how humans understand natural language. Because of this, LLMs can perform a wide range of tasks —such as answering questions, summarizing documents, generating code, or producing creative content—without requiring task-specific programming. As an ANN, an LLM **predicts the next word** in a sequence of words.

In a 2018 research paper titled *Improving Language Understanding by Generative Pre-Training*, a team of researchers at **OpenAI** proposed a two-staged model training framework which paved the way for developing the large-scale LLMs as we know them today:

- **Unsupervised Pre-training:** This involves training a Transformer based language model on a huge amount of text with the objective of predicting the next word. This step helps the model understand language syntax, semantics and world knowledge.
- **Supervised Fine-tuning:** Taking the model from stage 1, adapt the model on a specific task like summarization, question answering, sentiment analysis and so on.

This simple paradigm of pre-train + fine-tune led to the explosion of general purpose language understanding systems that are now redefining technological and human history.

Before we jump into the core architecture and components of LLMs, let us take a trip down memory lane to see how the field of **Natural Language Processing** (NLP) evolved.

A Brief History of NLP

Natural Language Processing (NLP), the field of artificial intelligence that focuses on enabling computers to understand, interpret, generate, and interact using human language has evolved significantly over the past several decades, transitioning from early rule-based approaches to the development of advanced LLMs. Each stage in this progression addressed

certain challenges while presenting new ones, ultimately leading to breakthroughs such as the Transformer architecture and GPT models.

1964 – 1967: ELIZA — The First "Chatterbot"

The story of NLP starts as early as the 1950s, a major milestone came in 1966, when a Computer Scientist and MIT Professor **Joseph Weizenbaum** built ELIZA – an NLP program that used pattern matching and hardcoded rules to simulate communication between humans and machines. Its most famous script, called DOCTOR, mimicked a psychotherapist that could talk to humans as if it understood them.

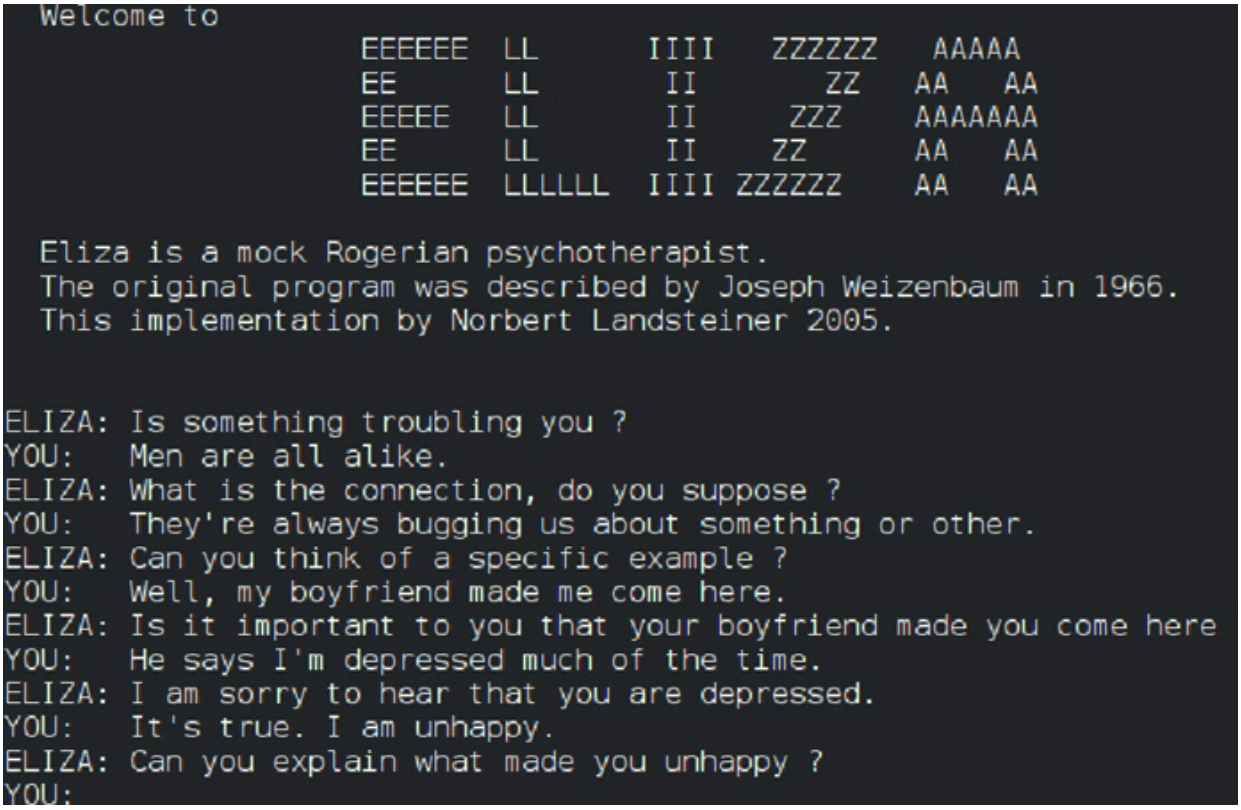


Figure 1.1: *A Conversation with ELIZA*

ELIZA essentially kicked off the **chatbot era** and raised questions about *simulation vs. true intelligence*—questions still debated with modern LLMs like GPT and Claude. Despite having no true understanding of human language, many of ELIZA's users were convinced of its capabilities and intelligence.

1990s – 2000s: NLP Starts Using Machine Learning

The earlier NLP models which were rule based were impractical. Considering the huge amount of text to be processed for every human language, this idea becomes unmaintainable quickly. Instead of manually encoding grammar or semantics, NLP started adopting early Machine Learning models that used data-driven approaches. The idea was to make models learn patterns from a big corpus of texts and make predictions based on probabilities.

These statistical models were a turning point to end the so-called **AI Winter** of the 1980s, which was a period of reduced interest (and funding) in the field. The most notable advancements during this time were:

- **N-gram Models**: These models estimate the probability of a word occurring given the preceding n-1 words. They make predictions for the next word based on how likely the word appeared in previously seen documents. While these models worked well on seen data, the biggest issue was Out-of-Vocabulary (OOV) words, which happens when the model sees a word during inference it has never seen during training. An unseen word in this model gets a zero probability, even if it is a perfectly valid word.
- **Hidden Markov Models (HMMs):** While N-gram models could predict next words, they had no understanding of parts of speech or named entities (people, locations, organizations and so on). HMMs solved this problem by capturing relationships between sentence structure and the words. They found their way into speech recognition systems and Part-of-Speech (POS) tagging. OOV words and long texts were still a problem for these models since HMMs work under the assumption that to predict what comes next, we only need to look at the most recent step. In other words, the model remembers just one step back, as if language only had a one-word memory.
- **Recurrent Neural Networks (RNNs):** In the early 90s, researchers started experimenting with the idea of designing NLP models that could actually think and work like the human brain. This gave birth to the idea of a **Neural Network (NN)** which is a computational model inspired by the brain, made up of interconnected units (“neurons”) that process and transform inputs. While NNs are a subject of its own, it would be worth mentioning a special type of NN called the Recurrent Neural Network (RNN). RNNs introduced a neural architecture

capable of modeling sequences with theoretically **unlimited memory**, addressing the short-range limitations of n-grams and HMMs. RNNs were groundbreaking in showing that a neural network could, in principle, learn the sequential structure of language without explicit hard coding structure. Later innovations like **Long Short-Term Memory networks (LSTMs, 1997)** would overcome many of these limitations, setting the stage for the deep learning revolution of the 2010s.

2010s: The Rise of Neural Networks

By the early 2010s, traditional statistical NLP and early neural networks like **RNNs** showed limitations with large datasets. As digital text grew, researchers shifted to **representation learning** using improved hardware and new architectures, marking a neural renaissance in NLP. During this time, content from social media, the web, and enterprise data was exploding and created both a pressing need and an opportunity for more powerful models. Some key breakthroughs in this period were:

- **Word2Vec:** Word2Vec was a technique developed by Google AI in 2013. At its core, it represents words as dense vectors in a continuous space. This approach allowed models to capture semantic relationships between words, something traditional n-gram models could not do. In practical terms, Word2Vec enabled NLP systems to reason about meaning dramatically, improving tasks such as information retrieval, analogy solving, and sentiment analysis.
- **Encoder-Decoder Architectures:** Encoder-decoder models introduced a two-part neural framework for processing sequences: the encoder reads and compresses an input sequence into a fixed-length representation, and the decoder generates a corresponding output sequence. This approach was a major advancement in NLP because it made tasks like machine translation and text summarization feasible end-to-end without task-specific feature engineering.
- **Attention Mechanism:** The attention mechanism as introduced in the 2017 paper *Attention is All You Need* allows a model to focus selectively on different parts of the input when generating each output word, rather than relying on a single fixed representation. This fundamentally changed NLP: models could now capture long-range

dependencies efficiently, scale to billions of parameters, and ultimately serve as the backbone for modern LLMs like GPT, Claude, and Llama.

2017: Attention is All You Need

The landmark paper **"Attention is All You Need"** by Vaswani et al. introduced the **Transformer architecture**, which completely reimagined how text could be processed. The Transformer's main innovation is **self-attention**, which lets the model simultaneously evaluate all words in a sequence. Unlike RNNs, Transformers process text in parallel for greater **training efficiency** and better handle **long-range dependencies**. They use **positional encodings** to maintain word order without relying on recurrence.

Transformers unified the lessons of prior neural approaches — embeddings, encoder-decoder architectures, and attention — into a single, scalable framework that has become the foundation of modern AI. In short, the Transformer made it practical to train general-purpose, highly expressive language models at a scale never before possible.

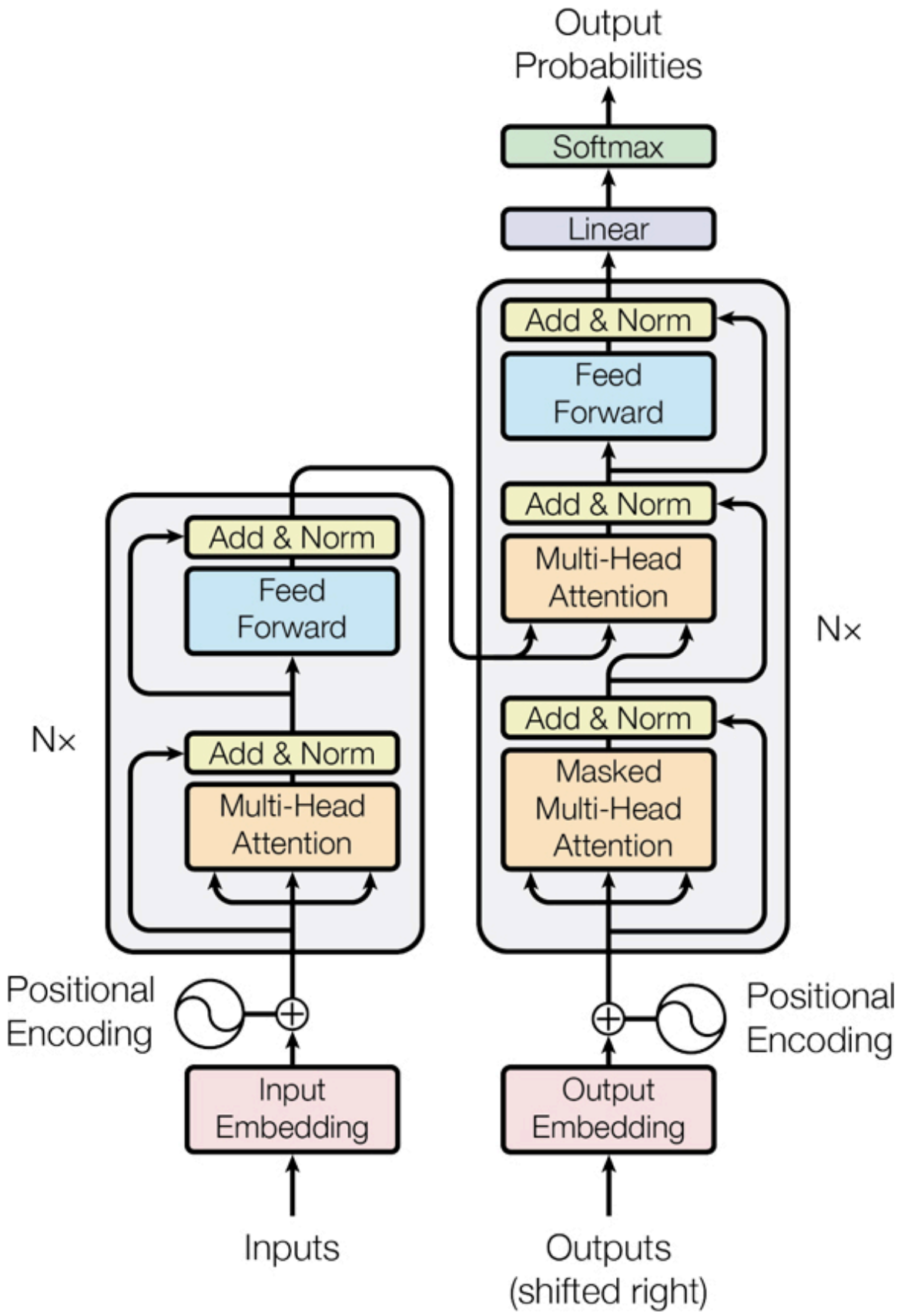

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Figure 1.2: *The Transformer - Model Architecture*
*(**Source:** — https://arxiv.org/pdf/1706.03762)*

The Tipping Point: ChatGPT

The release of ChatGPT in late 2022 marked a cultural turning point. Within weeks, it had reached millions of users and entered the mainstream conversation about AI. Unlike earlier models, ChatGPT combined raw language modeling power with **instruction tuning** and **Reinforcement Learning from Human Feedback (RLHF),** making it far more aligned with user expectations.

For many people, ChatGPT was the first tangible proof that AI could converse naturally, draft essays, write code, and even brainstorm creative ideas. Its viral adoption put LLMs at the centre of public debate and business strategy.

With this history lesson we are ready to dive into the core architecture of the contemporary LLM.

Architecture and Components of LLMs

Large Language Models such as GPT, Claude, and Llama are all built on the transformer architecture that we saw in the earlier section. While implementation details differ across providers, the underlying design remains consistent. At a high level, LLMs take text as input, transform it through a series of specialized layers, and then generate the most likely output word one by one. To understand how monitoring works later, it helps to break down both the architecture (the overall structure) and the components (the building blocks) that make up the modern LLMs.

Tokenization

The inputs to an LLM are not the actual text. Text is first broken into **tokens**—which are parts of words or sometimes even individual characters. Each token is then converted into a numerical ID. These IDs are mapped to dense vectors called **embeddings**, which serve as the mathematical representation of the input. In the following figure, we can see an example of how GPT 4o would tokenize the sentence "*You are reading this book called Ultimate LLM Monitoring using Langfuse"*. Notice how Langfuse is converted to three

tokens: **Lang, f,** and **use** since it is not a standard dictionary word. The numbered list shows the numerical ID for each token for GPT4o.

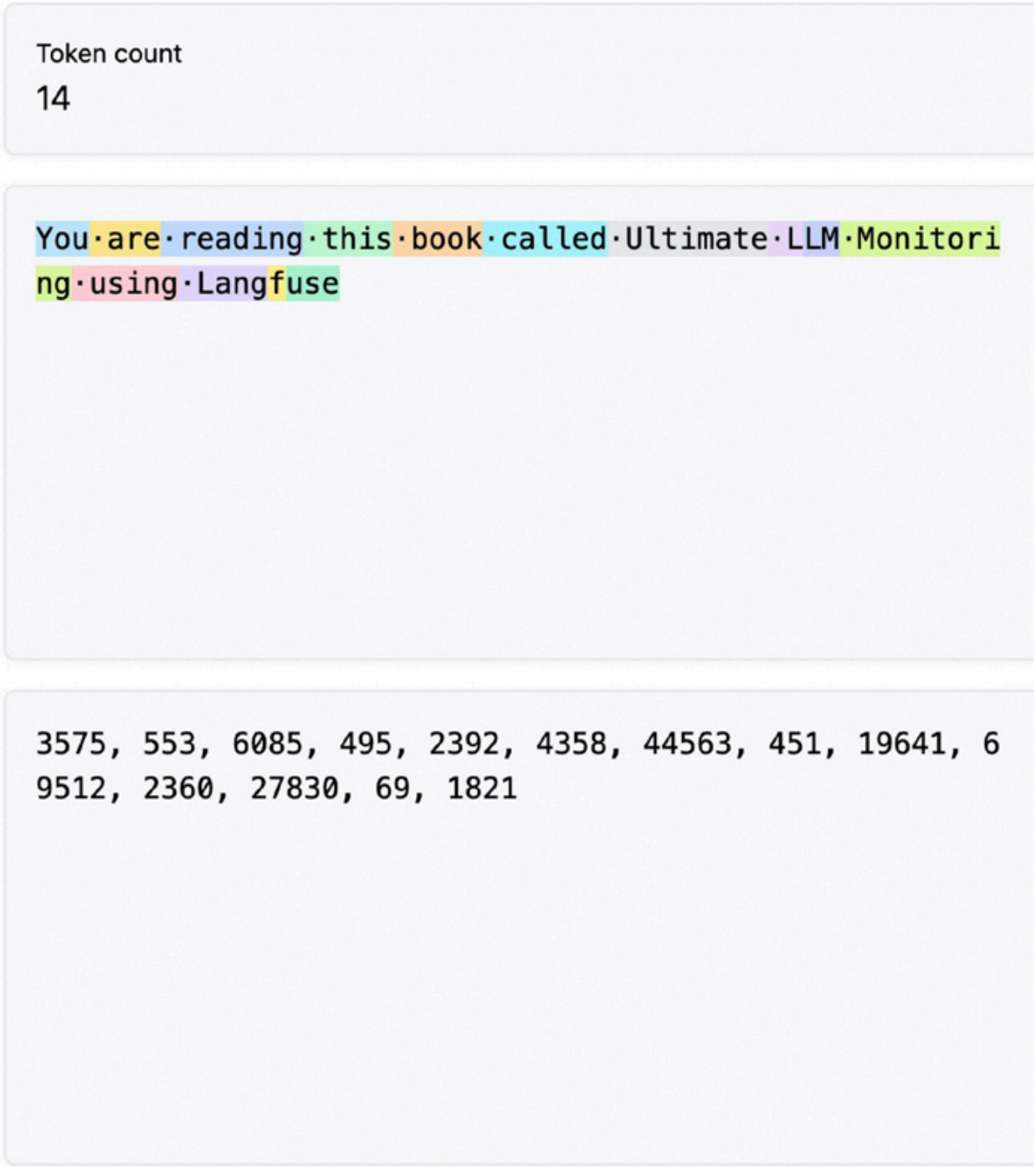


Figure 1.3: *Example of Tokenization*

Embeddings

Since computers only understand numbers, think of embeddings as the numerical equivalent of text that keeps its semantic meaning intact. This also helps the model to understand and compare meaning between texts.

- Instead of storing "You" as just text, the model represents it like:
  You as [0.12, -0.55, 0.89, …]
- "book" might be:
  [0.33, 0.41, -0.20, …]
- "Langfuse" might be:
  [0.02, 1.10, -0.75, …]

Embeddings capture **meaning** and **relationships**, not just exact words.

- Words with similar meaning are close together in vector space.

For instance, "book" and "novel" as vectors will be nearby.

- Words with different meanings are far apart.

This includes "book" versus "monitoring".

These numbers come from the training stage where words that appear in similar contexts get similar vectors. Different models could use different embeddings to store the same text. Embeddings are kind of a trade secret of the model and **not universal open knowledge**. The embedding for "Langfuse" in the embedding model of OpenAI is completely different from its embedding in a Hugging Face model. The numbers are meaningful only within the context of that trained model.

Positional Encoding

To retain word order, each embedding is combined with a **positional encoding**. This gives the model the ability to understand the difference between words depending on the order in which they appear in a sequence. As we just saw, in Transformers, every word is turned into an embedding (which is a list of numbers). But vectors for words do not say anything about order. So, we need positional encoding. Positional encoding are also numbers. These numbers come from **sine and cosine waves**. We add the two vectors (word embedding + positional encoding) element by element:

- **For "You" (Position 1):**

    - `Final vector = Embedding("You") + PE (1)`
- **For “are” (Position 2):**
    - `Final vector = Embedding("are”) + PE (2)`

… and so on, until “Langfuse”.

Now each word’s vector not only says what the word means, but also where it is in the sentence.

Multi-Head Self-Attention

The multi-head self-attention is the heart of the transformer architecture. It lets each word look at all the other words and decide which are important for understanding it. This is represented by the following equation:

$$Attention(Q, K, V) = softmax(QK^T/\sqrt{d_k})V$$

Each word creates three vectors: *Query (Q), Key (K), Value (V)*.

Similar to: “What am I asking about?” (Q), “What do I have to offer?” (K), “What info should I pass along?” (V).

Example:

- “reading” (Q) strongly matches “book” (K), so it pays a lot of attention to it.
- “Langfuse” attends to “LLM Monitoring”.
- “are” attends a bit to “You” (subject) and “reading” (verb).

We have multiple sets of Q, K, V so that the model learns different perspectives:

- One head might track grammar (“subject → verb → object”).
- Another might track the meaning (“LLM <-> Monitoring <-> Langfuse”).

After attention, the representation of each word is updated with info from the words it cares about.

Feed Forward Network (FFN)

The FFN is a small Neural Network applied to each word separately which helps transform features into more abstract ones.

Example:

- After attention, "book" knows it is connected to "reading".
- FFN turns that into a higher-level feature: "book-as-object-of-reading".

Think of this as polishing/refining the representation.

Add (Residual) + Normalization Layers

In this architecture, **residual + normalization** acts like the **quality control system** that ensures the assembly line never breaks down.

- **Residual:** It keeps the original meaning intact at every step.
- **Normalization:** This keeps numbers stable so the next layer can process them properly.

Without these, the response of the model would quickly become unstable. One can refer to the example of our previous sentence: *"You are reading this book called Ultimate LLM Monitoring using Langfuse."*

1. **Residual + Norm (First Time)** → This ensures that while "reading" learns from "book", it does not lose its original identity.
    - If attention overemphasized one word, normalization reins it back in.
2. **Residual + Norm (Second Time)** → Again, it prevents the FFN from distorting the signal too much.

This repeats through many layers. Each time, residual + norm guarantees that the meaning builds up gradually instead of exploding or vanishing.

Layer Stacking

If we recall the transformer diagram above, both the encoder (left) and decoder (right) stack the above layers multiple times because one pass isn't enough for the model to deeply understand language. Each layer refines and deepens the model's understanding.

- Lower layers capture local relationships (specifically "*reading <-> book*").
- Middle layers capture phrase-level (particularly "*called Ultimate LLM Monitoring*").
- Higher layers capture global meaning (such as "*The sentence is about a book related to Langfuse*").

After all the layers are applied to the text, it is sent to the final layer for the output.

Final Output Layer

This layer has two main operations:

- **Linear:** Projects vectors into vocabulary size of the model (for example, 50,000 possible words).
- **Softmax:** Turns them into probabilities ("next word is X with 92% chance").

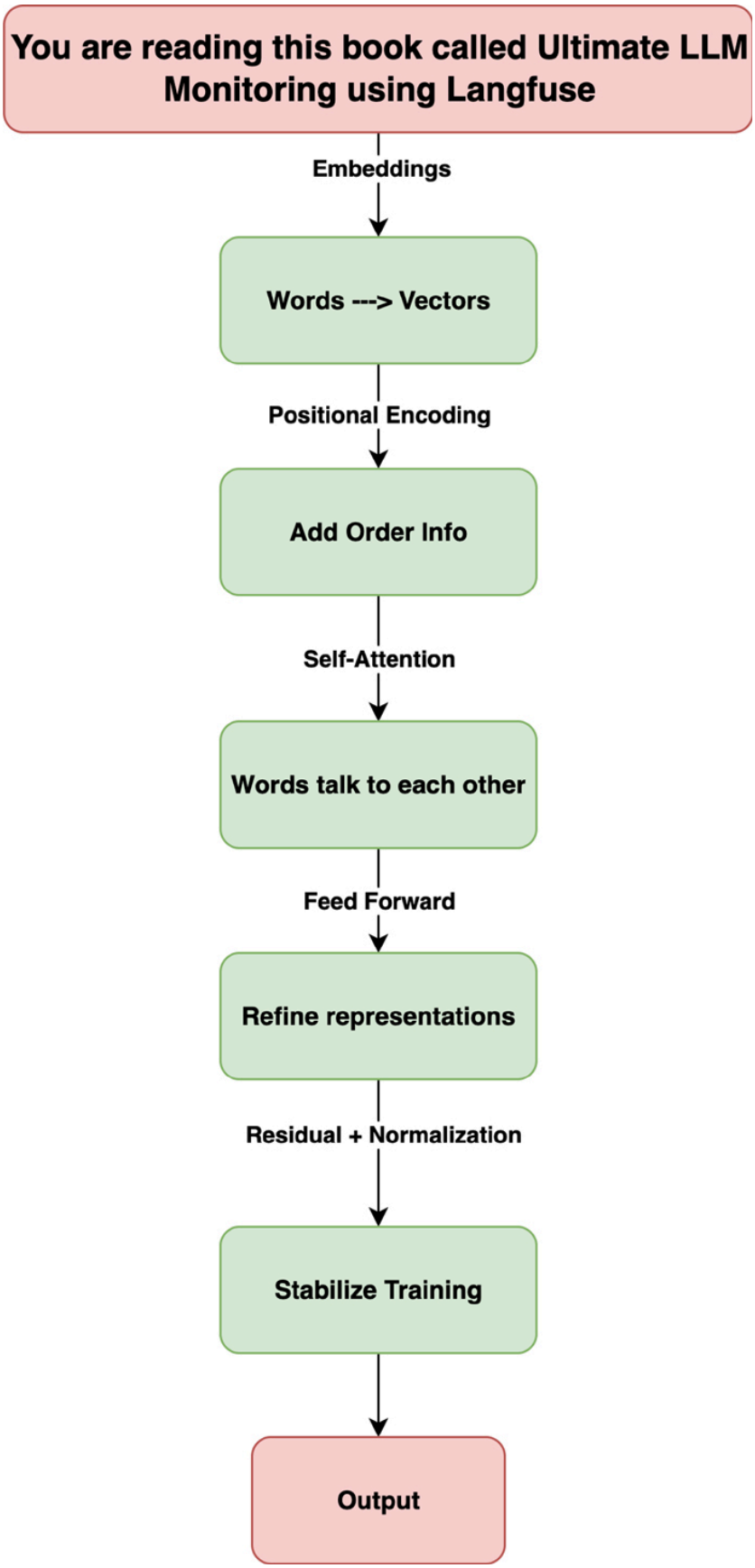

You are reading this book called Ultimate LLM Monitoring using Langfuse
Embeddings
Words ---> Vectors
Positional Encoding
Add Order Info
Self-Attention
Words talk to each other
Feed Forward
Refine representations
Residual + Normalization
Stabilize Training
Output

Figure 1.4: *Summary of LLM Architecture and Components*

Having reviewed the components of an LLM, let us now look at how modern LLMs are used in practice.

LLM Use Cases across Industries

Large Language Models are having a significant impact across various sectors of the society and technology. Before we examine the uses of LLMs in these fields, let's look at what LLMs can generally do today and then go in depth for some of the industry specific examples.

One of the most niche skills of an LLM is to perform NLP tasks at a scale and level which is either humanly not possible or would take weeks/months of effort. This includes: **summarization** of large texts across domains, finding **similarity** between texts, **translating** texts between different languages and **classifying** texts (for example, sentiment analysis). Of course, even when we don't have text to start with, LLMs can **generate text** too. The most notable example of this is **code generation**, which till date is one of the most widely used features of ChatGPT. Most LLMs are also **instruction-based**, meaning that users can simply prompt them to generate various types of text as they wish for.

Besides text, there are now hundreds or even thousands of models that can solve **Computer Vision** problems such as: **Image Classification, Object Detection, Image Segmentation, and** converting **Image-to-text** and vice versa. The list does not end with images; **audio classification, automatic speech recognition and text-to-speech** models are also now widely used in the industry.

One of the more recent advancements in LLMs is the development of so-called **Multimodal** models which can simultaneously work with more than one type of media for example: **Document Question Answering** which can help answer questions about the content of a document or **Image text-to-text** which can answer a question about an image.

As we can see here, the possibilities are endless and this list is only growing. Let us now look at some of the more common uses of LLMs today.

Content Generation

Since LLMs are advancements in the NLP field, one of their most popular uses is to generate textual content. This includes but is not limited to blog posts, product descriptions for e-commerce websites, social media content, and creative writing. After using ChatGPT or any similar app, we have experienced this firsthand. It's amazing how these tools can generate engaging and detailed content on almost any topic with only a few lines of prompting.

Content Generation does not stop at text only. Image generation using AI is now finding its way in every field too. Tools like **Midjourney** and **DALL-E** are able to generate high-resolution and artistic images by simply prompting them. The advancement does not end here either; video generation AI platforms such as **Veo** from Google and **Sora** from OpenAI have exhibited exceptional proficiency in producing realistic videos, all initiated through prompts.

Customer Support and Knowledge Management

AI-Powered **Chatbots** and Virtual Assistants are rewriting the rules of Customer Support systems. LLMs enable conversational agents that are adept at interpreting natural language queries and providing contextually appropriate responses. These chatbots provide **24/7 customer support**, reduce wait times, and can handle high volumes of repetitive queries.

Companies and e-commerce platforms deploy LLM-based chatbots for troubleshooting, account inquiries, and order tracking. These assistants can escalate complex queries to human agents seamlessly. In more advanced cases, LLMs can analyse incoming support tickets to determine urgency, category, and optimal agent assignment.

LLMs can also automatically create, update, and summarize knowledge base articles, FAQs, and documentation. Companies can then integrate AI tools that auto-suggest solutions from documentation when support tickets are logged. This technique along with the ability of LLMs to identify trends, sentiment, and recurring issues can feed insights to product teams to help them develop new features or address customer complaints quickly.

Software Engineering

Generative AI using LLMs has permanently transformed software coding. AI streamlines software development by cutting boilerplate, automating testing and documentation, and boosting team speed and productivity. LLMs can assist developers by suggesting code snippets, completing functions, or even generating entire programs from natural language descriptions. AI tools like **GitHub Copilot** and **Cursor IDE** act as intelligent coding partners, helping developers brainstorm solutions, debug faster, and learn on the fly.

LLMs can review codebases to detect bugs, recommend solutions, and provide explanations for error messages using straightforward language. LLMs can produce unit tests, integration tests, and edge cases based on code or requirements descriptions.

LLM-based tools like **vanna.ai** can generate SQL statements for databases simply by asking it questions in plain language. This makes data more accessible to people who are not SQL native, reduces dependency on specialists, and accelerates analytics workflows.

AI powered tools like **mintlify.com** can convert code repos into talking documentation in a matter of minutes. It can keep documentation updated and reduces the overhead in engineering teams who are always struggling to keep their code docs up to date.

Finance

LLMs are steadily transforming the financial sector. By streamlining research, enhancing compliance, and improving customer interactions, AI is becoming an essential part of how modern finance operates. The advent of LLMs in finance is not a recent thing. In March of 2023, Bloomberg announced **BloombergGPT**, a 50-billion parameter LLM aimed specifically for the finance industry. Bloomberg trained this model on its own data sources (one of the largest in the domain). This model outperformed other general purpose LLMs on financial tasks by a huge margin. While BloombergGPT itself is not a public product, Bloomberg integrates its capabilities into the **Bloomberg Terminal** and related services.

Organizations are employing artificial intelligence solutions to oversee billions of daily transactions, effectively identifying potential indicators of fraud. LLMs are reshaping Anti-Money Laundering (AML) and financial crime prevention by improving tasks that depend on language and unstructured data. They enhance sanctions and name screening with better

multilingual matching, streamline KYC by interpreting documents and spotting inconsistencies, and make adverse media monitoring faster through classification and summarization.

LLMs power robot-advisors that provide customized portfolio recommendations, retirement planning, and tax optimization. **However, it is important to note**—these models may provide simplistic or inaccurate suggestions because they do not possess financial judgment and may not consider every aspect of the financial circumstances or regulations of an individual. They are intended as decision-support resources rather than substitutes for certified financial advisors.

Healthcare

Generative AI with LLMs is starting to streamline healthcare administration, enhance patient communication, and assist clinical decisions. Though careful regulation is needed, early results suggest it can ease clinician workloads and boost patient outcomes.

Microsoft Dragon Copilot is an AI-powered clinical assistant made specifically for healthcare professionals. The tool can automatically generate clinical notes, draft referral letters, and create post-visit summaries from patient conversations.

LLMs enable professionals to rapidly access key information from medical databases and accelerate clinical decisions.

And even if someone is not a professional, LLMs can simplify complex medical terminology into clear, friendly explanations for information and educational purposes.

LLMs show promise in healthcare, but errors can be dangerous. They might hallucinate, overlook key details, or give misleading advice. Therefore, regulatory oversight and rigorous human validation are crucial.

Retail, E-Commerce, and Marketing

In all the industries we have seen so far, the impact of LLMs and AI is perhaps the most profound in the Retail, e-Commerce and Marketing space. This is also accelerated by the fact that online platforms are racing towards providing more and more AI features to their customers. Since this is such a broad domain there are limitless applications of AI here.

LLMs generate product titles, descriptions, and advertising copy designed to improve **Search Engine Optimization (SEO)**. This has massively reduced manual effort in content optimization and also gives a consistent tone to the catalogue of products.

The use of AI based chatbots is more prominent on e-tailers websites than anywhere else. From **personalized shopping** recommendations to query resolution, these chatbots are able to manage everything between converting customers to reducing call centre loads in the customer service department.

Behind the scenes, LLMs process customer reviews, feedback, and social media to detect trends, assess emerging needs, and observe changes in brand perception. This helps companies react quickly to changes and adapt their marketing strategies.

Thanks to multimodal LLMs, it's now even possible to search for similar products by uploading a photo of another product using visual search capabilities.

While LLMs are already transforming industries, deploying a system that uses them in real-world environments is far from straightforward. Understanding the challenges such as reliability, scalability, security, and ethical risks is essential and will be the focus of the next section.

Challenges in Deploying LLM Applications

Deploying LLM-based applications involves complexities that are distinct from those found in traditional Software-as-a-Service (SaaS) applications. While both types of applications share common concerns—like uptime, scaling, and security—LLM-based applications present unique challenges due to their reliance on **probabilistic models**, high computational requirements, and dynamic behaviour. Understanding these differences is critical for aiming to deliver robust and reliable LLM-powered solutions in production.

Non-Deterministic Outputs

As we have seen earlier, the output of an LLM is essentially a prediction (based on probability) of what the next word (or token) might be having seen the earlier words in a sentence. Considering that human language is vast, the answer to the question *"What should the next word be?"* has theoretically

infinite possibilities. This makes the very nature of the output of LLMs **non-deterministic.** This is different from something like an API which would usually give a deterministic response for the same query. If you have ever used ChatGPT or any other LLM- based AI app, you would have experienced this already. In most of these applications, business logic cannot be explicitly defined, and the possible output paths could be too big to explicitly handle every edge case. Applications such as AI chatbots, summarizers, or recommendation engines must include additional layers—such as output validation, ranking, or human-in-the-loop review—to ensure reliability. Additionally, large models sometimes generate outputs that appear reasonable but contain inaccurate or irrelevant information. **Hallucinations,** as they are called**,** continue to be one of the trickiest problems to monitor and mitigate in LLMs.

Infrastructure and Performance Requirements

Whether we choose to host our own model or rely on proprietary APIs like GPT, Claude, or Gemini, infrastructure considerations play a central role in determining cost, latency, and scalability.

If we are implementing open-source LLMs (such as Llama, Falcon, or Mistral) we will encounter significant operational challenges. These models require substantial GPU resources and specialized hardware with high memory capacity to deliver acceptable latency. Supporting multiple simultaneous users necessitates advanced orchestration strategies, including batching requests, managing GPU allocation, and distributing workloads across clusters. Scaling inference frequently involves investment in costly GPU infrastructure, distributed serving solutions, and technical optimizations such as quantization and model distillation. In the absence of these approaches, we may experience increased latency, reduced throughput, and escalating costs.

While proprietary APIs abstract away model hosting and GPU infrastructure, this does not eliminate performance challenges; it merely shifts them to a different layer of the stack.

- **Latency:** Even with optimized inference, **API calls can take seconds**. Applications that need real-time responses must adapt UX flows—showing typing indicators, streaming partial outputs, or managing user expectations.

- **Throughput and Limits:** Providers enforce strict **rate limits** and concurrency caps. If usage spikes, applications can hit ceilings on requests per minute or tokens per second, leading to degraded user experience. Scaling is not simply a matter of adding more servers; it requires careful workload management and sometimes direct negotiation with the provider.
- **Cost-Performance Tradeoffs:** Token-based pricing means that performance is inseparable from cost. Long prompts, multi-turn conversations, and rich outputs like images/videos drive up expenses quickly.

Observability Requirements

Observability for LLM applications involves tracking multiple aspects, including performance metrics, output quality, correctness, safety, and bias. Detecting drift or degradation typically requires ongoing evaluation using benchmarks or human-reviewed data. Monitoring outputs that may vary at scale presents operational challenges.

We will see in the next section the observability metrics that are important for LLM applications in detail.

To monitor these applications, teams often build **evaluation pipelines**: capturing model outputs in production, sampling them, and reviewing against benchmarks, custom metrics or human feedback.

In the later chapters, we will see how Langfuse helps in building these pipelines end-to-end.

Ethical, Safety, and Regulatory Concerns

The next challenge is quite unique to LLM applications. Outputs from an LLM may be biased, harmful, or offensive, even unintentionally. If left unmonitored, even a single bad interaction can undermine user trust or expose an organization to legal and reputational damage.

Safety features for applications must be implemented from the beginning of development and not added as an afterthought.

The concept of **guardrailing** is a common practice when designing applications with LLMs. *Guardrailing* refers to the techniques and systems

that constrain or monitor an LLM's behaviour to ensure safe, reliable, and compliant outputs. Guardrails filter responses from the model before sending them to the user. The filters can score responses on toxic or biased content and also implement some domain level or rule based constraints.

Guardrailing is also never perfect and cannot be used as a replacement for continuous monitoring of an LLM.

Across all these challenges—whether infrastructure complexity, latency and cost trade-offs, safety risks, or regulatory pressures—one thing is clear: **monitoring is indispensable**. Let us now look at some of the key metrics in LLM observability.

Key LLM Monitoring and Observability Metrics

Before we start monitoring, it is important to understand what metrics matter for an LLM-based application. It is evident that LLMs represent a significant advancement beyond previous technologies, necessitating the development of specialized metrics to accurately assess their performance. Having said that, the metrics we will see here are not an exhaustive list but the most useful and common ones that we will be inclined to use when working with LLMs. We will also see in later chapters how Langfuse can help to monitor and trace these metrics.

Latency

Latency in general refers to the total time between sending a request and receiving the **first meaningful** response. This metric is kind of a no-brainer when it comes to monitoring. Fast responses are crucial in any system and latency should be one of the first things we should monitor when working with an LLM-based application.

We have not discussed this in detail in the architecture section, but LLMs usually **stream** responses, which means they produce output word by word (or rather token by token). We may have noticed how ChatGPT shows words one by one when answering. This is nothing but streaming of the response.

In this context, latency for an LLM means the **time until the first token** is generated. This is different from the response time which is a general term used to measure the total time taken for the entire response to be seen. For an LLM, response time will be the **time until the last token** is seen.

The lower the latency, the faster the user gets the first feedback. Latency is usually measured in milliseconds.

Cost

Just like latency, cost is another basic metric that we would also want to track from the very beginning. In the case of LLMs, **every token counts** whether we use open-source models or the proprietary ones, cost is one thing we cannot ignore in our monitoring profile.

Most commercial LLMs like GPT, Claude or Gemini offer a *pay as you use* pricing model. Although the costs may seem trivial at first glance (for instance, GPT-5 costs $1.25 for a million input tokens and $10 for a million output tokens), these numbers can add up very fast. If we are developing an LLM application, it is not uncommon to try tens or even hundreds of different prompts across a handful of models. Add that up and we are spending real money even before we land on a working prototype.

Even if we choose to work with open-source models from Huggingface or Llama which do not cost by token usage, GPU inference costs could be high or even more costly, depending on the hardware we are using.

Since LLM usage is token based, costs will scale up with traffic. In such situations, even a slightly longer prompt could cause big expenses when multiplied by high no. of user hits. Long story short, cost monitoring is imperative for an LLM powered application.

Token Usage

Token usage and cost would most likely go hand in hand. Since cost is a direct function of token usage in most LLMs, by monitoring one of these metrics, we are implicitly monitoring the other one as well. Token usage like cost is also split between input token usage and output token usage.

However, there are some nuances where monitoring token usage independent of cost/latency can be insightful and necessary. By keeping our prompts the same, a jump in input token usage could indicate longer questions or queries by the users. This can also help detect unnecessary addition to context that might be sent to the LLM.

More output tokens mean longer inference time. Token usage can help explain spikes in latency/response times. In many cases, more output tokens

could indicate verbose responses by the LLM which could be counterproductive if the goal is to keep the output concise.

Prompt injection is an attack where a user deliberately crafts input text to manipulate an LLM into ignoring its original instructions or policies and instead follow the instructions of the attacker. The first sign of this would be an unusual number of input tokens.

In summary, token usage is not just about costs but could be an early indicator of efficiency and security of our application.

Hallucination

The metrics we have seen so far are quantitative in nature and in the case of LLMs, monitoring quality is the crux of ensuring reliability, trustworthiness, and real-world usefulness of the system.

Hallucination refers to instances where an LLM generates content that is **not grounded in factual information** or strays from the context provided. These fabricated or misleading outputs can undermine the reliability of our application, especially in scenarios where accuracy is critical. Hallucination is, till date, one of the biggest challenges for LLMs and while significant improvements have been made to mitigate it, a "Hallucination free" application is very unlikely. By knowing when or where hallucination occurs in our system, appropriate steps can be taken to prevent our LLM from hallucinating.

We will see in later chapters how an LLM itself can infer if the output of another LLM is exhibiting signs of hallucination and can warn developers even before this hits the ground.

Completeness

Completeness measures if the response of LLM fully covers every part of the query of the user, not just a portion. In many cases, missing information can be as harmful as incomplete information. Especially in domains like health, legal or finance, the response generated by an LLM should not leave anything behind or open the possibility for the user to use their imagination.

This metric is closely tied to accuracy, where accuracy answers the question, *Is the answer correct?* Completeness covers *Does the answer cover everything that was asked for.*

Relevance

While we want the LLMs response to be complete, it will not be useful if the answer is not focused on the query of the user. The basic definition of Relevance is whether the LLM answered the question without getting distracted by unrelated, unnecessary or off-topic content.

LLMs are trained on massive amounts of information so inherently they are very brainy. This makes them susceptible to being verbose in their responses and including more information than what was asked for.

This is not always desirable, especially when a concise answer is all that is needed.

Relevance is a crucial metric in LLM systems which are based on **Retrieval Augmented Generation (RAG)**. In RAG based apps, relevance is two-tiered:

1. *Were the relevant documents retrieved to answer the question?*
2. *Does the final response stay relevant based on the retrieved documents?*

If the retrieved documents are not relevant, the answer may be off. However, if the generation part hallucinates, the answer may still be wrong even if the retrieval was correct.

Correctness

Correctness is quite self-explanatory. It measures if the LLM's responses **are factually correct**. In a traditional Machine Learning system, we may use accuracy as a metric for this. However, for an LLM the concept of accuracy could be very broad or not even applicable in some cases. For example, if we ask ChatGPT: "*Help me plan a trip to New York city",* there is no accurate answer to this question, and responses could vary a lot.

This does not mean that correctness is not widely used for judging the LLMs performance. Correctness is often the **core quality metric** for evaluating LLM usefulness. If our application has the notion of a correct answer, or a ground truth, correctness can be simply measured by comparing the response of the LLM with the ground truth.

It is not uncommon for AI applications to have the concept of a **model answer** instead of a correct one. To understand this, we can consider an example: In a translation tool, a text could be translated in many different ways and all of them could technically be "correct". If an ideal translation is known, correctness can be measured by checking how close the generated response is to this desired answer.

Although **Hallucination**, **Completeness**, **Relevance**, and **Correctness** are quality metrics, they can be quantified into numeric representation to compare, monitor and evaluate an LLM. We will see in *Chapter 7, Evaluating LLMs in Langfuse*, how the LLM-as-a-judge feature of Langfuse can help doing exactly this. Before we jump to the next section, let us see a short summary of these metrics:

Metric	Short Definition	When to Use
Latency	Time to first/last token	When user experience depends on fast responses (such as chat, search).
Cost	Money spent per request/token	When scaling usage and keeping AI expenses predictable is critical.
Token Usage	Number of tokens consumed	When optimizing prompts or detecting inefficiencies.
Hallucination	Incorrect/fabricated information	When factual reliability and trust are crucial (healthcare, legal, finance).
Completeness	Coverage of all required aspects	When tasks require full coverage of instructions (summaries, lists, reports).
Relevance	Staying on-topic to the query	When answers must stay aligned with user intent or retrieved context (RAG).
Correctness	Factual accuracy of the answer	When factual truth matters.

Table 1.1: *Summary of Key LLM Metrics*

Introduction to the LLM Application Lifecycle

Before we jump into monitoring, let us try to understand some of the common stages that an LLM application might go through. Developing an LLM-powered app involves more than prompt-response exchanges; it needs a structured approach that blends testing, solid engineering, user focus, and ongoing enhancement. Although there is **no one universal lifecycle** for this kind of application, the succeeding figure shows the typical steps that LLM

applications could follow. We can see how Langfuse can be used in almost all of these stages. In later chapters, we will go into more details about each of these topics.

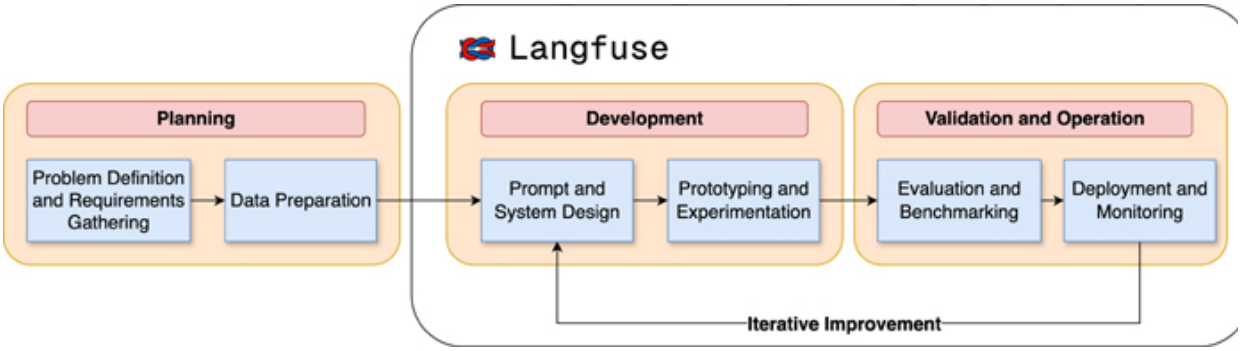


Figure 1.5: *A Typical Lifecycle of an LLM Application*

Problem Definition and Requirements Gathering

This stage focuses on defining the problem the application will address, ensuring it aligns with business or user needs. Teams set scope, success criteria, and constraints like latency, cost, compliance, and so on. At this stage, we also need to assess If an LLM is the right tool for this problem. Sometimes a traditional ML model or even a SaaS application like an API could be enough. In these cases, an LLM could even be an overkill.

A good rule of thumb in deciding whether our solution needs an LLM is, if our problem involves **understanding or generating natural language in a flexible way**, and can tolerate some uncertainty if outputs are monitored, validated, or improved with continuous iterations. If the problem is highly deterministic, structured, or safety-critical, a non-LLM solution should be our first choice.

A clear **problem statement and success metrics** are the expected outcome of this stage.

Data Preparation

LLMs rely heavily on the **quality and relevance of data provided**. The approach to data preparation differs depending on whether we are working with a foundational/self-developed LLM or a proprietary, API-based model like GPT from OpenAI.

For Foundational Models:

- **Large-Scale Dataset Collection**: We need vast, diverse, and domain-relevant corpora to train or fine-tune the model.
- **Cleaning and Pre-Processing**: Remove duplicates, normalize, and remove low-quality or harmful text. Ensure consistent formatting.
- **Domain Adaptation**: Curate specialized datasets based on domain-specific performance.
- **Bias and Safety Filtering**: Proactively filter sensitive or biased content, since the model inherits patterns from the training data.

When Using Proprietary LLMs (like GPT):

- **Prompt-Focused Preparation**: Instead of large-scale training, focus on curating high-quality input examples that represent real user queries.
- **Grounding Knowledge**: Organize domain-specific data (databases, documents, FAQs) for Retrieval-Augmented Generation (RAG) if applicable.
- **Few-Shot Examples**: Prepare representative demonstration prompts (few-shot examples) to guide the model's behavior.
- **Validation Datasets**: Build evaluation sets to test how well the proprietary LLM uses our data during prompting or retrieval.

Prompt and System Design

The prompt is essentially one of the core components in an LLM application. It is the primary input that the LLM needs to produce any output. This makes prompt designing critical to the overall functioning of the application.

Typically, designing the prompt(s) for our application would be a major chunk of effort in the overall lifecycle. **Prompt Engineering** involves designing inputs for LLMs to shape their responses. Well-crafted prompts can significantly enhance the accuracy, quality, and consistency of outputs without altering the model itself. Prompts are often broken into **system messages** (defining overall behavior), **user messages** (the task at hand), and **few-shot examples** (demonstrating the desired format or reasoning).

In theory, there are an infinite number of ways in how a prompt can be written, which makes it difficult to find the perfect prompt that will maximize the output quality of our application. Prompt design needs to be

tightly coupled with experimentation, evaluation and benchmarking to iteratively reach the best possible outcome over several iterations. In modern applications, it is very common to even version control prompts like code and maintain a log of success metrics for each of the versions.

Since Prompt Engineering is a huge subject by itself, we will not go into the details here. In *Chapter 7, Evaluating LLMs in Langfuse*, we will see how Langfuse enables **prompt management** and versioning which allows us to use prompts directly in applications without having the need to maintain prompts in code.

System Design principles apply to LLM applications like they apply to any software-based system. As the name suggests, System Design defines the **architecture** for a system. LLM applications may be a new genre in application development, but foundationally they are governed by the same pillars that go into designing any robust, scalable and maintainable system which delivers quality output as required by its users. These include but are not limited to:

- **Modularity:** Breaking down a system into smaller, independent, reusable components.
- **Scalability:** Systems should perform reliably under increasing loads.
- **Availability:** The system should be available and accessible when needed.
- **Reliability:** The system's ability to give consistent results over time.
- **Performance:** The speed and responsiveness of the system.

Here are some key design decisions that are specific to the current LLM applications:

1. **Data and Context Handling:**
    - How will external knowledge be integrated — via Retrieval-Augmented Generation (RAG), fine-tuning, or API calls?
    - What caching or pre-computation strategies can reduce repeated calls and cost?
2. **Reliability and Safety:**
    - How will inputs/outputs be validated?

- What moderation or filtering layers are needed to handle unsafe, biased, or irrelevant content?

3. **User Experience:**
    - Should the application return streaming responses for faster perceived performance?
    - How will the system handle ambiguous queries?
4. **Scalability and Cost Efficiency:**
    - Will smaller models handle simple queries, reserving larger models for complex ones?
    - Can batching or rate-limiting be used to optimize performance at scale?
5. **Monitoring and Feedback Loops:**
    - What metrics should be logged (latency, token usage, cost and so on)?
    - How will user feedback feed into prompt refinement or fine-tuning?

Prototyping and Experimentation

With the previous stage we are now already in the iteration phase. Once the problem is defined, the data prepared, and the initial prompt/system design outlined, the next step is to validate assumptions through prototyping. In LLM application development, prototyping is about testing how the model behaves in realistic scenarios and uncovering gaps before committing to a final solution.

Prototyping should typically be done in an **offline** fashion, that is, before going live with an iteration of the application. It's more efficient to test prompts, retrieval pipelines, and outputs in lightweight experiments than to build full systems upfront. Prototypes are able to simulate interactions and provide initial feedback on the usefulness, reliability, and quality of responses.

What Experimentation looks like:

- **Prompt Comparison:** Compare different types of prompts with different structures and styles.
- **Models:** Test different LLMs (some of which includes GPT, Llama, Claude) to balance cost, speed, and accuracy.
- **Context Strategies (if Applicable):** Evaluate how different knowledge sources impact performance.
- **Output Comparison:** Try different output (JSON schemas, bullet points, summaries and so on) to see what yields the best result in terms of usability.

Once we have some evidence that the prototype works, we can move onto evaluation, otherwise we refine the prototype.

Evaluation and Benchmarking

Now that we have a prototype, it is time to see how it performs in a real-world like environment. The success metrics we defined in the first stage form the backbone of the evaluation and benchmarking stage. In the Data Preparation stage, we also have our validation dataset which will be used to make equal comparisons of different evaluation stages.

It is important to re-iterate that choosing the **right success metrics** is crucial. Building the right application but tracking the wrong metrics could lead to a final product that does not fulfil the user's needs and cause unnecessary wastage of resources due to impeding rework.

While automatic evaluation may be the fastest and the most scalable option if the volume of the test is huge, it might also be the perfect time to get **Human-in-the-loop** at this stage. Depending on the application, a human evaluation could be necessary if the system is being designed for a critical domain like health, finance or legal. Business needs and security requirements could also mandate this.

The perfect blend of automatic and human evaluation can also be achieved where the first round of evaluations is automatic (sort of a screening based on basic metrics like cost, latency and so on) and the then a subset of this sample is sent to human review for checking things like factual correctness, compliance, ethical safety and so on. This dual level of evaluations is also a trademark feature of Langfuse, which we will cover in detail in *Chapter 8*.

A basic framework for an evaluation and benchmarking report could look like this:

1. **Comparison of Options:** Cross-checking which combination of prompt, model, context and retrieval strategy (if applicable) gives the best result for each of the success metrics.
2. **Balancing Trade-Offs:** It is likely that no option ranks number one in all success metrics. For example, the best performing model for correctness might also be the most expensive one or the prompt that makes the LLM hallucinate the least is also the slowest of the lot. In these cases, the winning combo could be selected by objectively prioritizing the success metrics depending on the needs of the system.
3. **Errors and Failures:** Try to identify reasons (if) and why certain options failed. Does a model hallucinate a lot? Is a certain prompt never leading to a correct answer? Are irrelevant documents being picked up for RAG. This kind of analysis helps uncover system design/prompt design problems and also influence choice of models.
4. **Recommendations:** Decide on the best configuration for deployment based on the metrics and trade-offs. Have a mitigation strategy in case the best option does not work in production.

Deployment and Monitoring

Once our application has been evaluated and chosen for production, the focus shifts to deployment and monitoring. This stage is not the end of the lifecycle but the beginning of the system's real-world journey, where it is monitored and tested continuously.

While our prototype was being run in offline mode, our actual application will be in **online** mode. The first thing to have in place for an online application is **Observability.** It means we capture interactions between the system and the user, set up alerts for application downtime and monitor all critical system parameters in real time.

Online evaluation should at least cover evaluating all success metrics for all requests or a sample of request (say 20%) if the volume is too huge. In a live application, it's also possible to monitor and gather real user feedback in addition to preset metrics. User feedback is the most valuable take away at this stage.

This feedback and evaluation complete our loop of a single iteration of the lifecycle of LLM application. Based on the results there could be several more rounds of Development/Validation/Deployment; each of which lead to a better end product.

In the next and final section of this chapter we will introduce the frameworks and tools that we will be using throughout this book. This will include some basic introductions and code blocks to help us get familiarized with integrating Langfuse in our LLM applications.

Exploring the LLM Toolkit for the Book

Now let us explore the frameworks and libraries we will use throughout the book. The LLM development landscape changes rapidly and there are new tools and frameworks almost every day. Our focus in this book is Langfuse which itself supports dozens of integration options but to keep the content on subject, we chose a subset of libraries to work with. This will keep things concise but also open up opportunities to experiment with a different tech stack. We will try and stick to Open-Source Software (OSS) where possible so that readers can freely learn about Langfuse without having to make monetary investments upfront.

Our choice of programming language in the book is **Python 3.12**. Although Langfuse also supports JavaScript/TypeScript, we will not be covering them in the book. Code examples shown in the book are easily adaptable to JS/TS with minimal effort. Having said that, readers should be familiar with coding in Python and install Python 3.12 as a pre-requisite. We will be working with virtual environments in python in this book. Official documentation is here: **https://docs.python.org/3/library/venv.html**. *We suggest creating a single virtual environment for the entire course of the book.*

OpenAI SDK

Our first library is the OpenAI Python SDK. The creators of ChatGPT have created this SDK to offer programmatic access to the underlying OpenAI REST API. It uses simple http-based calls and is the easiest way to get started with GPT based models.

We will start by creating a virtual environment to work with. In the terminal or shell, run:

```
python3 -m venv /path/to/environment # for Linux/Mac based systems
python -m venv C:\path\to\environment # for Windows
```

Then activate the environment by running:

```
source /path/to/environment/bin/activate # for Linux/Mac based systems
C:\> path\to\environment\Scripts\activate.bat # for Windows
```

To install the Open AI SDK in a Python application, use the command (in terminal/shell):

```
pip install openai python-dotenv
```

We will also install another python package called **`python-dotenv`** which is used to load environment variables. This is a pretty standard way to set up variables like API keys in the application. The package can be installed from here - **https://pypi.org/project/python-dotenv.**

This OpenAI API is a *pay as you go* service, and we need to buy credits before we can get started. We will also need an API key and purchase credits to use this model. We can do both things here - **https://platform.openai.com/settings/organization/apikeys.**

The API key needs to be added to a file named "**`.env`**" in the root directory of the project like this:

```
OPENAI_API_KEY=<your_api_key>
```

After this, we can start using the API client to make calls to GPT.

```
from dotenv import load_dotenv
load_dotenv()

from openai import OpenAI
client = OpenAI()

response = client.responses.create(
  model="gpt-5-nano",
  input="Write a short introduction about Langfuse, the LLM
  Engineering Platform"
)

print(response.output_text)
```

This will print an output similar to the following code (although the response could possible vary):

Langfuse is an LLM engineering platform designed to help teams build, deploy, and govern production-ready language-model applications. It provides end-to-end tooling for prompt and chain management, evaluation, monitoring, and governance—so you can ship faster with confidence. With versioned prompts, test suites, real-time observability dashboards (latency, accuracy, errors, costs), and guardrails, Langfuse makes it easier to iterate on models while maintaining safety and reliability. It integrates with major LLM providers and data systems, enabling seamless collaboration across engineering, product, and data science teams.

In this basic example, we have used the GPT-5-nano model, but we can use any of the GPT models supported by Open AI.

Another common usage of this SDK is to use the **Chat Completions API** which simulates a conversation like ChatGPT. We will use this API frequently in the book.

```
completion = client.chat.completions.create(
  model="gpt-5-nano",
  messages=[
    {"role": "system", "content": "You are an expert in Langfuse,
    the LLM Engineering Platform"},
    {
      "role": "user",
      "content": "What are the top 3 features of Langfuse?",
    },
  ],
)
print(completion.choices[0].message.content)
```

Output:

Langfuse is an LLM engineering platform that focuses on production-grade management of LLM services. The top features users commonly highlight are:

- End-to-end LLM orchestration and lifecycle management
- Unified workflow for deploying, routing, and versioning prompts and models
- Fine-grained control over routing policies, multi-model ensembles, and A/B tests

- Observability baked in metrics, traces, and dashboards for prompt variants, latency, and accuracy
- Deployment, monitoring, and reliability tooling
- Seamless deployment to multiple environments and providers with versioned models
- Real-time monitoring and alerts for latency, errors, and model drift
- Proven reliability features like retries, fallbacks, and circuit breakers to keep production endpoints healthy
- Observability and measurement at scale
- Detailed telemetry for prompts, completions, and user interactions
- Evaluation hooks to compare model performance, safety, and alignment across versions
- Data-driven optimization through dashboards, dashboards, and event-level insight to improve prompts and routing logic

Although this SDK has a lot of other features, we will not cover them in detail. The official GitHub page for the SDK is here: **https://github.com/openai/openai-python**

LangChain

We are going to be using **LangChain (https://www.langchain.com/langchain)** as the main development framework in this book. LangChain is an open-source, orchestration framework that provides a standard interface for every model, tool, and database. **LangChain** provides so-called **chains** which are sequences of steps where the LLM processes input, calls tools or APIs, and generates outputs. It works very well for retrieval-augmented generation (RAG) applications, chatbots, AI Agents, and many other types of LLM-based workflows.

One of its most powerful features is the **`plug and play`** of components which makes it very easy to swap models, databases or even prompts without re-writing the whole application.

To Install:

```
pip install langchain langchain_openai
```

To perform exactly the same operation as the Chat Completions API, the LangChain equivalent would be:

```
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
load_dotenv()

model = init_chat_model("gpt-5-nano", model_provider="openai")
model_response = model.invoke(
  input=[
    {
      "role": "system",
      "content": "You are an expert in Langfuse, the LLM
      Engineering Platform",
    },
    {
      "role": "user",
      "content": "What are the top 3 features of Langfuse?",
    },
  ]
)
print(model_response.content)
```

The **`init_chat_model`**, method from LangChain automatically takes care of creating the appropriate model client from the provider's implementation (in this case OpenAI SDK). The **`invoke`** method will also format the prompt messages into the required structure as needed by the OpenAI API.

Hypothetically speaking, if we wanted to switch GPT with another model say, Google Gemini, this could be done very easily in LangChain with just one line of change:

```
model = init_chat_model ("gemini-2.5-flash",
model_provider="google_genai")
```

The rest of the code will remain the same. We will additionally need to provide our **`GOOGLE_API_KEY`** instead of OpenAI API key. This swapping also works for most components in the chain and makes LangChain as one of the developers' favorite libraries to build LLM applications. Many of the examples we will see in the book will also use LangChain. **LangChain is**

not a pre-requisite for this book, and we will cover the necessary learning as we go along.

Langfuse

Finally, we will be using a lot of **Langfuse (https://langfuse.com/)** as this is the main subject of the book. Langfuse is an open-source **LLM Engineering Platform** for LLM-based applications. It provides developers and ML engineers with powerful tools for tracing, prompt management, evaluation, and performance monitoring—all in one place.

We have made several mentions of Langfuse throughout the book, most notably in the section about the LLM application lifecycle where we saw how it can be used for the full development, evaluation, benchmarking and monitoring flow. Langfuse is also compatible with any LLM and supports a variety of integrations.

We will not cover any code examples in this section, but rest assured, they will be covered in much detail starting from *Chapter 4, Introduction to Langfuse*.

The next couple of chapters will deep dive a bit more into the conceptual parts of monitoring of LLMs and topics like Model Drift and Bias, setting the stage for the sections of the book dedicated to Langfuse.

Conclusion

This chapter introduces Large Language Models (LLMs), tracing their evolution from early NLP systems like ELIZA to modern Transformer-based architectures. It details the components and industry applications of LLMs, explores deployment challenges—including performance, cost, and ethical risks—and highlights the necessity of monitoring key metrics like latency, cost, token usage, and output quality. The chapter also outlines the LLM application lifecycle and presents essential tools such as OpenAI SDK, LangChain, and Langfuse for effective development and observability. In the next chapter, we will deep dive into monitoring concepts for LLMs including the core principles and various dimensions of monitoring. We also see how we can create a robust monitoring strategy for typical LLM applications.

References

- **Improving Language Understanding by Generative Pre-Training**: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
- **ELIZA - A Computer Program for the Study of Natural Language Communication Between Man and Machine**: https://engineering.purdue.edu/~ece495nl/protected/weizenbaum1966.pdf
- **N-gram models**: https://towardsdatascience.com/introduction-to-language-models-n-gram-e323081503d9/
- **Attention is All You Need**: https://arxiv.org/pdf/1706.03762
- **NLP Tasks**: https://huggingface.co/tasks
- **Vanna AI**: http://vanna.ai
- **Mintlify**: https://mintlify.com
- **Bloomberg GPT**: https://www.bloomberg.com/company/press/bloomberggpt-50-billion-parameter-llm-tuned-finance/
- **Fraud detection in Finance**: https://www.businessinsider.com/mastercard-ai-credit-card-fraud-detection-protects-consumers-2025-5
- **Microsoft Dragon Copilot**: https://www.microsoft.com/en-us/health-solutions/clinical-workflow/dragon-copilot
- **Open AI SDK**: https://github.com/openai/openai-python
- **LangChain docs**: https://python.langchain.com/docs/introduction/
- **Langfuse**: https://langfuse.com/