**Chapter 1\_ Prompt Chaining**

Chapter 1: Prompt Chaining

**Prompt Chaining Pattern Overview**

Prompt chaining, sometimes referred to as Pipeline pattern, represents a powerful paradigm for handling intricate tasks when leveraging large language models (LLMs). Rather than expecting an LLM to solve a complex problem in a single, monolithic step, prompt chaining advocates for a divide-and-conquer strategy. The core idea is to break down the original, daunting problem into a sequence of smaller, more manageable sub-problems. Each sub-problem is addressed individually through a specifically designed prompt, and the output generated from one prompt is strategically fed as input into the subsequent prompt in the chain.

This sequential processing technique inherently introduces modularity and clarity into the interaction with LLMs. By decomposing a complex task, it becomes easier to understand and debug each individual step, making the overall process more robust and interpretable. Each step in the chain can be meticulously crafted and optimized to focus on a specific aspect of the larger problem, leading to more accurate and focused outputs.

The output of one step acting as the input for the next is crucial. This passing of information establishes a dependency chain, hence the name, where the context and results of previous operations guide the subsequent processing. This allows the LLM to build on its previous work, refine its understanding, and progressively move closer to the desired solution.

Furthermore, prompt chaining is not just about breaking down problems; it also enables the integration of external knowledge and tools. At each step, the LLM can be instructed to interact with external systems, APIs, or databases, enriching its knowledge and abilities beyond its internal training data. This capability dramatically expands the potential of LLMs, allowing them to function not just as isolated models but as integral components of broader, more intelligent systems.

The significance of prompt chaining extends beyond simple problem-solving. It serves as a foundational technique for building sophisticated AI agents. These agents can utilize prompt chains to autonomously plan, reason, and act in dynamic environments. By strategically structuring the sequence of prompts, an agent can engage in tasks requiring multi-step reasoning, planning, and decision-making. Such agent workflows can mimic human thought processes more closely, allowing for more natural and effective interactions with complex domains and systems.

**Limitations of single prompts:** For multifaceted tasks, using a single, complex prompt for an LLM can be inefficient, causing the model to struggle with constraints and instructions, potentially leading to instruction neglect where parts of the prompt are overlooked, contextual drift where the model loses track of the initial context, error propagation where early errors amplify, prompts which require a longer context window where the model gets insufficient information to respond back and hallucination where the cognitive load increases the chance of incorrect information. For example, a query asking to analyze a market research report, summarize findings, identify trends with data points, and draft an email risks failure as the model might summarize well but fail to extract data or draft an email properly.

**Enhanced Reliability Through Sequential Decomposition:** Prompt chaining addresses these challenges by breaking the complex task into a focused, sequential workflow, which significantly improves reliability and control. Given the example above, a pipeline or chained approach can be described as follows:

1. Initial Prompt (Summarization): "Summarize the key findings of the following market research report: [text]." The model's sole focus is summarization, increasing the accuracy of this initial step.
2. Second Prompt (Trend Identification): "Using the summary, identify the top three emerging trends and extract the specific data points that support each trend: [output from step 1]." This prompt is now more constrained and builds directly upon a validated output.
3. Third Prompt (Email Composition): "Draft a concise email to the marketing team that outlines the following trends and their supporting data: [output from step 2]."

This decomposition allows for more granular control over the process. Each step is simpler and less ambiguous, which reduces the cognitive load on the model and leads to a more accurate and reliable final output. This modularity is analogous to a computational pipeline where each function performs a specific operation before passing its result to the next. To ensure an accurate response for each specific task, the model can be assigned a distinct role at every stage. For example, in the given scenario, the initial prompt could be designated as "Market Analyst," the subsequent prompt as "Trade Analyst," and the third prompt as "Expert Documentation Writer," and so forth.

**The Role of Structured Output:** The reliability of a prompt chain is highly dependent on the integrity of the data passed between steps. If the output of one prompt is ambiguous or poorly formatted, the subsequent prompt may fail due to faulty input. To mitigate this, specifying a structured output format, such as JSON or XML, is crucial.

For example, the output from the trend identification step could be formatted as a JSON object:

|  |
| --- |
| {  "trends": [  {  "trend\_name": "AI-Powered Personalization",  "supporting\_data": "73% of consumers prefer to do business with brands that use personal information to make their shopping experiences more relevant."  },  {  "trend\_name": "Sustainable and Ethical Brands",  "supporting\_data": "Sales of products with ESG-related claims grew 28% over the last five years, compared to 20% for products without."  }  ]  } |

This structured format ensures that the data is machine-readable and can be precisely parsed and inserted into the next prompt without ambiguity. This practice minimizes errors that can arise from interpreting natural language and is a key component in building robust, multi-step LLM-based systems.

**Practical Applications & Use Cases**

Prompt chaining is a versatile pattern applicable in a wide range of scenarios when building agentic systems. Its core utility lies in breaking down complex problems into sequential, manageable steps. Here are several practical applications and use cases:

**1. Information Processing Workflows:** Many tasks involve processing raw information through multiple transformations. For instance, summarizing a document, extracting key entities, and then using those entities to query a database or generate a report. A prompt chain could look like:

* Prompt 1: Extract text content from a given URL or document.
* Prompt 2: Summarize the cleaned text.
* Prompt 3: Extract specific entities (e.g., names, dates, locations) from the summary or original text.
* Prompt 4: Use the entities to search an internal knowledge base.
* Prompt 5: Generate a final report incorporating the summary, entities, and search results.

This methodology is applied in domains such as automated content analysis, the development of AI-driven research assistants, and complex report generation.

**2. Complex Query Answering:** Answering complex questions that require multiple steps of reasoning or information retrieval is a prime use case. For example, "What were the main causes of the stock market crash in 1929, and how did government policy respond?"

* Prompt 1: Identify the core sub-questions in the user's query (causes of crash, government response).
* Prompt 2: Research or retrieve information specifically about the causes of the 1929 crash.
* Prompt 3: Research or retrieve information specifically about the government's policy response to the 1929 stock market crash.
* Prompt 4: Synthesize the information from steps 2 and 3 into a coherent answer to the original query.

This sequential processing methodology is integral to developing AI systems capable of multi-step inference and information synthesis. Such systems are required when a query cannot be answered from a single data point but instead necessitates a series of logical steps or the integration of information from diverse sources.

For example, an automated research agent designed to generate a comprehensive report on a specific topic executes a hybrid computational workflow. Initially, the system retrieves numerous relevant articles. The subsequent task of extracting key information from each article can be performed concurrently for each source. This stage is well-suited for parallel processing, where independent sub-tasks are run simultaneously to maximize efficiency.

However, once the individual extractions are complete, the process becomes inherently sequential. The system must first collate the extracted data, then synthesize it into a coherent draft, and finally review and refine this draft to produce a final report. Each of these later stages is logically dependent on the successful completion of the preceding one. This is where prompt chaining is applied: the collated data serves as the input for the synthesis prompt, and the resulting synthesized text becomes the input for the final review prompt. Therefore, complex operations frequently combine parallel processing for independent data gathering with prompt chaining for the dependent steps of synthesis and refinement.

**3. Data Extraction and Transformation:** The conversion of unstructured text into a structured format is typically achieved through an iterative process, requiring sequential modifications to improve the accuracy and completeness of the output.

* Prompt 1: Attempt to extract specific fields (e.g., name, address, amount) from an invoice document.
* Processing: Check if all required fields were extracted and if they meet format requirements.
* Prompt 2 (Conditional): If fields are missing or malformed, craft a new prompt asking the model to specifically find the missing/malformed information, perhaps providing context from the failed attempt.
* Processing: Validate the results again. Repeat if necessary.
* Output: Provide the extracted, validated structured data.

This sequential processing methodology is particularly applicable to data extraction and analysis from unstructured sources like forms, invoices, or emails. For example, solving complex Optical Character Recognition (OCR) problems, such as processing a PDF form, is more effectively handled through a decomposed, multi-step approach.

Initially, a large language model is employed to perform the primary text extraction from the document image. Following this, the model processes the raw output to normalize the data, a step where it might convert numeric text, such as "one thousand and fifty," into its numerical equivalent, 1050. A significant challenge for LLMs is performing precise mathematical calculations. Therefore, in a subsequent step, the system can delegate any required arithmetic operations to an external calculator tool. The LLM identifies the necessary calculation, feeds the normalized numbers to the tool, and then incorporates the precise result. This chained sequence of text extraction, data normalization, and external tool use achieves a final, accurate result that is often difficult to obtain reliably from a single LLM query.

**4. Content Generation Workflows:** The composition of complex content is a procedural task that is typically decomposed into distinct phases, including initial ideation, structural outlining, drafting, and subsequent revision

* Prompt 1: Generate 5 topic ideas based on a user's general interest.
* Processing: Allow the user to select one idea or automatically choose the best one.
* Prompt 2: Based on the selected topic, generate a detailed outline.
* Prompt 3: Write a draft section based on the first point in the outline.
* Prompt 4: Write a draft section based on the second point in the outline, providing the previous section for context. Continue this for all outline points.
* Prompt 5: Review and refine the complete draft for coherence, tone, and grammar.

This methodology is employed for a range of natural language generation tasks, including the automated composition of creative narratives, technical documentation, and other forms of structured textual content.

**5. Conversational Agents with State:** Although comprehensive state management architectures employ methods more complex than sequential linking, prompt chaining provides a foundational mechanism for preserving conversational continuity. This technique maintains context by constructing each conversational turn as a new prompt that systematically incorporates information or extracted entities from preceding interactions in the dialogue sequence.

* Prompt 1: Process User Utterance 1, identify intent and key entities.
* Processing: Update conversation state with intent and entities.
* Prompt 2: Based on current state, generate a response and/or identify the next required piece of information.
* Repeat for subsequent turns, with each new user utterance initiating a chain that leverages the accumulating conversation history (state).

This principle is fundamental to the development of conversational agents, enabling them to maintain context and coherence across extended, multi-turn dialogues. By preserving the conversational history, the system can understand and appropriately respond to user inputs that depend on previously exchanged information.

**6. Code Generation and Refinement:** The generation of functional code is typically a multi-stage process, requiring a problem to be decomposed into a sequence of discrete logical operations that are executed progressively

* Prompt 1: Understand the user's request for a code function. Generate pseudocode or an outline.
* Prompt 2: Write the initial code draft based on the outline.
* Prompt 3: Identify potential errors or areas for improvement in the code (perhaps using a static analysis tool or another LLM call).
* Prompt 4: Rewrite or refine the code based on the identified issues.
* Prompt 5: Add documentation or test cases.

In applications such as AI-assisted software development, the utility of prompt chaining stems from its capacity to decompose complex coding tasks into a series of manageable sub-problems. This modular structure reduces the operational complexity for the large language model at each step. Critically, this approach also allows for the insertion of deterministic logic between model calls, enabling intermediate data processing, output validation, and conditional branching within the workflow. By this method, a single, multifaceted request that could otherwise lead to unreliable or incomplete results is converted into a structured sequence of operations managed by an underlying execution framework.

**7. Multimodal and multi-step reasoning:** Analyzing datasets with diverse modalities necessitates breaking down the problem into smaller, prompt-based tasks. For example, interpreting an image that contains a picture with embedded text, labels highlighting specific text segments, and tabular data explaining each label, requires such an approach.

* Prompt 1: Extract and comprehend the text from the user's image request.
* Prompt 2: Link the extracted image text with its corresponding labels.
* Prompt 3: Interpret the gathered information using a table to determine the required output.

**Hands-On Code Example**

Implementing prompt chaining ranges from direct, sequential function calls within a script to the utilization of specialized frameworks designed to manage control flow, state, and component integration. Frameworks such as LangChain, LangGraph, Crew AI, and the Google Agent Development Kit (ADK) offer structured environments for constructing and executing these multi-step processes, which is particularly advantageous for complex architectures.

For the purpose of demonstration, LangChain and LangGraph are suitable choices as their core APIs are explicitly designed for composing chains and graphs of operations. LangChain provides foundational abstractions for linear sequences, while LangGraph extends these capabilities to support stateful and cyclical computations, which are necessary for implementing more sophisticated agentic behaviors. This example will focus on a fundamental linear sequence.

The following code implements a two-step prompt chain that functions as a data processing pipeline. The initial stage is designed to parse unstructured text and extract specific information. The subsequent stage then receives this extracted output and transforms it into a structured data format.

To replicate this procedure, the required libraries must first be installed. This can be accomplished using the following command:

|  |
| --- |
| pip install langchain langchain-community langchain-openai langgraph |

Note that langchain-openai can be substituted with the appropriate package for a different model provider. Subsequently, the execution environment must be configured with the necessary API credentials for the selected language model provider, such as OpenAI, Google Gemini, or Anthropic.

|  |
| --- |
| import os  from langchain\_openai import ChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.output\_parsers import StrOutputParser  # For better security, load environment variables from a .env file  # from dotenv import load\_dotenv  # load\_dotenv()  # Make sure your OPENAI\_API\_KEY is set in the .env file  # Initialize the Language Model (using ChatOpenAI is recommended)  llm = ChatOpenAI(temperature=0)  # --- Prompt 1: Extract Information ---  prompt\_extract = ChatPromptTemplate.from\_template(  "Extract the technical specifications from the following text:\n\n{text\_input}"  )  # --- Prompt 2: Transform to JSON ---  prompt\_transform = ChatPromptTemplate.from\_template(  "Transform the following specifications into a JSON object with 'cpu', 'memory', and 'storage' as keys:\n\n{specifications}"  )  # --- Build the Chain using LCEL ---  # The StrOutputParser() converts the LLM's message output to a simple string.  extraction\_chain = prompt\_extract | llm | StrOutputParser()  # The full chain passes the output of the extraction chain into the 'specifications'  # variable for the transformation prompt.  full\_chain = (  {"specifications": extraction\_chain}  | prompt\_transform  | llm  | StrOutputParser()  )  # --- Run the Chain ---  input\_text = "The new laptop model features a 3.5 GHz octa-core processor, 16GB of RAM, and a 1TB NVMe SSD."  # Execute the chain with the input text dictionary.  final\_result = full\_chain.invoke({"text\_input": input\_text})  print("\n--- Final JSON Output ---")  print(final\_result) |

This Python code demonstrates how to use the LangChain library to process text. It utilizes two separate prompts: one to extract technical specifications from an input string and another to format these specifications into a JSON object. The ChatOpenAI model is employed for language model interactions, and the StrOutputParser ensures the output is in a usable string format. The LangChain Expression Language (LCEL) is used to elegantly chain these prompts and the language model together. The first chain, extraction\_chain, extracts the specifications. The full\_chain then takes the output of the extraction and uses it as input for the transformation prompt. A sample input text describing a laptop is provided. The full\_chain is invoked with this text, processing it through both steps. The final result, a JSON string containing the extracted and formatted specifications, is then printed.

**Context Engineering and Prompt Engineering**

Context Engineering (see Fig.1) is the systematic discipline of designing, constructing, and delivering a complete informational environment to an AI model prior to token generation. This methodology asserts that the quality of a model's output is less dependent on the model's architecture itself and more on the richness of the context provided.

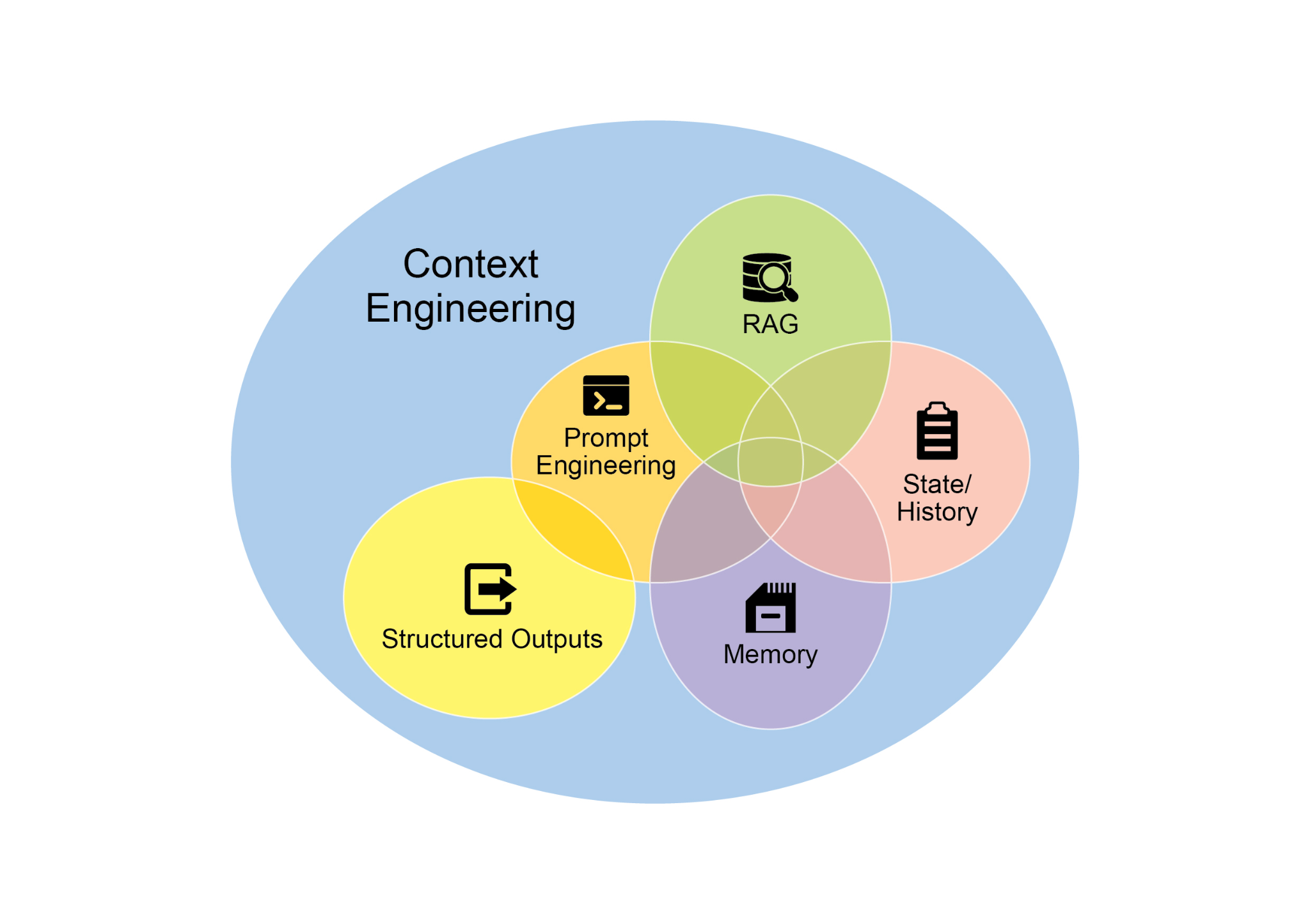


Fig.1:Context Engineering is the discipline of building a rich, comprehensive informational environment for an AI, as the quality of this context is a primary factor in enabling advanced Agentic performance.

It represents a significant evolution from traditional prompt engineering, which focuses primarily on optimizing the phrasing of a user's immediate query. Context Engineering expands this scope to include several layers of information, such as the **system prompt**, which is a foundational set of instructions defining the AI's operational parameters—for instance, *"You are a technical writer; your tone must be formal and precise."* The context is further enriched with external data. This includes retrieved documents, where the AI actively fetches information from a knowledge base to inform its response, such as pulling technical specifications for a project. It also incorporates tool outputs, which are the results from the AI using an external API to obtain real-time data, like querying a calendar to determine a user's availability. This explicit data is combined with critical implicit data, such as user identity, interaction history, and environmental state. The core principle is that even advanced models underperform when provided with a limited or poorly constructed view of the operational environment.

This practice, therefore, reframes the task from merely answering a question to building a comprehensive operational picture for the agent. For example, a context-engineered agent would not just respond to a query but would first integrate the user's calendar availability (a tool output), the professional relationship with an email's recipient (implicit data), and notes from previous meetings (retrieved documents). This allows the model to generate outputs that are highly relevant, personalized, and pragmatically useful. The "engineering" component involves creating robust pipelines to fetch and transform this data at runtime and establishing feedback loops to continually improve context quality.

To implement this, specialized tuning systems can be used to automate the improvement process at scale. For example, tools like Google's Vertex AI prompt optimizer can enhance model performance by systematically evaluating responses against a set of sample inputs and predefined evaluation metrics. This approach is effective for adapting prompts and system instructions across different models without requiring extensive manual rewriting. By providing such an optimizer with sample prompts, system instructions, and a template, it can programmatically refine the contextual inputs, offering a structured method for implementing the feedback loops required for sophisticated Context Engineering.

This structured approach is what differentiates a rudimentary AI tool from a more sophisticated and contextually-aware system. It treats the context itself as a primary component, placing critical importance on what the agent knows, when it knows it, and how it uses that information. The practice ensures the model has a well-rounded understanding of the user's intent, history, and current environment. Ultimately, Context Engineering is a crucial methodology for advancing stateless chatbots into highly capable, situationally-aware systems.

**At a Glance**

**What:** Complex tasks often overwhelm LLMs when handled within a single prompt, leading to significant performance issues. The cognitive load on the model increases the likelihood of errors such as overlooking instructions, losing context, and generating incorrect information. A monolithic prompt struggles to manage multiple constraints and sequential reasoning steps effectively. This results in unreliable and inaccurate outputs, as the LLM fails to address all facets of the multifaceted request.

**Why:** Prompt chaining provides a standardized solution by breaking down a complex problem into a sequence of smaller, interconnected sub-tasks. Each step in the chain uses a focused prompt to perform a specific operation, significantly improving reliability and control. The output from one prompt is passed as the input to the next, creating a logical workflow that progressively builds towards the final solution. This modular, divide-and-conquer strategy makes the process more manageable, easier to debug, and allows for the integration of external tools or structured data formats between steps. This pattern is foundational for developing sophisticated, multi-step Agentic systems that can plan, reason, and execute complex workflows.

**Rule of thumb:** Use this pattern when a task is too complex for a single prompt, involves multiple distinct processing stages, requires interaction with external tools between steps, or when building Agentic systems that need to perform multi-step reasoning and maintain state.

**Visual summary**

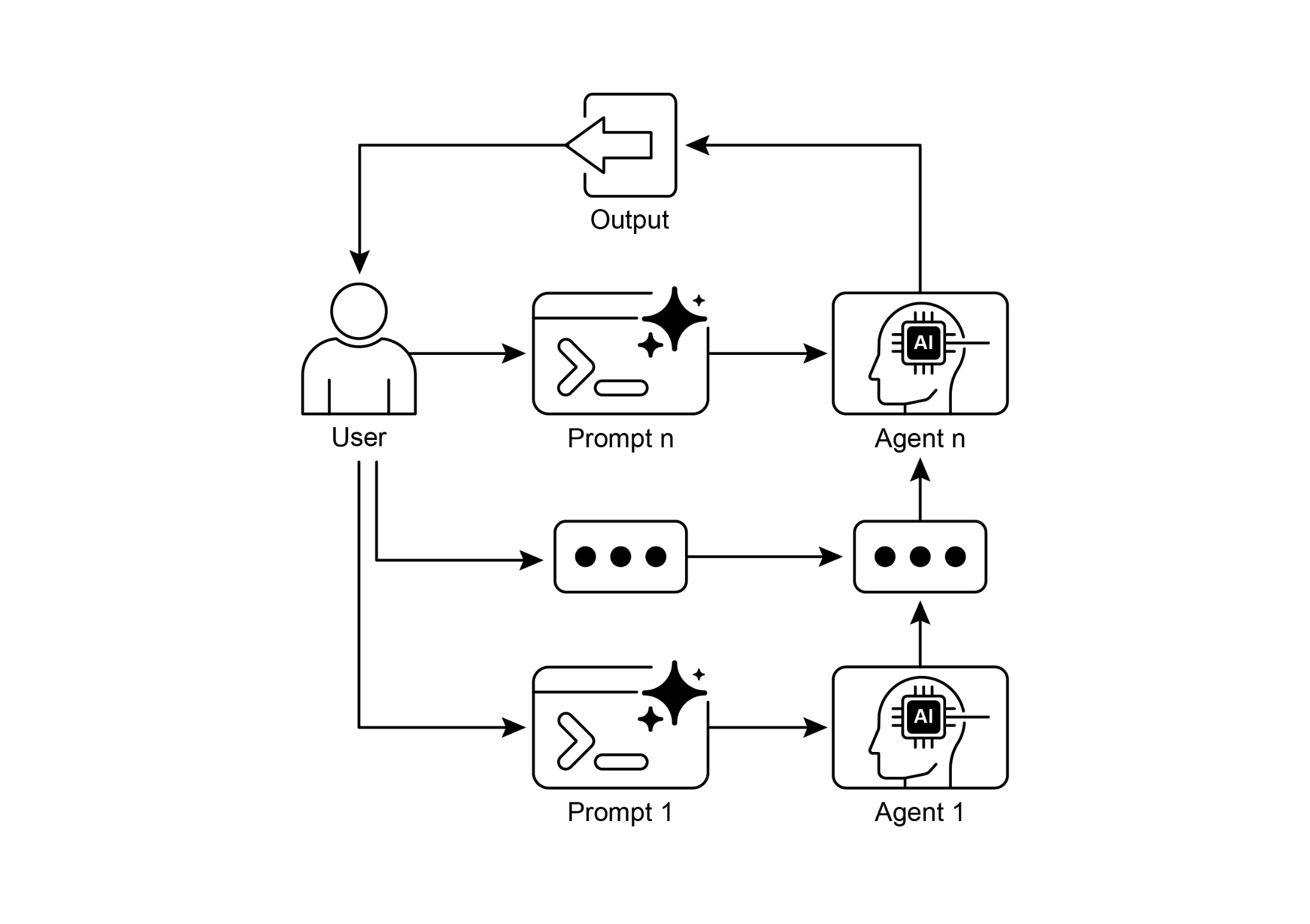


Fig. 2: Prompt Chaining Pattern: Agents receive a series of prompts from the user, with the output of each agent serving as the input for the next in the chain.

**Key Takeaways**

Here are some key takeaways:

* Prompt Chaining breaks down complex tasks into a sequence of smaller, focused steps. This is occasionally known as the Pipeline pattern.
* Each step in a chain involves an LLM call or processing logic, using the output of the previous step as input.
* This pattern improves the reliability and manageability of complex interactions with language models.
* Frameworks like LangChain/LangGraph, and Google ADK provide robust tools to define, manage, and execute these multi-step sequences.

**Conclusion**

By deconstructing complex problems into a sequence of simpler, more manageable sub-tasks, prompt chaining provides a robust framework for guiding large language models. This "divide-and-conquer" strategy significantly enhances the reliability and control of the output by focusing the model on one specific operation at a time. As a foundational pattern, it enables the development of sophisticated AI agents capable of multi-step reasoning, tool integration, and state management. Ultimately, mastering prompt chaining is crucial for building robust, context-aware systems that can execute intricate workflows well beyond the capabilities of a single prompt.

**References**

1. LangChain Documentation on LCEL: <https://python.langchain.com/v0.2/docs/core_modules/expression_language/>
2. LangGraph Documentation: <https://langchain-ai.github.io/langgraph/>
3. Prompt Engineering Guide - Chaining Prompts: <https://www.promptingguide.ai/techniques/chaining>
4. OpenAI API Documentation (General Prompting Concepts): <https://platform.openai.com/docs/guides/gpt/prompting>
5. Crew AI Documentation (Tasks and Processes): <https://docs.crewai.com/>
6. Google AI for Developers (Prompting Guides): <https://cloud.google.com/discover/what-is-prompt-engineering?hl=en>
7. Vertex Prompt Optimizer <https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-optimizer>

**第1章\_提示链**

第一章：提示链

**提示链模式概述**

提示链（有时也被称为管道模式）是一种强大的范式，在利用大语言模型（LLM）处理复杂任务时非常有效。提示链并不期望大语言模型在单一的、整体的步骤中解决复杂问题，而是主张采用分而治之的策略。其核心思想是将原本令人望而生畏的问题分解为一系列更小、更易于管理的子问题。每个子问题都通过专门设计的提示单独处理，一个提示生成的输出会被策略性地作为输入提供给链中的下一个提示。

这种顺序处理技术本质上为与大语言模型（LLMs）的交互引入了模块化和清晰性。通过分解复杂任务，更容易理解和调试每个单独的步骤，使整个过程更加稳健和可解释。链中的每个步骤都可以精心设计和优化，以专注于更大问题的特定方面，从而产生更准确、更有针对性的输出。

一步的输出作为下一步的输入至关重要。这种信息传递建立了一个依赖链，因此得名，其中先前操作的上下文和结果指导后续处理。这使得大语言模型（LLM）能够在其先前工作的基础上继续发展，完善其理解，并逐步接近所需的解决方案。

此外，提示链不仅在于分解问题；它还能实现外部知识和工具的整合。在每一步，大语言模型（LLM）都可以被指示与外部系统、应用程序编程接口（API）或数据库进行交互，从而使其知识和能力超越内部训练数据的范畴。这种能力极大地拓展了大语言模型的潜力，使其不仅能作为孤立的模型运行，还能成为更广泛、更智能系统的重要组成部分。

提示链的重要性不仅限于简单的问题解决。它是构建复杂AI智能体的基础技术。这些智能体可以利用提示链在动态环境中自主规划、推理和行动。通过战略性地组织提示序列，智能体可以执行需要多步推理、规划和决策的任务。这样的智能体工作流程可以更紧密地模拟人类的思维过程，从而实现与复杂领域和系统更自然、更有效的交互。

**单一提示的局限性：**对于多方面的任务，使用单一、复杂的提示来驱动大语言模型（LLM）可能效率低下，导致模型在处理约束和指令时遇到困难，可能会出现指令忽视（即提示的部分内容被忽略）、上下文漂移（即模型失去对初始上下文的跟踪）、错误传播（即早期错误被放大）、需要更长上下文窗口的提示（即模型获得的信息不足以做出回应）以及幻觉（即认知负荷增加了产生错误信息的可能性）等问题。例如，一个要求分析市场研究报告、总结调查结果、确定带有数据点的趋势并起草电子邮件的查询可能会失败，因为模型可能总结得很好，但却无法正确提取数据或起草电子邮件。

**通过顺序分解提高可靠性：**提示链通过将复杂任务分解为聚焦的顺序工作流程来应对这些挑战，这显著提高了可靠性和可控性。以上述示例为例，管道或链式方法可描述如下：

1. 初始提示（总结）：“总结以下市场研究报告的主要发现：[文本]”。模型的唯一重点是总结，提高了这一初始步骤的准确性。
2. 第二个提示（趋势识别）：“利用摘要，确定前三大新兴趋势，并提取支持每个趋势的具体数据点：[步骤1的输出]。”这个提示现在更具约束性，并且直接基于已验证的输出。
3. 第三个提示（撰写电子邮件）：“起草一封简洁的电子邮件给营销团队，概述以下趋势及其支持数据：[步骤2的输出]。”

这种分解方式能够对流程进行更细致的控制。每个步骤都更简单、歧义更少，这减轻了模型的认知负担，从而产生更准确、可靠的最终输出。这种模块化类似于计算管道，其中每个函数在将结果传递给下一个函数之前执行特定操作。为确保针对每个特定任务都能给出准确响应，模型在每个阶段都可以被赋予不同的角色。例如，在给定的场景中，初始提示可以指定为“市场分析师”，后续提示为“交易分析师”，第三个提示为“专家文档撰写人”，依此类推。

**结构化输出的作用：**提示链的可靠性在很大程度上取决于步骤之间传递的数据的完整性。如果一个提示的输出含糊不清或格式不佳，后续提示可能会因输入错误而失败。为了减轻这种情况，指定一种结构化输出格式，如JSON或XML，至关重要。

例如，趋势识别步骤的输出可以格式化为一个JSON对象：

|  |
| --- |
| {  "trends": [  {  "trend\_name": "AI驱动的个性化",  "支持数据": "73%的消费者更愿意与使用个人信息来提升购物体验相关性的品牌开展业务。"  },  {  "trend\_name": "可持续和道德品牌",  "支持数据": "带有ESG相关声明的产品销售额在过去五年中增长了28%，而没有此类声明的产品销售额增长了20%。"  }  ]  } |

这种结构化格式确保数据具有机器可读性，能够被精确解析并毫无歧义地插入到下一个提示中。这种做法可将因解读自然语言而产生的错误降至最低，是构建稳健的、基于大语言模型（LLM）的多步骤系统的关键要素。

**实际应用与用例**

提示链是一种通用模式，在构建智能体系统时适用于广泛的场景。其核心效用在于将复杂问题分解为一系列可管理的步骤。以下是几个实际应用和用例：

**1. 信息处理工作流程：**许多任务涉及通过多次转换来处理原始信息。例如，总结文档、提取关键实体，然后使用这些实体查询数据库或生成报告。提示链可能如下所示：

* 提示 1：从给定的 URL 或文档中提取文本内容。
* 提示2：总结清理后的文本。
* 提示3：从摘要或原文中提取特定实体（例如，姓名、日期、地点）。
* 提示4：使用实体来搜索内部知识库。
* 提示5：生成一份包含总结、实体和搜索结果的最终报告。

这种方法论应用于自动内容分析、AI驱动的研究助手开发和复杂报告生成等领域。

**2. 复杂查询回答：**回答需要多步推理或信息检索的复杂问题是一个主要用例。例如，“1929年股市崩盘的主要原因是什么，政府政策又是如何应对的？”

* 提示1：确定用户查询中的核心子问题（崩溃原因、政府应对措施）。
* 提示2：专门研究或检索有关1929年股市崩盘原因的信息。
* 提示3：专门研究或检索有关政府对1929年股市崩盘的政策应对的信息。
* 提示4：将步骤2和3中的信息整合为对原始查询的连贯回答。

这种顺序处理方法论对于开发能够进行多步推理和信息综合的AI系统至关重要。当一个查询无法从单个数据点得到答案，而是需要一系列逻辑步骤或整合来自不同来源的信息时，就需要这样的系统。

例如，一个旨在就特定主题生成全面报告的自动化研究代理执行混合计算工作流程。最初，系统会检索大量相关文章。从每篇文章中提取关键信息的后续任务可以针对每个来源同时执行。这个阶段非常适合并行处理，即同时运行独立的子任务以实现效率最大化。

然而，一旦各个提取步骤完成，整个过程就会本质上变成顺序执行。系统必须首先整理提取的数据，然后将其综合成一个连贯的草稿，最后审查并完善这个草稿以生成最终报告。这些后续阶段中的每一个在逻辑上都依赖于前一个阶段的成功完成。这就是提示链发挥作用的地方：整理后的数据作为综合提示的输入，而生成的综合文本则成为最终审查提示的输入。因此，复杂操作经常将用于独立数据收集的并行处理与用于综合和完善等依赖步骤的提示链结合起来。

**3. 数据提取与转换：**将非结构化文本转换为结构化格式通常通过迭代过程实现，需要进行顺序修改以提高输出的准确性和完整性。

* 提示1：尝试从发票文档中提取特定字段（例如，姓名、地址、金额）。
* 处理：检查是否提取了所有必填字段，以及这些字段是否符合格式要求。
* 提示2（条件性）：如果字段缺失或格式错误，编写一个新提示，要求模型专门查找缺失或格式错误的信息，可能的话，提供失败尝试的相关背景信息。
* 处理：再次验证结果。如有必要，重复操作。
* 输出：提供提取并验证过的结构化数据。

这种顺序处理方法论特别适用于从表单、发票或电子邮件等非结构化来源中提取和分析数据。例如，解决复杂的光学字符识别（OCR）问题，如处理PDF表单，通过分解的多步骤方法能更有效地解决。

首先，使用大语言模型从文档图像中进行主要的文本提取。随后，该模型处理原始输出以对数据进行归一化，在此步骤中，它可能会将数字文本（如“一千零五十”）转换为其数值等价形式1050。大语言模型面临的一个重大挑战是进行精确的数学计算。因此，在后续步骤中，系统可以将任何所需的算术运算委托给外部计算器工具。大语言模型识别出必要的计算，将归一化后的数字提供给该工具，然后将精确结果整合进来。这种文本提取、数据归一化和外部工具使用的链式序列能够实现最终的准确结果，而这通常很难通过单个大语言模型查询可靠地获得。

**4. 内容创作工作流程：**复杂内容的创作是一项程序性任务，通常会分解为不同阶段，包括初步构思、结构大纲拟定、初稿撰写以及后续修订

* 提示1：根据用户的普遍兴趣生成5个主题创意。
* 处理：允许用户选择一个想法或自动选择最佳想法。
* 提示2：根据选定的主题，生成详细的大纲。
* 提示3：根据大纲中的第一点撰写一个草稿章节。
* 提示4：根据大纲中的第二点撰写一个草稿部分，并提供上一部分作为上下文。对大纲中的所有要点都继续这样做。
* 提示5：审查并完善完整草稿，确保其连贯性、语气和语法无误。

这种方法论被用于一系列自然语言生成任务，包括自动创作创意叙事、技术留档和其他形式的结构化文本内容。

**5. 具有状态的对话代理：**尽管全面的状态管理架构采用的方法比顺序链接更为复杂，但提示链提供了一种保持对话连续性的基础机制。这种技术通过将每个对话轮次构建为一个新的提示来维护上下文，该提示系统地整合了对话序列中先前交互的信息或提取的实体。

* 提示1：处理用户话语1，识别意图和关键实体。
* 处理中：根据意图和实体更新对话状态。
* 提示2：根据当前状态，生成一个响应和/或确定下一个所需的信息片段。
* 对后续轮次重复此操作，每个新的用户话语都会启动一个利用累积对话历史（状态）的链条。

这一原则对于对话代理的开发至关重要，它使对话代理能够在多轮的长对话中保持上下文和连贯性。通过保留对话历史，系统能够理解并适当地响应用户依赖于先前交换信息的输入。

**6. 代码生成与优化：**功能性代码的生成通常是一个多阶段的过程，需要将问题分解为一系列离散的逻辑操作，并逐步执行

* 提示1：理解用户对代码功能的请求。生成伪代码或大纲。
* 提示2：根据大纲编写初始代码草稿。
* 提示3：识别代码中潜在的错误或需要改进的地方（也许可以使用静态分析工具或另一个大语言模型调用）。
* 提示4：根据已识别的问题重写或优化代码。
* 提示5：添加留档或测试用例。

在诸如AI辅助软件开发等应用中，提示链的效用源于其将复杂编码任务分解为一系列可管理子问题的能力。这种模块化结构降低了大语言模型在每一步的操作复杂性。至关重要的是，这种方法还允许在模型调用之间插入确定性逻辑，从而在工作流中实现中间数据处理、输出验证和条件分支。通过这种方式，原本可能导致不可靠或不完整结果的单一、多方面请求被转换为由底层执行框架管理的结构化操作序列。

**7. 多模态和多步骤推理：**分析具有不同模态的数据集需要将问题分解为基于提示的较小任务。例如，解读包含嵌入文本的图片、突出显示特定文本段的标签以及解释每个标签的表格数据的图像，就需要采用这种方法。

* 提示1：从用户的图像请求中提取并理解文本。
* 提示2：将提取的图像文本与其对应的标签关联起来。
* 提示3：使用表格解读收集到的信息，以确定所需输出。

**实践代码示例**

实现提示链的方式多种多样，从脚本内直接、顺序的函数调用，到利用专门设计的框架来管理控制流、状态和组件集成。像LangChain、LangGraph、Crew AI和谷歌代理开发工具包（ADK）这样的框架，为构建和执行这些多步骤流程提供了结构化环境，这对于复杂架构尤为有利。

为便于演示，LangChain和LangGraph是合适的选择，因为它们的核心API是专门为组合操作链和操作图而设计的。LangChain为线性序列提供了基础抽象，而LangGraph则扩展了这些功能，以支持有状态和循环计算，这对于实现更复杂的智能体行为是必要的。本示例将重点关注一个基本的线性序列。

以下代码实现了一个两步提示链，其功能类似于数据处理管道。初始阶段旨在解析非结构化文本并提取特定信息。随后的阶段接收提取的输出，并将其转换为结构化数据格式。

要复制此过程，必须首先安装所需的库。这可以使用以下命令完成：

|  |
| --- |
| pip install langchain langchain-community langchain-openai langgraph |

请注意，langchain-openai可以用适用于不同模型提供商的相应包来替代。随后，执行环境必须配置所选语言模型提供商（如OpenAI、Google Gemini或Anthropic）所需的API凭证。

|  |
| --- |
| import os  from langchain\_openai import ChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  从langchain\_core.output\_parsers导入StrOutputParser  # 为提高安全性，请从.env 文件加载环境变量  # 从 dotenv 导入 load\_dotenv  # load\_dotenv()  # 确保你的 OPENAI\_API\_KEY 已在.env 文件中设置  # 初始化语言模型（建议使用 ChatOpenAI）  llm = ChatOpenAI(temperature=0)  # --- 提示 1：提取信息 ---  prompt\_extract = ChatPromptTemplate.from\_template(  从以下文本中提取技术规格：\n\n{text\_input}  )  # --- 提示 2：转换为 JSON ---  prompt\_transform = ChatPromptTemplate.from\_template(  将以下规格转换为以 'cpu'、'memory' 和 'storage' 为键的 JSON 对象：\n\n{specifications}  )  # --- 使用 LCEL 构建链 ---  # StrOutputParser() 将大语言模型（LLM）的消息输出转换为简单字符串。  extraction\_chain = prompt\_extract | llm | StrOutputParser()  # 完整链将提取链的输出传递到 'specifications' 中  # 转换提示的变量。  full\_chain = (  {"规格": extraction\_chain}  | 提示转换  | 大语言模型  | StrOutputParser()  )  # --- 运行链 ---  新款笔记本电脑采用了3.5 GHz的八核处理器、16GB的RAM和1TB的NVMe SSD。  # 使用输入文本字典执行链。  final\_result = full\_chain.invoke({"text\_input": input\_text})  print("\n---最终JSON输出---")  print(final\_result) |

这段Python代码展示了如何使用LangChain库来处理文本。它使用了两个独立的提示：一个用于从输入字符串中提取技术规格，另一个用于将这些规格格式化为JSON对象。ChatOpenAI模型用于语言模型交互，而StrOutputParser确保输出为可用的字符串格式。LangChain表达式语言（LCEL）用于优雅地将这些提示和语言模型链接在一起。第一个链extraction\_chain用于提取规格。full\_chain然后将提取的输出作为转换提示的输入。提供了一个描述笔记本电脑的示例输入文本。full\_chain使用此文本进行调用，通过两个步骤对其进行处理。最终结果是一个包含提取和格式化规格的JSON字符串，然后将其打印出来。

**上下文工程与提示工程**

上下文工程（见图1）是在生成标记之前，为AI模型设计、构建并提供完整信息环境的系统学科。这种方法论认为，模型输出的质量较少依赖于模型本身的架构，而更多地依赖于所提供上下文的丰富程度。

图1：上下文工程是为AI构建丰富、全面的信息环境的学科，因为这种上下文的质量是实现高级智能体性能的主要因素。

它代表了从传统提示工程的重大演变，传统提示工程主要侧重于优化用户即时查询的措辞。上下文工程将这一范围扩展到包括多个层次的信息，例如**系统提示**，它是一组定义AI操作参数的基本指令，例如*"你是一名技术作家； 你的语气必须正式且精确。*通过外部数据进一步丰富上下文。这包括检索到的文档，其中AI会主动从知识库中获取信息以提供响应依据，例如提取项目的技术规格。它还会整合工具输出，即AI使用外部API获取实时数据的结果，例如查询日历以确定用户的可用性。这种明确的数据与关键的隐含数据（如用户身份、交互历史和环境状态）相结合。核心原则是，即使是先进的模型，在对操作环境的了解有限或构建不佳时，也会表现不佳。

因此，这种做法将任务从单纯回答问题重新定义为为智能体构建全面的操作图景。例如，经过上下文工程处理的智能体不会仅仅对查询做出响应，而是会首先整合用户的日历可用性（工具输出）、与电子邮件收件人的专业关系（隐式数据）以及之前会议的记录（检索到的文档）。这使得模型能够生成高度相关、个性化且实用的输出。“工程”部分涉及创建强大的管道，以便在运行时获取和转换这些数据，并建立反馈循环，以持续提高上下文质量。

为实现这一点，可以使用专门的调优系统来大规模自动执行改进过程。例如，像谷歌的Vertex AI提示优化器这样的工具，可以通过系统地根据一组样本输入和预定义的评估指标来评估响应，从而提高模型性能。这种方法对于在不同模型中调整提示和系统指令非常有效，无需大量手动重写。通过向这样的优化器提供样本提示、系统指令和模板，它可以以编程方式优化上下文输入，为实现复杂上下文工程所需的反馈循环提供一种结构化方法。

这种结构化方法正是区分初级AI工具与更复杂、具备上下文感知能力的系统的关键所在。它将上下文本身视为主要组成部分，高度重视智能体知道什么、何时知道以及如何使用这些信息。这种做法确保模型能够全面理解用户的意图、历史和当前环境。最终，上下文工程是将无状态聊天机器人提升为高度智能、具备情境感知能力的系统的关键方法论。

**概览**

**问题：**复杂任务在单个提示中处理时，往往会让大语言模型不堪重负，导致严重的性能问题。模型的认知负荷增加了出现错误的可能性，如忽略指令、丢失上下文和生成错误信息。单一提示难以有效管理多个约束和顺序推理步骤。这导致输出不可靠且不准确，因为大语言模型无法处理多方面请求的所有方面。

**原因：**提示链通过将复杂问题分解为一系列相互关联的较小子任务，提供了一种标准化的解决方案。链中的每个步骤都使用针对性的提示来执行特定操作，从而显著提高了可靠性和可控性。一个提示的输出作为输入传递给下一个提示，形成一个逻辑工作流，逐步构建出最终的解决方案。这种模块化、分而治之的策略使过程更易于管理、调试，并且允许在步骤之间集成外部工具或结构化数据格式。这种模式是开发能够规划、推理和执行复杂工作流的复杂多步智能体系统的基础。

**经验法则：**当任务对于单个提示来说过于复杂、涉及多个不同的处理阶段、步骤之间需要与外部工具交互，或者在构建需要执行多步推理并保持状态的智能体系统时，请使用此模式。

**可视化总结**

图2：提示链模式：智能体接收用户的一系列提示，链中每个智能体的输出作为下一个智能体的输入。

**关键收获**

以下是一些关键收获：

* 提示链将复杂任务分解为一系列更小、更有针对性的步骤。这有时也被称为管道模式。
* 链中的每一步都涉及大语言模型调用或处理逻辑，将上一步的输出作为输入。
* 这种模式提高了与语言模型复杂交互的可靠性和可管理性。
* 像LangChain/LangGraph和谷歌ADK这样的框架提供了强大的工具，用于定义、管理和执行这些多步骤序列。

**结论**

通过将复杂问题分解为一系列更简单、更易管理的子任务，提示链为引导大语言模型提供了一个强大的框架。这种“分而治之”的策略通过让模型一次专注于一个特定操作，显著提高了输出的可靠性和可控性。作为一种基础模式，它使开发能够进行多步推理、工具集成和状态管理的复杂AI智能体成为可能。最终，掌握提示链对于构建强大的、具有上下文感知能力的系统至关重要，这些系统能够执行远远超出单个提示能力的复杂工作流程。

**参考文献**

1. LangChain关于LCEL的文档：<https://python.langchain.com/v0.2/docs/core_modules/expression_language/>
2. LangGraph文档：<https://langchain-ai.github.io/langgraph/>
3. 提示工程指南 - 链式提示：<https://www.promptingguide.ai/techniques/chaining>
4. OpenAI API文档（通用提示概念）：<https://platform.openai.com/docs/guides/gpt/prompting>
5. 船员AI文档（任务和流程）：<https://docs.crewai.com/>
6. 面向开发者的 Google AI（提示指南）：<https://cloud.google.com/discover/what-is-prompt-engineering?hl=en>
7. Vertex提示优化器<https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-optimizer>