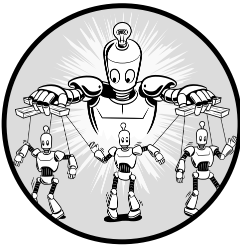
The Developer's Guide to AI (Sample Chapter) © 3/31/26 by Jacob Orshalick, Jerry M. Reghunadh, and Danny Thompson

1

UNDERSTANDING LARGE LANGUAGE MODELS



Large language models (LLMs) are very large, deep learning models that are *pretrained* on vast amounts of data. The sheer amount of publicly available data on the internet makes this training possible. Data scientists combine this data with licensed sources and curated datasets to train LLMs to understand patterns and relationships in natural language.

Web pages, books, articles, research papers, code, and more form the foundation for LLM training, illustrating how words connect to create sentences, ideas, and concepts (Figure 1-1). By training AI models on massive amounts of text, researchers discovered that these systems could develop an inherent understanding of language, accumulate knowledge, and even demonstrate elements of reasoning.

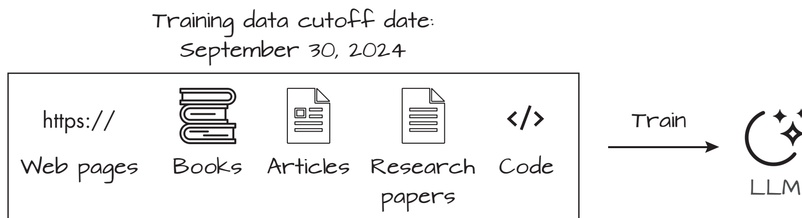


Figure 1-1: LLMs are trained on vast amounts of publicly available and licensed data.

This was the major breakthrough in the field of natural language processing (NLP) when Google researchers released Bidirectional Encoder Representations from Transformers (BERT) in 2018. Table 1-1 shows how the amount of data used to train LLMs has steadily increased over time. Unsurprisingly, research has shown that more training data is better.

Table 1-1: Data Used to Train Some Well-Known LLMs

LLM	Year	Amount of training data	Training data sources
BERT	2018	≈ 3.3 billion words	BookCorpus (800 million words), English Wikipedia (2.5 billion words)
GPT-3	2020	≈ 300 billion tokens	Common Crawl, WebText2, books (multiple sets), Wikipedia
LLaMA (Llama 1)	2023	≈ 1–1.4 trillion tokens	Publicly available web data and curated corpora (Common Crawl, GitHub, Wikipedia, books, and so on)

The computing costs to train your own LLM are estimated at anywhere from hundreds of thousands to millions of dollars, depending on the size of the model. Fortunately for us, companies are racing to release more powerful proprietary models (these cost money to use) and open weight models (similar to open source) every day.

Thankfully, we are being given access to all these AI models through something all developers know how to use: APIs and SDKs.

THE BERT BREAKTHROUGH

Before Google's BERT, NLP models had difficulty understanding complex sentence structures. Researchers at Google started to experiment with masking random words in text and training a model to guess the missing words. The model learned to pay attention to the entire sentence, looking at the words before and after each blank.

Researchers then swapped the order of random sentences and had the model guess whether the sentences followed one another. With this training, BERT began to handle tasks like answering questions and summarizing information with impressive accuracy. The published results of BERT drove incredible advances in NLP, eventually leading to today's LLMs.

Understanding Text Generation

Anyone who has used ChatGPT knows the inputs and outputs of an LLM. We *prompt* the LLM with questions, instructions, and examples, and it generates a *response* (Figure 1-2). The quality of the response generally depends on the quality of the prompt (see Part II).

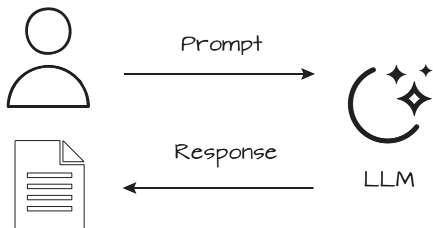


Figure 1-2: Prompts in natural language are the inputs to LLMs.

The model predicts the word most likely to appear next in the sequence, over and over, until the response generation is complete. Each word is referred to as a *token*. Tokens are not exactly words, but it is easy for us to think of them that way (Figure 1-3).

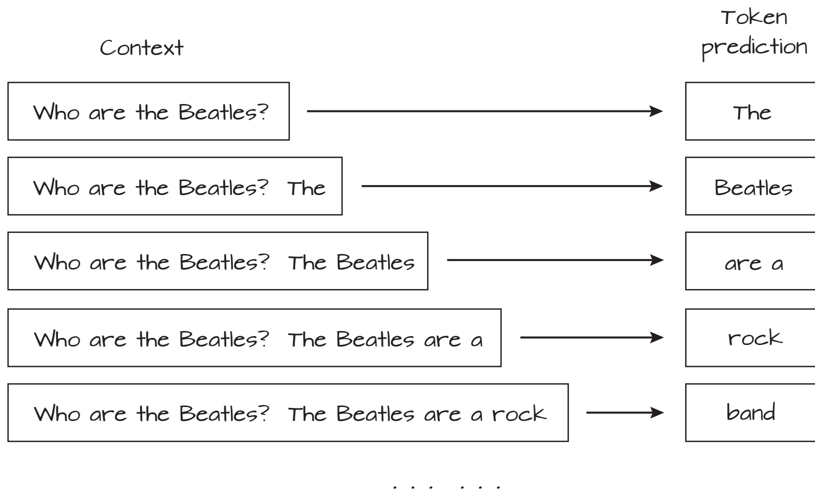


Figure 1-3: LLMs generate responses by predicting a sequence of tokens.

As you can see, the prompt provides the initial *context* for the completion that will be generated by the model (in this case, "Who are the Beatles?"). The model then predicts the *most probable* next token and adds it to the context for the next prediction until a complete response is generated. This step-by-step prediction process is called an *auto-regressive* approach. Each new token is generated based on all the tokens that came before it.

The process of an AI model predicting a response when given new, unseen data (a prompt in the case of an LLM) is referred to as *inference*. It's

why AI models are so useful. Once they've been trained, they can use what they've learned to *infer* a response to new input. This ability to generalize is something that humans easily do but that traditional programming has never been able to achieve.

NOTE *You don't have to understand the architectural underpinnings of LLMs in order to use these models. We'll introduce you to the concepts and technical details you need to build production-ready systems with these models. But if you're curious and really want to understand the transformer architecture that supports nearly all modern LLMs, start with Ashish Vaswani et al.'s paper titled "Attention Is All You Need," which introduces the core ideas behind transformers and remains the foundational reference for anyone looking to dive deeper into how models like GPT and BERT work.*

AI vs. Traditional Logic

Traditional programming is based on Boolean logic. It's deterministic. We know that once we've tested our code, a given input will result in an expected output. Code is reliable and programmed to perform a very specific task.

This can also be a drawback. Our software is limited to the tasks we explicitly program it to accomplish. It can't adapt to new scenarios. Coding our solution correctly often means that an unexpected input results in either some type of validation failure or an error.

AI models are different. They are trained to handle complex rules and are extremely flexible. But this comes at a cost: The results you get with any given input are probabilistic. This means that sending the same prompt to an LLM twice may yield slightly different results.

If you were to ask ChatGPT, "Who are the Beatles?," you would get a response with members of the band, chart-topping songs and albums, and their legacy. If you start a new conversation with ChatGPT and ask the same question, you would get a similar but slightly different response.

This is a key starting point when deciding whether an AI model makes sense for your specific use case. For example, if you're using an LLM, you may need to be able to tolerate variability in the response. Even with a well-constructed prompt, token generation is still based on probability.

Also, remember that testing may prove that your AI model responds correctly 99.9 percent of the time. That may be more accurate than a human at a particular task, but 0.1 percent of the time, AI is still wrong. You need to consider the impact of a wrong answer and how it would be handled.

WHEN WE NEED A HUMAN IN THE LOOP

Always keep in mind that having a human involved in AI decision-making may be desirable or even necessary. This is referred to as having a *human in the loop*. The more critical the decision, the more likely a human should verify an AI response for accuracy.

> A life-critical system is an obvious example. If there is even a small probability that a decision could lead to someone's death or severe injury, a human in the loop is almost certainly required.

The Versatility of LLMs

If you've used ChatGPT, you've already seen how flexible these models are. You can ask it to draft an email, summarize a long article, or suggest ideas for your next trip, all with plain-language instructions.

Under the hood, the model wasn't trained for your exact request. First, it's pretrained on diverse text to learn a general, task-agnostic skill: predicting the next token. Then it's instruction-tuned with examples so it learns to follow directions and produce helpful, well-formatted answers when prompted. With a clear prompt (and sometimes a couple of examples), the same model adapts to many tasks, no retraining required.

You can use these models for a wide range of practical tasks:

- Writing and editing (email, drafts, and rewrites)
- Summarizing (articles, reports, and transcripts)
- Classifying and extracting (tagging, pulling key fields)
- Translating (between many languages)
- Reasoning (explaining steps, outlining approaches)
- Coding (snippets, refactors, and tests)
- Performing as chatbots and AI assistants

You can even find LLMs that have been fine-tuned to better handle these and other specific tasks.

If this seems overwhelming, don't worry. We'll walk through examples and prompting techniques for all these use cases and more throughout the book.

Understanding the Limitations

The panacea of AI research is creating a truly general-purpose artificial general intelligence (AGI). LLMs are not AGI, at least not yet. LLMs are incredibly powerful in their ability to understand what we are asking and to generate a response, but they have limits.

The limitations introduced in this section are not intended to be comprehensive. We'll continue to discuss limitations and demonstrate strategies to address them through the book. Just don't fall into the trap of thinking that LLMs will always be your solution. As the saying goes: *When all you have is a hammer, everything starts to look like a nail.*

Our job as developers is to make informed technology choices based on the specific business problem we're solving. We need to understand the tools available, their limitations, and the trade-offs that come with them.

Some problems simply aren't a good fit for LLMs. The same can be said for any technology choice.

Limited Training Knowledge

An LLM's built-in knowledge is limited to the data it was trained on (Figure 1-4). The training data is bound by a cutoff date. The model does not know facts that arose after the cutoff, nor does it know anything about your private data unless you provide that information in the prompt. The model can produce confident but outdated or incomplete answers, especially on recent news, proprietary information, or highly specialized topics.

Figure 1-4: The knowledge of an LLM is limited to its training data.

You can work around this constraint by providing the missing context within your prompt at inference time. This is referred to as *AI grounding*.

In our AI support agent example, the LLM being called had no knowledge of the product because it wasn't trained or fine-tuned with that information. If the team had a knowledge base of how-to support articles, they could include them in the prompt to properly ground the response (Figure 1-5).

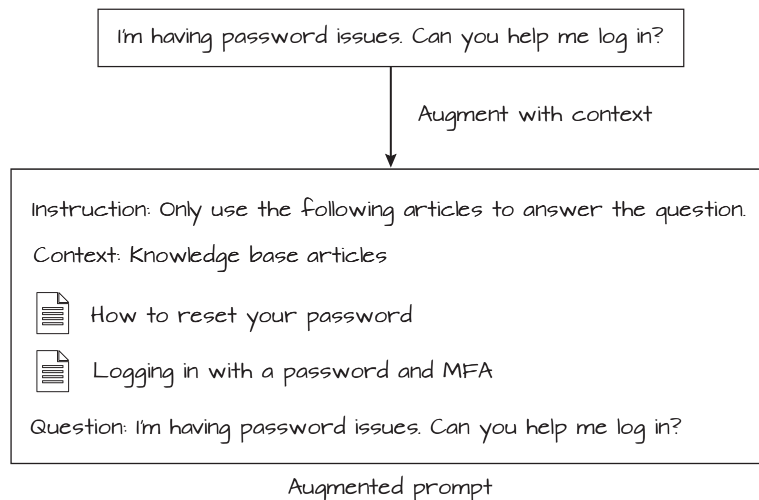


Figure 1-5: The prompt contains the user's question along with how-to articles for context.

In Chapter 8, you'll learn about retrieval-augmented generation (RAG), which is a targeted strategy for supplying only the context that's needed to generate a response.

Hallucinations

If an LLM has not been trained on something you are asking, it may predict a sequence of tokens to answer your question based on patterns it has seen before. The response will seem confident, even if the model has no idea what you are talking about. The LLM may even seem to make up fictitious information! This is referred to as a *hallucination*.

We can all relate. You're having a conversation with someone and think you know exactly what they're talking about. They ask you a question, and you respond confidently with amazing insight. Suddenly, they look at you as if you have three heads. You were so sure, but now you realize you confused fragments of what you thought you knew into something that doesn't quite apply. This has never happened to any of us, right?

Hallucinations can also be caused by the probabilistic nature of LLMs. Remember that each token is predicted based on its probability of occurring next in a sequence of tokens. Even tokens with a high probability of being next could be wrong, depending on the context.

We'll cover a variety of strategies to help prevent hallucinations when we cover prompt engineering in Part II. Just keep in mind that hallucinations are always possible because of the nature of LLMs.

Model Bias

Model bias happens when AI learns patterns from data that already contains human biases or gaps. If the training data leans a certain way (say, with content that is mostly English or from a specific region) or repeats stereotypes, the model can repeat those patterns. It's not the model "choosing" to be unfair; it's simply echoing what it saw.

While the vast amount of data available on the internet makes LLM training possible, we all know that human bias is everywhere. Data scientists do their best to filter out bad training data, but models learn language patterns without moral or ethical considerations.

This means their output can be skewed by their training data. If we recognize bias in our responses, we may need to add filtering, adjust our prompts, or fine-tune the model. We'll discuss this more in later chapters.

Multi-Step Reasoning

Multi-step reasoning is hard for LLMs because they don't "think" through a problem the way people do. They generate the next likely word based on patterns, not a plan. Therefore, LLMs can skip steps, mix up facts, or make confident leaps that look right but aren't.

Errors also compound. If one early step is off, later steps build on that mistake. Long prompts make this worse. Important details can get buried, the model loses track of earlier points, and it fills gaps with guesses instead of asking for missing information.

LLMs also lack reliable self-checking. They don't know when they're wrong, they are sensitive to the way you phrase the question, and they won't

The Developer's Guide to AI (Sample Chapter) © 3/31/26 by Jacob Orshalick, Jerry M. Reghunadh, and Danny Thompson

verify math or logic unless you force them to. They don't maintain a true working memory across turns, so context can drift in longer tasks.

We'll discuss some prompting techniques in Chapter 5 that help with handling multi-step reasoning tasks.

Improving LLM Responses

An LLM is essentially a black box. We don't know exactly what data the model has been trained on. For any pretrained model, we have to experiment to determine whether it can provide a useful response as is. As you've already learned, we can influence the accuracy of the response based on a number of factors. The old adage always applies: *Garbage in, garbage out.*

Figure 1-6 shows the core customization strategies aimed at improving model accuracy and how they compare in terms of time, cost, flexibility, and precision.

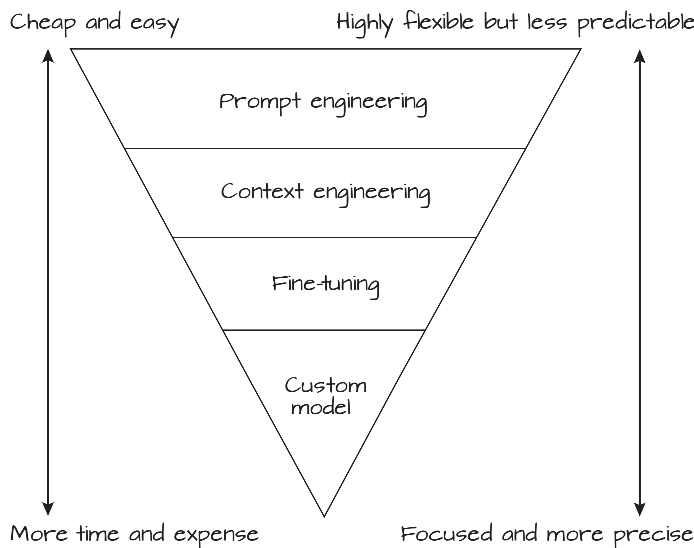


Figure 1-6: The hierarchy of model customization strategies

Throughout the remainder of the book, we'll give you the knowledge you need to progress down the pyramid. The following provides a quick overview of each of these strategies.

Prompt Engineering

Prompt engineering requires creativity and trial and error. The goal is to create prompts that provide context, instructions, and examples to help an LLM understand your intent and generate a desirable response. Part II is dedicated to the art and science of prompt engineering. For now, it's key to understand that your ability to write good prompts will determine your success with LLMs.

The Developer's Guide to AI (Sample Chapter) © 3/31/26 by Jacob Orshalick, Jerry M. Reghunadh, and Danny Thompson

It's always easiest (and the most cost-effective) to start with prompt engineering. We simply create prompts and experiment with the accuracy of the LLM response. If the results are promising, but we need more precision in our responses, we can continue down the pyramid.

Context Engineering

Context engineering encompasses prompt engineering along with the system components necessary to efficiently manage context within a prompt. In the words of Andrej Karpathy (former senior director of AI at Tesla and co-founder of OpenAI):

> People associate prompts with short task descriptions you'd give an LLM in your day-to-day use. [But] in every industrial-strength LLM app, context engineering is the delicate art and science of filling the context window with just the right information for the next step.

You'll learn more about context windows and the importance of managing the size of your prompts in Chapter 4. In the simplest terms, the context window defines the limit on the size of a prompt that can be sent to an LLM.

As you'll see, when we're supplying context to an LLM, it's important to include only the facts it needs (snippets from documents, search results, or memory), and put them in a clear order. This becomes even more important in multi-turn scenarios (for example, chatbots or agents), as conversation history can quickly fill the context window.

Fine-Tuning

Even context engineering may not be enough. Sometimes you need the model itself to be fundamentally better at a particular skill or to adopt a specific style that is crucial for your application. This is the realm of fine-tuning: taking an already powerful LLM and training it further on specialized data to mold its core capabilities precisely to your needs. In Part IV, you'll learn everything you need to know to fine-tune a pretrained AI model.

Custom Models

Custom models are beyond the scope of this book. Remember, we're the AI chefs. We use APIs and SDKs to integrate pretrained AI models. We leave the model building to the AI architects. That said, if you want to become an AI architect, this book will give you the knowledge to know where to start.

Building Autonomous Systems

Once you've learned how to bend LLMs to your will, you can use this knowledge to build truly autonomous systems. As we discussed earlier, traditional software systems are programmed to work in a specific way. Given a set of

inputs, the system will do exactly what it was programmed to do. Any new workflow requires you to change the system.

Agentic AI systems are different. These systems can operate autonomously, making decisions, taking actions, and adjusting their behavior over time to achieve specific goals. They do this by coordinating tools, APIs, or even other AI models to complete complex tasks.

Unlike traditional systems that follow fixed workflows, agents dynamically adapt based on the situation. They can interpret a user's intent, assess the surrounding context, and make decisions on the user's behalf. As they operate, they learn from new information and refine their approach to better meet their objectives.

Agents are a common agentic AI implementation pattern, and the Model Context Protocol (MCP) is the open standard introduced to make tools, APIs, and data sources accessible to AI models. You'll learn all about agents and MCP in Part V so you can start building your own agentic AI solutions.

Plugging LLMs into Your Code

All of this is interesting, but just like our startup team, we need a way to plug these LLMs into our applications. This is where the rubber meets the road.

AI companies (such as OpenAI) and open source tools (such as Ollama) provide APIs and SDKs that make it easy for developers to call these AI models in our code. Figure 1-7 shows how ChatGPT uses these APIs to generate a response.

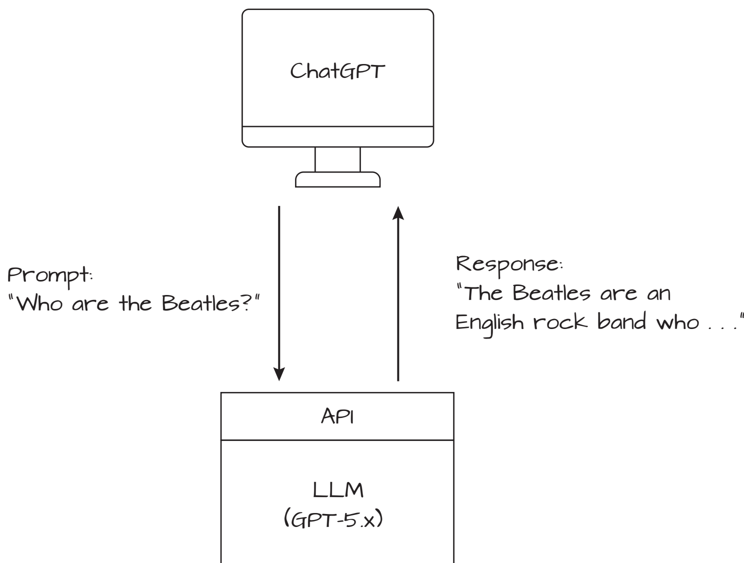


Figure 1-7: ChatGPT is an interface that uses OpenAI APIs to interact with LLMs.

The Developer's Guide to AI (Sample Chapter) © 3/31/26 by Jacob Orshalick, Jerry M. Reghunadh, and Danny Thompson

As you can see, ChatGPT is really just an interface. When you send a prompt, ChatGPT is calling an OpenAI API to interact with the actual LLM generating the response.

In the upcoming chapters, we'll connect to a locally run, open weight LLM (Llama 3.2) through the Ollama APIs. We'll demonstrate through examples the use of low-level SDKs, lightweight wrappers, and high-level frameworks to interact with models. The Llama models are built by Meta and provided to the community as open weight models. This allows you to run the LLM locally without paying for usage.

You can use the same code and tools you'll find in this book to experiment with any pretrained LLMs. As you'll see in Chapter 4, the concepts remain the same, but your testing may reveal that a specific model provides better responses or performance, or is more cost-effective.

OPEN SOURCE VS. OPEN WEIGHT MODELS

As developers, we're very familiar with open source software whose code is freely available to inspect, modify, and share. It's the foundation of countless tools we use daily, from operating systems like Linux to libraries on GitHub that power nearly every modern application. We'll use several open source libraries to integrate AI models throughout this book.

In contrast, the AI community refers to models like Llama 3.2 as *open weight*, meaning the model's trained parameters are available, but the full source code, dataset, and training process may not be. This means that Llama 3.2 is not truly open source. Keep in mind that the license also restricts redistribution and certain types of commercial use. We'll revisit this topic in Chapter 4.

Summary

This chapter provided a high-level introduction to LLMs and general AI concepts. You started by learning how LLMs are trained on large datasets of publicly available and licensed text including books, articles, and research papers in order to understand natural language.

You then saw how LLMs generate text by predicting a sequence of tokens. You learned how AI models differ from traditional programming in that their outputs (predictions) are probabilistic, not deterministic. This gives them the flexibility to adapt to new situations but runs the risk of mistakes.

You saw the versatility of LLMs and the variety of use cases they're able to handle through prompting, including writing, summarizing, classifying, translating, coding, and much more. You were also introduced to the limitations that have to be considered and addressed when using an LLM, such as limited training knowledge, hallucinations, model bias, and challenges with handling multi-step reasoning.

Finally, we introduced some of the major topics we'll cover in the book: prompt engineering, context engineering, fine-tuning, and agents. You learned that we'll be interacting with LLMs through APIs and SDKs.

Now it's time to dive into the code so we can get our hands dirty. In Chapter 2, we'll start just like our startup team did by calling an LLM from code, using JavaScript. We'll introduce some basic concepts along the way. Then, in Chapter 3, we'll introduce Python as we port over the example from JavaScript and prepare you for the chapters to come.

So come on, let's get coding!