Inside Large Language Models

Foundations to Production

Ritesh Modi

Inside Large Language Models: Foundations to Production

First Edition

ISBN: [to be assigned]

First printing: 2026

To the curious,

the ones who refuse to treat a model as a black box

and insist on seeing how it actually works.

Preface

Large language models moved from research curiosity to everyday tool in the span of a few years, and yet most of the people using them every day have no idea how they actually work. They type a prompt, the model produces fluent text, and the mechanism behind it stays hidden. That is a problem. When you treat a model as magic, you cannot debug its failures, improve its outputs, or decide when to trust it. You are stuck asking someone else for answers.

This book exists because the gap between "I use a language model" and "I understand how a language model works" is far smaller than it looks from the outside. The ideas at the core, tokens, embeddings, attention, transformer blocks, training loss, sampling, are explainable from first principles. You do not need a research background. You do not need a graduate math class. You need someone willing to walk through each concept slowly, with concrete numbers, real examples, and the patience to explain what every symbol means and why every design choice exists.

That is the book I wanted when I was learning. Every chapter starts with intuition before mechanics. Every concept gets a worked numerical example before any formula. Every piece of code comes with a line-by-line explanation and an execution-flow trace showing what happens under the hood. The book moves from the simplest building blocks, what a token is, what an embedding does, through single-head attention, multi-head attention, the complete transformer block, training loops, inference, alignment, and fine-tuning for real applications.

By the end of this book, you will have built a small transformer-based language model from scratch, trained it on real text, run inference on your own inputs, and fine-tuned a pretrained model for four concrete tasks. More importantly, when a new model is announced next year, you will be able to read its paper and know exactly which parts are genuinely new and which parts are familiar ideas wearing a new name.

What This Book Is About

This book teaches large language models from the ground up. Every abstraction is built from what came before it. The book starts with text, the raw material a language model consumes, and explains how text becomes numbers, how numbers become vectors in a learned space, how those vectors interact through attention, and how a stack of such interactions produces the next-token predictions that form the basis of every modern LLM.

The book is organized into nine parts. Part I (Chapters 1 through 3) builds the foundation: what a language model is, what the architectural families are, and the mathematical objects every subsequent chapter assumes. Part II (Chapters 4 and 5) introduces positional encoding and single-head attention, the mechanism at the heart of the transformer. Part III (Chapters 6 and 7) assembles the full transformer

block, with multi-head attention, residual connections, layer normalization, the feed-forward network, and the language-model head.

Part IV (Chapters 8 and 9) shifts from architecture to training: the loss function, backpropagation, optimizers, the data pipeline, and a capstone chapter that puts every piece together to train a complete GPT-style model from scratch. Part V (Chapter 10) covers inference, how a trained model actually produces text, token by token, with the optimizations that make it feasible in production. Part VI (Chapter 11) covers post-training alignment: supervised fine-tuning, reward modeling, and reinforcement learning from human feedback.

Part VII (Chapters 12 and 13) covers fine-tuning theory and the inference optimizations used to serve these models at scale. Part VIII (Chapters 14 through 17) is the applied portion: four hands-on fine-tuning projects taking a pretrained model through a contract-classification task, a legal-document assistant, a natural-language-to-SQL system, and a function-calling agent. Part IX (Chapter 18) ties every concept in the book into a single coherent picture so you leave with one mental model, not eighteen fragmented ones.

What This Book Is Not

This book is not a framework tutorial. It is not a manual for any specific library, and it deliberately keeps the teaching independent of whichever framework happens to be popular this year. Frameworks change. The ideas in this book do not. When a code example needs to run, it runs, but the point of the code is always to illustrate a concept, not to showcase an API.

This book is not a research survey. It does not attempt to catalogue every model variant or every paper published in the last five years. It covers the ideas that have survived long enough to matter, the core architectural choices, the training objectives, the alignment techniques, and the fine-tuning methods that appear again and again across systems. Once you understand those, reading any recent paper becomes a matter of spotting what is new and what is familiar.

This book is not an API reference. It does not list every argument to every function. It does not enumerate every configuration option. Reference material is best consumed as reference material, on a web page, searchable, up to date. This book teaches the concepts the reference assumes you know.

This book is not about deploying models in production infrastructure. It explains how inference works, including the optimizations that production systems rely on, but questions about cluster orchestration, autoscaling, request routing, and cost accounting belong to a different discipline with its own dedicated resources.

Who This Book Is For

This book is for anyone who wants to genuinely understand how large language models work. You do not need prior machine learning experience. You do not need a deep math background. You should be comfortable with Python at an intermediate level, functions, classes, list comprehensions, and working with libraries , and you should have enough high-school algebra to follow along when a concept uses a formula. Every mathematical object that appears in the book is explained from scratch the first time it is introduced.

Software engineers who have been told "add an LLM to the product" will find this book invaluable because it teaches you what you are actually adding. Data scientists and analysts working adjacent to modeling will find the concept-first approach lets them move from using models to building and tuning them. Recent graduates and career switchers will find a self-contained path from basic Python to a working understanding of the architecture that underpins every major LLM released today.

If you are already building with LLMs, calling APIs, writing prompts, stitching together retrieval-augmented applications, this book fills in the mechanisms behind the tools you use every day. The chapters on fine-tuning (Chapters 12 through 17) are particularly relevant if you want to move beyond prompt engineering into actually customizing models for your own domain.

How to Use This Book

The book is designed to be read front to back. Each chapter builds on what came before. The cross-references are specific, when a chapter points you back to "Chapter 5 (Single-Head Attention), Section 5.3," it means exactly that section, and your understanding of the current chapter will be deeper if you have that earlier material in mind.

That said, the book supports different reading paths. If you already understand the transformer architecture at a high level but want to see every component taught from first principles, read Chapters 1 through 9 carefully and skim the applied fine-tuning chapters (14 through 17) based on interest. If you are here primarily to fine-tune models for your own tasks, read Chapter 11 (Post-Training) and Chapter 12 (Fine-Tuning Theory) carefully, then pick the applied chapter closest to your problem. If you want to understand inference and serving, Chapter 10 and Chapter 13 are self-contained enough to read as a pair.

Every chapter ends with a set of hands-on exercises ("Try It Yourself") and a set of scenario-based knowledge-check questions. The questions are deliberately not definitional, they present a situation and ask you to reason about what the model or the training loop will do. Answers for every knowledge-check question across all eighteen chapters are in Appendix E. Work through the questions before checking the answers. Attempting a question and getting it wrong builds more understanding than reading the correct answer directly.

Every non-trivial code block in the book is followed by an Execution Flow trace , a small step card that shows the call tree the runtime follows when your code runs, with the tensor shapes at each step. The traces are how you build intuition for what is actually happening underneath the one-line API calls. Reading them is one of the fastest ways to internalize the architecture.

Conventions Used in This Book

Typography

Body text is set in Garamond. Code in the body text appears in `Courier New`, for example, softmax(), d_model, and attention_mask. When you see a term in monospace font, it refers to a specific variable name, function, tensor, or configuration in the code.

Code Blocks

Standalone code examples appear in numbered listings with a color-coded bar on the left that indicates the semantic role of each section, setup, forward pass, loss computation, or output. Every listing number follows the pattern "Listing [chapter].[sequence]", Listing 5.3 is the third code block in Chapter 5. When the text refers to a specific listing, it uses this numbering so you can find it quickly.

Figures and Diagrams

Diagrams are numbered per chapter, Figure 9.2 is the second diagram in Chapter 9. The book uses a consistent four-color system across every diagram, table, and value display. Every chapter includes a color legend at the bottom, but the system is summarized here in full:

Color System

	Color	Meaning	Used For
0	Blue	Input / original / source	Input tensors, token IDs, source values, cached values
1	Amber	Computed / transformed / derived	Intermediate activations, scores, derived metrics
2	Green	Output / result / final	Final tensors, predicted tokens, loss

			values, results
3	Purple	Aggregated / architectural	Batch-level summaries, module outputs, architecture elements

The color coding is semantic, not decorative. When you see a blue-shaded value in a step card, it came from the input. Amber means it was computed in that step. Green is the final result. Purple is an aggregated or architectural-level value. The colors make multi-step transformations traceable at a glance.

Callout Boxes

Three kinds of callout boxes appear throughout the book, each with a distinct visual treatment:

> *Insight boxes (green left border) surface deeper connections, real-world applications, and the moments where a concept clicks into place. They connect what you just learned back to the bigger picture of how a language model works.*

> *Gotcha boxes (amber left border) warn about common mistakes, surprising behavior, and the traps that catch experienced engineers. When a gotcha says "this will silently produce wrong outputs," it means exactly that, pay attention.*

> *Error boxes (red left border) show the exact error message you will see when something goes wrong, followed by what caused it and how to fix it. These are the errors you will actually encounter in practice, with the diagnoses you will find yourself reaching for again and again.*

Step Cards and Execution Flow Traces

Multi-step processes, tokenization pipelines, attention computations, training loops, are shown as a sequence of step cards. Each card has a numbered pill badge (Step 1, Step 2, and so on) and a title describing what happens in that step. The pill color signals the nature of the step: amber for computation, blue for input or cached data, green for output, and purple for aggregated or architectural operations.

After every non-trivial code example you will find an Execution Flow trace, a step card with a green pill that shows the complete call tree underneath the code. The traces include the tensor shapes at each step so you can follow not just what is being called but what is flowing through. They are the fastest way to build real intuition for how these models actually execute.

Tables and Data Displays

Tables are numbered per chapter, Table 10.1 is the first table in Chapter 10. When showing tensor contents or intermediate values, the book uses styled tables with color-coded cells rather than raw printouts, because the semantic coloring carries meaning about where each value came from and where it is going.

Cross-References

When the book points back to material covered elsewhere it always includes the chapter number, chapter name, and section: “as we saw in Chapter 5 (Single-Head Attention), Section 5.3.” This specificity is deliberate. It lets you flip to the exact spot if you need a refresher, and it lets you skip the reference if the concept is already fresh. The book never says “as mentioned earlier” without telling you exactly where.

Companion Code and Data

All code examples from this book are available in a companion code repository. Each chapter has its own directory with independently runnable Python scripts. The scripts include any setup code that is omitted from the book to keep the listings focused on the concept being taught. Setup blocks are clearly marked so you can tell them apart from the teaching examples.

The repository also includes the small datasets used throughout the book, the toy corpora used for tokenization examples, the short training set used in the capstone chapter, and the task-specific datasets used in the applied fine-tuning projects. A requirements file pins the exact library versions that the book was tested against, so reproducing any example is a matter of cloning the repository, installing the requirements, and running the script.

To run the examples, you need Python 3.10 or later and a machine with at least 16 GB of RAM. Most of the code can be executed on a laptop CPU, and where a GPU makes a meaningful difference the book calls it out and provides a lighter-weight fallback. The capstone training chapter and the larger fine-tuning projects are written to work on either a modest consumer GPU or a free-tier cloud notebook, with configuration options for faster hardware when you have it.

How to Contact Us

Getting the details right matters. If you find a technical error, a code example that does not run as written, an explanation that confuses more than it clarifies, or a typo that makes you stop reading, I want to hear about it. Errata and corrections are tracked in the companion repository's issues section. Before opening a new issue, check the existing ones to see if somebody has already reported the same thing.

For general questions about the book, suggestions for future editions, or to share something you built after reading it, you can reach me at www.riteshmodi.com or through the companion repository. Messages from readers who shipped something real after working through the book are the ones I enjoy most. I read every one.

If you are using this book in a course, training program, or reading group and would like supplementary material, extra exercises, slide decks, or dataset variations, please reach out. I am glad to support educators teaching the next round of engineers who are going to build with these models.

Acknowledgments

A book like this is shaped by more people than any list can name. It is shaped by every engineer who asked a question that made me realize I did not understand something as well as I thought. It is shaped by every junior colleague who pushed back on an explanation until I had to rebuild it from first principles. It is shaped by every debate over a whiteboard about what a particular gradient actually meant.

Thank you to the open-source communities that built and continue to build the tools the whole field runs on. The pace at which ideas turn into working code in this space is only possible because thousands of contributors have chosen to share their work freely. Much of what this book teaches was only made teachable because somebody else built a reference implementation you could read.

Thank you to the reviewers who read early drafts, ran every example, challenged my explanations, and caught the errors that would otherwise have shipped. The mistakes that remain are mine, not theirs.

Thank you to every reader who chose this book over the many alternatives now available. Writing a technical book is an act of optimism, a bet that your particular way of framing an idea will click for someone in a way that other resources did not. If this book helps you build something you are proud of, debug a problem that had you stuck, or land an opportunity you wanted, that is the entire point.

And finally, thank you to my family, who tolerated the long evenings and early mornings this book demanded, who listened patiently while I explained attention mechanics over dinner, and who never suggested, even once, that maybe I had enough hobbies already.

Table of Contents

Part I: Foundations

What an LLM is, the architectural families, and the math we will assume.

Chapter 1
Understanding Large Language Models: From Mystery to Mastery

On November 30, 2022, OpenAI released ChatGPT. Within five days it had one million users. Within two months, one hundred million. But what was actually happening under the hood remained a mystery to nearly everyone using it. That gap, between using a large language model and understanding one, has never been wider, and it has never mattered more. As these models move from answering trivia questions to executing code in production environments, booking flights, filing pull requests, and orchestrating multi-step workflows, the engineers who understand the internals are the ones building the future. The ones who don't are guessing. And guessing gets expensive fast.

This chapter builds the mental model you need before a single line of math or code appears. You will learn the core mechanism behind every **Large Language Model (LLM)**, **next-token prediction**, and why something so deceptively simple produces behavior that looks like reasoning, translation, and creativity. We will trace the path from statistical n-gram models of the 1950s through recurrent neural networks to the **Transformer** architecture that changed everything in 2017, so that when we build a Transformer from scratch in later chapters, every design choice will make sense. You will also get an honest picture of what today's frontier models can and cannot do (reasoning models, agentic systems, and fine-tuning breakthroughs, along with persistent limitations like **hallucination** and adversarial fragility) because if you are building products on top of LLMs, that honest picture matters a lot.

We start by defining what an LLM actually is and how predicting the next word leads to code generation and multi-step reasoning. From there we trace a brief history through three eras of language modeling that explain why Transformers won. We then look at the current state of the field, covering capabilities, limitations, and why understanding internals gives you a genuine edge over engineers who treat models as black boxes. After surveying real-world applications across code, law, medicine, and everyday productivity, we close with the book's roadmap and the math prerequisites you will need. This is the first chapter of the book, so there are no prior chapters to build on, but everything you read here sets the stage for Chapter 2 (Architecture Archetypes: Encoders, Decoders, and Everything In Between), where we will meet the four architecture families that define modern natural language processing.

1.1 What Actually Is a Large Language Model?

You have probably used ChatGPT, Claude, or one of the other large language models that have exploded into the mainstream over the past couple of years. Maybe you have been impressed, maybe confused, maybe both. But here is the thing: using these models and understanding them are two very different skills, and the second one is about to become much more valuable than the first. This opening section tackles the most fundamental question head-on: what is a large language model, actually? Not the

marketing version, not the science-fiction version, but the real mechanism. Once you see it, the rest of the book clicks into place, and the core idea is simpler than most people expect.

A large language model is a neural network trained on massive amounts of text to predict the next **token** in a sequence. A token is a piece of text the model works with, sometimes a whole word, sometimes a fragment like "ing" or "un"; we'll open that up properly in Chapter 3 (Foundations of Large Language Models), Section 3.2. That is it. Given "The cat sat on the," the model assigns a probability to every possible next word, so "mat" gets a high probability while "quantum" gets a low one. Stack that operation billions of times during training, and something remarkable emerges: the model picks up grammar, facts, reasoning patterns, even some common sense. Not because anyone programmed those things in, but because they are useful for prediction. That phrase "something emerges" is worth pausing on, because it is the idea of emergence itself: behavior that was not explicitly trained for showing up anyway once the model is big enough. The "large" part refers to both the number of **parameters** (modern models have billions to trillions) and the scale of training data, which can span trillions of tokens drawn from books, web pages, code repositories, and scientific papers. Parameters are the numerical knobs the model tunes during training; think of them as the dials a sound engineer adjusts, only here there are a hundred billion of them and the tuning happens automatically.

That sense of wonder is a great starting point, but wonder alone will not help you build anything. The engineer who knows why a prompt works can reproduce the result. The one who does not is guessing. So let's nail down the definition and then ask: how does guessing the next word lead to code generation, multi-step reasoning, or translation between languages nobody explicitly taught the model?

Next-token prediction sounds too simple to explain what these models actually do. But think about what "predicting the next token" actually requires. To predict the next token in a calculus proof, the model has to internalize calculus. To predict the next token in a legal brief, it has to absorb the structure of legal reasoning. The simplicity of the objective is deceptive, because what the model must learn to do it well is anything but simple. The training objective forces the model to build internal representations of syntax, semantics, logic, and world knowledge, because all of those things help it make better predictions. One goal produces a whole family of **emergent capabilities**.

This is what makes LLMs so versatile, and it is why people call them "**foundation models**." A model trained only to predict the next token somehow learns to translate languages, answer factual questions, and write working code, because all of those skills make it better at prediction. One model writes poems, generates Python code, translates between languages, and explains quantum physics, all from the same weights, the same architecture, the same training objective. That versatility is why the industry shifted from building a separate model for each task to starting from one model and adapting it. Instead of training a sentiment classifier, a translation model, and a summarizer, you start from one foundation model and fine-tune or prompt it for each task. That is a big deal, and by the time you finish this book, you will understand exactly how that works, be able to build one from scratch, fine-tune it for a specific task, and deploy it in production.

To understand where LLMs are going, it helps to know where they came from. The journey from rule-based systems to Transformers is shorter than you would think, but it is packed with breakthroughs that each solved a real problem and left behind a real limitation.

1.2 A Brief History: From Statistical Models to Transformers

On the previous pages we nailed down what an LLM actually does: predict the next token, over and over, and let intelligent-seeming behavior emerge from that one operation. That is the "what." Now we need the "how did we get here." The answer involves three distinct eras of language modeling, each one making the next possible, and the final leap (the Transformer) is the reason you are reading this book at all. Understanding the history is not just trivia. Each era solved a real problem and left behind a real limitation, and those limitations motivated the next breakthrough. When we get to the Transformer architecture in later chapters, you will see exactly why it won, and what it replaced.

The first era was the statistical era, stretching from the 1950s through the early 2000s. Alan Turing proposed the Turing Test in 1950, asking whether a machine could exhibit intelligent behavior indistinguishable from a human. Early systems tried rule-based approaches: handcrafted grammar rules, dictionaries, decision trees. They were brittle. Change one word and the system broke. **N-gram models** were a leap forward, because instead of rules they counted word patterns in real text. A bigram model learns the probability of "sat" following "cat" from millions of examples. It worked, sort of. N-grams were fast, simple, and surprisingly effective for speech recognition and basic text prediction. But they had fundamental limits. A 5-gram model can only look at the previous four words. The "**curse of dimensionality**" meant larger context windows required exponentially more data. And crucially, n-grams had no concept of meaning: the word "bank" was the same token whether you were talking about rivers or money. Anything more than a few words back was invisible.

Statistical models got one big thing right: treat language as data, not rules. Count patterns in real text instead of trying to hand-code grammar. That insight still holds today. But n-gram models had no way to represent meaning, and their fixed context windows meant they could never capture the long-range dependencies that real language requires. Neural networks changed that. They could learn continuous representations of words (vectors that actually encode meaning) and process sequences with real memory. Limited memory, sure. But real.

The second era was the neural era, running from roughly the mid-2000s through 2017. Neural networks had been around since the 1950s, but they were not practical for language until the 2000s, because the hardware just was not there. Then researchers like Yoshua Bengio and Tomas Mikolov showed that neural nets could learn much richer word representations. **Word2Vec**, published in 2013, was a turning point. It learned that "king" minus "man" plus "woman" approximately equals "queen," just from reading text. The model had picked up something about meaning. That was new.

Recurrent Neural Networks, or RNNs, and later **Long Short-Term Memory** networks, or LSTMs, became the go-to architecture for language. They could process text sequentially, keeping some memory of what came before, and they powered the first machine translation systems that were genuinely impressive. Some of those early chatbots felt almost human. But RNNs had a critical weakness that could not be engineered away. They process tokens one at a time, left to right, which created two problems. First, information from early tokens gradually faded as the sequence got longer, like a game of telephone where the message degrades with each retelling. LSTMs helped with gating mechanisms, but long-range dependencies remained fragile. Second, sequential processing meant you could not parallelize training across Graphics Processing Units (GPUs). Each token had to wait for the previous one. That made scaling to larger models and datasets painfully slow.

> *The sequential bottleneck of RNNs was not a bug that could be patched; it was fundamental to the architecture. No matter how clever the gating mechanism, processing tokens one at a time meant training time scaled linearly with sequence length, and information from the beginning of a document inevitably decayed by the end. These two limitations (fading memory and sequential processing) are exactly what the Transformer was designed to eliminate.*

So neural networks could handle language; that was settled. Word2Vec proved they could capture meaning, and RNN-based translation systems actually worked. But two ceilings would not budge. Sequential processing meant each token waited for the one before it, which made training painfully slow. And the fixed-size hidden state meant information from early in a document just faded. The field needed a structurally different approach. In 2017, it got one.

The third era, the Transformer era, began with a single paper. In 2017, Vaswani and seven co-authors at Google published "Attention Is All You Need," which replaced recurrence entirely with a mechanism called **self-attention**. Picture every word in the input sentence glancing at every other word and deciding which ones matter for understanding its meaning; that is all self-attention is at heart, and Chapter 5 (Single-Head Attention) builds it from first principles. The Transformer itself is a stack of these **attention** layers with some feed-forward computation and normalization in between (Chapter 7 (Feed-Forward Network, LM Head & The Complete Transformer) assembles the full architecture piece by piece). Instead of reading tokens one at a time, the Transformer could process every token simultaneously, with each one attending to every other, which made it parallelizable, scalable, and able to capture dependencies across the entire input rather than just a few words back. This was the breakthrough, and everything since (from **GPT** and BERT to LLaMA, Claude, and Gemini) is built on the Transformer architecture from that paper.

Once you could train in parallel, you could scale, and scaling brought surprises. Models did not just get incrementally better with more parameters and data; they exhibited qualitative jumps in capability, with tasks that were impossible at one billion parameters becoming trivial at one hundred billion. Google's BERT, released in 2018, showed that **pre-training** a Transformer on massive text and then **fine-tuning** it on a specific task crushed every benchmark. BERT was **encoder-only**, good at understanding but not at generating, yet it proved the pre-train then fine-tune recipe that the entire field now follows. OpenAI's GPT-2 showed that a **decoder-only** Transformer could generate remarkably coherent text, and GPT-3 (released in 2020 with 175 billion parameters) demonstrated **few-shot learning**, the ability to perform tasks it was never explicitly trained on, just from a few examples in the prompt. Since then we have seen Meta's LLaMA, Anthropic's Claude, Google's Gemini, and plenty of others, each generation bigger, more capable, and better aligned than the last.

Think of the statistical era as a tourist with a phrasebook; they can look up common phrases but are completely lost with anything novel. "Where is the bathroom?" works, but "Can you recommend a quiet restaurant with outdoor seating near the river?" does not exist in the book. The neural era with RNNs is like someone who studied the language in school: they have real understanding but can only hold a short conversation before losing track of what was said earlier, processing one word at a time, sequentially, with memory that fades. The Transformer era is like a native speaker who has read every book in the library, able to hold complex, nuanced conversations, understand context from pages ago, and do it all while carrying on multiple conversations at once, because the architecture is parallelized.

Three eras, each one solving a problem the last one could not. Statistical models gave us data-driven patterns. Neural models added learned representations and memory. Transformers introduced parallel attention, unlocking scale that was previously impossible. This is not just history for history's sake; it explains why modern LLMs look the way they do, and why design choices like attention and residual connections matter as much as they do. So that is how we got here. Now, where are we? What can these models actually do, and where do they fall short?

1.3 Where We Are Now and Why Understanding Matters

We have traced the path from n-grams to RNNs to Transformers, the three eras that explain how we arrived at the models powering today's Artificial Intelligence (AI) revolution. But knowing the history is only half the picture. The field has changed dramatically in just the past two years. Models do not just predict the next word anymore; they reason through multi-step problems, call external tools, and operate as autonomous agents that can plan, execute, and self-correct. Understanding where things stand right now, and where they are still falling short, is what separates informed builders from casual users.

This section is also where we tackle a question you might already be asking: why bother learning the internals at all? Can you not just use the Application Programming Interface (API) and call it a day? You can. But the engineers who understand what is happening underneath consistently get better results, debug faster, and build things that others cannot. Let's look at why.

Reasoning models are the first of three recent shifts that have redefined what these models can do, and each one traces directly back to the architectural and training concepts you will learn in this book. Models like OpenAI's o1 and o3 series and DeepSeek-R1 do not just predict the next token; they generate chains of thought, explicitly working through multi-step problems before producing an answer. The technical mechanism behind this is fascinating: these models are trained with reinforcement learning rewards that incentivize extended reasoning rather than quick answers. The result is dramatically better performance on math, science, and logic problems, at the cost of more computation per query, a concept called "**test-time compute**." We will explore the training techniques that make this possible in Chapter 8 (Training: Loss, Backpropagation, Optimizers and the Data Pipeline) and Chapter 11 (RLHF and Instruction Tuning: Aligning a Language Model).

A second shift, almost as consequential, is **agentic systems**: the move from models that answer questions to models that take actions. Tools like Claude Code, Devin, and GitHub Copilot Workspace do not just suggest code; they create files, run tests, fix errors, and iterate. Under the hood, this is **function calling**: the model generates structured JavaScript Object Notation (JSON) that triggers real API calls. In Chapter 17 (Function Calling: Teaching an LLM to Use Your Tools), you will fine-tune your own function-calling model from scratch.

Alongside those two, the third shift is the democratization of fine-tuning. **Low-Rank Adaptation (LoRA)** and its quantized variant (**QLoRA**) let you adapt a model by training less than one percent of its parameters. You can fine-tune a seven-billion parameter model on a single GPU in a few hours. This has unlocked domain-specific LLMs, models specialized for legal drafting, Structured Query Language (SQL) generation, and medical question-answering, trained by small teams on modest budgets. Chapters 12 through 17 cover fine-tuning theory and four hands-on projects.

The frontier keeps moving. Reasoning gets deeper, agents get more reliable, fine-tuning gets cheaper, and the gap between what is possible in research labs and what developers can actually deploy shrinks every quarter. But every one of these advances (chain-of-thought reasoning, tool use, LoRA adapters, **Key-Value (KV) cache** optimization) builds directly on the Transformer architecture you are about to learn.

Here is the shift that matters most: LLMs are no longer just information providers. For years, the mental model was "ask a question, get an answer," with the model acting as a very smart search engine. That era is over. Today's models take actions. They book flights, file pull requests, execute database migrations, send emails, and orchestrate multi-step workflows across dozens of tools. An agent given the instruction "deploy the hotfix to staging" does not hand you a checklist; it runs the tests, builds the container, pushes to the registry, and updates the deployment manifest. The model has gone from advisor to operator. That is a completely different relationship, and it demands a much deeper level of understanding from the people building these systems.

At the same time, the models themselves keep getting better: more calibrated, more honest about uncertainty, and dramatically cheaper to run. Hallucination rates have dropped significantly thanks to **Reinforcement Learning from Human Feedback (RLHF)** and constitutional AI techniques, while math and coding benchmarks show year-over-year improvements of twenty to forty percent. **Context windows** have expanded from four thousand to over one million tokens, enabling document-scale processing, and multimodal capabilities are now standard (with text, image, audio, and video all handled in one model). On top of all that, inference costs have dropped ten to one hundred times through quantization, speculative decoding, and hardware improvements.

But they still hallucinate, stating false information with the same confident tone they use for accurate answers. Long-horizon planning remains fragile, and multi-step agent workflows often fail silently. There is no true persistent memory either: each conversation starts from scratch, and the context window is not the same as memory. Adversarial reliability is weak (carefully crafted inputs can bypass safety measures), and evaluation is genuinely difficult because benchmarks saturate faster than capabilities improve. On top of all that, training frontier models still costs tens of millions of dollars. When a model is just answering questions, a hallucination is an inconvenience; when it is executing actions in your production environment, a hallucination is a production incident. The stakes have changed.

Now here is the question that matters: if anyone can call an API, what advantage does understanding the internals give you? A lot, it turns out. Understanding changes how you write prompts, how you debug failures, and how you decide between fine-tuning and **Retrieval-Augmented Generation (RAG)**. Knowing attention patterns helps you structure prompts that work consistently rather than accidentally, and when output is wrong, knowing the architecture tells you where to look (is it a **tokenization** issue, an attention pattern problem, or a sampling artifact?). New papers make sense once you understand the foundations they build on. Choosing between full fine-tuning, LoRA, and QLoRA requires understanding which layers to target and why, while production deployment is full of trade-offs (quantization, KV cache sizing, batch strategies) that demand architectural understanding. You can only mitigate risks like hallucination, adversarial attacks, and memorization of training data if you understand them, and the people pushing the frontier forward all understand the fundamentals deeply.

Let's make this concrete. Where are LLMs actually being used right now, and what does understanding the internals mean for each use case?

1.4 Real-World Applications: Where LLMs Are Being Used Today

We have covered what LLMs are, where they came from, and why understanding the internals gives you an edge. Now let's ground all of that in reality. Where are these models actually being deployed today? The answer is: almost everywhere. From code editors that write functions for you to medical systems that summarize patient records, LLMs have moved from research curiosity to production infrastructure faster than almost any technology in history. What is striking about these applications is not their variety; it is that they all rely on the same underlying operation we defined in Section 1.1, namely next-token prediction. The model that writes Python is architecturally identical to the one that drafts legal briefs. The difference is in the training data and the fine-tuning. That insight will matter a lot when we get to the fine-tuning chapters later in the book.

Code generation was the first killer application for LLMs, a domain with clear correctness signals and massive open-source training data. GitHub Copilot, Cursor, and Windsurf have transformed software development. These tools use LLMs fine-tuned on code to provide real-time suggestions, complete functions, explain unfamiliar codebases, and even generate entire modules from natural language descriptions. Studies report thirty to fifty percent productivity gains. Under the hood, it is the same next-token prediction, just trained on billions of lines of code instead of books. And the same models that complete Python functions can also draft emails, synthesize research, and generate marketing copy. It is all the same operation underneath: predict the next token.

Customer service has been another early adoption area. LLM-powered chatbots now handle first-line customer support for major companies, accessing knowledge bases, understanding nuanced queries, and escalating to humans when confidence is low. The key challenge is ensuring accuracy, because a wrong answer from a support bot erodes trust fast, which is why production systems add verification steps and human oversight. Content creation and marketing teams use LLMs for product descriptions, marketing copy, email campaigns, and social media posts. The model handles the first draft, and humans refine. The most effective setups use **prompt chaining**: one call to generate, another to critique, a third to revise. Some companies fine-tune models on their brand voice for consistency.

But LLMs are also transforming knowledge work in fields like law, medicine, and research, where accuracy matters more than speed. Researchers use LLMs to summarize papers, extract key findings, and even generate hypotheses. Legal teams use them to review contracts and find relevant precedents. Medical professionals use them to synthesize patient histories. In every case, the model's output is a starting point, not a final answer, and domain experts verify and refine. The ability to summarize vast bodies of text is genuinely useful, but hallucination makes human oversight non-negotiable. If you do not understand where the model's confidence is misplaced, you cannot build safe systems.

Beyond code generation and knowledge work, LLMs are being embedded as assistants in virtually every software category. Natural-language queries run against databases, conversational interfaces sit on top of Customer Relationship Management (CRM) systems, and AI-assisted design tools are moving into every major design suite. The pattern is the same: take a complex interface and let users interact with it in plain language. Maybe the biggest transformation is the most personal. Millions of people (teachers, students, freelancers, hobbyists) use LLMs every day for drafting emails, learning new topics, brainstorming ideas, debugging code, and planning trips. The same architecture powering a Fortune 500 customer service platform also helps a student draft an essay. That breadth of adoption is remarkable.

One pattern keeps showing up, regardless of industry or use case: the people getting the most out of LLMs are the ones who understand how they work. They write better prompts, anticipate hallucinations, and choose the right model for the job, and that is not a coincidence. Understanding the architecture is not academic; it is a competitive advantage.

> *Users who understand LLM internals consistently get better outcomes across every domain. They debug faster, prompt more effectively, and build more reliable systems. The rest of this book is built around giving you exactly that kind of understanding, piece by piece.*

The math behind LLMs is more accessible than you would expect. Let's talk about what this book covers and who it is for.

1.5 Your Journey Ahead: What You Will Learn in This Book

You have seen the definition, the history, the current state, and the real-world applications. At this point you might be thinking: "Okay, I am convinced this stuff matters. But can I actually learn it?" The answer is yes, and this final section lays out exactly what the book covers, what math you will need, and how each chapter builds on the last. Treat it as a quick tour of the terrain before we start walking it.

One thing worth saying up front: the math in this book is more accessible than it looks. We are not writing a research paper. Every equation gets a plain-English explanation, every concept gets a concrete example, and every mechanism gets working code you can run yourself. If you can follow a Python function and remember what a dot product is, you are ready. A lot of people bounce off LLMs because of the math. Transformers, attention, backpropagation, gradient descent: those words sound intimidating. But the math itself? More accessible than you would think.

The prerequisites reduce to three things, and you may already know two of them. Linear algebra is the first: matrices, vectors, and dot products. Every Transformer layer is, at the lowest level, a matrix multiplication, so if you can multiply a matrix by a vector, you can follow the forward pass of a Transformer. You will also want a working memory of basic calculus, specifically derivatives and the chain rule, because **backpropagation** (how the model learns) is the chain rule applied repeatedly, and we will walk through it step by step with actual numbers. Finally, a light touch of probability rounds things out: LLMs output a probability distribution over the next token, so understanding **softmax**, **cross-entropy**, and sampling strategies rests on knowing how probabilities combine.

Every concept in this book follows the same pattern: intuition first, then the formal math, then code. You will always see why the math works, not just that it works. Take attention mechanisms: they sound complex, but the core idea is straightforward. When processing a word, figure out which other words matter most, and weight them accordingly. Those three building blocks (linear algebra, calculus, probability) are all you need. Every formula in this book is motivated by an actual architectural question, implemented in Python, and grounded in a running example so you can verify each step by hand. No math appears in a vacuum.

Here is how the book is structured. Part one covers the foundations: the mathematical building blocks, neural networks, embeddings, and tokenization. These are the ingredients that everything else is built

from. Part two covers the architecture, including attention mechanisms, the Transformer block, the difference between encoder and decoder architectures, and how the pieces fit together. This is where you build a Transformer from scratch. Part three covers training, from pre-training on massive text through **gradient descent**, fine-tuning with LoRA and QLoRA, and alignment with RLHF and **Direct Preference Optimization (DPO)**. How the model learns. Part four covers deployment and optimization, including quantization, pruning, KV cache, Flash Attention, continuous batching, and tensor parallelism. Making models fast and useful in production. Part five covers applied fine-tuning projects, with four hands-on deep dives where you fine-tune models for classification, legal document drafting, SQL generation, and function calling.

That is a lot of ground. But you do not need to be a specialist to follow it. Each part builds on the one before, so finish Part one and you will have everything you need for Part two, and so on. The book is designed for a wide range of readers: software engineers who want to understand the AI tools they are using and building with, data scientists looking to add LLM expertise to their toolkit, ML engineers who want deeper architectural understanding, researchers and students entering the field, entrepreneurs and product managers who need to make informed technical decisions, and anyone who is curious about how the technology actually works and willing to put in the effort to find out. Comfortable with Python is assumed. A machine learning background helps but is not required, because we build everything from scratch.

Wherever you are starting from, the goal is the same: go from using LLMs to understanding what is happening inside them. By the final chapter, you will be able to trace how a sentence becomes tokens, how tokens become embeddings, how attention transforms those embeddings, and how the whole system gets trained, aligned, and deployed. LLMs are reshaping work, learning, and creative practice all at once, and the engineers who really understand the mechanics (not just the Application Programming Interface) will have an outsized role in shaping what comes next. As these models become more powerful and more deeply integrated into everything, that understanding will only become more important. Turn the page and we begin.

Before you turn the page to Chapter 2 (Architecture Archetypes: Encoders, Decoders & Everything In Between), there are a handful of misconceptions about LLMs that trip up nearly everyone who is learning this material for the first time. Naming them now, while the core ideas are still fresh, will save you from a whole class of bugs and bad assumptions later.

Common Mistakes and Gotchas

> *Confusing "large" with "smart." The "large" in Large Language Model refers to the number of parameters and the scale of training data, not to some qualitative measure of intelligence. A model with more parameters is not automatically better at every task. Smaller, well-fine-tuned models frequently outperform larger general-purpose models on specific domains.*

> *Treating LLMs as databases of facts. LLMs do not "know" things the way a database stores records. They have learned statistical patterns from training data. When a model states a fact confidently, it is pattern-matching, not retrieving a verified record. This is why hallucination happens and why you should never trust LLM output on factual claims without verification.*

> *Assuming the model "understands" your intent. When a model produces a correct answer, it is tempting to conclude it understood the question the way a human would. It did not. It predicted tokens that are statistically likely given the input. Rephrasing the same question can produce a completely different (and sometimes wrong) answer, which a genuine understanding would not allow.*

> *Ignoring the role of tokenization. Many confusing model behaviors trace back to tokenization. The model does not see words; it sees tokens, which may split words in unexpected ways. "ChatGPT" might be one token or three, depending on the tokenizer. Understanding tokenization, which we cover in Chapter 3 (Foundations of Large Language Models), Section 3.2, prevents a whole class of debugging headaches.*

> *Thinking prompt engineering is all you need. Prompt engineering is valuable, but it has limits. If you need consistent, reliable behavior for a production application, fine-tuning (covered in Chapters 12 through 17) is almost always more reliable than trying to get the perfect prompt.*

Reading about next-token prediction, hallucination, and emergent behavior is one thing. Poking at a live model is another. The three exercises below give you hands-on experience with the ideas we just covered, from watching a model assign probabilities to its own predictions to catching it hallucinating in real time.

Try It Yourself

★☆☆ Exercise 1.1: Next-Token Prediction Intuition

Open any LLM (ChatGPT, Claude, or a local model) and give it the prompt "The capital of France is," then ask it to show you its top five most likely next tokens with probabilities (some APIs support this via logprobs). Write down what you observe. Now change the prompt to "The capital of the country that built the Eiffel Tower is" and compare. Are the probabilities different? Why might the model assign different confidence levels to the same factual answer depending on how the question is phrased?

Expected insight: The model's confidence depends on the token sequence it has seen, not on "knowing" the fact. Different phrasings activate different attention patterns.

★★☆ Exercise 1.2: Hallucination Detection

Ask an LLM to write a short biography of a real but relatively obscure person (for example, a mid-career researcher or a local politician). Then fact-check every claim in the output. How many statements are verifiably true? How many are plausible-sounding but fabricated? Write a one-paragraph analysis of where the model's output diverges from reality and why next-token prediction makes hallucination an inherent risk rather than a bug to be fixed.

Expected insight: The model generates text that is statistically plausible, not factually verified. Hallucination is a natural consequence of the training objective.

★★☆ Exercise 1.3: Three Eras Comparison

For each of the three eras of language modeling described in this chapter (statistical, neural, Transformer), write one concrete example of a language task that era could handle and one it could not. For instance: an n-gram model can complete "New York ___" with "City" but cannot resolve a pronoun reference across a paragraph. Explain why the architectural limitation of each era prevents it from handling the harder task.

Expected insight: Each era's limitations are not just about scale but about fundamental architectural constraints: fixed windows, sequential processing, and attention span.

Once you have tried the exercises, test whether the concepts have genuinely landed. The questions that follow are not definition drills; they are the kinds of scenarios you will actually face on the job, in design reviews, and in technical interviews when someone asks you to justify an architectural choice.

Knowledge Check Q&A

These questions are modeled on real interview scenarios at major technology companies. They test not just recall but understanding. Try to answer each one before reading the answer.

Q1.1. A colleague is building a customer support chatbot and says, "We do not need to understand how the LLM works internally, we just call the API and tweak the prompt until it works." What are the specific risks of this approach, and what could go wrong in production that prompt-tweaking alone cannot fix?

Q1.2. Your team is evaluating whether to use a single large foundation model or train separate smaller models for three different tasks: sentiment analysis, document summarization, and code generation. What are the trade-offs, and under what circumstances would you choose one approach over the other?

Q1.3. Someone claims that GPT-4 "truly understands language just like humans do" because it can pass the bar exam and write poetry. Using what you learned in this chapter about next-token prediction and emergence, construct a technically precise response.

Q1.4. A developer fine-tuned an LLM for medical question-answering, but the model sometimes generates confident answers that contain dangerous misinformation. The developer says, "The model scored 92 percent on our test set, so it should be safe." What is wrong with this reasoning?

Q1.5. A junior engineer asks why your team uses a Transformer instead of a larger LSTM, arguing that it has fewer parameters and trains faster. Walk them through the three key limitations of recurrent

architectures that Transformers solve, using specific examples from the chapter.

Q1.6. Your company wants to deploy an LLM-powered agent that can execute database queries, send emails, and modify files on behalf of users. What safeguards would you put in place, and why does understanding the internals matter more for agentic systems than for question-answering?

Q1.7. A product manager asks you to explain "emergence" in LLMs and whether the team should count on emergent capabilities appearing when they scale up their model. What would you advise?

Q1.8. A colleague complains that their LLM gives different answers to the same factual question depending on how they phrase it. One phrasing produces the correct answer confidently, while a slight rephrasing produces a hallucination with equal confidence. Explain why.

Q1.9. A junior engineer wants to skip straight to the fine-tuning chapters because that is what they need for their current project. Explain the dependency chain and what problems they will encounter.

Q1.10. An executive reads that inference costs have dropped 10 to 100 times and concludes that cost is no longer a concern for LLM deployment at scale. What nuances is the executive missing?

With the interview scenarios behind you, it is worth pausing to tie the whole chapter together in one place. The summary below does not just rehash the sections; it connects next-token prediction, the three historical eras, and today's agentic shift into a single narrative you can carry into the rest of the book.

Summary

A large language model is, at the lowest level, a mathematical function: it takes a sequence of tokens as input and outputs a probability distribution over the next token. That single operation, next-token prediction, is the entire mechanism. Everything else that appears to be intelligence, creativity, or reasoning emerges from applying that operation billions of times during training on trillions of tokens of text. The model picks up grammar, facts, reasoning patterns, and common sense not because anyone programmed those things in, but because they are useful for making better predictions. This simplicity of objective combined with scale of training is what makes LLMs simultaneously powerful and surprising, and it is why the term "foundation model" has taken hold, reflecting the reality that one pre-trained model can be adapted to thousands of downstream tasks.

The path to today's LLMs runs through three distinct eras, each solving a fundamental problem that the previous era could not. Statistical n-gram models proved that treating language as data rather than rules was the right approach, but they were limited to fixed context windows and had no concept of meaning.

Neural networks (first with Word2Vec's learned word representations, then with RNNs and LSTMs) added real semantics and sequential memory, but they could not parallelize training and their memory faded over long sequences. The Transformer, introduced in 2017, replaced sequential processing with self-attention, letting every token attend to every other token simultaneously. That one architectural change unlocked the parallelism that made scaling possible, and scaling is what turned language models from academic curiosities into the most rapidly adopted technology in history.

Understanding these internals is not an academic exercise. The field has moved beyond question-answering into reasoning models that chain thoughts, agentic systems that take real-world actions, and fine-tuning techniques that let small teams specialize models on a single GPU. At the same time, persistent challenges (hallucination, adversarial fragility, the absence of true memory) mean that the engineers who understand what is happening inside the model are the ones who build reliable systems, debug failures efficiently, and make informed architecture decisions. Whether you are writing prompts, fine-tuning for a domain, or deploying at scale, understanding the Transformer architecture gives you an edge that no amount of API calls alone can provide.

The summary closed the loop on what LLMs are and why they matter. The natural question now is structural: if every frontier model is built on the Transformer, what does that architecture actually look like, and why do some models read while others write? Chapter 2 (Architecture Archetypes: Encoders, Decoders & Everything In Between) splits the family tree open.

What's Next

In Chapter 2 (Architecture Archetypes: Encoders, Decoders, and Everything In Between), we will meet the four architecture families that define modern natural language processing: encoders, decoders, encoder-decoders, and auto-encoders. You will learn what each architecture does, when to use it, and, most importantly, why modern LLMs are almost exclusively decoder-only. We will look at how causal masking works mechanically in the attention layer, which is the mechanism that makes decoders autoregressive. By the end of that chapter, you will have the architectural vocabulary to understand why GPT, LLaMA, and Claude are built the way they are, and you will be ready to start building the attention mechanism from scratch in the chapters that follow.

Chapter 2
Architecture Archetypes: Encoders, Decoders & Everything In Between

In Chapter 1 (Understanding Large Language Models: From Mystery to Mastery), we learned that every **Large Language Model (LLM)** is, at the lowest level, a **next-token prediction** machine. But that raises an immediate question: if the basic idea is always "predict the next piece of text," why are there so many different kinds of models? Why does **Bidirectional Encoder Representations from Transformers (BERT)** work differently from **Generative Pre-trained Transformer (GPT)**? Why can ChatGPT write you a poem but cannot tell you which words in a sentence are people's names as accurately as a model half its size? The answer comes down to **architecture**, the floor plan that determines how data flows through the model. Two buildings can use the same bricks (the same math, the same neural network layers), but if one is designed as a library and the other as a factory, they will be good at very different things. This chapter gives you the vocabulary to tell those blueprints apart.

Every model you will meet in this book, and every model deployed in the real world, belongs to one of four **architecture archetype** families. **Encoders** read and understand text, and BERT is the canonical example. **Decoders** write and generate text one word at a time, which is what GPT does inside ChatGPT, Claude, and **LLaMA**. **Encoder-decoders** combine reading and writing for tasks like translation, the pattern used by **Text-to-Text Transfer Transformer (T5)** and **BART**. **Auto-encoders** compress data through a narrow bottleneck and reconstruct it, learning what matters most, which is how **Variational Auto-Encoders (VAEs)** power image generation inside Stable Diffusion. By the end of this chapter you will be able to look at any model and immediately know which family it belongs to, and that single skill tells you what the model can and cannot do.

We start with encoders, the architecture that reads and understands, and build every concept from scratch with everyday pictures and concrete examples before any technical detail appears. Then we introduce decoders and auto-regressive generation, walking through the exact mechanism that powers every chatbot you have ever used. Next we combine the two into encoder-decoder models and see how cross-attention bridges reading and writing. We close with auto-encoders and a grand taxonomy table that puts all four families side by side with runnable code. If Chapter 1 (Understanding Large Language Models: From Mystery to Mastery) gave you the map of the LLM field, this chapter hands you the compass: a precise vocabulary for the four architectural families that every model in this book belongs to.

2.1 Encoders: The Reading Machine

Before we look at any specific model, let us start with a question you can answer from everyday experience. Read this sentence: "I deposited money at the bank." Now read this one: "I sat on the bank of the river." The word "bank" appears in both, but it means completely different things. You knew that instantly because you read the entire sentence before deciding what "bank" meant. You looked at "deposited" and "money" in the first sentence and concluded: financial institution. You looked at "river" in the second and concluded: the edge of a waterway. That ability to read everything first, then use the full context to understand each word, is exactly what an **encoder-only** model does. An encoder is the "reader" of the model world: it takes in a full sentence and produces, for each word, a rich description that captures what that word means in this particular context.

What Is a Vector? The Word's Report Card

When we say the encoder produces a "rich description" of each word, what does that actually look like? It is a list of numbers, hundreds or thousands of them. In the world of mathematics and machine learning, a list of numbers is called a **vector**. But do not let the terminology intimidate you. Imagine every word in a sentence gets a report card. Instead of grades like A, B, C, this report card has hundreds of numerical scores, each measuring a different aspect of the word's meaning in context. One score might capture "how much does this word relate to finance?" Another might capture "is this word the subject of the sentence?" Another: "is this word positive or negative in tone?" No single score tells you much, but together they paint a detailed portrait of what the word means right here, right now, in this specific sentence.

BERT, the most famous encoder, produces a vector of 768 numbers for each word. Why 768? There is no deep reason; it was a design choice by the researchers, like choosing the number of rooms in a building. Bigger models use more numbers (1024, 4096, or even more), capturing finer-grained distinctions. The key point is this: the vector for "bank" in "I deposited money at the bank" will be a different list of 768 numbers than the vector for "bank" in "I sat on the bank of the river", even though the word itself is identical. The encoder figured out the difference by reading the full sentence.

Bidirectional Attention: Reading the Whole Sentence at Once

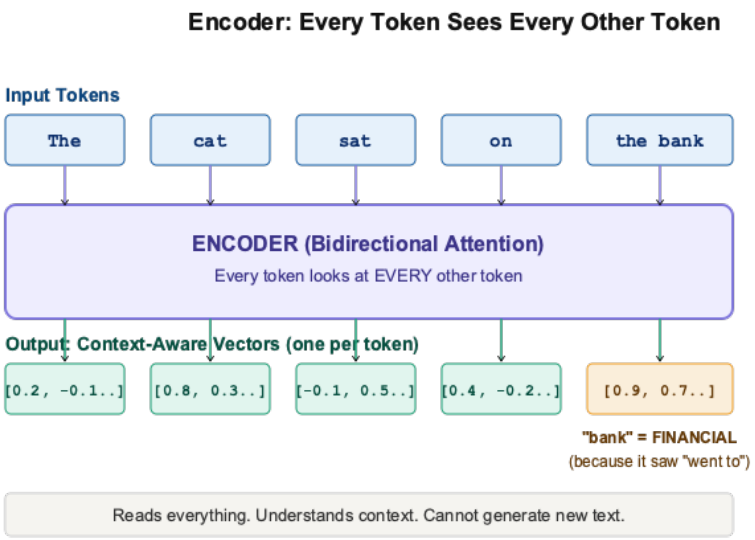
How does the encoder manage to consider every word in the sentence when building the report card for one specific word? Through a mechanism called **attention**. "Bidirectional" just means each word is allowed to look in both directions at once, at words to its left and to its right, with nothing blocked. We will cover attention in deep mathematical detail in Chapters 4 through 6 (Positional Encoding & Attention Projections, Single-Head Attention, Multi-Head Attention, Residuals & Layer Norm), but the intuition is simple enough to grasp right now.

Imagine you are in a study group with four other students. You are all reading the same paragraph. When you encounter a confusing word, you glance around the room: "Does anyone have context that helps me understand this word?" Each student holds up a sign showing how relevant their word is to yours. The student holding "money" holds up a big sign when you are trying to understand "bank" in a financial context. The student holding "river" holds up a big sign in the geographical context. You combine everyone's contributions, weighted by how relevant they are, to form your understanding. That is

attention: every word looks at every other word and asks, "How much should I pay attention to you when figuring out what I mean?"

In an encoder, this attention is **bidirectional attention**, which means every word can look at every other word, both to its left and to its right, at the same time, with no restrictions. The word "bank" at position 5 can look at "river" at position 8 just as easily as it can look at "I" at position 1. This is the encoder's great strength and also its core limitation: because it reads everything at once, it builds rich, context-aware representations, and because it reads everything at once, it cannot generate new text (generation requires producing words one at a time, in order, which is incompatible with seeing everything simultaneously). We will explore exactly why in Section 2.2.

Here is a concrete example. Consider: "The trophy didn't fit in the suitcase because it was too big." What does "it" refer to, the trophy or the suitcase? You know the answer because you read the whole sentence, including the phrase "too big," which rules out the suitcase. An encoder can figure this out too, because when it processes "it," it simultaneously sees "trophy," "suitcase," and "too big." A decoder, which we will meet in Section 2.2, cannot do this during training: when processing the word at position 5, it is forbidden from looking at positions 6, 7, 8 and can only look backward. That "only look backward" rule is called **causal masking** (or **causal attention**), because the cause (earlier words) must come before the effect (the word being predicted), never the other way around.



Encoder architecture: all input tokens are processed simultaneously with bidirectional attention. Each token can see every other token. The output is one context-aware vector per token. In the figure, the vector for "bank" is drawn in amber because its meaning was shaped by the full surrounding context.

How Encoders Learn: The Fill-in-the-Blank Game

If the encoder can see every word in the sentence, you might wonder: how do you train it? You cannot use the approach from Chapter 1 (Understanding Large Language Models: From Mystery to Mastery), "predict the next word," because the encoder can already see the next word and would just cheat by looking at the answer. So the researchers behind BERT invented a clever trick: cover up some words and

ask the model to guess what is underneath. This is exactly like the fill-in-the-blank exercises you did in school.

Take the sentence: "The cat sat on the mat." The training procedure randomly replaces about 15 percent of the words with a special placeholder **token** called [MASK]: "The [MASK] sat on the [MASK]." The model's job is to predict that the first blank should be "cat" and the second should be "mat." To get this right, the model must read all the surrounding words, because it needs "sat" and "on the" to guess that "cat" fits the first gap. Because it must look in both directions to fill the gap, it naturally develops bidirectional attention. This training technique is called **masked language modeling (MLM)**, and it is how BERT was trained. The intuition is simple: hide a few words at random, force the model to guess them from everything around them, and over billions of sentences the model learns what words normally fit where. BERT was originally trained on a second, smaller task too, called Next Sentence Prediction (NSP), which showed the model two sentences and asked whether the second actually followed the first in the source text; later encoders dropped NSP because MLM on its own already teaches most of what the model needs.

> *The mask prevents cheating. If we did not hide any words, the model could just copy the input to the output and learn nothing. The mask forces the model to actively reason: "I cannot see this word. Based on all the words around it, what should it be?" That is exactly the kind of contextual reasoning we want the encoder to develop.*

There is one important detail about BERT's masking strategy. Of the 15 percent of tokens selected for masking, 80 percent are replaced with [MASK], 10 percent are replaced with a random wrong token, and 10 percent are left unchanged. Why? If the model only ever saw [MASK] tokens as the ones it needed to predict, it would learn to ignore every position that is not visibly masked. The random replacements and unchanged tokens force the model to treat every position as potentially needing contextual support, which produces better representations for all tokens, not just the masked ones.

A quick term worth naming now, because it will come up again. This fill-in-the-blanks exercise is called **pre-training**: the model learns general language skill from huge amounts of plain text, with no human labels. Once pre-training is done, the same model is then **fine-tuned** on a specific task like classifying contracts or tagging names, using a much smaller labeled dataset. Pre-training teaches the model language; fine-tuning teaches it your job. Chapter 12 (Fine-Tuning Large Language Models) covers fine-tuning in full.

The Expert Reader at the Publishing House

Picture a highly trained editor at a publishing house. Their job is to read an entire manuscript before annotating a single word. They read the full sentence, sometimes the full paragraph, and then annotate every word with its role: "this 'bank' is financial," "this 'set' is a noun," "this 'run' is a verb." The editor never writes new sentences, only annotates existing ones, so their output is the same document with every word now carrying a rich sticky note describing its role, meaning, and relationship to every other word. That sticky note is the encoder's output vector, the report card we discussed earlier.

Compare this to a casual reader who reads word by word from left to right and has to annotate each word before reading the next one. This reader will label "bank" as "financial" even in the sentence "I stood on

the river bank," because they have not read "river" yet. The encoder is that expert editor, while the casual reader is the causal decoder we will meet next: BERT is the editor, GPT is the casual reader who happens to be very fast and very well read.

The expert-editor image gives you the intuition, and the track record of real systems is what turned encoders into the backbone of an entire generation of NLP systems.

What Encoders Excel At, and What They Cannot Do

Encoders dominated Natural Language Processing (NLP) from 2018 to 2022, where NLP covers everything a computer does with human language, from tagging names in a document to answering questions about a passage. BERT and its variants (RoBERTa, ALBERT, DeBERTa) set records on reading comprehension, sentiment analysis, and question answering, because reading the full context gives encoders richer representations for understanding tasks. Today, encoders are still the backbone of semantic search and retrieval: when you type a query into a search engine and it finds semantically similar documents rather than just keyword matches, an encoder is running under the hood, turning every document into a vector, turning your query into a vector, and returning the nearest neighbors. In a Retrieval-Augmented Generation (RAG) system, which is the technology behind many modern AI assistants, an encoder turns documents into searchable vectors while a decoder generates the answer.

Encoders excel at **classification** ("Is this review positive or negative?"), **Named-Entity Recognition (NER)** ("Which words are people or places?"), question answering ("Where in this passage is the answer?"), semantic similarity ("Are these two sentences about the same thing?"), and feature extraction for retrieval and search. What encoders cannot do is generate new text: they produce vectors rather than words, so they cannot complete a sentence, write a story, answer open-ended questions, or carry a conversation. Anything that requires producing a sequence of new tokens is beyond what an encoder can do.

One more design detail worth knowing. When you need to classify a whole sentence rather than individual words, you need a single vector that represents the entire input. BERT's solution is a special token called [CLS] (short for "classification") that is prepended to every input. After the encoder runs, the output vector at the [CLS] position is used as a summary of the entire sentence. How can a single vector of 768 numbers summarize a whole sentence? Through attention. During encoding, the [CLS] token attends to every other token, and over multiple layers it aggregates information from the whole sequence, so by the final layer its vector reflects a weighted mixture of everything the model has read, much like a student who has read the whole chapter and written a one-paragraph summary. In practice, to classify a sentence, you feed it to BERT, take only the [CLS] output vector, and attach a small classifier on top. That one vector carries enough information about the whole sentence to make the decision.

Encoders are outstanding readers, with full bidirectional context that captures nuance, ambiguity, and long-range relationships, but they cannot write a single new word. For that, we need the opposite architecture: one that gives up the ability to see everything at once in exchange for the ability to generate text, one token at a time. That is the decoder, and it is the engine behind every chatbot, code assistant, and text generator you have ever used.

2.2 Decoders & Auto-Regressive Models

Now imagine you are writing a story, one word at a time. You can look back at what you have already written, but you cannot peek at what comes next, because what comes next does not exist yet; you are creating it. That is exactly what a decoder does. While an encoder reads a complete sentence and understands it, a decoder generates text from scratch, choosing each word based solely on the words it has already produced. This left-to-right, one-word-at-a-time process is called **autoregressive** generation (sometimes written auto-regressive), and it is the mechanism behind **decoder-only** models like GPT-4, Claude, LLaMA, Gemini, and every other chatbot you have used.

The word "auto-regressive" might sound intimidating, but it describes something you do every time you speak. When you say a sentence, you choose each word based on the words you have already said. You do not plan the entire sentence in your head and then recite it; you build it incrementally, word by word, adjusting as you go. A decoder works the same way: given "The cat sat," it predicts the most likely next word ("on"), appends it, then predicts the next word given "The cat sat on" ("the"), and so on. Each prediction depends on the entire history of what has come before, but never on what comes after.

The Generation Loop: Step by Step

Let us walk through exactly how a decoder generates text, with a concrete example. Suppose we give the model the prompt "The cat" and ask it to continue.

Input: "The cat" → Predict next token

The model takes the two tokens "The" and "cat" as input. It processes them through its layers and outputs a probability distribution over its entire vocabulary, typically 50,000 or more possible tokens. A probability distribution just means every word in the vocabulary gets a number between zero and one, and those numbers add up to one, so you can read each number as "how much of the total likelihood lives on this word." The model might assign P("sat") = 0.31, P("is") = 0.12, P("ran") = 0.08, and so on for all 50,000+ options. It picks "sat" (the highest probability).

With "sat" chosen, it joins the input and becomes part of the history for the next prediction. Step 2 restarts the whole process with one extra token.

Input: "The cat sat" → Predict next token

Now the model takes three tokens as input: "The", "cat", "sat". It runs another forward pass and produces a new probability distribution. This time, "on" gets a high score (P = 0.42), so it picks "on". The key point: the model sees all three previous tokens when making this prediction, but it cannot see what will come after "on", because that does not exist yet.

"on" is now part of the input. Step 3 makes the pattern unmistakable: every new token simply extends the history and triggers another pass.

Input: "The cat sat on" → Predict next token

Four tokens in. The model predicts "the" (P = 0.38). The pattern continues: predict, append, predict, append. Each step uses the entire history but never peeks ahead. This loop runs until the model produces a special end-of-sequence token, or until a maximum length is reached.

This predict-one-then-append loop is the entire generation mechanism. Every sentence ChatGPT has ever written was produced exactly this way: one token at a time, left to right, never looking ahead. The mathematical name for this process is the chain rule of probability: the probability of a full sentence equals the product of each token's probability given all the tokens before it. In notation: P("The cat sat on the mat") = P("The") multiplied by P("cat" given "The") multiplied by P("sat" given "The cat") and so on. Every large language model you have ever chatted with uses this exact formula.

The Blindfolded Storyteller

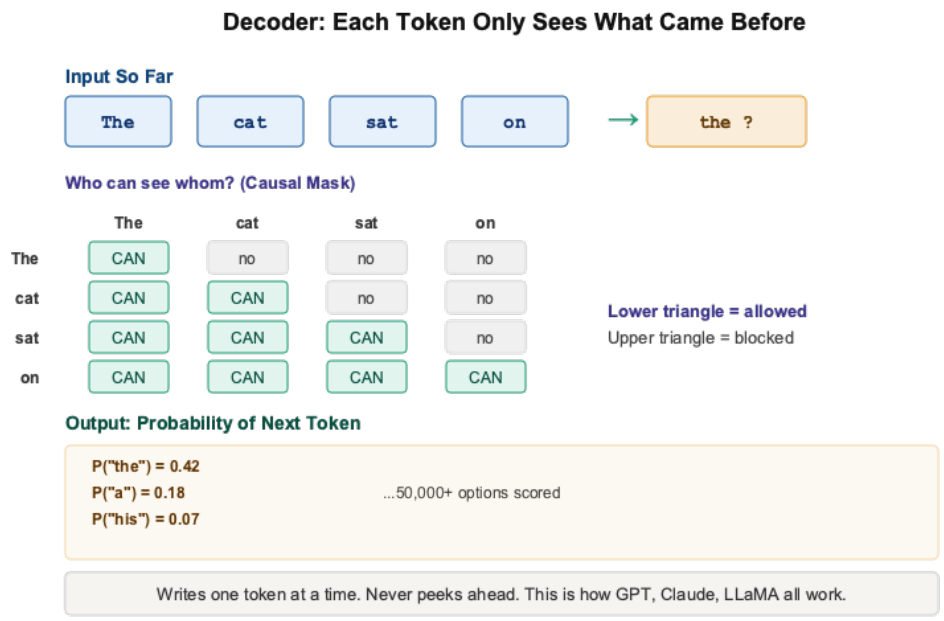
Picture a storyteller sitting on a stage, blindfolded, while someone reads her the story so far, word by word. After hearing each new word, she says aloud the next word she thinks should come next, with no way to glance at the script ahead of her and no way to re-read the beginning on her own. She can only hear what has been read so far and speak one word at a time, which is exactly what a decoder does: it only knows the left context, it generates exactly one token per step, and its whole job is to answer the question "given everything I have heard so far, what word comes next?" The loop is simple: hear the history, speak the next word, add that word to the history, repeat.

"Hear the history, speak one word, add it to the history, repeat" is literally what GPT-4 does, billions of times a day, at a scale no stage storyteller could match.

2.2.1 How Causal Masking Actually Works in Attention

Here is a question that should bother you. During training, the model already has the full sentence, because it is training on text that already exists. So how does it enforce the "no peeking ahead" rule while processing everything at the same time? The answer is a mechanism called causal masking, and it is the single architectural difference between an encoder and a decoder. The word "causal" means cause must come before effect: each token can be influenced only by the tokens that came earlier, never by anything that comes later.

Remember that attention lets every token look at every other token to figure out context. In an encoder, there are no restrictions, so every token sees every other token. In a decoder, we add one constraint: when computing the attention for the token at position 3, block it from seeing tokens at positions 4, 5, 6, and so on. Only let it see positions 0, 1, 2, and itself at position 3. The mechanism is simple: before the attention calculation, set all the scores in the "future" positions to negative infinity. **Softmax** is the function that turns a list of numbers into a probability distribution; the big numbers become big probabilities and the small numbers become small probabilities, and everything adds up to one. Negative infinity is as small as a number can be, so when scores pass through softmax those positions become exactly zero. The model literally cannot attend to future tokens, because they contribute nothing to the weighted sum.



Decoder architecture with causal masking. Each row shows which tokens a given token can attend to (green = allowed, gray = blocked). 'The' can only see itself. 'cat' sees 'The' and itself. This lower-triangular pattern is the causal mask. The output is a probability distribution over the next token.

In code, causal masking is a single line:

code.py: example

```
# scores: shape (seq_len, seq_len) - attention scores
mask = torch.tril(torch.ones(seq_len, seq_len))          lower triangle = 1
scores = scores.masked_fill(mask == 0, float('-inf'))    upper = -inf
weights = torch.nn.functional.softmax(scores, dim=-1)
```

Step through what this code does. The function torch.tril creates a lower-triangular matrix, meaning ones on and below the diagonal and zeros above it. For a sequence of 4 tokens, this matrix looks like: row 0 = [1,0,0,0], row 1 = [1,1,0,0], row 2 = [1,1,1,0], row 3 = [1,1,1,1]. The masked_fill call replaces every zero position with negative infinity. After softmax, those negative-infinity positions become exactly zero probability. Token 0 can only attend to itself. Token 1 can attend to tokens 0 and 1. Token 2 can attend to tokens 0, 1, and 2. And so on. This is the autoregressive property: each token sees only the past.

Causal-mask execution

```
causal_mask(scores)  # scores shape: [seq_len=4, seq_len=4]
│
├─ torch.ones(4, 4)
│   └─ tensor of 1s, shape [4, 4]
│
├─ torch.tril(ones)
│   └─ lower-triangular tensor, shape [4, 4]
│       row 0 = [1, 0, 0, 0]
│       row 1 = [1, 1, 0, 0]
│       row 2 = [1, 1, 1, 0]
│       row 3 = [1, 1, 1, 1]
│
├─ scores.masked_fill(mask == 0, -inf)
│   └─ upper-triangle cells become -inf, shape [4, 4]
│
└─ F.softmax(scores, dim=-1)
    └─ each row sums to 1; -inf slots become exactly 0
       final attention weights shape [4, 4]
```

> *Why negative infinity instead of just zero? If you set masked scores to zero instead of negative infinity, softmax would still assign a small positive probability to those positions (because e to the power of 0 equals 1). The model could then "leak" information from future tokens. Using negative infinity guarantees exactly zero probability, which is mathematically equivalent to those tokens not existing at all.*

Here is the remarkable thing: the Transformer architecture is identical for both encoders and decoders. Same self-attention layers, same feed-forward blocks, same layer normalization. The entire difference is this one mask. That single change flips the model from "understander" to "generator." In modern implementations like Flash Attention and PyTorch's scaled_dot_product_attention, the causal mask is built implicitly without materializing the full matrix, saving memory for long sequences.

2.2.2 Big Picture: Why Decoders Power Modern AI

If encoder-decoders can both read and write, and encoders produce richer representations, why do the biggest models (GPT-4, Claude, LLaMA, Gemini) use decoders alone? The strongest reason is that training is effortless, because feeding a decoder any text document means the training target is already sitting inside it (namely token number N+1) and no human labels are needed, which is why LLMs can be trained on something close to the entire internet. On top of that, in-context learning emerged naturally from the autoregressive objective, giving models the ability to perform new tasks from just a description and a few examples in the prompt, with no fine-tuning required. And because one unified model handles every task from translation to coding to reasoning, there is no separate encoder or cross-attention to design and scale, which keeps the engineering simple at every level.

There is one critical optimization that makes decoder generation practical: the **Key-Value (KV) cache**. Naively, generating 100 tokens would require 100 full forward passes (each processing an increasingly long sequence), which adds up to enormous computation. Inside attention, every token produces three vectors known as the Query, Key, and Value (Q, K, V). The Query is what the current token is looking for, while Keys and Values describe what every other token has to offer; we will unpack all three in Chapter 5

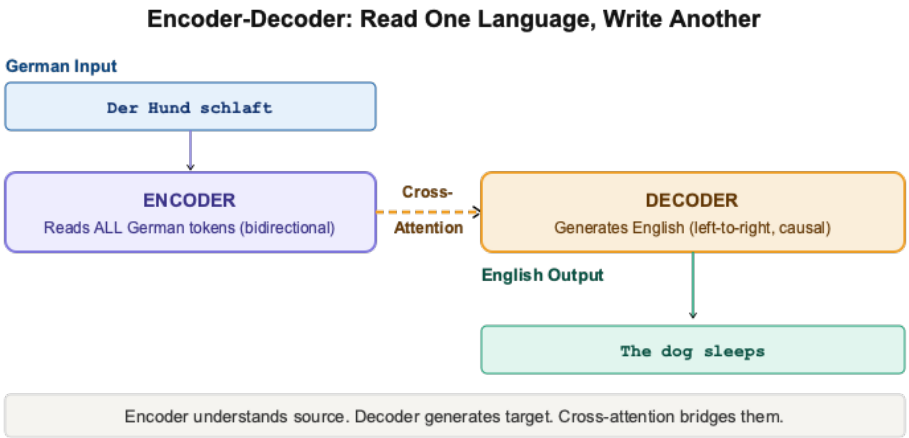
(Single-Head Attention). Because of causal masking, the Key and Value vectors for tokens already generated never change, so the model caches them. When generating token 100 the model only computes the Query, Key, and Value for the new token, then attends to the 99 cached Key-Value pairs from previous tokens, which reduces each new step's cost dramatically. Chapter 10 (LLM Inference: From Prompt to Generated Text) covers the KV cache in full mathematical detail.

Encoders read with bidirectional attention but cannot generate, while decoders generate with causal attention but only look backward. What if you need both, so that you can deeply understand one sequence and then produce a different one? That is what encoder-decoder models do, and translation is the clearest example.

2.3 Encoder-Decoders and Cross-Attention

Suppose you need to translate "Der Hund schläft auf dem Sofa" from German to English: "The dog is sleeping on the sofa." An encoder-only model would produce rich contextual vectors for the German words, but it cannot generate English. A decoder-only model could be prompted to translate, but it has no dedicated mechanism for deeply reading the source. The encoder-decoder solves this by splitting the job in two: an encoder that reads the German with full bidirectional attention, and a decoder that generates the English one token at a time. Those two halves need a way to talk to each other, and the bridge between them is called **cross-attention**. The combined design is often called **sequence-to-sequence**.

Picture a court interpreter in a courtroom. A witness speaks in Spanish for three minutes, and the interpreter listens to the entire statement, building a complete understanding; that listening stage is the encoder. The interpreter then turns to the judge and renders the statement in English, one phrase at a time, and while speaking each English phrase she mentally refers back to the Spanish statement to choose the right words, which is exactly what cross-attention does. The interpreter never invents content, because every English word is anchored in the Spanish source, and unlike a pure decoder she already read the whole source before starting to speak.



Encoder-decoder architecture for translation. The encoder reads the full German input bidirectionally. The decoder generates English left-to-right. Cross-attention (amber dashed arrow) lets the decoder consult the encoder's representations at every step.

The interpreter picture and the diagram both point at the same piece of machinery: the connection between the encoder's understanding and the decoder's next word. That connection has a precise mechanical definition.

How Cross-Attention Bridges Two Languages

In a standard encoder or decoder, self-attention lets each token attend to other tokens in the same sequence. Cross-attention is different: the Query vectors come from the decoder ("I am currently generating the word at position 3 in the English output"), while the Key and Value vectors come from the encoder's hidden states (the full German representation). In intuition terms, the decoder raises its hand with a question (Query), and every German token shows what it can offer (Keys and Values) so the model can pick the best match; Chapter 5 (Single-Head Attention) works all three out in detail. This means the decoder token can attend to any position in the encoder output, with no causal mask on the encoder side. The result is that when generating the English word "sleeping," the cross-attention mechanism puts high weight on the German token "schläft," even though the two words are in different languages and appear at different positions.

What makes cross-attention powerful is that the decoder does not just get a single summary from the encoder; it consults the full set of encoder hidden states at every generation step, picking which source tokens matter most for the word it is currently generating. A useful side effect falls out of this: when you train a translation model end-to-end, meaning all the pieces are trained together on translated pairs rather than being stitched from separate components, the cross-attention heads discover word-level correspondences between source and target on their own. Classical NLP researchers spent decades building explicit alignment algorithms for exactly this job, and the model learns the same thing from data alone.

> *If you visualize the cross-attention weight matrix (decoder position on one axis, encoder position on the other), you get a near-diagonal pattern for languages with similar word order such as English to German, and an inverted diagonal for languages with a different order such as English to Japanese, where the verb comes at the end of the sentence instead of in the middle. The model reinvented classical word alignment as a byproduct of end-to-end training.*

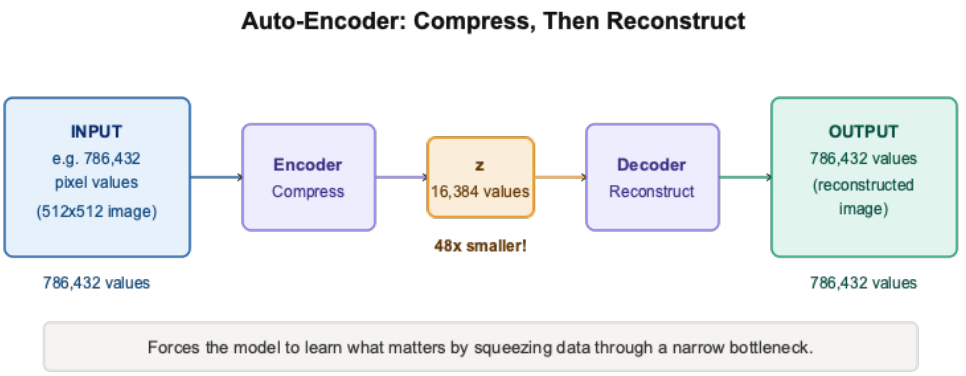
T5 (Text-to-Text Transfer Transformer), BART, and the original 2017 "Attention Is All You Need" Transformer all use this encoder-decoder pattern. T5 treats every NLP task as a text-to-text problem: prefix with "translate English to German:" for translation, "summarize:" for summarization, "question:" for question-answering. One architecture, many tasks. But if decoder-only models like GPT-4 can also translate and summarize, when would you actually use an encoder-decoder? Use an encoder-decoder when the output length and structure can differ substantially from the input. For tasks where source and output are the same type (open-ended generation, chat), decoder-only models dominate today because they scale better and are simpler to train.

That covers three of our four architecture families. The fourth is the auto-encoder, an architecture that looks superficially similar to the encoder-decoder but serves a very different purpose: learning to compress.

2.4 Auto-Encoders: Learning to Compress

Picture yourself moving to a new city. You have 50 boxes of stuff, but you can only take one small suitcase on the train, so you pack ruthlessly, keeping only the essentials. When you arrive, you use what you packed to reconstruct your life. The suitcase is the bottleneck, what you chose to pack are the learned features, and the act of unpacking and setting up your new home is the decoder reconstructing the output. If you were forced to do this packing thousands of times, you would get very good at knowing what is truly essential and what is noise, which is exactly what an auto-encoder learns.

An auto-encoder has an encoder half and a decoder half, but the goal is different from an encoder-decoder. Instead of translating from one sequence to another, an auto-encoder tries to compress its input down to a small set of numbers (called the latent representation or bottleneck) and then reconstruct the original from that compressed representation. The training target is the input itself, so no human labels are needed, and the loss is simply the difference between input and reconstruction. Because the signal has to squeeze through a narrow bottleneck, the encoder is pushed to keep only the most essential features and throw away noise.



Auto-encoder architecture. The input (786,432 pixel values for a 512x512 image) is compressed by the encoder to a small latent vector z (16,384 values, 48 times smaller). The decoder reconstructs the image from z. The bottleneck forces the model to learn what is essential.

Why would copying your own input teach you anything? Because you are not really copying; you are squeezing the signal through a much smaller representation, which forces the encoder to figure out what actually matters. Consider a small sentence example: if you compress "The cat sat on the mat" through a 4-dimensional bottleneck and can recover it perfectly, those 4 dimensions must encode the subject, the verb, the preposition, and the location. The model has learned the structure of the sentence, not by being told what structure is, but by trying to reconstruct it.

Variational Auto-Encoders: When You Want to Generate

Standard auto-encoders are great for compression, but they have a problem with generation: the latent space is unstructured, which means similar inputs can map to wildly different latent vectors and a random point you pick from the space might decode to garbage. The fix is the Variational Auto-Encoder (VAE), and the core idea is to encode every input not to a single point but to a small fuzzy cloud of possible

points. A useful way to picture this cloud: imagine pinning an input to a location on a map (the mean) and drawing a fuzzy circle around it (the variance) that says how far the cloud can spread. That bell-shaped cloud is called a **Gaussian**, and the mean is its center while the variance controls its width.

Once every input has its own fuzzy cloud in the latent space, we need the clouds to behave nicely. That is what Kullback-Leibler (KL) divergence does: it measures how different two probability clouds are, and adding a KL term to the training loss gently pulls every cloud toward one shared reference cloud, centred at the origin with a width of one, called a standard normal distribution. The effect is that similar inputs end up in nearby regions, no part of the latent space is empty, and you can walk smoothly from one decoded output to another. Pick any point and decode it, and you get a plausible output. Pick a point halfway between "cat" and "dog" and you might get a blurry cat-dog hybrid.

VAEs were foundational for generative modeling. They are now largely superseded by diffusion models for images, but the encoder-bottleneck-decoder pattern lives on in latent diffusion models like Stable Diffusion, which uses a VAE to compress images to a small latent space before the diffusion process operates.

Unlike encoders, decoders, and encoder-decoders, which all process sequences of tokens, auto-encoders can process data of any shape: images, audio, three-dimensional objects. They are trained by reconstruction loss (make the output match the input), not by next-token prediction or masked-token prediction. In the LLM context, auto-encoders appear as components inside larger systems rather than as standalone language models.

That is all four architecture families. Let us put them side by side.

2.5 Grand Taxonomy: All Four Architectures Side by Side

The table below summarizes everything we have learned. For each architecture, it shows the attention type, what goes in and what comes out, how it is trained, which famous models use it, and what it is best at. Refer back to this table whenever you encounter a new model and need to classify it.

Architecture Comparison Table

	Attention	In → Out	Training	Examples	Best For
Encoder	Bidirectional (full)	Tokens → Vectors	Masked LM (fill blanks)	BERT, RoBERTa	Classification, NER, search
Decoder	Causal (left-only)	Tokens → Next Token	Next-token prediction	GPT-4, Claude, LLaMA,	Generation, chat, code, reasoning

				Gemini	
Enc-Dec	Bidir enc + causal dec + cross-attn	Seq A → Seq B	Conditional generation	T5, BART, original Transformer	Translation, summarization
Auto-Enc	Varies	Input → z → Recon.	Reconstruction loss (+KL)	VAE, Stable Diff. encoder	Compression, generation

Each Architecture in Code: HuggingFace in 5 Lines

Theory is only half the picture. Let us see each architecture in action with minimal, runnable Python code using the HuggingFace Transformers library. Each snippet is fewer than ten lines. If you have never used HuggingFace before: AutoTokenizer loads the correct tokenizer for any model, AutoModel loads the model weights, and return_tensors="pt" tells the tokenizer to return PyTorch tensors rather than plain Python lists.

Listing 2.1: Encoder (BERT), Bidirectional Contextual Embeddings

transformers_example.py: usage

```python
from transformers import AutoModel, AutoTokenizer

tok = AutoTokenizer.from_pretrained("bert-base-uncased")
mdl = AutoModel.from_pretrained("bert-base-uncased")
inputs = tok("The cat sat", return_tensors="pt")
out = mdl(**inputs)
# out.last_hidden_state: (1, 5, 768)
# one 768-dim vector per token
```

Why 5 tokens, not 3? The tokenizer automatically adds two special tokens: [CLS] at the start and [SEP] at the end. So "The cat sat" becomes [CLS, The, cat, sat, SEP] = 5 tokens. Each gets its own 768-dimensional contextual vector. BERT returns hidden states, not generated text. To use these vectors, you add a task-specific head on top: a linear layer for classification, a token-level classifier for NER, or a similarity function for search.

BERT forward-pass execution

```
mdl(**tok("The cat sat", return_tensors="pt"))
│
├─ tokenizer splits + adds special tokens
│   ├─ [CLS] The cat sat [SEP]
│   └─ input_ids shape [1, 5]
│
├─ token embeddings + position embeddings
│   └─ shape [1, 5, 768]
│
├─ 12 Transformer encoder layers (bidirectional self-attention)
│   └─ each preserves shape [1, 5, 768]
│
└─ returns BaseModelOutput
    └─ last_hidden_state shape [1, 5, 768]
```

Listing 2.2: Decoder (GPT-2), Autoregressive Text Generation

transformers_example.py: usage

```python
from transformers import pipeline

gen = pipeline("text-generation", model="gpt2")

result = gen("The cat sat on",

             max_new_tokens=20,
             do_sample=True)

# result[0]["generated_text"]

# auto-regressive generation loop
```

The pipeline handles the entire auto-regressive loop internally: tokenize the prompt, run the forward pass, sample a token from the output distribution, append it, repeat for max_new_tokens iterations. The do_sample=True flag enables stochastic sampling (instead of always picking the highest-probability token), which produces more diverse and creative outputs. Each iteration uses the KV cache to avoid recomputing attention for already-generated tokens.

GPT-2 generation loop execution

```
gen("The cat sat on", max_new_tokens=20, do_sample=True)
│
├─ tokenize prompt
│    └─ input_ids shape [1, 4]
│
├─ loop for 20 steps:
│    ├─ forward pass on current tokens (KV cache reused)
│    │    └─ logits shape [1, seq_len, 50257]
│    ├─ take last position logits → shape [1, 50257]
│    ├─ softmax → probability vector
│    ├─ torch.multinomial(probs, 1) → sampled token id
│    └─ append token to input_ids (seq_len grows by 1)
│
└─ decode final token ids back to text
     └─ result[0]['generated_text'] : str
```

Listing 2.3: Encoder-Decoder (T5), Sequence-to-Sequence Translation

transformers_example.py: usage

```python
from transformers import T5ForConditionalGeneration, T5Tokenizer

tok = T5Tokenizer.from_pretrained("t5-small")
mdl = T5ForConditionalGeneration.from_pretrained("t5-small")
ids = tok("translate English to German: The cat sat",
          return_tensors="pt").input_ids

out = mdl.generate(ids)
# tok.decode(out[0]) -> "Die Katze sass"
```

The encoder processes "translate English to German: The cat sat" and produces contextual vectors. The decoder starts with a special start token and generates one German token at a time. At each step, the decoder's cross-attention layers let the current decoder state query the encoder's output, allowing "Katze" to attend heavily to "cat" in the encoder representations.

T5 translate execution

```
mdl.generate(input_ids)  # input_ids shape [1, src_len]
│
```

```
├─ encoder forward pass (bidirectional)
│   └─ encoder_hidden_states shape [1, src_len, 512]
│
├─ start decoder with <pad> start token → shape [1, 1]
│
├─ loop until <eos> or max_length:
│   ├─ decoder self-attention (causal) on decoder ids
│   ├─ cross-attention: Q from decoder, K/V from encoder_hidden_states
│   ├─ logits shape [1, 1, vocab_size]
│   ├─ argmax (greedy default) → next_token
│   └─ append next_token to decoder ids
│
└─ returns generated token ids, shape [1, tgt_len]
    └─ tok.decode(out[0]) → "Die Katze sass"
```

Listing 2.4: Auto-Encoder (VAE), Compress and Reconstruct

diffusers_example.py: usage

```python
from diffusers import AutoencoderKL
import torch

vae = AutoencoderKL.from_pretrained("stabilityai/sdxl-vae")
# encode to latent space
latent = vae.encode(image_tensor).latent_dist.sample()

# decode back to pixels
reconstructed = vae.decode(latent).sample
```

The VAE encoder outputs the parameters of a Gaussian distribution (mean and variance), not a fixed vector. The .sample() call draws a random sample from that distribution, introducing controlled randomness, and that random draw is what makes the approach "variational": it forces the latent space to be smooth and continuous, because the model has to decode sensible images even from slightly different points in the cloud. The input image tensor has shape [1, 3, 512, 512] (786,432 values), and the latent has shape [1, 4, 64, 64] (16,384 values), which is 48 times smaller. The reconstructed image is nearly identical to the original.

VAE encode-decode execution

```
vae.encode(image_tensor).latent_dist.sample()
│
├─ image_tensor shape [1, 3, 512, 512]  (RGB image)
```

```
│
├─ encoder CNN downsamples 8x
│   └─ feature map shape [1, 8, 64, 64]
│       (channels 0-3 = mean, channels 4-7 = log-variance)
│
├─ latent_dist = DiagonalGaussian(mean, logvar)
│
├─ .sample()  → mean + std * eps   (reparameterization)
│   └─ latent shape [1, 4, 64, 64]   (48x smaller than input)
│
└─ vae.decode(latent).sample
    └─ decoder CNN upsamples 8x → shape [1, 3, 512, 512]
```

With the four architectures mapped out and the code that exercises each one, you now have the vocabulary to classify any model you meet. A handful of easy-to-make mistakes about these architectures show up again and again in practice, and the callouts below walk through each one: the trap, why it bites, and how to avoid it.

Common Mistakes and Gotchas

> *Thinking encoders can generate text. They cannot. Encoders produce vectors, not words. If you need to generate text, you need a decoder.*

> *Confusing "bidirectional" with "better." Bidirectional attention gives richer representations for understanding tasks, but it prevents generation. Causal attention enables generation. Neither is universally better; they serve different purposes.*

> *Using causal masking values of 0 instead of negative infinity. Setting masked scores to 0 still allows softmax to assign small positive probabilities to future tokens, leaking information. Always use negative infinity.*

> *Assuming encoder-decoders are always better than decoders. For most tasks today (chat, code, reasoning), decoder-only models outperform encoder-decoders because of better scaling properties and simpler training. Encoder-decoders shine when input and output are structurally different (translation, summarization).*

> *Thinking auto-encoders are language models. Auto-encoders compress and reconstruct data. They appear as components inside larger systems (the VAE in Stable Diffusion) but are not standalone language models.*

> *Forgetting about the KV cache. Without caching Key and Value vectors from previous tokens, decoder generation would be prohibitively slow. Every production LLM system uses the KV cache. Chapter 10 (LLM Inference: From Prompt to Generated Text) covers it in detail.*

Knowing the pitfalls is only useful once you have run the architectures yourself. The exercises below turn the concepts from this chapter into hands-on work: comparing encoder and decoder output shapes, implementing a causal mask from scratch, and classifying real-world systems into the four architecture families.

Try It Yourself

★☆☆ Exercise 2.1: Encoder vs Decoder Output

Run the BERT code from Listing 2.1 and the GPT-2 code from Listing 2.2. Compare the outputs. BERT gives you a tensor of shape [1, 5, 768]. GPT-2 gives you a string of generated text. Explain in your own words why these outputs are completely different and what each is useful for.

Expected insight: BERT outputs vectors (useful for classification, search, similarity). GPT-2 outputs text (useful for generation, chat, completion). The architecture determines the output type.

★★☆ Exercise 2.2: The Causal Mask by Hand

Write out the 5x5 causal mask matrix for a sequence of 5 tokens by hand. Then implement it in PyTorch using torch.tril(torch.ones(5, 5)). Create a fake attention score matrix (any 5x5 matrix of random numbers), apply the causal mask using masked_fill, and verify that after softmax, no token attends to future positions.

```
import torch
mask = torch.tril(torch.ones(5, 5))
scores = torch.randn(5, 5)
scores = scores.masked_fill(mask == 0, float('-inf'))
weights = torch.softmax(scores, dim=-1)
print(weights)  # upper triangle should be all zeros
```

Expected insight: The upper triangle of the weight matrix is exactly zero. Each row sums to 1.0, but all the probability mass is on positions at or before the current token.

★★☆ Exercise 2.3: Architecture Detective

For each of these models, determine which architecture family it belongs to and explain your reasoning: (a) A model that takes a legal document and classifies it as 'contract', 'brief', or 'ruling'. (b) A model that takes an English sentence and produces a French translation. (c) A chatbot that generates responses to user messages. (d) A model that detects anomalous network traffic by comparing it to learned normal patterns.

Expected insight: (a) Encoder, because classification is an understanding task. (b) Encoder-decoder, because input and output are different languages. (c) Decoder, because chat is open-ended generation. (d) Auto-encoder, because it learns normal patterns via compression and flags inputs that reconstruct poorly.

After the exercises, sharpen the same concepts against the kinds of questions that come up in technical interviews and design reviews, where you have to justify architectural choices, explain trade-offs, and argue for one family over another.

Knowledge Check Q&A

The questions below are modeled on real interview scenarios at major technology companies. They test not just recall but understanding. Try to answer each one before looking it up in Appendix E.

Q2.1. If encoders are so good at understanding language, why can you not just use a BERT encoder to build a chatbot that answers questions?

Q2.2. "The man walked to the bank and withdrew money" vs "The man walked to the bank and sat by the water." If you encode both with BERT, will the vector for "bank" actually differ between them? Why?

Q2.3. If masking 15 percent of tokens is how BERT trains, what happens to the other 85 percent? Is the model also learning from unmasked tokens?

Q2.4. If decoders are so simple (just predict the next token), how can GPT-4 reason, translate, and write code?

Q2.5. A colleague says: "Why can I not just run a decoder in parallel like an encoder? It would be so much faster." How do you respond?

Q2.6. What stops a decoder from just repeating the same word forever?

Q2.7. Can you use a decoder-only model for translation instead of an encoder-decoder?

Q2.8. What is the latent space of an auto-encoder, really?

Q2.9. Your team needs to build three systems: (a) a document classifier, (b) a customer support chatbot, (c) a translation service. For each, which architecture family would you choose and why?

Q2.10. Is GPT's context window a kind of encoder? After all, the prompt is "read" by the model before generation begins.

With the interview-grade questions behind you, the four architectures should now sit clearly in your head as distinct tools with distinct jobs. The summary that follows threads the big ideas, the single design choice that separates readers from writers and why decoders came to dominate, into one narrative worth remembering.

Summary

Every model in the LLM universe belongs to one of four architecture families, and the differences between them come down to a single design choice: what can each token see? Encoders use bidirectional attention, letting every token see every other token simultaneously. This makes them outstanding readers because they produce rich, context-aware vectors for each word and power tasks like classification, named-entity recognition, and semantic search. Bidirectional attention is incompatible with generation, because if the model can see the answer it cannot learn to predict it. BERT (the canonical encoder) was trained with masked language modeling, where some words are hidden and the model has to fill in the blanks, and that single objective forced it to develop deep contextual understanding.

Decoders flip the trade-off. By applying a causal mask, a single line of code that blocks each token from seeing future tokens, the decoder gives up bidirectional reading in exchange for the ability to generate text. The auto-regressive loop (predict one token, append it, predict the next) is the mechanism behind every chatbot, code assistant, and text generator in existence. GPT-4, Claude, LLaMA, and Gemini are all decoder-only models. They won because of effortless training (no labels needed), emergent in-context learning, and architectural simplicity, with the KV cache making their sequential generation practical by caching Key and Value vectors from previous tokens.

Encoder-decoders combine both halves, connected by cross-attention that lets the decoder read the encoder's representations while generating. They are the natural fit for translation and summarization, where the input and output are different kinds of sequences. Auto-encoders serve a different purpose entirely: compressing data through a narrow bottleneck and reconstructing it, learning the essential structure of the data in the process. VAEs add a probabilistic layer that makes the latent space smooth enough for generation. Together, these four families (encoder, decoder, encoder-decoder, and auto-encoder) form the complete vocabulary you need to understand any model you will encounter in this book.

The summary gave you the taxonomy; the next chapter gives you the pipeline. Knowing a decoder exists is one thing, but tracing exactly how your raw text becomes a predicted token, stage by stage, is the next step.

What's Next

You now have the map (from Chapter 1, Understanding Large Language Models: From Mystery to Mastery) and the compass (from this chapter). In Chapter 3 (Foundations of Large Language Models), we zoom into the decoder, the architecture behind every modern LLM, and trace the full path from raw text to predicted tokens. You will learn how text becomes numbers (tokenization), how those numbers become meaningful vectors (embeddings), and how the Transformer processes those vectors through multiple layers to produce a probability distribution over the next token. The approach is the same as this chapter: first principles, everyday pictures, concrete examples, then code. By the end of Chapter 3, you will understand the complete data pipeline that every LLM uses, and you will be ready to build the attention mechanism from scratch in the chapters that follow.

Chapter 3

Foundations of Large Language Models

In Chapter 1 (Understanding Large Language Models: From Mystery to Mastery), you learned that a **Large Language Model (LLM)** is, at the lowest level, a **next-token prediction** machine. In Chapter 2 (Architecture Archetypes: Encoders, Decoders & Everything In Between), Section 2.2, you learned that the decoder is the architecture behind every modern chatbot and text generator, and that a single design choice, the causal mask, separates readers from writers. But we left a critical question unanswered: what exactly happens between the moment you type a sentence and the moment the model produces its reply? How does raw human text become numbers that a machine can process? How do those numbers travel through dozens of layers to emerge as a probability distribution over the next word? This chapter answers those questions by tracing the complete pipeline from the first character you type to the last token the model generates.

The chapter opens with the big picture: a bird's-eye view of the entire LLM pipeline, from raw text through **tokenization**, **embedding**, **positional encoding**, Transformer blocks, and finally the **language model head (LM head)** that produces the next token. Think of a jazz musician improvising. She has internalized thousands of songs during rehearsal (training), and when she plays live, she hears the notes played so far (input tokens), processes them through her musical intuition (Transformer layers), and produces the next note (inference). The LLM does exactly this, billions of times a day. From there the chapter moves into tokenization itself: why computers cannot read words directly, how text is broken into tokens, and how the **Byte-Pair Encoding (BPE)** algorithm builds an efficient **vocabulary** from scratch. The final sections cover token **embeddings**: what a **vector** actually is, how the model converts token numbers into rich mathematical objects, and why the embedding layer is both the simplest and one of the most important components of any LLM.

Every concept in this chapter gets the same treatment: why the problem exists before how the solution works, a concrete comparison before any mathematics, and a step-by-step trace with actual numbers before any abstract formulas. By the end you will understand the complete data path that every LLM uses, and you will be ready to tackle the attention mechanism in Chapter 4 (Positional Encoding & Attention Projections), Section 4.1, and Chapter 5 (Single-Head Attention), Section 5.1.

3.1 The Big Picture: What an LLM Actually Does

Before we examine any component in detail, you need the complete map. If you have ever used ChatGPT, Claude, or any other chatbot, you have interacted with a system that runs the same seven-stage pipeline every single time it produces a token. Understanding this pipeline at a high level is like understanding the floor plan of a building before you tour each room. You will always know where you are and why each room exists.

Here is the pipeline in plain English. You type a sentence (say, "The cat sat") and the **tokenizer** breaks that sentence into pieces called **tokens**, converting each piece into a number called a **token ID**. The

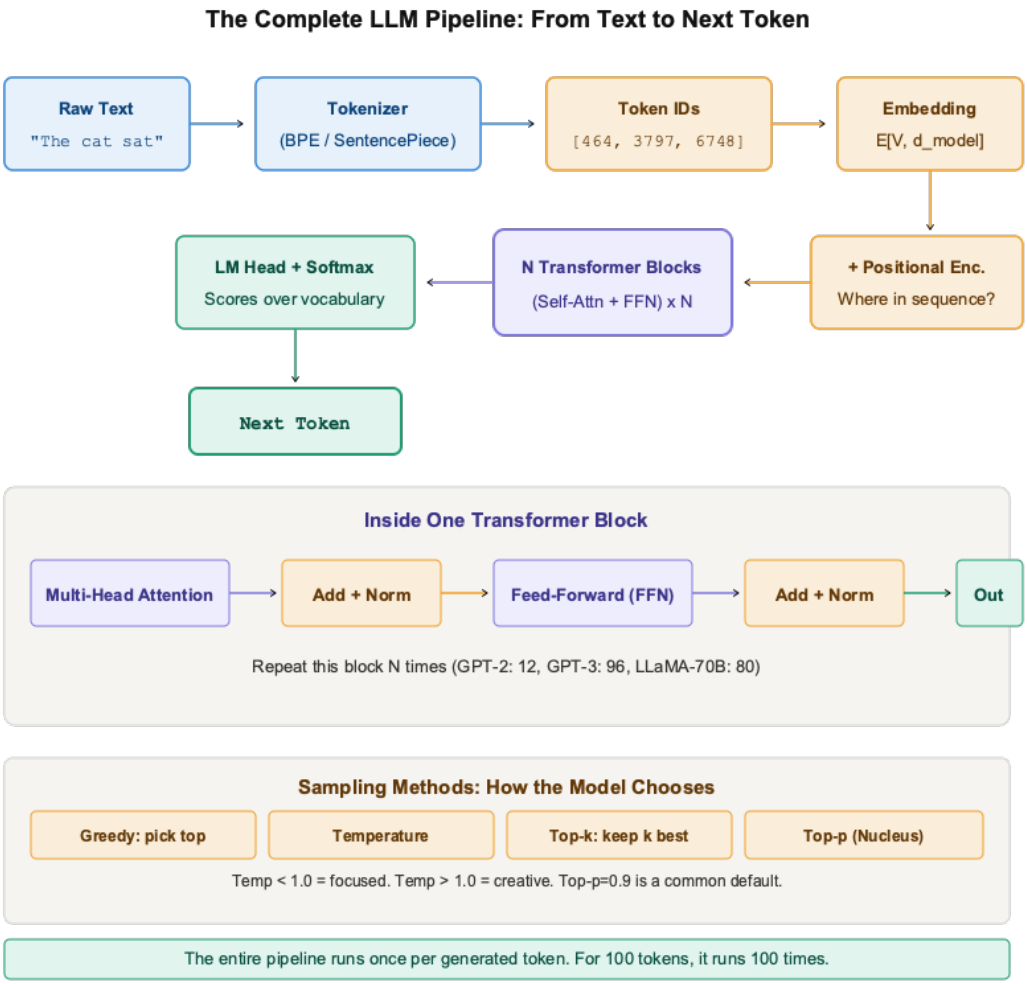
embedding layer converts each token ID into a vector, a long list of numbers that represents the token's meaning. **Positional encoding** then adds information about where each token sits in the sequence (first? fifth? hundredth?). The resulting vectors pass through a stack of Transformer blocks, each of which applies **self-attention** ("which other tokens should I pay attention to?") followed by a **feed-forward network (FFN)** ("what should I do with the information I gathered?"). After all blocks, a final layer called the language model head (LM head) converts the last token's vector into a score for every token in the vocabulary. A function called **softmax** turns those scores into probabilities, and the model picks a token to become your next word.

That is the pipeline in abstract terms. To make it visceral, imagine what the same process looks like in a completely different domain.

A Jazz Musician on Stage

Think of a jazz musician sitting in with a band she has never played with before. During rehearsals (training) she listened to thousands of songs, absorbing scales, chord progressions, rhythmic patterns, and the unwritten rules of jazz, and those patterns are now stored in her musical intuition, her parameters. When the band starts playing a new song live (inference), she hears every note that has been played so far (the input tokens), and her brain processes those notes through layers of musical understanding: she recognizes the key, the chord progression, the rhythmic feel (Transformer blocks). And then she plays the next note, just one note, before listening again, this time hearing the note she just played as part of the history, and producing the next one. She never looks at a score that tells her what comes next; she generates one note at a time, based on everything she has heard so far. That is exactly what a decoder-based LLM does.

The mapping is precise. Training is the rehearsals: the model processes billions of text passages and adjusts its parameters to predict the next token better. Inference is the live performance: the model takes a prompt, processes it through its layers, and generates one token at a time. The model's parameters are the musician's internalized knowledge, encoding patterns about language, facts, reasoning strategies, and style. The model never memorizes specific passages (just as the musician does not memorize every note of every song); instead, it learns statistical patterns that let it improvise fluently on any input.



The complete LLM pipeline. Raw text enters on the left, passes through the tokenizer, embedding layer, positional encoding, N Transformer blocks, and the LM head + softmax. The output is a probability distribution over the vocabulary, from which the next token is sampled. Blue = input, amber = computed/transformed, purple = processing, green = output.

The diagram above gives you the bird's-eye view. Now let us zoom in and walk through each stage with a concrete example, so you can see exactly what happens to a piece of text as it moves through the pipeline.

The Seven Stages, Step by Step

Let's trace the pipeline with a concrete example. The user types "The cat sat" and the model needs to predict what comes next.

Step 1 **Raw Text → Tokenizer**

Input: the string "The cat sat". The tokenizer splits this into tokens. Different tokenizers produce different splits. GPT-2's tokenizer produces three tokens: "The", " cat", " sat". Notice the leading spaces; BPE treats spaces as part of the token. Each token is mapped to its integer ID in the vocabulary: [464, 3797, 6748]. We now have a tensor (a

multi-dimensional array) of integers with shape [3].

Token IDs are integers, labels with no built-in meaning. Step 2 hands them to the embedding layer, which swaps each one for a dense vector that actually encodes something about the token.

Step 2 **Token IDs → Embedding Layer**

Each token ID is used to look up a row in the embedding matrix E. This matrix has shape [V, d_model], where V is the vocabulary size (50,257 for GPT-2) and d_model is the embedding dimension (768 for GPT-2). Token 464 retrieves row 464 of E: a vector of 768 numbers. Token 3797 retrieves row 3797. Token 6748 retrieves row 6748. The result is a matrix X0 of shape [3, 768], three vectors, one per token. There is no math here; it is a table lookup.

Those embedding vectors tell the model what each token means but nothing about order. Step 3 fixes that by stamping a position signature onto each vector.

Step 3 **Embedding → Positional Encoding**

The embedding vectors tell the model what each token means, but not where it appears in the sequence. "Cat sat the" and "The cat sat" would produce the same set of embedding vectors. Positional encoding fixes this by adding a unique pattern to each position. After this step, the vector for "The" at position 0 is different from the vector for "The" at position 5. We cover positional encoding in full detail in Chapter 4 (Positional Encoding & Attention Projections), Section 4.1.

The vectors now carry both meaning and position. Step 4 is where the real work happens: they pass through the stack of Transformer blocks that mix information across tokens and extract the patterns the model has learned.

Step 4 **N Transformer Blocks**

The position-aware vectors now pass through N Transformer blocks stacked on top of each other. Each block has two sub-layers: (1) **Multi-Head Self-Attention**, which lets each token look at every previous token and compute a weighted combination of their information, and (2) a **Feed-Forward Network (FFN)**, which processes each token's vector independently through two linear transformations with a non-linearity in between. Each sub-layer is wrapped in a **residual connection** (add the input back to the output) and **layer normalization**. GPT-2 has 12 blocks, GPT-3 has 96, and LLaMA 70B has 80, so more blocks means deeper understanding at the cost of more computation.

After all those blocks, every token has its own rich vector, but we only need one prediction. Step 5 plucks out the last token's vector and projects it onto the entire vocabulary.