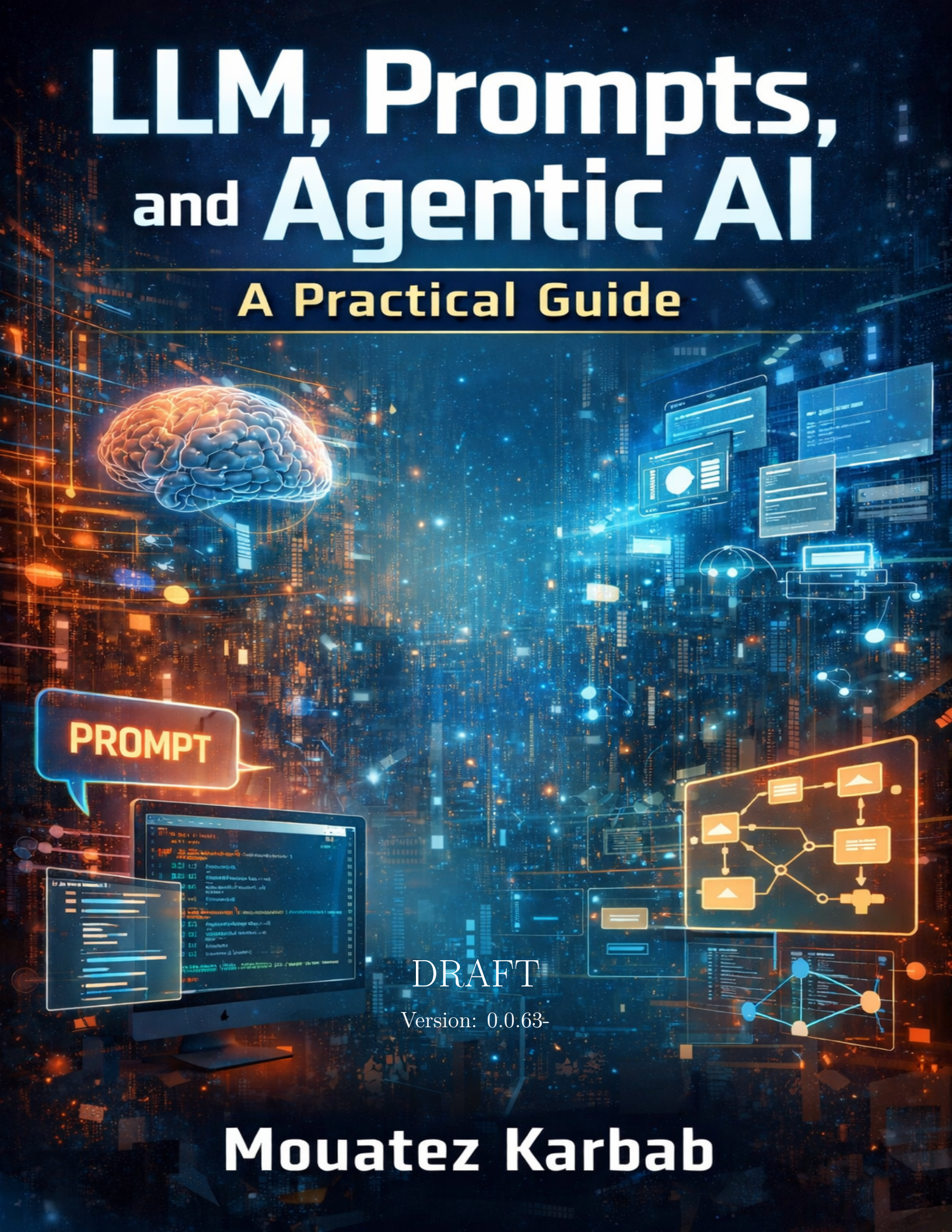
LLM, Prompts, and Agentic AI

A Practical Guide



DRAFT

Version: 0.0.63-

Mouatez Karbab

LLM, Prompts, and Agentic AI
A Practical Guide

ElMouatez Karbab

January 11, 2026
Version: 0.0.63-

Contents

1 Introduction to LLMs and Their Ecosystem **2**

1.1 From Text to Numbers: Bag of Words vs. Word Embeddings . . . 3
1.2 Vocabulary Construction and Tokenization 3
1.3 What is an LLM? 4
1.3.1 Conceptual Evolution 4
1.3.2 The Age of Neural Networks 5
1.3.3 Real-World Relevance 6
1.4 Core Architecture: The Transformer 6
1.4.1 The Transformer Model Explained 6
1.4.2 Self-Attention Mechanisms 7
1.4.3 Key Foundational Concepts 7
1.5 How They Learn 9
1.5.1 Phase 1: Pre-training 9
1.5.2 Phase 2: Fine-Tuning and Alignment 10
1.6 The Modern Landscape 10
1.6.1 Proprietary/Closed Models 11
1.6.2 Open-Source/Open-Weight Models 11
1.6.3 Comparative Table 12
1.7 Practical Choices 12
1.7.1 Using APIs (e.g., OpenAI API, Google AI Platform) . . . 13
1.7.2 Running Open-Source Models 13
1.7.3 Decision Framework 14
1.8 Core Capabilities and Inherent Limitations 14
1.8.1 What LLMs Excel At 14
1.8.2 Inherent Limitations & Ethical Considerations 15

Chapter 1

Introduction to Large Language Models and Their Ecosystem

Introduction

Welcome to the foundational chapter of our journey into Large Language Models (LLMs), prompt engineering, and agentic AI. This chapter serves as the starting point, establishing the essential concepts that underpin the entire field. We will trace the evolution of language models, from simple statistical methods to the sophisticated Transformer architecture that powers today's most advanced systems. Understanding this history is crucial, as it reveals the solutions developed to overcome fundamental challenges in natural language processing.

This chapter has several key learning objectives. By its conclusion, you will be able to:

- Understand the historical evolution and fundamental architecture of LLMs.
- Grasp core concepts like tokens, attention, and parameters.
- Differentiate between major LLM types and access models (API vs. Open-Source).
- Recognize the primary capabilities and inherent limitations of modern LLMs.
- Apply these concepts through practical code examples, such as querying a model via an API.

We will begin by defining what an LLM is, then dissect the Transformer architecture, explore the training process, survey the current landscape of models, and finally, discuss the practical trade-offs between using APIs and hosting

open-source models. This knowledge will provide a solid platform for the more advanced topics covered in the rest of this book.

1.1 From Text to Numbers: Bag of Words vs. Word Embeddings

Deep Learning models, including Large Language Models (LLMs), operate fundamentally on mathematical operations. Therefore, raw text must be converted into numerical representations before processing. This translation involves two critical steps: *tokenization* (converting text to atomic units) and *vectorization* (mapping those units to numerical vectors). This section compares two distinct approaches to this problem: the statistical Bag of Words (BoW) model and the semantic Word Embedding model.

1.2 Vocabulary Construction and Tokenization

Regardless of the vectorization method, the first step is building a dictionary (vocabulary). Given a corpus C, we extract a set of unique tokens $V = \{w_1, w_2, \ldots, w_{|V|}\}$. Each unique token w_i is assigned a unique integer index i.

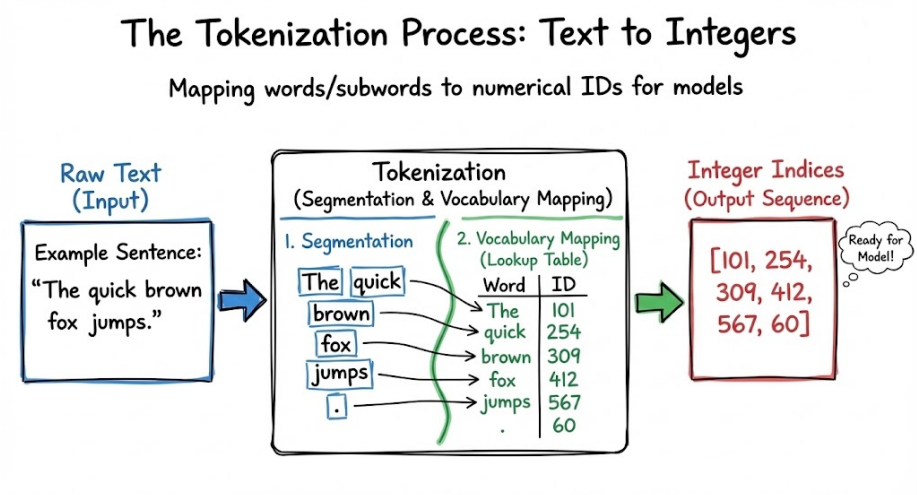


Figure 1.1: The Tokenization Process: Mapping text to integer indices.

1.3 What is an LLM?: From Statistical Models to Transformers

A *Large Language Model (LLM)* is a type of artificial intelligence model specifically designed to understand, generate, and process human language. These models are "large" because they contain billions of parameters and are trained on massive datasets of text and code. Their journey from simple predictive text to complex reasoning systems is a story of innovation in computer science and mathematics.

1.3.1 Conceptual Evolution

The path to modern LLMs began with much simpler statistical techniques. Early models used methods like *n-grams*, which predict the next word in a sequence based on the previous 'n' words. For example, a *trigram* (n=3) model might predict the word "day" after seeing the sequence "a beautiful". While effective for simple tasks, *n-gram* models faced significant limitations. They struggled to capture long-range dependencies—the connection between words far apart in a text—and could not grasp the nuanced context of language. This led researchers to explore more powerful methods.

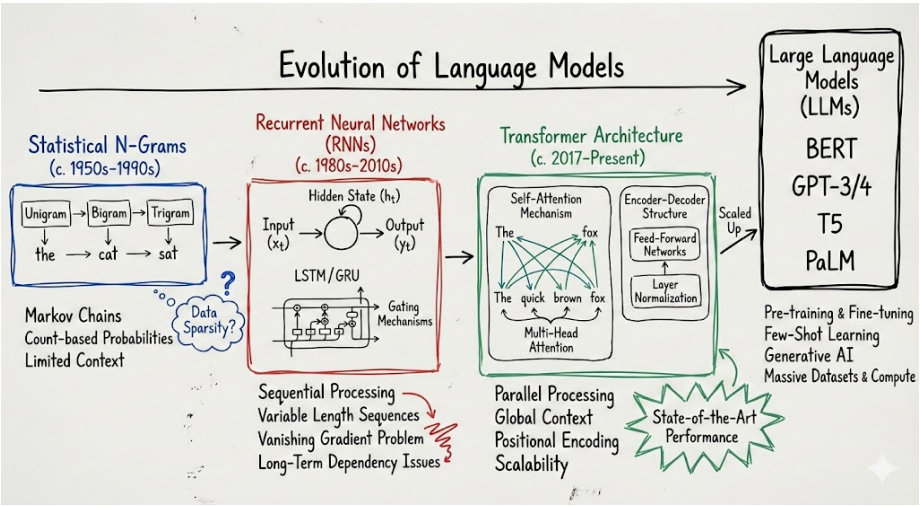


Figure 1.2: A timeline illustrating the evolution of language models, from statistical n-grams to Recurrent Neural Networks (RNNs) and culminating in the Transformer architecture.

1.3.2 The Age of Neural Networks

Neural networks introduced the ability to learn complex, non-linear relationships in data, offering a significant leap forward in modeling language.

Recurrent Neural Networks (RNNs) A *Recurrent Neural Network (RNN)* is a type of neural network designed specifically for sequential data, like text. It processes a sequence step-by-step, maintaining an internal *memory* or state that captures information from previous steps. However, standard RNNs suffered from the *vanishing gradient problem*, where the influence of earlier inputs would fade over time, making it difficult to learn long-range dependencies. To solve this, more advanced architectures were developed.

- **Long Short-Term Memory (LSTM):** LSTMs are a special kind of RNN that introduce a "cell state" and "gates" (input, forget, and output) to regulate the flow of information. These gates allow the network to selectively remember or forget information over long sequences, effectively mitigating the vanishing gradient problem.
- **Gated Recurrent Units (GRU):** GRUs are a simplified version of LSTMs that combine the forget and input gates into a single "update gate." They are often computationally more efficient than LSTMs while delivering similar performance on many tasks.

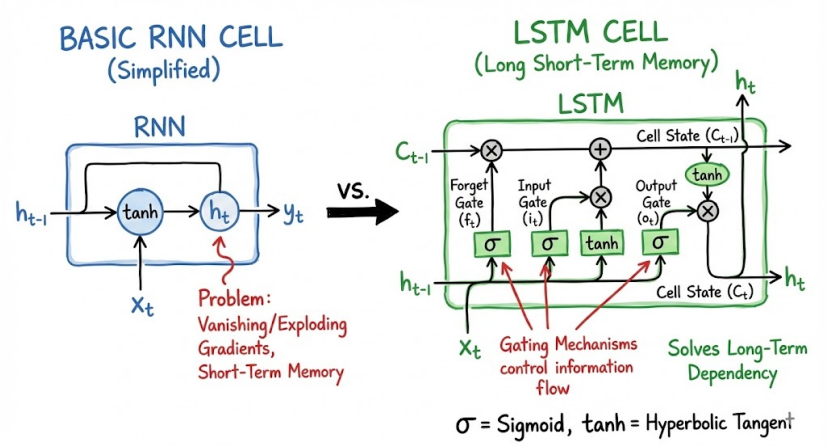


Figure 1.3: A simplified comparison of a basic RNN cell versus an LSTM cell, highlighting the gating mechanisms in the LSTM that control information flow.

The Transformer Revolution While RNNs, LSTMs, and GRUs represented a major advance, they had a fundamental bottleneck: their sequential nature. They had to process text token by token, which made it difficult to

parallelize training on modern hardware like GPUs. In 2017, a paper from Google titled "Attention is All You Need" introduced the *Transformer* architecture. The Transformer model dispensed with recurrence entirely and instead relied on a mechanism called *self-attention*. This allowed the model to process all tokens in a sequence simultaneously, enabling massive parallelization and training on much larger datasets than ever before. This was a pivotal moment that paved the way for modern LLMs.

1.3.3 Real-World Relevance

This architectural evolution directly enables many applications we use daily. For instance, real-time translation apps depend on the efficiency and contextual understanding of Transformer-based models. Developers can easily access these capabilities using libraries like Hugging Face *Transformers*, which provides pre-trained models and tools for a wide range of natural language processing tasks. Listing 1 shows a simple example of using this library for a text generation task.

```python
from transformers import pipeline

# Initialize a text-generation pipeline with a small model
generator = pipeline('text-generation', model='gpt2')

# Generate text based on a prompt
prompt = "The future of artificial intelligence is"
result = generator(prompt, max_length=50, num_return_sequences=1)

print(result[0]['generated_text'])

# Expected Output (will vary due to model's probabilistic nature):
# The future of artificial intelligence is not as simple as we think.
# It's not as simple as the future of the human race. It's a question of how we use it.
```

Listing 1: A basic query using the Hugging Face Transformers library.

1.4 Core Architecture: The Transformer

The Transformer is the foundational architecture for nearly all modern LLMs. Its design allows for unprecedented scale and performance in language tasks by processing entire sequences of data in parallel.

1.4.1 The Transformer Model Explained

At a high level, the original Transformer model consists of two main parts: an *encoder* stack and a *decoder* stack. The encoder processes the input sequence (e.g., a sentence in German) and creates a rich numerical representation. The decoder then takes this representation and generates the output sequence (e.g., the translated sentence in English).

However, many modern LLMs are variants of this design.

- **Decoder-Only Models:** Models like the GPT series are "decoder-only." They are optimized for generative tasks, where the goal is to predict the next token in a sequence based on the preceding ones.
- **Encoder-Decoder Models:** Models like T5 (Text-to-Text Transfer Transformer) retain the full encoder-decoder structure, making them well-suited for sequence-to-sequence tasks like translation and summarization.

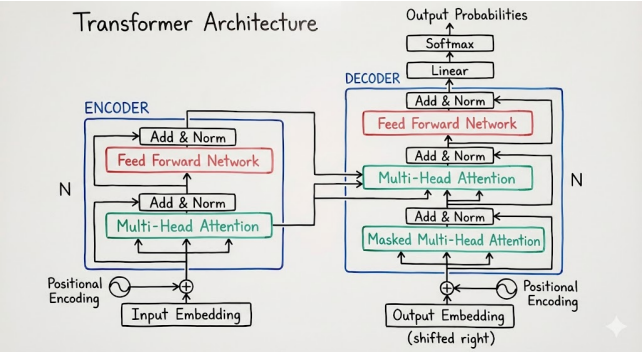


Figure 1.4: A high-level diagram of the Transformer's encoder-decoder architecture.

1.4.2 Self-Attention Mechanisms

The core innovation of the Transformer is the *self-attention* mechanism.

Core Idea You can think of self-attention as a dynamic weighting system. For each token in the input sequence, self-attention calculates an "attention score" with every other token in the sequence. A high score means two tokens are highly relevant to each other. This allows the model to focus on the most important parts of the input when processing a particular word.

Queries, Keys, and Values To compute these scores, the model projects the embedding of each token into three different vectors: a *Query* (Q), a *Key* (K), and a *Value* (V).

- **Query:** Represents the current token that is "looking" for context.
- **Key:** Represents a token that is being compared against the query. The compatibility between a query and a key determines the attention score.
- **Value:** Represents the information that a token carries. The final output for a token is a weighted sum of all value vectors, where the weights are the attention scores.

This process is often done in parallel multiple times through *multi-head attention*, allowing the model to capture different types of relationships (e.g., grammatical, semantic) simultaneously.

The computation follows a few key steps, as outlined in Algorithm 1.

Algorithm 1: Self-Attention Computation Steps

1: For each input token, create a Query (Q), Key (K), and Value (V) vector.
2: Compute the attention scores by taking the dot product of the Query vector of the current token with the Key vectors of all other tokens in the sequence.
3: Normalize the scores by scaling them (dividing by the square root of the dimension of the key vectors).
4: Apply a *softmax* function to the scores to convert them into probabilities. These probabilities sum to 1 and represent the attention weights.
5: Compute the final output for the token by taking a weighted sum of all the Value vectors, using the attention weights.

1.4.3 Key Foundational Concepts

Several other concepts are fundamental to understanding how LLMs work.

Tokens LLMs do not see text as words but as *tokens*. *Tokenization* is the process of breaking down a piece of text into smaller units. Common methods include *Byte-Pair Encoding (BPE)*, used by GPT models, which merges frequent pairs of bytes to create new tokens. This allows the model to handle any word, even those it has not seen before. For example, the text "Hello world" might be tokenized into '["Hel", "lo", " world"]'. We can use libraries like 'tiktoken' to see how text is tokenized, as shown in Listing 2.

```python
import tiktoken

# Load an encoding for a specific model (e.g., gpt-4)
enc = tiktoken.get_encoding("cl100k_base")

# Encode a string into token integers
tokens = enc.encode("Hello world, let's learn about tokenization!")
print(f"Tokens: {tokens}")
print(f"Number of tokens: {len(tokens)}")

# Decode the tokens back into a string
text = enc.decode(tokens)
print(f"Decoded text: {text}")

# Expected Output:
# Tokens: [9906, 1917, 11, 852, 347, 1867, 439, 21957, 32, 0]
# Number of tokens: 10
# Decoded text: Hello world, let's learn about tokenization!
```

Listing 2: An example of tokenizing text using the 'tiktoken' library.

Context Window The *context window* is the maximum number of tokens the model can process at one time. This includes both the input prompt and the generated output. Early models had small context windows (e.g., 2,048 tokens), but modern models have expanded significantly (e.g., 128k for GPT-4 Turbo, and up to 1 million for Claude 3.5 Sonnet). A larger context window is crucial for tasks like summarizing long documents or maintaining a coherent conversation.

Parameters The *parameters* of an LLM are the weights and biases of the neural network, learned during training. The number of parameters is a rough measure of the model's size and potential complexity. Models range from smaller ones with 7 billion parameters (e.g., Llama 3 8B) to frontier models with rumored trillions. Training involves adjusting these parameters using an algorithm called *backpropagation*. During *inference* (when the model is used), these parameters determine the output. To make large models more efficient, techniques like *quantization* are used, which reduces the precision of the parameters (e.g., from 32-bit to 4-bit numbers) to lower memory and computational costs.

1.5 How They Learn: The Two-Phase Training Process

Training an LLM is a complex, two-phase process involving pre-training on a massive, general dataset, followed by fine-tuning and alignment to make the model useful and safe for specific tasks.

1.5.1 Phase 1: Pre-training

Pre-training is where the model learns general knowledge about language, grammar, reasoning, and the world.

Objective The primary objective is typically unsupervised learning. Two common approaches are:

- **Masked Language Modeling (MLM):** Used in models like BERT, this involves masking (hiding) some tokens in the input text and training the model to predict them.
- **Next-Token Prediction:** Used in models like GPT, this involves training the model to predict the next token in a sequence given all the previous tokens. This is also called *causal language modeling*.

Process The process starts with a massive corpus of text, such as the Common Crawl dataset, which contains a large portion of the public internet. This data is cleaned, tokenized, and fed into the model in batches. The model makes predictions, compares them to the actual text, calculates an error (loss), and adjusts its parameters to reduce this error over many iterations.

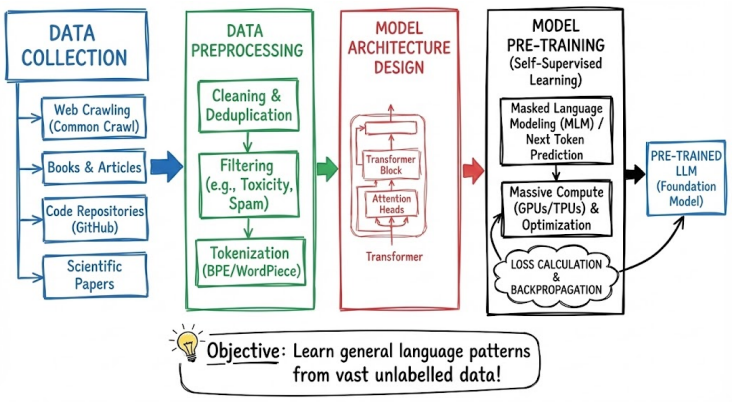


Figure 1.5: A flowchart outlining the main steps of the pre-training process, from data collection to model training.

Scale and Cost Pre-training is incredibly resource-intensive. It requires thousands of high-end GPUs running for weeks or months, consuming vast amounts of electricity. The computational cost is often measured in *FLOPs* (Floating Point Operations Per Second), with large models requiring trillions of FLOPs to train. This scale necessitates *distributed training*, where the training process is split across multiple machines.

1.5.2 Phase 2: Fine-Tuning and Alignment

After pre-training, the model is a powerful but general language predictor. Fine-tuning and alignment are necessary to steer its behavior to be helpful, honest, and harmless.

Instruction Tuning This step teaches the model to follow instructions. The model is fine-tuned on a smaller, high-quality dataset of prompt-response pairs. Datasets like Alpaca, which was generated using an existing LLM to create instruction-following examples, are commonly used for this purpose.

Alignment (RLHF/DPO) Alignment ensures the model's behavior aligns with human values.

- **Reinforcement Learning from Human Feedback (RLHF):** This is a three-step process. First, humans rank different model responses to the same prompt. Second, a "reward model" is trained on this preference data to predict which responses humans would prefer. Third, the LLM is fine-tuned using reinforcement learning (often with an algorithm called PPO) to maximize the score from the reward model.

- **Direct Preference Optimization (DPO):** DPO is a more recent and stable alternative to RLHF. It achieves a similar goal by directly optimizing the LLM on the preference data, bypassing the need to train a separate reward model.

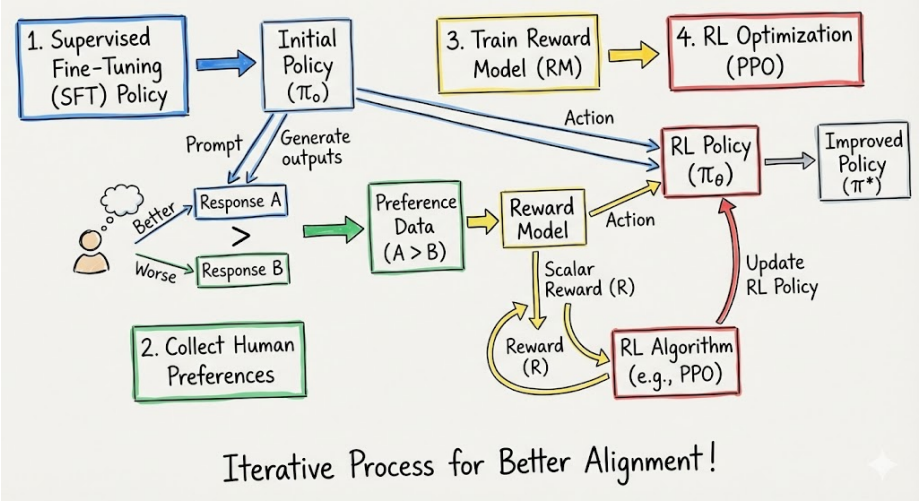


Figure 1.6: A diagram illustrating the workflow of Reinforcement Learning from Human Feedback (RLHF), from human preference collection to policy optimization.

Emerging techniques like constitutional AI, where the model is guided by a set of principles or a "constitution," are also being explored to enhance safety and alignment.

1.6 The Modern Landscape: A Tour of Major Models

The LLM landscape in late 2025 is dynamic, with rapid innovation from both large tech companies and the open-source community. Models can be broadly categorized as proprietary or open-source.

1.6.1 Proprietary/Closed Models

These models are typically accessed via APIs and are developed and controlled by a single company.

- **OpenAI GPT Series:** The release of **GPT-5** has shifted the paradigm toward "System 2" deep reasoning and autonomous agentic workflows. It features a unified multimodal architecture that handles text, audio, and

vision inputs and outputs simultaneously without relying on separate submodels.

- **Google Gemini Family: Gemini 3 Pro** continues to push the boundaries of context, now handling over 2 million tokens. It introduces "Deep Think" capabilities for complex problem-solving and maintains its status as the leader in native multimodal research, effectively processing entire codebases or long-form video archives.
- **Anthropic's Claude Series:** The **Claude 4** family, particularly the Opus variant, is renowned for its human-like nuance and reduced refusal rates. It remains a top choice for complex coding tasks and creative writing, with significant improvements in following complex, multi-step instructions.

1.6.2 Open-Source/Open-Weight Models

These models have their weights publicly available, allowing anyone to download, modify, and run them on their own hardware.

- **Meta's Llama Series: Llama 4** has cemented itself as the standard for open-weight AI, offering context windows ranging from 1M to 10M tokens in its largest variants. With distinct "Scout" and massive-parameter versions, it provides a robust alternative to closed models for enterprise-grade applications.
- **Mistral AI Models:** Mistral continues to prioritize efficiency with **Mistral Large 3**. Utilizing advanced sparse Mixture of Experts (MoE) architectures, it delivers high-performance inference at a fraction of the compute cost, with notably strong performance in multilingual European contexts.
- **Other Notable Models:** The open ecosystem is thriving with specialized tools. **DeepSeek-V3.1** (and the R1 reasoning series) has become a favorite for cost-effective coding and logic tasks. **Qwen 3 Max** from Alibaba dominates in Asian language support and dual-mode (fast vs. thinking) operation, while **Grok-3** offers unique real-time integration with X (formerly Twitter) data.

1.6.3 Comparative Table

Table **??** provides a high-level comparison of some of the major models available today. Note that parameters for proprietary models are often not disclosed and are estimated.

Table 1.1: Comparison of Major LLMs (Late 2025)

Model	Developer	Context Window	Key Strengths
GPT-5	OpenAI	400k	Unified multimodal reasoning, autonomous agen
Gemini 3 Pro	Google	2M+	Deep thinking mode, native multimodal research
Claude 4 Opus	Anthropic	200k	Complex coding, nuance, reduced refusal rates
Llama 4 (Maverick)	Meta	1M – 10M	Top open-weight model, massive context handlir
DeepSeek-V3.1	DeepSeek	128k	Efficient MoE, hybrid "thinking" inference
Mistral Large 3	Mistral AI	128k	Strong multilingualism, sparse MoE efficiency
Qwen 3 Max	Alibaba	1M	Exceptional Asian language support, dual-mode

1.7 Practical Choices: APIs vs. Open-Source Models

Choosing between a proprietary API and a self-hosted open-source model is a critical decision that depends on factors like cost, privacy, customization, and scalability.

1.7.1 Using APIs (e.g., OpenAI API, Google AI Platform)

Using an API involves sending requests to a model hosted by a provider and receiving the output.

Pros

- **Ease of Use:** Minimal setup is required—just sign up for an API key.
- **Scalability:** Providers handle all the infrastructure, which scales automatically with your usage.
- **Access to Frontier Models:** You get immediate access to the latest and most powerful models.
- **Integrated Safety:** Providers typically include built-in safety filters.

Cons

- **Cost:** Pay-per-use pricing can become expensive at scale.
- **Vendor Lock-in:** Your application becomes dependent on a single provider.
- **Data Privacy:** You must send your data to a third-party service.
- **Latency and Downtime:** Performance is subject to network conditions and provider availability.

Listing 3 shows a simple API call using the OpenAI Python library.

```python
import os
from openai import OpenAI

# It's best practice to set the API key as an environment variable
# client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))
# For this example, we'll use a placeholder key.
client = OpenAI(api_key="YOUR_API_KEY_HERE")


try:
    completion = client.chat.completions.create(
      model="gpt-4o",
      messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "What is the capital of France?"}
      ]
    )
    print(completion.choices[0].message.content)
except Exception as e:
    print(f"An error occurred: {e}")

# Expected Output:
# The capital of France is Paris.
```

Listing 3: An example of a basic API call to OpenAI's GPT-4o model.

1.7.2 Running Open-Source Models

This approach involves downloading a model's weights and running it on your own infrastructure.

Pros

- **Control and Privacy:** Data never leaves your servers.
- **Customization:** You can fine-tune the model on your own data using techniques like *LoRA (Low-Rank Adaptation)*.
- **No Per-Use Cost:** After the initial hardware investment, there are no ongoing API fees.
- **Offline Capability:** The model can run in environments without internet access.

Cons

- **Infrastructure Cost:** Requires powerful GPUs (e.g., NVIDIA A100s or H100s), which are expensive.
- **Complexity:** Requires significant technical expertise to set up, maintain, and optimize for inference.
- **Maintenance Overhead:** You are responsible for updates, security, and scaling.

Tools like the Hugging Face Hub and inference servers like vLLM simplify the process of deploying open-source models. Listing 4 shows a basic example.

```python
from transformers import AutoTokenizer, AutoModelForCausalLM

# Specify the model and tokenizer
model_name = "gpt2" # Using a small model for demonstration
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Prepare the input
prompt = "Running an open-source model locally gives you"
inputs = tokenizer(prompt, return_tensors="pt")

# Generate output
outputs = model.generate(**inputs, max_length=50)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))

# Expected Output (will vary):
# Running an open-source model locally gives you the ability to run
# your own code on your own hardware. This means you can run your own
# code on your own hardware. This means you can run your own code on
# your own hardware.
```

Listing 4: Running a small open-source model locally using Hugging Face Transformers.

1.7.3 Decision Framework

The choice between APIs and open-source models often comes down to a trade-off between convenience and control. A hybrid approach is also common: using an API for rapid prototyping and then migrating to a fine-tuned open-source model for production to optimize cost and privacy.

1.8 Core Capabilities and Inherent Limitations

While LLMs are incredibly powerful, it is essential to understand both their strengths and their weaknesses to use them effectively and responsibly.

1.8.1 What LLMs Excel At

LLMs have demonstrated remarkable performance across a wide range of language tasks.

- **Text Generation:** This is their most fundamental capability, including writing essays, code, marketing copy, and creative stories. Tools like Gradio can be used to quickly create a web interface to demonstrate this, as shown in Listing 5.
- **Information Synthesis:** LLMs can read and summarize long documents, answer questions based on a provided context, and extract key information from unstructured text.
- **Classification and Extraction:** They can perform tasks like sentiment analysis, topic classification, and named entity recognition, often with little to no task-specific training (*zero-shot* or *few-shot* learning).

- **Translation:** While dedicated translation models exist, general-purpose LLMs have become very effective at translating between languages, including low-resource ones.

```python
import gradio as gr
from transformers import pipeline

# Load a pre-trained model via the Hugging Face pipeline
generator = pipeline('text-generation', model='gpt2')

def generate_text(prompt):
    """A function to generate text based on a prompt."""
    result = generator(prompt, max_length=50, num_return_sequences=1)
    return result[0]['generated_text']

# Create a Gradio interface
iface = gr.Interface(
    fn=generate_text,
    inputs=gr.Textbox(lines=2, placeholder="Enter your prompt here..."),
    outputs="text",
    title="Simple Text Generation Demo",
    description="This is a basic demo of a GPT-2 model using Gradio."
)

# Launch the interface
# iface.launch() # This line would start a local web server.
```

Listing 5: A simple Gradio demo for a text generation application.

1.8.2 Inherent Limitations & Ethical Considerations

Despite their capabilities, LLMs have significant limitations that users must be aware of. We will explore mitigation strategies in detail in Chapter 9, but here is an overview.

- **Hallucinations:** LLMs can generate text that is plausible but factually incorrect or nonsensical. This is often called *hallucination*. Mitigation strategies include using verification prompts or cross-referencing generated information with reliable sources.
- **Knowledge Cutoff:** A model's knowledge is frozen at the time its training data was collected. It is not aware of events that occurred after its training date. This can be addressed using *Retrieval-Augmented Generation (RAG)*, a technique we will cover in Chapter 5, which allows the model to access external, up-to-date information.
- **Bias:** LLMs are trained on vast amounts of text from the internet, which contains human biases. As a result, models can perpetuate or even amplify harmful stereotypes. Detecting and mitigating bias is an active and critical area of research.
- **Opacity:** The decision-making process of large neural networks is often described as a "black box," making it difficult to understand why a model

produced a specific output. Techniques like attention visualization, which creates a heatmap showing which input words the model focused on, can provide some insight.

Conclusion

This chapter provided a comprehensive introduction to the world of Large Language Models. We journeyed from the statistical origins of language modeling to the revolutionary Transformer architecture that defines the modern era. We dissected the core components of this architecture, including the crucial self-attention mechanism, and demystified foundational concepts like tokens, context windows, and parameters.

We also explored the two-phase training process—pre-training and fine-tuning—that endows these models with their remarkable capabilities. A survey of the current landscape revealed the key players and models, highlighting the practical choice between using proprietary APIs and self-hosting open-source alternatives. Finally, we balanced the discussion by outlining both the powerful capabilities and the significant limitations of LLMs, such as hallucinations and bias.

The concepts introduced here—from tokenization to the trade-offs of model deployment—form the bedrock upon which all advanced applications, prompt engineering techniques, and agentic systems are built. With this foundation, you are now prepared to move forward and explore how to effectively harness the power of these transformative tools.

Acronyms and Terminology