**Chapter 2: Understanding LLMs**

**1. What Are LLMs?**

* **Definition**: LLMs (Large Language Models) are services that take a string of text as input (called a **prompt**) and return another string of text as output (called a **completion** or **response**).
* **Training**: LLMs are initially untrained and produce random, nonsensical outputs. They need to be **trained** on large datasets (e.g., books, articles, code) to learn how to generate coherent and contextually relevant text. Training involves teaching the model to mimic the patterns found in the training data.
* **Fine-Tuning**: Instead of training from scratch, many LLMs are **fine-tuned** on specific datasets. For example, OpenAI's Codex was fine-tuned on GitHub code to specialize in generating source code.

**2. How LLMs See the World**

* **Tokenization**: LLMs don’t process text as characters or words but as **tokens**, which are chunks of text typically 3-4 characters long. The model uses a **tokenizer** to break down the input text into tokens, which are then processed by the LLM.
* **Differences from Human Processing**:
  + **Deterministic Tokenizers**: Unlike humans, who can interpret typos or garbled text, LLMs use deterministic tokenizers that make typos stand out.
  + **No Letter-Level Processing**: LLMs can’t slow down to examine individual letters or reassemble tokens easily, making tasks like reversing letters in words difficult.
  + **Different Text Perception**: LLMs see text as sequences of tokens, not as words or letters, which affects how they handle tasks like capitalization or ASCII art.

**3. One Token at a Time**

* **Auto-Regressive Models**: LLMs generate text **one token at a time**. They predict the next token based on the previous tokens, and this process repeats until the completion is generated. This is called **autoregressive** generation.
* **Patterns and Repetitions**: LLMs can fall into repetitive patterns because they are good at recognizing and continuing patterns. This can lead to long, repetitive outputs if not controlled.

**4. Temperature and Probabilities**

* **Sampling**: LLMs compute the probability of all possible next tokens and then **sample** one based on these probabilities. The **temperature** parameter controls how "creative" or random the sampling process is.
  + **Temperature = 0**: The model always chooses the most likely token, leading to deterministic and predictable outputs.
  + **Temperature > 0**: The model introduces randomness, allowing for more varied and creative outputs. Higher temperatures can lead to more errors and less coherent text.
* **Beam Search**: An alternative to temperature-based sampling, beam search looks ahead to find the most likely sequence of tokens, but it is computationally expensive.

**5. The Transformer Architecture**

* **Minibrains**: The core of an LLM is the **transformer architecture**, which consists of thousands of identical **minibrains** (neural network layers). Each minibrain processes a token and shares information with others through an **attention mechanism**.
* **Attention Mechanism**: Minibrains communicate by asking and answering questions about the text. This allows the model to focus on relevant parts of the text when generating the next token.
* **Parallelism**: The transformer architecture allows for parallel processing of tokens, making it efficient for training and inference. However, generating text is slower than reading it because the model must wait for each token to be processed before generating the next one.
* **Unidirectional Processing**: LLMs process text from left to right, meaning they can only look at previous tokens, not future ones. This limits their ability to "look ahead" or revise previous outputs.

**6. Hallucinations**

* **Definition**: LLMs can produce **hallucinations**, which are factually incorrect but plausible-sounding completions. This happens because LLMs are trained to mimic patterns in text, not to verify facts.
* **Truth Bias**: LLMs tend to assume that the prompt is true, which can lead to hallucinations if the prompt contains false or hypothetical information.
* **Mitigation**: To reduce hallucinations, prompts can ask the model to provide reasoning, calculations, or sources that can be independently verified.

**7. Human Thought vs. LLM Processing**

* **Differences**: Humans produce text as part of a broader process that may involve research, editing, and reasoning. LLMs, on the other hand, generate text in a single pass without the ability to pause, edit, or verify information.
* **Guessing vs. Knowing**: LLMs always guess the next token based on patterns in their training data. They don’t "know" facts in the way humans do, and they can’t express doubt or uncertainty.

**8. Prompt Engineering Implications**

* **Order Matters**: The order of information in a prompt is critical because LLMs process text in a single pass from beginning to end. If important information appears too late in the prompt, the model may miss it.
* **Avoiding Repetition**: To prevent repetitive outputs, prompt engineers can use techniques like filtering or adjusting the temperature to introduce variability.
* **Task Suitability**: When designing prompts, consider whether the task is something an LLM can handle in a single pass without backtracking or editing. Tasks that require multiple steps or revisions may not be well-suited for LLMs.

**9. Conclusion**

* **Key Takeaways**:
  1. LLMs are **document completion engines** that mimic the patterns in their training data.
  2. They generate text **one token at a time**, with no ability to edit or revise previous tokens.
  3. LLMs process text in a **single pass**, from beginning to end, which affects how they handle complex tasks.
  4. Understanding these limitations is crucial for effective **prompt engineering**.

**Summary:**

This chapter provides a foundational understanding of how LLMs work, from tokenization and autoregressive generation to the transformer architecture and attention mechanisms. It also highlights the limitations of LLMs, such as hallucinations and their inability to edit or verify information, which are important considerations when designing prompts. The chapter sets the stage for more advanced topics in prompt engineering, which are covered in later chapters.

Chapter 3: Moving to Chat

**1. From Document Completion to Chat**

* **Base Models**: LLMs like GPT-3 are initially trained to complete documents. They can generate plausible continuations for any given text, but they lack the ability to act as conversational assistants.
* **Limitations of Base Models**:
  + **Unsafe Content**: Base models can generate harmful or inappropriate content if prompted with dangerous or unethical queries (e.g., instructions for illegal activities).
  + **Lack of Assistance**: When asked a question, base models may generate a list of related questions instead of providing a direct answer, making them less useful for conversational applications.
* **Need for Fine-Tuning**: To make LLMs more suitable for real-world applications, they need to be fine-tuned to act as **helpful, honest, and harmless (HHH)** assistants.

**2. Reinforcement Learning from Human Feedback (RLHF)**

* **RLHF Overview**: RLHF is a training technique that uses human feedback to fine-tune LLMs, making them more aligned with user expectations. It involves multiple steps and models:
  1. **Base Model**: The starting point, trained on a large corpus of internet data.
  2. **Supervised Fine-Tuning (SFT) Model**: Fine-tuned on a smaller dataset of human-generated examples to make it better at following instructions.
  3. **Reward Model**: Trained on human-ranked completions to judge the quality of responses.
  4. **RLHF Model**: Fine-tuned using the reward model to optimize for helpful, honest, and harmless behavior.
* **Key Concepts**:
  1. **HHH Alignment**: The goal of RLHF is to make models **Helpful** (follow user instructions), **Honest** (avoid hallucinations), and **Harmless** (avoid harmful content).
  2. **Alignment Tax**: Fine-tuning for HHH alignment can sometimes reduce the model's general intelligence or performance on certain tasks. This is mitigated by mixing in some of the original training data.

**3. Instruct Models**

* **Instruct Models**: These are LLMs fine-tuned to follow user instructions rather than just complete documents. They are trained on examples of prompts and ideal completions, such as answering questions, summarizing text, or generating code.
* **Limitations**:
  + **Ambiguity**: Without clear indicators, the model may not know whether to complete a document or answer a question.
  + **Alignment Tax**: Instruct models may lose some of their general capabilities due to fine-tuning for specific tasks.

**4. Chat Models**

* **ChatML**: OpenAI introduced **ChatML**, a markup language that structures conversations between a user and an assistant. It uses tags like <|im\_start|> and <|im\_end|> to define roles (system, user, assistant) and organize the conversation.
* **Benefits of ChatML**:
  + **Unambiguous Structure**: ChatML makes it clear when the model should respond as an assistant, avoiding ambiguity.
  + **System Messages**: The system message sets the behavior and tone of the assistant (e.g., "You are a helpful assistant.").
  + **Preventing Prompt Injection**: ChatML prevents users from injecting malicious prompts by restricting the use of special tokens.

**5. The Changing API**

* **Chat Completion API**: OpenAI's chat API allows developers to interact with LLMs using a structured format. The API converts user inputs into ChatML behind the scenes, making it easier to build conversational applications.
* **Key Parameters**:
  + **Temperature**: Controls the creativity of the model's responses. Lower values (e.g., 0) produce more deterministic outputs, while higher values (e.g., 1) introduce more randomness.
  + **Max Tokens**: Limits the length of the output.
  + **Logprobs**: Returns the probability of each token, useful for understanding the model's confidence.
  + **Stop Sequences**: Stops the generation when a specific sequence of tokens is encountered.
* **Comparison with Completion API**:
  + **Alignment Tax**: Chat models may lose some general capabilities due to fine-tuning for conversational tasks.
  + **Control**: Completion APIs offer more control over the output, especially for tasks like code generation, where you want the model to return only the code without additional commentary.

**6. Moving Beyond Chat to Tools**

* **Tool Execution API**: OpenAI introduced an API that allows LLMs to call external tools (e.g., APIs) and incorporate the results into the conversation. This extends the capabilities of LLMs beyond text generation, enabling them to perform tasks like searching the web or executing code.
* **Core Idea**: Even with tool execution, LLMs are still fundamentally completing documents—just more complex ones that include tool calls and results.

**7. Prompt Engineering as Playwriting**

* **Metaphor**: Building a chat-based LLM application is like writing a play, where:
  + **Characters**: The user, assistant, system, and tools.
  + **Playwrights**: The prompt engineer, the user, the LLM, and external APIs.
  + **Script**: The prompt, which is a transcript of the conversation.
* **Role of the Prompt Engineer**: The prompt engineer designs the structure of the conversation, adds context, and ensures the model behaves as expected.

**8. Conclusion**

* **LLMs as Document Completion Engines**: Despite the advancements in chat and tool execution, LLMs are still fundamentally completing documents. In the case of chat models, the document is a conversation transcript.
* **Future of LLMs**: While chat models are becoming the norm, document completion models are still relevant for tasks like code generation. The challenge for prompt engineers is to effectively convey the user's problem and context within the limited space of a prompt or transcript.

**Key Takeaways:**

1. **RLHF** is a powerful technique for fine-tuning LLMs to make them more aligned with user expectations (helpful, honest, harmless).
2. **Chat Models** use **ChatML** to structure conversations, making them more predictable and safer.
3. **Temperature** and other API parameters allow developers to control the creativity and behavior of the model.
4. **Prompt Engineering** is like writing a play, where the prompt engineer designs the conversation and ensures the model behaves as expected.
5. **LLMs are still document completion engines**, even when they appear to be engaging in chat or executing tools.

Chapter 4: Designing LLM Applications

**1. The Anatomy of the Loop**

* **LLM Application as a Loop**: The LLM application is represented as a loop that transforms the user's problem into the model's domain (text) and then converts the model's completion back into a solution for the user.
* **User Domain**: The user's problem can vary widely, from simple tasks like proofreading to complex tasks like travel planning.
* **Model Domain**: The model's role is to complete documents, which can include emails, code, stories, or even chat transcripts.
* **Transformations**:
  + **User Problem → Model Domain**: The application converts the user's problem into a prompt that the model can complete.
  + **Model Completion → User Solution**: The model's output is transformed back into a solution for the user.

**2. The User’s Problem**

* **Dimensions of Complexity**:
  + **Medium**: How the problem is conveyed (e.g., text, voice).
  + **Level of Abstraction**: How abstract or concrete the problem is.
  + **Context Required**: How much additional information is needed to solve the problem.
  + **Statefulness**: Whether the problem requires memory of past interactions.
* **Examples**:
  + **Proofreading**: Low complexity in all dimensions.
  + **IT Support Assistance**: Medium complexity, requiring access to documentation and conversation history.
  + **Travel Planning**: High complexity, requiring access to calendars, APIs, and user preferences.

**3. Converting the User’s Problem to the Model Domain**

* **Prompt Engineering**: The key to converting the user's problem into the model's domain lies in crafting the right prompt. The prompt must:
  1. **Resemble Training Data**: The prompt should mimic documents from the model's training set (the **Little Red Riding Hood principle**).
  2. **Include Relevant Information**: The prompt must contain all the information needed to solve the user's problem.
  3. **Lead to a Solution**: The prompt should condition the model to generate a completion that addresses the problem.
  4. **Have a Natural Stopping Point**: The model should know when to stop generating text.
* **Examples**:
  1. **Completion Models**: The prompt must explicitly indicate when the model should provide a solution (e.g., using headings like "## Solution").
  2. **Chat Models**: The model is fine-tuned to stop after generating a helpful assistant message.

**4. Using the LLM to Complete the Prompt**

* **Model Selection**: The choice of model (e.g., GPT-3.5 vs. GPT-4) depends on factors like cost, latency, and quality of completions.
* **Fine-Tuning**: Fine-tuning can improve performance for specific tasks or languages, but it requires additional effort and resources.
* **Completion Models vs. Chat Models**:
  + **Completion Models**: Require more explicit instructions to generate solutions and stop at the right point.
  + **Chat Models**: Simplify the process by automatically generating helpful responses and stopping after the assistant's message.

**5. Transforming Back to User Domain**

* **Simple Applications**: For chat apps, the model's completion can be directly presented to the user.
* **Complex Applications**: The model's output may need to be transformed or parsed to extract useful information. For example:
  + **Function Calling**: The model can generate function calls (e.g., booking a flight), which the application executes and then presents the results to the user.
  + **Medium Conversion**: The model's text output may need to be converted into speech or integrated into a user interface.

**6. Zooming In to the Feedforward Pass**

* **Feedforward Pass**: The process of converting the user's problem into the model's domain involves several steps:
  1. **Context Retrieval**: Gather relevant information from the user and external sources.
  2. **Snippetization**: Break down the context into smaller, relevant chunks.
  3. **Scoring and Prioritizing**: Assign scores or priorities to snippets based on their relevance.
  4. **Prompt Assembly**: Assemble the snippets into a well-structured prompt that resembles the model's training data.
* **Boilerplate Text**: Boilerplate text introduces the problem and connects the snippets in a way that makes sense to the model.

**7. Exploring the Complexity of the Loop**

* **State Management**: More complex applications require maintaining state between interactions (e.g., chat history).
* **External Context**: LLMs often need access to external information (e.g., recent news, corporate documentation) through **Retrieval Augmented Generation (RAG)**.
* **Reasoning Depth**: Techniques like **chain-of-thought prompting** can help the model reason through complex problems by generating intermediate steps before providing the final answer.
* **Tool Usage**: LLMs can interact with external tools (e.g., APIs) to perform tasks like searching the web, booking flights, or executing code.

**8. Evaluating LLM Application Quality**

* **Offline Evaluation**: Test the application before deploying it to users. For example:
  + **Code Completion**: Test whether the generated code passes tests.
  + **Chat Applications**: Use an LLM to judge the quality of chat transcripts.
* **Online Evaluation**: Gather user feedback after deployment. Metrics include:
  + **Explicit Feedback**: Thumbs-up/down buttons.
  + **Implicit Feedback**: User behavior, such as how often completions are accepted or modified.

**9. Conclusion**

* **LLM Applications as Transformation Layers**: LLM applications act as a bridge between the user's problem domain and the model's text domain.
* **Complexity**: As applications become more complex, they require state management, external context, sophisticated reasoning, and tool integration.
* **Future Chapters**: The next chapters will dive deeper into the topics introduced here, including context retrieval, snippetization, and advanced applications like conversational agents and workflows.

**Key Takeaways:**

1. **LLM Applications** are built around a loop that transforms the user's problem into the model's domain and then converts the model's output back into a solution.
2. **Prompt Engineering** is crucial for ensuring the model generates helpful and relevant completions.
3. **State Management**, **External Context**, and **Tool Usage** are essential for building complex LLM applications.
4. **Evaluation** is critical for ensuring the quality of LLM applications, both before and after deployment.

Chapter 5: **Prompt Content**

**1. Introduction: The Power of LLMs in Contextual Recommendations**

* **Example**: A book recommendation app using LLMs can leverage not just the user's reading history (like traditional recommendation systems) but also additional contextual information such as demographics, preferences, and recent experiences.
* **Key Insight**: LLMs can process **messy textual data** (e.g., user reviews, preferences, and experiences) to make **more targeted and personalized recommendations** compared to traditional algorithms.

**2. Sources of Content**

* **Static Content**: This is **fixed content** that clarifies the task or problem for the LLM. It includes:
  + **Explicit Instructions**: Direct commands or rules (e.g., "Use markdown," "Don’t use hyperlinks").
  + **Few-Shot Prompting**: Providing examples to guide the model's behavior (e.g., showing how to format answers or what kind of responses are expected).
* **Dynamic Content**: This is **context-specific content** that changes based on the user or the situation. It includes:
  + **User-Specific Information**: Details about the user’s preferences, past behavior, or current context (e.g., "The last book I read was *Moby Dick*").
  + **Relevant External Information**: Data retrieved from external sources (e.g., recent news, user reviews, or API data).

**3. Static Content**

* **Clarifying the Question**:
  + **Explicit Instructions**: Clear, direct commands that help the model understand the task (e.g., "Don’t refer to dates after your knowledge cutoff of 2024-03-03").
  + **Implicit Instructions**: Using **few-shot prompting** to show the model how to respond by providing examples.
* **Few-Shot Prompting**:
  + **Definition**: Providing a few examples (or "shots") to guide the model's behavior.
  + **Advantages**:
    - Helps the model understand the expected format and style of responses.
    - Can teach the model to mimic a specific tone or persona (e.g., grumpy vs. genial reviewer).
  + **Drawbacks**:
    1. **Scales Poorly with Context**: If the main question has a lot of context, the examples may not fit within the model's token limit.
    2. **Biases the Model**: The examples can anchor the model's expectations, leading to biased or skewed responses.
    3. **Introduces Spurious Patterns**: The model may pick up unintended patterns from the examples (e.g., ascending or descending order of numbers).

**4. Dynamic Content**

* **Definition**: Dynamic content is **user-specific or context-specific information** that changes with each request. It includes:
  + **User Preferences**: Past behavior, demographics, or preferences.
  + **External Context**: Information retrieved from APIs, databases, or other sources.
* **Considerations**:
  + **Latency**: How quickly the content can be retrieved (e.g., low urgency for email summarization vs. high urgency for real-time chat).
  + **Preparability**: Whether the content can be prepared in advance (e.g., user profile information vs. real-time data).
  + **Comparability**: How to prioritize and compare different pieces of context (e.g., scoring snippets based on relevance).

**5. Retrieval-Augmented Generation (RAG)**

* **Definition**: RAG is a technique where the application **retrieves relevant information** from external sources and incorporates it into the prompt to help the model generate more informed responses.
* **Key Steps**:
  1. **Retrieval**: Search for relevant snippets of information (e.g., user reviews, Wikipedia summaries).
  2. **Incorporation**: Add the retrieved snippets to the prompt to provide context for the model.
* **Chekhov’s Gun Fallacy**: The model may overinterpret irrelevant snippets, assuming they must be important. To avoid this, ensure that only **relevant snippets** are retrieved.
* **Retrieval Methods**:
  1. **Lexical Retrieval**: Based on word overlap (e.g., Jaccard similarity, TF\*IDF).
  2. **Neural Retrieval**: Uses **embedding models** to convert text into vectors and find semantically similar snippets.
* **Neural Retrieval Process**:
  1. **Indexing**: Convert documents into vectors and store them in a vector database (e.g., FAISS, Pinecone).
  2. **Querying**: Convert the user's query into a vector and search for the nearest vectors in the database.
  3. **Retrieval**: Return the most relevant snippets based on vector similarity.

**6. Summarization**

* **Definition**: Summarization is the process of **condensing large amounts of text** into shorter, more manageable summaries.
* **Hierarchical Summarization**: For very long texts, break the content into smaller chunks (e.g., chapters), summarize each chunk, and then summarize the summaries.
* **General vs. Specific Summaries**:
  + **General Summaries**: Provide a broad overview of the text (e.g., summarizing a book).
  + **Specific Summaries**: Focus on specific aspects of the text (e.g., summarizing a user's social media posts to find book preferences).

**7. Conclusion**

* **Static Content**: Used to **clarify the task** and provide **examples** for the model. It remains the same across different users and requests.
* **Dynamic Content**: Provides **user-specific or context-specific information** that changes with each request. It includes details about the user, their preferences, and relevant external data.
* **RAG and Summarization**: Techniques like **Retrieval-Augmented Generation (RAG)** and **summarization** help the model access and process large amounts of information efficiently.
* **Next Steps**: The next chapter will focus on **structuring prompts** and **prioritizing content** to ensure the model can make sense of the prompt and generate high-quality completions.

**Key Takeaways:**

1. **Static Content** is used to define the task and provide examples, while **Dynamic Content** provides user-specific or context-specific details.
2. **Few-Shot Prompting** is a powerful tool for guiding the model's behavior, but it can introduce biases and spurious patterns if not used carefully.
3. **Retrieval-Augmented Generation (RAG)** allows the model to access external information, improving the relevance and accuracy of its responses.
4. **Summarization** is essential for condensing large amounts of text, especially when dealing with long documents or user-generated content.
5. **Hierarchical Summarization** is a useful technique for summarizing very long texts by breaking them into smaller chunks and summarizing each chunk.

Chapter 6: Assembling the Prompt

**1. Anatomy of the Ideal Prompt**

The chapter begins by emphasizing the importance of structuring prompts effectively. It introduces the concept of an **"ideal prompt"** and breaks it down into key components:

* **Introduction**: Sets the context for the model. It clarifies the type of document or task and guides the model's focus from the start.
* **Prompt Elements**: These are the building blocks of the prompt, which can vary in size and number. They can be dynamic (contextual) or static (instructions).
* **Refocus**: A reminder to the model of the main question, especially in longer prompts. This is often done using the **"sandwich technique"**, where the main question is stated at the beginning and end of the prompt.
* **Transition**: The final part of the prompt that shifts from explaining the problem to solving it. This is crucial for getting the model to provide a solution.

The chapter also introduces the concept of the **"Valley of Meh"**, where information in the middle of the prompt is less effectively used by the model compared to the beginning and end. To mitigate this, key information should be placed outside this "valley."

**2. Types of Documents**

The chapter explores different types of documents that can be used as prompts, depending on the task:

* **Advice Conversations**: These are structured as dialogues between a user (advice seeker) and the model (advice provider). This format is natural for chat models and allows for multi-round interactions.
  + **Inception**: A technique where the prompt engineer starts the answer for the model, guiding it to complete the response in a desired way.
* **Analytic Reports**: These are structured like traditional reports, with introductions, analyses, and conclusions. They are useful for tasks requiring objective analysis and are easy to structure.
* **Structured Documents**: These follow formal specifications like XML or YAML, making them easier to parse. Examples include Anthropic’s **Artifacts**, which are self-contained documents like Python scripts or diagrams.

The chapter also recommends using **Markdown** for reports due to its simplicity, universality, and ability to organize content hierarchically.

**3. Formatting Snippets**

The chapter discusses how to format snippets of information within the prompt:

* **Modularity**: Snippets should be easy to insert or remove from the prompt.
* **Naturalness**: Snippets should feel like an organic part of the document.
* **Brevity**: Snippets should be concise to save tokens.
* **Inertness**: The tokenization of one snippet should not affect the tokenization of adjacent snippets.

The chapter also explains how to format **few-shot examples** (examples provided to the model to guide its responses) and introduces the concept of **elastic snippets**, which are flexible versions of the same snippet that can be adjusted based on available space.

**4. Relationships Among Prompt Elements**

Prompt elements are not isolated; they interact in three key ways:

* **Position**: The order in which elements appear in the prompt. This is crucial for maintaining coherence, especially in narratives or reference documents.
* **Importance**: The relative value of each element. Some elements (like the introduction) are more important than others.
* **Dependency**: How elements depend on or exclude each other. For example, one element may require another to make sense, or two elements may be incompatible (e.g., a summary vs. a detailed explanation).

The chapter suggests using **priority tiers** or numerical scores to assess the importance of elements and manage dependencies.

**5. Putting It All Together**

The final section of the chapter explains how to assemble the prompt by solving an **optimization problem**:

* **Constraints**: The prompt must respect dependencies and stay within the token limit.
* **Approaches**:
  + **Additive Greedy Approach**: Start with an empty prompt and iteratively add the highest-value elements that fit.
  + **Subtractive Greedy Approach**: Start with all elements and iteratively remove the least valuable ones.

The chapter emphasizes that these approaches are basic prototypes and may need to be customized based on specific application requirements.

**6. Conclusion**

The chapter concludes by summarizing the key steps in crafting an effective prompt:

1. Choose the right document format (e.g., advice conversation, analytic report, structured document).
2. Convert gathered information into prompt elements (snippets).
3. Refine the elements based on position, importance, and dependency.
4. Assemble the prompt using a custom prompt-crafting engine.

The chapter sets the stage for the next step: ensuring that the model provides meaningful and accurate responses.

**Key Takeaways**

* **Prompt Structure**: A well-structured prompt includes an introduction, context, refocus, and transition.
* **Document Types**: Different tasks require different document formats (e.g., advice conversations, analytic reports, structured documents).
* **Snippet Formatting**: Snippets should be modular, natural, brief, and inert.
* **Element Relationships**: Prompt elements must be organized based on position, importance, and dependency.
* **Prompt Assembly**: The final prompt is assembled by solving an optimization problem, balancing constraints like token limits and dependencies.

Chapter 7: Taming the model

**1. Anatomy of the Ideal Completion**

The chapter begins by breaking down the components of an LLM completion, similar to how prompts were dissected in Chapter 6. The goal is to ensure that completions are clear, effective, and free from unnecessary delays or confusing details. The key components of an ideal completion are:

* **Preamble**: The initial part of the generated text that sets the stage for the main content. There are three types of preambles:
  + **Structural Boilerplate**: Text between the end of the prompt and the start of the completion. This can often be minimized by including deterministic boilerplate in the prompt.
  + **Reasoning**: Detailed explanations or chain-of-thought processes that help the model arrive at a correct answer. This is particularly useful in complex tasks where the model needs to break down the problem into smaller steps.
  + **Fluff**: Unnecessary verbosity or politeness, often introduced by RLHF-trained models. This can be managed by reformatting prompts or using few-shot examples to guide the model toward more concise responses.
* **Recognizable Start and End**: The completion should have a clear beginning and end to make it easy to parse the main answer. This is especially important for programmatic use, where extraneous information can be costly in terms of time and compute resources.
* **Postscript**: The fluffy or irrelevant part of the completion that comes after the main answer. The chapter discusses techniques like **stop sequences** and **streaming** to control the length of the completion and avoid generating unnecessary tokens.

**2. Beyond the Text: Logprobs**

The chapter introduces **logprobs** (logarithmic probabilities), which provide insights into the model's confidence in its token choices. Logprobs can be used to:

* **Evaluate Answer Quality**: By summing or averaging logprobs, you can assess the model's confidence in its response. This is useful for determining the reliability of the answer.
* **Calibration**: Adjusting the model's decision-making process by shifting logprobs to better match the desired certainty threshold. This is particularly useful in classification tasks.
* **Critical Points in the Prompt**: Logprobs can also be used to detect surprising or unusual parts of the prompt, such as typos or high-information-density passages.

**3. LLMs for Classification**

The chapter discusses how LLMs can be used for classification tasks, where the model must choose from a fixed set of categories. Key points include:

* **Unique Token Starts**: To avoid ambiguity, each classification option should start with a unique token. This prevents the model from combining probabilities for options that share the same initial token.
* **Calibration**: Adjusting the model's logprobs to better match the desired decision thresholds. This is particularly important in applications like email filtering, where the model's default confidence levels may not align with user expectations.

**4. Choosing the Model**

The chapter emphasizes the importance of selecting the right model for your application. Key considerations include:

* **Intelligence**: How well the model performs on complex tasks requiring reasoning or accuracy.
* **Speed**: How quickly the model can generate responses, especially in interactive applications.
* **Cost**: The expense of running inference, which can vary significantly between models.
* **Ease of Use**: The level of support provided by the model provider, including deployment, routing, and caching.
* **Functionality**: Whether the model supports features like instruction following, chat, tool use, and logprobs.
* **Special Requirements**: Non-negotiable factors like data residency, open-source availability, or specific training data.

The chapter also discusses various model providers, including **OpenAI**, **Anthropic**, **Mistral**, **Cohere**, **Google**, and **Meta**, and recommends using comparison sites like **Artificial Analysis** to evaluate options.

**5. Fine-Tuning**

The chapter introduces the concept of **fine-tuning**, where an existing model is trained on specific tasks or domains to improve its performance. Key points include:

* **Full Fine-Tuning**: Adjusting all of the model's parameters to learn new domains or tasks. This requires a large number of training examples and significant computational resources.
* **Parameter-Efficient Fine-Tuning (e.g., LoRA)**: Adjusting only a subset of the model's parameters, making the process faster and more efficient. This is useful for tasks where the model already has the necessary capabilities but needs to adapt to specific formats or styles.
* **Soft Prompting**: Using machine learning to find a model state that encourages desired outputs, rather than crafting specific prompts. This is a more advanced technique and may not be supported by all model frameworks.

The chapter also discusses the **Little Red Riding Hood principle** in the context of fine-tuning, emphasizing that prompts should resemble the fine-tuned documents to avoid the model reverting to its original training.

**6. Conclusion**

The chapter concludes by summarizing the key techniques for "taming" the model, including:

* Clearly defining the desired completion format and style.
* Using logprobs to evaluate and calibrate the model's responses.
* Selecting the right model based on intelligence, speed, cost, and functionality.
* Considering fine-tuning to adapt the model to specific tasks or domains.

The chapter sets the stage for the next part of the book, which will explore advanced techniques for using LLMs as flexible agents and workflow execution systems.

**Key Takeaways**

* **Completion Structure**: A well-structured completion includes a clear preamble, main answer, and recognizable end.
* **Logprobs**: These provide insights into the model's confidence and can be used to evaluate and calibrate responses.
* **Model Selection**: The right model depends on factors like intelligence, speed, cost, and functionality.
* **Fine-Tuning**: Adapting an existing model to specific tasks can significantly improve performance, but requires careful consideration of training data and techniques.

Chapter 8: Conversational Agency

**1. Tool Usage**

The chapter begins by discussing the limitations of traditional chat models, which are confined to the information they were trained on and cannot access real-time or private data. To overcome these limitations, **tool usage** is introduced, allowing models to interact with external APIs and perform tasks like retrieving real-time information, performing calculations, or making changes to the environment.

* **Defining Tools**: Tools are represented as JSON schemas that describe their functionality, arguments, and expected outputs. For example, a tool to get or set room temperature might be defined as follows:

python

Copy

tools = [

{

"type": "function",

"function": {

"name": "get\_room\_temp",

"description": "Get the ambient room temperature in Fahrenheit",

},

},

{

"type": "function",

"function": {

"name": "set\_room\_temp",

"description": "Set the ambient room temperature in Fahrenheit",

"parameters": {

"type": "object",

"properties": {

"temp": {

"type": "integer",

"description": "The desired room temperature in °F",

},

},

"required": ["temp"],

},

},

},

]

* **Tool Invocation**: The model can invoke tools by generating a function call in its response. The application then executes the function and returns the result to the model, which incorporates it into the conversation. This process is demonstrated in the process\_messages function, which handles tool calls and responses.
* **Internal Representation**: The chapter explains how tools are represented internally in the prompt, often using **TypeScript-like syntax** to describe functions and their arguments. This helps the model understand how to invoke tools correctly.
* **Guidelines for Tool Definitions**:
  + **Keep tools simple**: Avoid overly complex tools with many parameters.
  + **Use meaningful names**: Tool and argument names should be self-explanatory.
  + **Handle errors gracefully**: Provide clear error messages that the model can understand and act upon.
  + **Prevent dangerous actions**: Ensure that tools with real-world consequences require explicit user authorization.

**2. Reasoning**

The chapter emphasizes that LLMs, by default, lack an **internal monologue**—they generate text token by token without deep reasoning. To improve the model's ability to reason through problems, several techniques are introduced:

* **Chain of Thought (CoT)**: This technique encourages the model to "think aloud" by breaking down problems into smaller steps before providing an answer. For example, instead of directly answering a question, the model might first explain its reasoning process:

Copy

Q: Will The Exorcist stimulate the limbic system?

A: The Exorcist is a horror movie. Horror movies are scary. The limbic system is involved in fear. Thus, The Exorcist will stimulate the limbic system. So the answer is yes.

* **ReAct (Reasoning and Acting)**: This approach combines reasoning with tool usage. The model iteratively **thinks**, **acts** (by invoking tools), and **observes** the results before arriving at a final answer. For example, to determine which magazine was started first, the model might:
  1. **Think**: "I need to search for both magazines and compare their start dates."
  2. **Act**: Invoke a search tool to retrieve information about each magazine.
  3. **Observe**: Compare the results and conclude which magazine was started first.
* **Plan-and-Solve Prompting**: This technique asks the model to first devise a plan before executing it. For example:

Copy

Let’s first understand the problem and devise a plan to solve the problem. Then, let’s carry out the plan and solve the problem step-by-step.

* **Reflexion**: This approach allows the model to review its work after the fact, identify mistakes, and make corrections. For example, if the model generates code that fails unit tests, it can use the error messages to improve its next attempt.
* **Branch-Solve-Merge**: This technique involves breaking a problem into multiple subproblems, solving them independently, and then merging the results. This can lead to more robust solutions, especially for complex tasks.

**3. Context for Task-Based Interactions**

The chapter discusses how to manage **context** in conversational agents, which is crucial for maintaining coherent and relevant interactions. Context can come from several sources:

* **Preamble**: Sets up the agent's behavior and defines the tools it can use.
* **Prior Conversation**: Includes all previous messages in the conversation, providing background information for the current request.
* **Artifacts**: Pieces of data relevant to the conversation, such as flight details or book excerpts. Artifacts can be attached to user or assistant messages.
* **Current Exchange**: The most recent user request, along with any tool calls and responses generated while handling the request.

The chapter provides guidelines for organizing context:

* **Selecting Tools**: Only include tools relevant to the current task to avoid confusing the model.
* **Presenting Artifacts**: Use formats like XML or Markdown to embed artifacts in the prompt. Consider summarizing large artifacts to avoid overwhelming the model.
* **Managing Conversation History**: Drop irrelevant parts of the conversation to keep the prompt concise and focused.

**4. Building a Conversational Agent**

The chapter walks through the process of building a **conversational agent** using Python and OpenAI's API. The key components are:

* **process\_messages**: Handles tool invocation and appends tool responses to the conversation.
* **run\_conversation**: Manages the full conversation loop, taking user input, processing it, and displaying the assistant's responses.

The chapter provides a complete example of a **thermostat assistant** that can get and set room temperatures based on user requests. The agent demonstrates **common sense reasoning** and **context awareness**, such as remembering the initial temperature when asked to reset it.

**5. User Experience (UX)**

The chapter concludes with a discussion of **UX considerations** for conversational agents:

* **Spinner**: Indicates when the agent is processing a response.
* **Tool Call Indicators**: Show when the agent is using tools in the background.
* **Tool Call Inspection**: Allow users to view and modify tool arguments to correct mistakes.
* **Authorization Requests**: Ensure that users approve potentially dangerous actions, such as purchasing tickets or transferring funds.
* **Artifact Visibility**: Let users see the artifacts the agent is considering, helping them understand the agent's thought process.

**Key Takeaways**

* **Tool Usage**: Conversational agents can interact with the real world by invoking tools, such as APIs for retrieving information or performing actions.
* **Reasoning**: Techniques like **Chain of Thought**, **ReAct**, and **Plan-and-Solve** improve the model's ability to reason through complex problems.
* **Context Management**: Effective use of context, including **preamble**, **prior conversation**, and **artifacts**, is crucial for maintaining coherent and relevant interactions.
* **Building Agents**: The chapter provides a step-by-step guide to building a conversational agent, complete with tool invocation and context management.
* **User Experience**: UX considerations, such as **tool call visibility** and **authorization requests**, are essential for creating user-friendly conversational agents.

Chapter 9: LLM Workflows

**1. Introduction: The Trade-Off Between Generality and Strength**

The chapter begins by discussing the limitations of current LLMs in achieving **Artificial General Intelligence (AGI)**, which would require human-like reasoning, problem-solving, and creativity. While LLMs are more general and powerful than traditional machine learning models, they still struggle with tasks requiring deep reasoning, especially in domains like mathematics and scientific discovery.

* **Generality vs. Strength**:
  + **Generality**: Conversational agents like ChatGPT can handle a wide range of topics but are limited in their ability to solve complex, multi-step tasks.
  + **Strength**: LLM workflows, on the other hand, are designed for specific tasks and can break down complex problems into smaller, manageable steps, making them more powerful but less general.

The chapter introduces the concept of **LLM workflows**, which are structured processes that decompose large tasks into smaller, well-defined tasks. These workflows are driven by a **supervisor process** (which may or may not use an LLM) that coordinates tasks, distributes work, and ensures the workflow achieves its goal.

**2. Would a Conversational Agent Suffice?**

The chapter uses a **Shopify plug-in marketing example** to illustrate the limitations of conversational agents in handling complex tasks. The goal is to:

1. Scrape Shopify storefronts.
2. Extract details about each store.
3. Generate plug-in ideas.
4. Compose and send marketing emails to store owners.

* **Conversational Agent Limitations**:
  + Without tools, the agent can only generate hypothetical plans.
  + Even with tools (e.g., web search, email sending), the agent struggles with multi-step tasks and produces low-quality results.
  + The agent lacks the structure needed to handle complex workflows efficiently.

This example demonstrates the need for **LLM workflows**, which provide the necessary structure to break down and execute complex tasks.

**3. Basic LLM Workflows**

The chapter outlines the steps to build a basic LLM workflow:

1. **Define the Goal**: Identify the desired outcome of the workflow.
2. **Specify Tasks**: Break the workflow into smaller tasks, each with clear inputs and outputs.
3. **Implement Tasks**: Build each task, ensuring it works correctly in isolation.
4. **Assemble the Workflow**: Connect the tasks into a coherent workflow.

**Defining Tasks**

Each task in the workflow should have a clear purpose, input, and output. For example:

* **Input**: A Shopify storefront's HTML.
* **Output**: A summary of the store's products, tone, values, and themes.

**Implementing Tasks**

Tasks can be implemented using **templated prompts** or **tool-based approaches**:

* **Templated Prompts**: Use prompt templates to guide the LLM in generating the desired output. For example, a prompt template for generating marketing emails might include placeholders for store details and plug-in concepts.
* **Tool-Based Approach**: Use LLMs with tool-calling capabilities to extract structured data from unstructured inputs (e.g., extracting restaurant details from HTML).

**Adding Sophistication to Tasks**

To improve task performance, the chapter suggests:

* **Chain-of-Thought Reasoning**: Encourage the model to "think step-by-step" before generating an answer.
* **ReAct (Reasoning and Acting)**: Combine reasoning with tool usage to solve multi-step problems.
* **Reflexion**: Allow the model to review and correct its output based on feedback.
* **Self-Correction**: Use techniques like **LLM-as-judge** to evaluate and improve task outputs.

**4. Assembling the Workflow**

Once tasks are defined and implemented, they are assembled into a **workflow**. The chapter discusses different workflow topologies:

* **Pipeline**: A linear sequence of tasks where the output of one task feeds into the next.
* **Directed Acyclic Graph (DAG)**: Tasks are connected in a way that allows multiple inputs and outputs, but without cycles.
* **Cyclic Graph**: Tasks can loop back to previous tasks, allowing for iterative improvements (e.g., retrying failed tasks).

The chapter provides an example of a **Shopify plug-in promoter workflow** implemented as a DAG, where tasks include:

1. Emitting storefront HTML.
2. Extracting store details.
3. Generating plug-in concepts.
4. Composing marketing emails.
5. Sending emails.

**5. Advanced LLM Workflows**

The chapter introduces more advanced workflow approaches that give LLMs greater autonomy and flexibility:

* **LLM-Driven Workflow**: Allow an LLM to act as a **workflow agent**, deciding how to route work items between tasks.
* **Stateful Task Agents**: Each task is implemented as an **agent** that maintains state and updates work items as needed.
* **Roles and Delegation**: Define agents with specific roles (e.g., Assistant, UserProxy) and delegate tasks to them. Frameworks like **AutoGen** and **CrewAI** facilitate this approach.

**AutoGen and CrewAI**

* **AutoGen**: Allows the creation of teams of conversational agents, each with specific roles and tools. The **UserProxy** agent acts as a stand-in for the human user, guiding the workflow.
* **CrewAI**: Similar to AutoGen, but organizes agents into "crews" with specific roles, goals, and tools. Agents can work sequentially, hierarchically, or consensually.

**6. Evaluation and Optimization**

The chapter emphasizes the importance of **evaluating** and **optimizing** workflows:

* **Task-Level Evaluation**: Test each task in isolation to ensure it performs as expected.
* **Workflow-Level Evaluation**: Monitor the overall workflow to identify and fix issues.
* **Reflexion**: Incorporate feedback loops to improve task outputs.
* **A/B Testing**: Compare different implementations of tasks to determine which performs better.

**7. Conclusion**

The chapter concludes by reiterating the trade-off between **generality** and **strength** in LLM workflows. While conversational agents are general, they lack the strength to handle complex tasks. LLM workflows, on the other hand, are more powerful but require careful design and implementation.

* **Key Takeaways**:
  + **Simplicity is Key**: Use traditional software or simpler machine learning models when possible. Reserve LLMs for tasks that require their unique capabilities.
  + **Modularity**: Break workflows into smaller, well-defined tasks that can be evaluated and optimized independently.
  + **Advanced Techniques**: For highly flexible workflows, consider using **LLM-driven workflows** or **stateful task agents**, but be aware of the increased complexity and potential instability.

The chapter sets the stage for the next chapter, which will focus on **evaluating LLM applications** to ensure they are performing as intended.

**Key Concepts**

* **LLM Workflows**: Structured processes that break down complex tasks into smaller, manageable steps.
* **Generality vs. Strength**: Conversational agents are general but weak; LLM workflows are strong but less general.
* **Task Implementation**: Use **templated prompts** or **tool-based approaches** to implement tasks.
* **Workflow Topologies**: **Pipeline**, **DAG**, and **Cyclic Graph** are common workflow structures.
* **Advanced Workflows**: **LLM-driven workflows**, **stateful task agents**, and **role-based delegation** allow for more flexible and autonomous workflows.
* **Evaluation and Optimization**: Continuously evaluate and optimize tasks and workflows to ensure high performance.