**Chapter 4\_ Reflection**

Chapter 4: Reflection

**Reflection Pattern Overview**

In the preceding chapters, we've explored fundamental agentic patterns: Chaining for sequential execution, Routing for dynamic path selection, and Parallelization for concurrent task execution. These patterns enable agents to perform complex tasks more efficiently and flexibly. However, even with sophisticated workflows, an agent's initial output or plan might not be optimal, accurate, or complete. This is where the **Reflection** pattern comes into play.

The Reflection pattern involves an agent evaluating its own work, output, or internal state and using that evaluation to improve its performance or refine its response. It's a form of self-correction or self-improvement, allowing the agent to iteratively refine its output or adjust its approach based on feedback, internal critique, or comparison against desired criteria. Reflection can occasionally be facilitated by a separate agent whose specific role is to analyze the output of an initial agent.

Unlike a simple sequential chain where output is passed directly to the next step, or routing which chooses a path, reflection introduces a feedback loop. The agent doesn't just produce an output; it then examines that output (or the process that generated it), identifies potential issues or areas for improvement, and uses those insights to generate a better version or modify its future actions.

The process typically involves:

1. **Execution:** The agent performs a task or generates an initial output.
2. **Evaluation/Critique:** The agent (often using another LLM call or a set of rules) analyzes the result from the previous step. This evaluation might check for factual accuracy, coherence, style, completeness, adherence to instructions, or other relevant criteria.
3. **Reflection/Refinement:** Based on the critique, the agent determines how to improve. This might involve generating a refined output, adjusting parameters for a subsequent step, or even modifying the overall plan.
4. **Iteration (Optional but common):** The refined output or adjusted approach can then be executed, and the reflection process can repeat until a satisfactory result is achieved or a stopping condition is met.

A key and highly effective implementation of the Reflection pattern separates the process into two distinct logical roles: a Producer and a Critic. This is often called the "Generator-Critic" or "Producer-Reviewer" model. While a single agent can perform self-reflection, using two specialized agents (or two separate LLM calls with distinct system prompts) often yields more robust and unbiased results.

1. The Producer Agent: This agent's primary responsibility is to perform the initial execution of the task. It focuses entirely on generating the content, whether it's writing code, drafting a blog post, or creating a plan. It takes the initial prompt and produces the first version of the output.

2. The Critic Agent: This agent's sole purpose is to evaluate the output generated by the Producer. It is given a different set of instructions, often a distinct persona (e.g., "You are a senior software engineer," "You are a meticulous fact-checker"). The Critic's instructions guide it to analyze the Producer's work against specific criteria, such as factual accuracy, code quality, stylistic requirements, or completeness. It is designed to find flaws, suggest improvements, and provide structured feedback.

This separation of concerns is powerful because it prevents the "cognitive bias" of an agent reviewing its own work. The Critic agent approaches the output with a fresh perspective, dedicated entirely to finding errors and areas for improvement. The feedback from the Critic is then passed back to the Producer agent, which uses it as a guide to generate a new, refined version of the output. The provided LangChain and ADK code examples both implement this two-agent model: the LangChain example uses a specific "reflector\_prompt" to create a critic persona, while the ADK example explicitly defines a producer and a reviewer agent.

Implementing reflection often requires structuring the agent's workflow to include these feedback loops. This can be achieved through iterative loops in code, or using frameworks that support state management and conditional transitions based on evaluation results. While a single step of evaluation and refinement can be implemented within either a LangChain/LangGraph, or ADK, or Crew.AI chain, true iterative reflection typically involves more complex orchestration.

The Reflection pattern is crucial for building agents that can produce high-quality outputs, handle nuanced tasks, and exhibit a degree of self-awareness and adaptability. It moves agents beyond simply executing instructions towards a more sophisticated form of problem-solving and content generation.

The intersection of reflection with goal setting and monitoring (see Chapter 11) is worth noticing. A goal provides the ultimate benchmark for the agent's self-evaluation, while monitoring tracks its progress. In a number of practical cases, Reflection then might act as the corrective engine, using monitored feedback to analyze deviations and adjust its strategy. This synergy transforms the agent from a passive executor into a purposeful system that adaptively works to achieve its objectives.

Furthermore, the effectiveness of the Reflection pattern is significantly enhanced when the LLM keeps a memory of the conversation (see Chapter 8). This conversational history provides crucial context for the evaluation phase, allowing the agent to assess its output not just in isolation, but against the backdrop of previous interactions, user feedback, and evolving goals. It enables the agent to learn from past critiques and avoid repeating errors. Without memory, each reflection is a self-contained event; with memory, reflection becomes a cumulative process where each cycle builds upon the last, leading to more intelligent and context-aware refinement.

**Practical Applications & Use Cases**

The Reflection pattern is valuable in scenarios where output quality, accuracy, or adherence to complex constraints is critical:

1. Creative Writing and Content Generation:

Refining generated text, stories, poems, or marketing copy.

* **Use Case:** An agent writing a blog post.
* **Reflection:** Generate a draft, critique it for flow, tone, and clarity, then rewrite based on the critique. Repeat until the post meets quality standards.
* **Benefit:** Produces more polished and effective content.

2. Code Generation and Debugging:

Writing code, identifying errors, and fixing them.

* **Use Case:** An agent writing a Python function.
* **Reflection:** Write initial code, run tests or static analysis, identify errors or inefficiencies, then modify the code based on the findings.
* **Benefit:** Generates more robust and functional code.

3. Complex Problem Solving:

Evaluating intermediate steps or proposed solutions in multi-step reasoning tasks.

* **Use Case:** An agent solving a logic puzzle.
* **Reflection:** Propose a step, evaluate if it leads closer to the solution or introduces contradictions, backtrack or choose a different step if needed.
* **Benefit:** Improves the agent's ability to navigate complex problem spaces.

4. Summarization and Information Synthesis:

Refining summaries for accuracy, completeness, and conciseness.

* **Use Case:** An agent summarizing a long document.
* **Reflection:** Generate an initial summary, compare it against key points in the original document, refine the summary to include missing information or improve accuracy.
* **Benefit:** Creates more accurate and comprehensive summaries.

5. Planning and Strategy:

Evaluating a proposed plan and identifying potential flaws or improvements.

* **Use Case:** An agent planning a series of actions to achieve a goal.
* **Reflection:** Generate a plan, simulate its execution or evaluate its feasibility against constraints, revise the plan based on the evaluation.
* **Benefit:** Develops more effective and realistic plans.

6. Conversational Agents:

Reviewing previous turns in a conversation to maintain context, correct misunderstandings, or improve response quality.

* **Use Case:** A customer support chatbot.
* **Reflection:** After a user response, review the conversation history and the last generated message to ensure coherence and address the user's latest input accurately.
* **Benefit:** Leads to more natural and effective conversations.

Reflection adds a layer of meta-cognition to agentic systems, enabling them to learn from their own outputs and processes, leading to more intelligent, reliable, and high-quality results.

**Hands-On Code Example (LangChain)**

The implementation of a complete, iterative reflection process necessitates mechanisms for state management and cyclical execution. While these are handled natively in graph-based frameworks like LangGraph or through custom procedural code, the fundamental principle of a single reflection cycle can be demonstrated effectively using the compositional syntax of LCEL (LangChain Expression Language).

This example implements a reflection loop using the Langchain library and OpenAI's GPT-4o model to iteratively generate and refine a Python function that calculates the factorial of a number. The process starts with a task prompt, generates initial code, and then repeatedly reflects on the code based on critiques from a simulated senior software engineer role, refining the code in each iteration until the critique stage determines the code is perfect or a maximum number of iterations is reached. Finally, it prints the resulting refined code.

First, ensure you have the necessary libraries installed:

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

You will also need to set up your environment with your API key for the language model you choose (e.g., OpenAI, Google Gemini, Anthropic).

|  |
| --- |
| import os  from dotenv import load\_dotenv  from langchain\_openai import ChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.messages import SystemMessage, HumanMessage  # --- Configuration ---  # Load environment variables from .env file (for OPENAI\_API\_KEY)  load\_dotenv()  # Check if the API key is set  if not os.getenv("OPENAI\_API\_KEY"):  raise ValueError("OPENAI\_API\_KEY not found in .env file. Please add it.")  # Initialize the Chat LLM. We use gpt-4o for better reasoning.  # A lower temperature is used for more deterministic outputs.  llm = ChatOpenAI(model="gpt-4o", temperature=0.1)  def run\_reflection\_loop():  """  Demonstrates a multi-step AI reflection loop to progressively improve a Python function.  """  # --- The Core Task ---  task\_prompt = """  Your task is to create a Python function named `calculate\_factorial`.  This function should do the following:  1. Accept a single integer `n` as input.  2. Calculate its factorial (n!).  3. Include a clear docstring explaining what the function does.  4. Handle edge cases: The factorial of 0 is 1.  5. Handle invalid input: Raise a ValueError if the input is a negative number.  """  # --- The Reflection Loop ---  max\_iterations = 3  current\_code = ""  # We will build a conversation history to provide context in each step.  message\_history = [HumanMessage(content=task\_prompt)]  for i in range(max\_iterations):  print("\n" + "="\*25 + f" REFLECTION LOOP: ITERATION {i + 1} " + "="\*25)  # --- 1. GENERATE / REFINE STAGE ---  # In the first iteration, it generates. In subsequent iterations, it refines.  if i == 0:  print("\n>>> STAGE 1: GENERATING initial code...")  # The first message is just the task prompt.  response = llm.invoke(message\_history)  current\_code = response.content  else:  print("\n>>> STAGE 1: REFINING code based on previous critique...")  # The message history now contains the task,  # the last code, and the last critique.  # We instruct the model to apply the critiques.  message\_history.append(HumanMessage(content="Please refine the code using the critiques provided."))  response = llm.invoke(message\_history)  current\_code = response.content  print("\n--- Generated Code (v" + str(i + 1) + ") ---\n" + current\_code)  message\_history.append(response) # Add the generated code to history  # --- 2. REFLECT STAGE ---  print("\n>>> STAGE 2: REFLECTING on the generated code...")  # Create a specific prompt for the reflector agent.  # This asks the model to act as a senior code reviewer.  reflector\_prompt = [  SystemMessage(content="""  You are a senior software engineer and an expert  in Python.  Your role is to perform a meticulous code review.  Critically evaluate the provided Python code based  on the original task requirements.  Look for bugs, style issues, missing edge cases,  and areas for improvement.  If the code is perfect and meets all requirements,  respond with the single phrase 'CODE\_IS\_PERFECT'.  Otherwise, provide a bulleted list of your critiques.  """),  HumanMessage(content=f"Original Task:\n{task\_prompt}\n\nCode to Review:\n{current\_code}")  ]  critique\_response = llm.invoke(reflector\_prompt)  critique = critique\_response.content  # --- 3. STOPPING CONDITION ---  if "CODE\_IS\_PERFECT" in critique:  print("\n--- Critique ---\nNo further critiques found. The code is satisfactory.")  break  print("\n--- Critique ---\n" + critique)  # Add the critique to the history for the next refinement loop.  message\_history.append(HumanMessage(content=f"Critique of the previous code:\n{critique}"))  print("\n" + "="\*30 + " FINAL RESULT " + "="\*30)  print("\nFinal refined code after the reflection process:\n")  print(current\_code)  if \_\_name\_\_ == "\_\_main\_\_":  run\_reflection\_loop() |

The code begins by setting up the environment, loading API keys, and initializing a powerful language model like GPT-4o with a low temperature for focused outputs. The core task is defined by a prompt asking for a Python function to calculate the factorial of a number, including specific requirements for docstrings, edge cases (factorial of 0), and error handling for negative input. The run\_reflection\_loop function orchestrates the iterative refinement process. Within the loop, in the first iteration, the language model generates initial code based on the task prompt. In subsequent iterations, it refines the code based on critiques from the previous step. A separate "reflector" role, also played by the language model but with a different system prompt, acts as a senior software engineer to critique the generated code against the original task requirements. This critique is provided as a bulleted list of issues or the phrase 'CODE\_IS\_PERFECT' if no issues are found. The loop continues until the critique indicates the code is perfect or a maximum number of iterations is reached. The conversation history is maintained and passed to the language model in each step to provide context for both generation/refinement and reflection stages. Finally, the script prints the last generated code version after the loop concludes.

**Hands-On Code Example (ADK)**

Let's now look at a conceptual code example implemented using the Google ADK. Specifically, the code showcases this by employing a Generator-Critic structure, where one component (the Generator) produces an initial result or plan, and another component (the Critic) provides critical feedback or a critique, guiding the Generator towards a more refined or accurate final output.

|  |
| --- |
| from google.adk.agents import SequentialAgent, LlmAgent  # The first agent generates the initial draft.  generator = LlmAgent(  name="DraftWriter",  description="Generates initial draft content on a given subject.",  instruction="Write a short, informative paragraph about the user's subject.",  output\_key="draft\_text" # The output is saved to this state key.  )  # The second agent critiques the draft from the first agent.  reviewer = LlmAgent(  name="FactChecker",  description="Reviews a given text for factual accuracy and provides a structured critique.",  instruction="""  You are a meticulous fact-checker.  1. Read the text provided in the state key 'draft\_text'.  2. Carefully verify the factual accuracy of all claims.  3. Your final output must be a dictionary containing two keys:  - "status": A string, either "ACCURATE" or "INACCURATE".  - "reasoning": A string providing a clear explanation for your status, citing specific issues if any are found.  """,  output\_key="review\_output" # The structured dictionary is saved here.  )  # The SequentialAgent ensures the generator runs before the reviewer.  review\_pipeline = SequentialAgent(  name="WriteAndReview\_Pipeline",  sub\_agents=[generator, reviewer]  )  # Execution Flow:  # 1. generator runs -> saves its paragraph to state['draft\_text'].  # 2. reviewer runs -> reads state['draft\_text'] and saves its dictionary output to state['review\_output']. |

This code demonstrates the use of a sequential agent pipeline in Google ADK for generating and reviewing text. It defines two LlmAgent instances: generator and reviewer. The generator agent is designed to create an initial draft paragraph on a given subject. It is instructed to write a short and informative piece and saves its output to the state key draft\_text. The reviewer agent acts as a fact-checker for the text produced by the generator. It is instructed to read the text from draft\_text and verify its factual accuracy. The reviewer's output is a structured dictionary with two keys: status and reasoning. status indicates if the text is "ACCURATE" or "INACCURATE", while reasoning provides an explanation for the status. This dictionary is saved to the state key review\_output. A SequentialAgent named review\_pipeline is created to manage the execution order of the two agents. It ensures that the generator runs first, followed by the reviewer. The overall execution flow is that the generator produces text, which is then saved to the state. Subsequently, the reviewer reads this text from the state, performs its fact-checking, and saves its findings (the status and reasoning) back to the state. This pipeline allows for a structured process of content creation and review using separate agents.**Note:** An alternative implementation utilizing ADK's LoopAgent is also available for those interested.

Before concluding, it's important to consider that while the Reflection pattern significantly enhances output quality, it comes with important trade-offs. The iterative process, though powerful, can lead to higher costs and latency, since every refinement loop may require a new LLM call, making it suboptimal for time-sensitive applications. Furthermore, the pattern is memory-intensive; with each iteration, the conversational history expands, including the initial output, critique, and subsequent refinements.

**At Glance**

**What:** An agent's initial output is often suboptimal, suffering from inaccuracies, incompleteness, or a failure to meet complex requirements. Basic agentic workflows lack a built-in process for the agent to recognize and fix its own errors. This is solved by having the agent evaluate its own work or, more robustly, by introducing a separate logical agent to act as a critic, preventing the initial response from being the final one regardless of quality.

**Why:** The Reflection pattern offers a solution by introducing a mechanism for self-correction and refinement. It establishes a feedback loop where a "producer" agent generates an output, and then a "critic" agent (or the producer itself) evaluates it against predefined criteria. This critique is then used to generate an improved version. This iterative process of generation, evaluation, and refinement progressively enhances the quality of the final result, leading to more accurate, coherent, and reliable outcomes.

**Rule of thumb:** Use the Reflection pattern when the quality, accuracy, and detail of the final output are more important than speed and cost. It is particularly effective for tasks like generating polished long-form content, writing and debugging code, and creating detailed plans. Employ a separate critic agent when tasks require high objectivity or specialized evaluation that a generalist producer agent might miss.

**Visual summary**

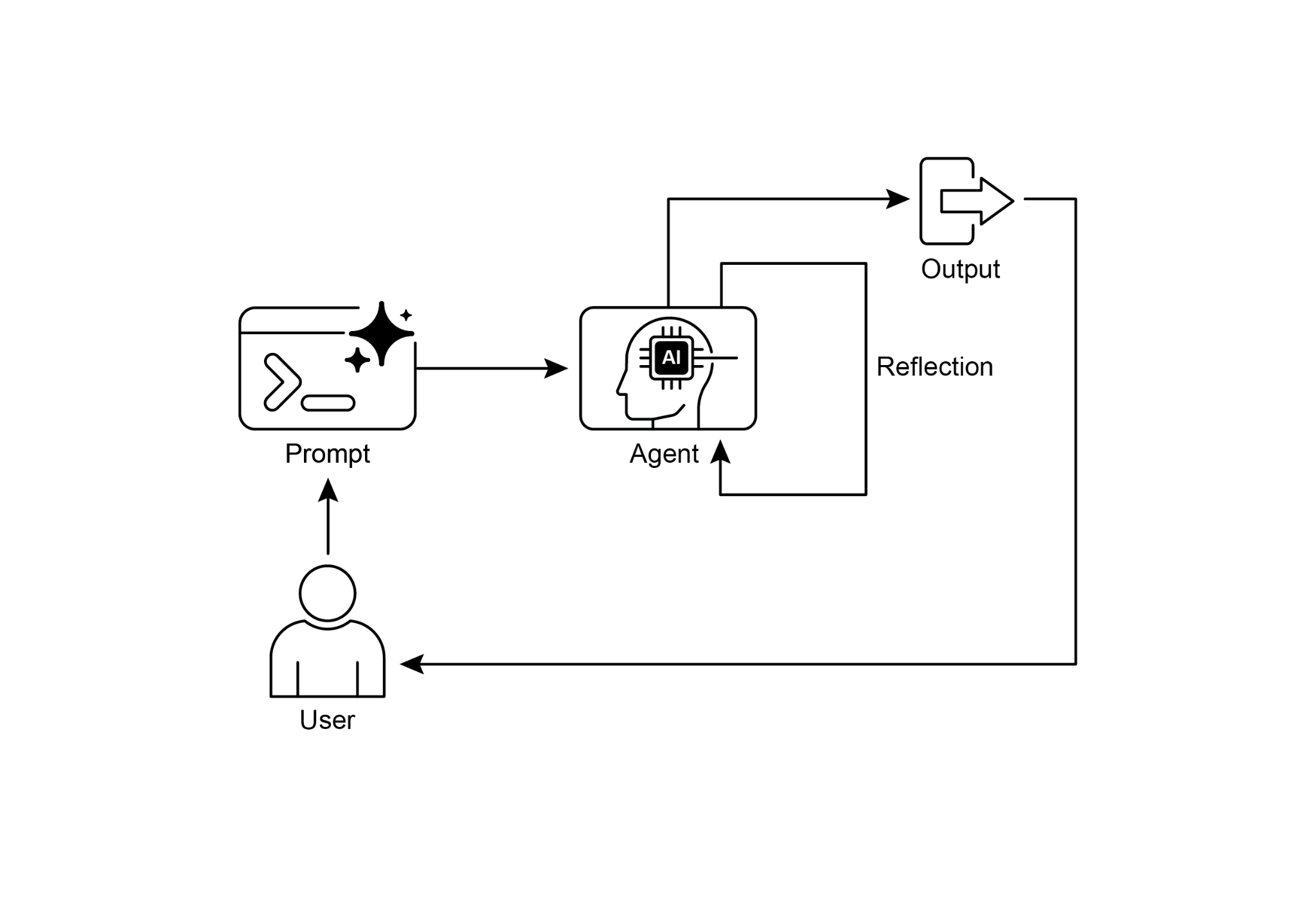


Fig. 1: Reflection design pattern, self-reflection

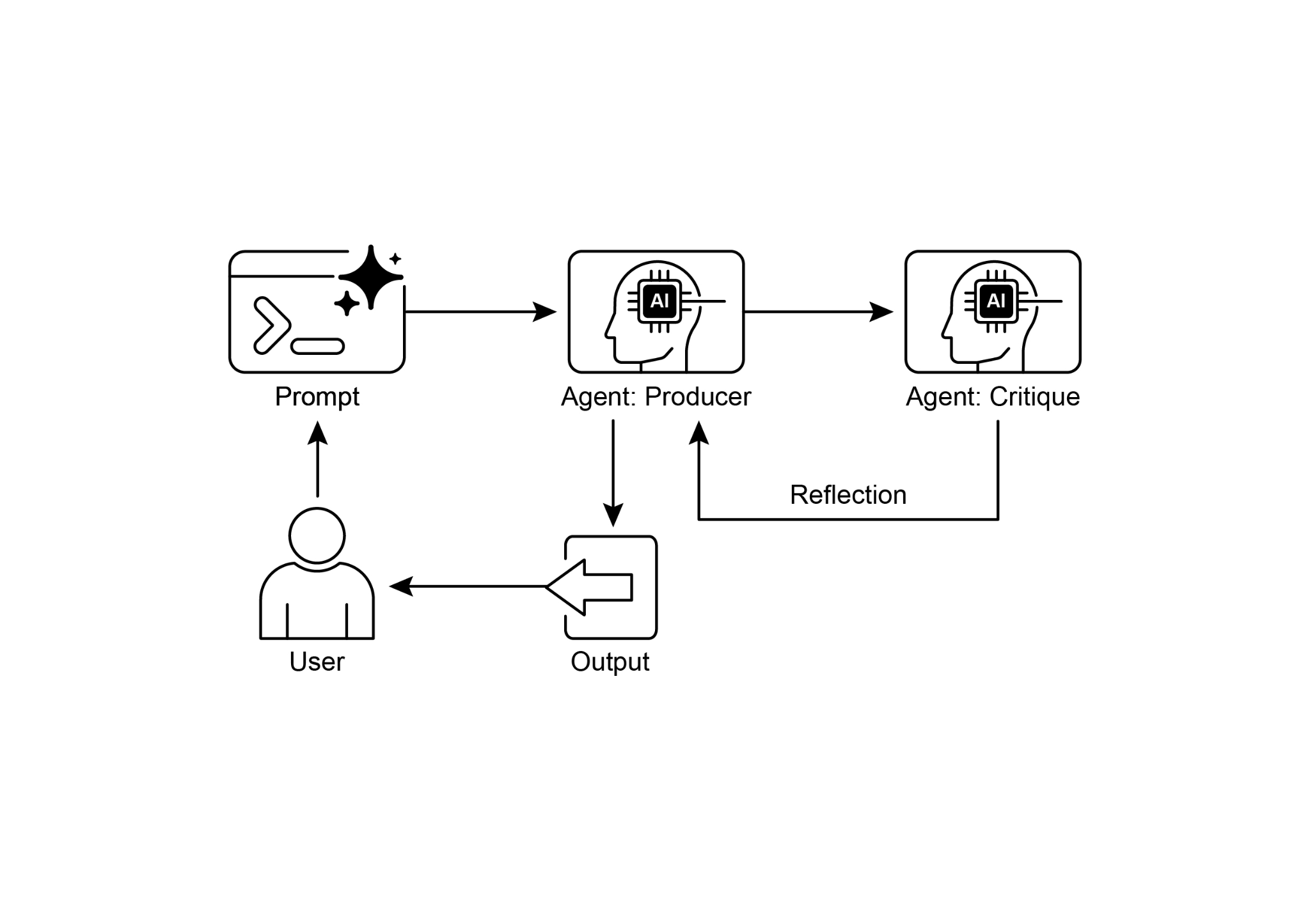


Fig.2: Reflection design pattern, producer and critique agent

**Key Takeaways**

* The primary advantage of the Reflection pattern is its ability to iteratively self-correct and refine outputs, leading to significantly higher quality, accuracy, and adherence to complex instructions.
* It involves a feedback loop of execution, evaluation/critique, and refinement. Reflection is essential for tasks requiring high-quality, accurate, or nuanced outputs.
* A powerful implementation is the Producer-Critic model, where a separate agent (or prompted role) evaluates the initial output. This separation of concerns enhances objectivity and allows for more specialized, structured feedback.
* However, these benefits come at the cost of increased latency and computational expense, along with a higher risk of exceeding the model's context window or being throttled by API services.
* While full iterative reflection often requires stateful workflows (like LangGraph), a single reflection step can be implemented in LangChain using LCEL to pass output for critique and subsequent refinement.
* Google ADK can facilitate reflection through sequential workflows where one agent's output is critiqued by another agent, allowing for subsequent refinement steps.
* This pattern enables agents to perform self-correction and enhance their performance over time.

**Conclusion**

The reflection pattern provides a crucial mechanism for self-correction within an agent's workflow, enabling iterative improvement beyond a single-pass execution. This is achieved by creating a loop where the system generates an output, evaluates it against specific criteria, and then uses that evaluation to produce a refined result. This evaluation can be performed by the agent itself (self-reflection) or, often more effectively, by a distinct critic agent, which represents a key architectural choice within the pattern.

While a fully autonomous, multi-step reflection process requires a robust architecture for state management, its core principle is effectively demonstrated in a single generate-critique-refine cycle. As a control structure, reflection can be integrated with other foundational patterns to construct more robust and functionally complex agentic systems.

**References**

Here are some resources for further reading on the Reflection pattern and related concepts:

1. Training Language Models to Self-Correct via Reinforcement Learning, <https://arxiv.org/abs/2409.12917>
2. LangChain Expression Language (LCEL) Documentation: <https://python.langchain.com/docs/introduction/>
3. LangGraph Documentation:<https://www.langchain.com/langgraph>
4. Google Agent Developer Kit (ADK) Documentation (Multi-Agent Systems): <https://google.github.io/adk-docs/agents/multi-agents/>

**第四章\_反思**

第四章：反思

**反射模式概述**

在前面的章节中，我们探讨了基本的智能体模式：用于顺序执行的链式模式、用于动态路径选择的路由模式，以及用于并发任务执行的并行模式。这些模式使智能体能够更高效、更灵活地执行复杂任务。然而，即使有复杂的工作流程，智能体的初始输出或计划也可能不是最优、准确或完整的。这就是**反思**模式发挥作用的地方。

反思模式涉及一个智能体评估自身的工作、输出或内部状态，并利用该评估来提高其性能或优化其响应。这是一种自我修正或自我提升的形式，使智能体能够根据反馈、内部批判或与期望标准的比较，迭代地优化其输出或调整其方法。反思有时可由一个专门负责分析初始智能体输出的独立智能体来推动。

与简单的顺序链（其中输出直接传递到下一步）或选择路径的路由不同，反思引入了一个反馈循环。智能体不仅产生输出；然后它会检查该输出（或生成该输出的过程），识别潜在问题或改进领域，并利用这些见解生成更好的版本或修改其未来的行动。

该过程通常包括：

1. **执行：**代理执行任务或生成初始输出。
2. **评估/批判：**代理（通常使用另一个大语言模型调用或一组规则）分析上一步的结果。这种评估可能会检查事实准确性、连贯性、风格、完整性、是否遵循指令或其他相关标准。
3. **反思/改进：**基于批评意见，智能体确定如何改进。这可能包括生成优化后的输出、调整后续步骤的参数，甚至修改整体计划。
4. **迭代（可选但常见）：**然后可以执行优化后的输出或调整后的方法，反思过程可以重复进行，直到达到满意的结果或满足停止条件。

反射模式的一个关键且高效的实现方式是将过程分为两个不同的逻辑角色：生产者和评论者。这通常被称为“生成器 - 评论者”或“生产者 - 审查者”模型。虽然单个智能体可以进行自我反思，但使用两个专门的智能体（或使用不同系统提示的两次单独的大语言模型调用）往往能产生更可靠、无偏见的结果。

1. 生产者代理：该代理的主要职责是执行任务的初始阶段。它完全专注于生成内容，无论是编写代码、起草博客文章还是制定计划。它接收初始提示并生成输出的第一个版本。

2. 批评者智能体：该智能体的唯一目的是评估生产者生成的输出。它被赋予了不同的指令集，通常是一个独特的角色设定（例如，“你是一位资深软件工程师”，“你是一位严谨的事实核查员”）。批评者的指令指导它根据特定标准分析生产者的工作，如事实准确性、代码质量、风格要求或完整性。它旨在发现缺陷、提出改进建议并提供结构化反馈。

这种关注点分离的方式非常强大，因为它避免了代理审查自身工作时产生的“认知偏差”。批评者代理以全新的视角审视输出，完全专注于发现错误和改进之处。批评者给出的反馈随后会被传回生产者代理，生产者代理将其作为指导，生成一个新的、经过优化的输出版本。所提供的LangChain和ADK代码示例都实现了这种双代理模型：LangChain示例使用特定的“反思者提示”来创建批评者角色，而ADK示例则明确定义了生产者和审查者代理。

实施反思通常需要对智能体的工作流程进行结构化设计，以纳入这些反馈循环。这可以通过代码中的迭代循环来实现，也可以使用支持状态管理和基于评估结果进行条件转换的框架来实现。虽然评估和改进的单个步骤可以在LangChain/LangGraph、ADK或Crew.AI链中实现，但真正的迭代反思通常涉及更复杂的编排。

反思模式对于构建能够产生高质量输出、处理细微任务，并展现出一定程度的自我认知和适应性的智能体至关重要。它使智能体超越简单的指令执行，迈向更复杂的问题解决和内容生成形式。

反思与目标设定和监控（见第11章）的交集值得关注。目标为智能体的自我评估提供了最终基准，而监控则跟踪其进展。在许多实际案例中，反思随后可能充当纠正引擎，利用监控反馈来分析偏差并调整其策略。这种协同作用将智能体从被动的执行者转变为一个有目的地自适应工作以实现其目标的系统。

此外，当大语言模型（LLM）保留对话记忆时，反思模式的有效性会显著增强（见第8章）。这种对话历史为评估阶段提供了关键背景，使智能体不仅能孤立地评估其输出，还能结合之前的交互、用户反馈和不断演变的目标进行评估。这使智能体能够从过去的批评中学习，避免重复错误。没有记忆，每次反思都是一个独立的事件； 有了记忆，反思就成为一个累积的过程，每个周期都建立在上一个周期的基础之上，从而实现更智能、更具情境感知的优化。

**实际应用与用例**

在输出质量、准确性或对复杂约束的遵循至关重要的场景中，反射模式具有重要价值：

1. 创意写作与内容创作：

润色生成的文本、故事、诗歌或营销文案。

* **用例：**代理撰写博客文章。
* **反思：**先撰写初稿，从行文流畅度、语气和清晰度方面进行审视，然后根据审视结果进行重写。重复此过程，直至文章达到质量标准。
* **优点：**生成更完善、更有效的内容。

2. 代码生成与调试：

编写代码、识别错误并修复它们。

* **用例：**一个智能体编写Python函数。
* **反思：**编写初始代码，运行测试或静态分析，识别错误或低效之处，然后根据发现的问题修改代码。
* **优点：**生成更健壮、更具功能性的代码。

3. 复杂问题解决能力：

在多步骤推理任务中评估中间步骤或提出的解决方案。

* **用例：**一个智能体解决逻辑谜题。
* **反思：**提出一个步骤，评估它是否更接近解决方案或引入矛盾，如果需要则回溯或选择不同的步骤。
* **优点：**提高代理在复杂问题空间中导航的能力。

4. 总结与信息综合：

提炼摘要，确保其准确性、完整性和简洁性。

* **用例：**代理总结长文档。
* **反思：**生成初步总结，将其与原始文档中的关键点进行比较，完善总结以包含遗漏信息或提高准确性。
* **优点：**生成更准确、全面的摘要。

5. 规划与战略：

评估拟议计划并识别潜在缺陷或改进之处。

* **用例：**一个智能体规划一系列行动以实现目标。
* **反思：**制定计划，模拟其执行过程或根据约束条件评估其可行性，根据评估结果修订计划。
* **好处：**制定更有效、更现实的计划。

6. 对话代理：

回顾对话中的先前轮次，以保持上下文连贯、纠正误解或提高回复质量。

* **用例：**客户支持聊天机器人。
* **反思：**在用户做出回应后，回顾对话历史和最后生成的消息，以确保连贯性并准确处理用户的最新输入。
* **好处：**促成更自然、更有效的对话。

反思为能动系统增添了一层元认知，使它们能够从自身的输出和过程中学习，从而产生更智能、可靠和高质量的结果。

**实践代码示例（LangChain）**

完整的迭代式反思过程的实现需要状态管理和循环执行的机制。虽然这些在基于图的框架（如LangGraph）中是原生处理的，或者通过自定义过程代码来处理，但单个反思周期的基本原理可以使用LCEL（LangChain表达式语言）的组合语法有效地展示出来。

此示例使用Langchain库和OpenAI的GPT-4o模型实现了一个反思循环，以迭代方式生成并优化一个计算数字阶乘的Python函数。该过程从任务提示开始，生成初始代码，然后根据模拟的高级软件工程师角色的评审反复反思代码，在每次迭代中优化代码，直到评审阶段确定代码完美或达到最大迭代次数。最后，它会打印出最终优化后的代码。

首先，确保你已经安装了必要的库：

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

你还需要使用你选择的语言模型（例如，OpenAI、谷歌Gemini、Anthropic）的API密钥来设置你的环境。

|  |
| --- |
| 导入 os  from dotenv import load\_dotenv  from langchain\_openai import ChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.messages import SystemMessage, HumanMessage  # --- 配置 ---  # 从.env 文件加载环境变量（用于 OPENAI\_API\_KEY）  load\_dotenv()  # 检查是否设置了 API 密钥  如果没有获取到环境变量 "OPENAI\_API\_KEY"：  raise ValueError("在.env文件中未找到OPENAI\_API\_KEY，请添加。")  # 初始化聊天大语言模型。我们使用 gpt-4o 以获得更好的推理能力。  # 较低的温度用于更具确定性的输出。  llm = ChatOpenAI(model="gpt-4o", temperature=0.1)  def run\_reflection\_loop():  """  展示了一个多步骤的AI反思循环，以逐步改进Python函数。  """  # ---核心任务---  task\_prompt = """  你的任务是创建一个名为 `calculate\_factorial` 的 Python 函数。  此函数应执行以下操作：  1. 接受一个整数 `n` 作为输入。  2. 计算其阶乘（n!）。  3. 包含一个清晰的文档字符串，解释该函数的功能。  4. 处理边界情况：0的阶乘是1。  5. 处理无效输入：如果输入为负数，则抛出 ValueError 异常。  """  # ---反思循环---  max\_iterations = 3  current\_code = ""  # 我们将构建对话历史，以便在每一步中提供上下文。  message\_history = [HumanMessage(content=task\_prompt)]  for i in range(max\_iterations):  print("\n" + "="\*25 + f" 反射循环：迭代 {i + 1} " + "="\*25)  # --- 1. 生成/优化阶段 ---  # 在第一次迭代中，它进行生成。在后续迭代中，它进行优化。  if i == 0:  print("\n>>> 阶段 1: 生成初始代码...")  # 第一条消息只是任务提示。  response = llm.invoke(message\_history)  current\_code = response.content  否则:  print("\n>>> 阶段1：根据之前的评审改进代码...")  # 消息历史记录现在包含任务，  # 最后一段代码，也是最后一次批评。  # 我们指示模型应用这些批评意见。  message\_history.append(HumanMessage(content="请根据提供的评论完善代码。"))  response = llm.invoke(message\_history)  current\_code = response.content  print("\n--- 生成的代码 (v" + str(i + 1) + ") ---\n" + current\_code)  message\_history.append(response) # 将生成的代码添加到历史记录中  # --- 2. 反思阶段 ---  print("\n>>> 阶段2：反思生成的代码...")  # 为反射器代理创建一个特定的提示。  # 这要求模型充当高级代码审查员。  reflector\_prompt = [  系统消息(内容="""  你是一位资深软件工程师和专家  在Python中。  你的职责是进行细致的代码审查。  批判性地评估所提供的Python代码  基于原始任务要求。  查找漏洞、风格问题、缺失的边界情况，  以及有待改进的方面。  如果代码完美且满足所有要求，  以单个短语“CODE\_IS\_PERFECT”进行响应。  否则，请提供一个带项目符号的批评列表。  """),  HumanMessage(content=f"原始任务:\n{task\_prompt}\n\n待审查的代码:\n{current\_code}")  ]  critique\_response = llm.invoke(reflector\_prompt)  critique = critique\_response.content  # --- 3. 停止条件 ---  if "CODE\_IS\_PERFECT" in critique:  print("\n--- 评估 ---\n未发现进一步的问题。代码令人满意。")  中断  print("\n--- 评论 --- \n" + critique)  # 将评论添加到历史记录中，以供下一轮细化循环使用。  message\_history.append(HumanMessage(content=f"对之前代码的批评：\n{critique}"))  print("\n" + "="\*30 + "最终结果" + "="\*30)  print("\n反思过程后最终优化的代码：\n")  print(current\_code)  if \_\_name\_\_ == "\_\_main\_\_":  run\_reflection\_loop() |

代码首先设置环境，加载 API 密钥，并初始化一个强大的语言模型，如 GPT-4o，使用低温以实现聚焦输出。核心任务由一个提示定义，要求编写一个 Python 函数来计算一个数的阶乘，包括对文档字符串、边界情况（0 的阶乘）以及对负输入的错误处理的具体要求。run\_reflection\_loop 函数负责协调迭代优化过程。在循环中，第一次迭代时，语言模型根据任务提示生成初始代码。在后续迭代中，它根据上一步的评审意见优化代码。一个单独的“评审者”角色，同样由语言模型扮演，但使用不同的系统提示，充当高级软件工程师，根据原始任务要求对生成的代码进行评审。评审意见以问题列表的形式提供，如果没有发现问题，则使用短语“CODE\_IS\_PERFECT”。循环会一直持续，直到评审表明代码完美无缺，或者达到最大迭代次数。对话历史会被保留，并在每一步传递给语言模型，为生成/优化和评审阶段提供上下文。最后，脚本在循环结束后打印最后生成的代码版本。

**实践代码示例（ADK）**

现在让我们来看一个使用谷歌ADK实现的概念性代码示例。具体来说，该代码通过采用生成器 - 评判器结构来展示这一点，其中一个组件（生成器）产生初始结果或计划，另一个组件（评判器）提供批判性反馈或评价，引导生成器朝着更完善或准确的最终输出发展。

|  |
| --- |
| from google.adk.agents import SequentialAgent, LlmAgent  # 第一个智能体生成初始草稿。  generator = LlmAgent(  name="草稿撰写器",  description="根据给定主题生成初始草稿内容。",  instruction="撰写一段简短且信息丰富的段落，内容围绕用户的主题。",  output\_key="draft\_text" # 输出将保存到这个状态键中。  )  # 第二个智能体对第一个智能体的草稿进行批评。  reviewer = LlmAgent(  name="事实核查员",  描述="审查给定文本的事实准确性，并提供结构化的批评。",  instruction="""  你是一位严谨的事实核查员。  1. 读取状态键 'draft\_text' 中提供的文本。  2. 仔细核实所有声明的事实准确性。  3. 你的最终输出必须是一个包含两个键的字典：  - "status": 字符串，值为 "ACCURATE" 或 "INACCURATE"。  - "reasoning": 一个字符串，用于清晰解释你的状态，如果发现任何问题，则需引用具体问题。  """,  output\_key="review\_output" # 结构化字典将保存于此。  )  # SequentialAgent确保生成器在审查器之前运行。  review\_pipeline = SequentialAgent(  name="撰写与审核流程",  sub\_agents=[生成器, 审查器]  )  # 执行流程：  # 1. 生成器运行 -> 将其段落保存到 state['draft\_text'] 中。  # 2. 审核者运行 -> 读取 state['draft\_text'] 并将其字典输出保存到 state['review\_output']。 |

此代码展示了如何在Google ADK中使用顺序代理管道来生成和审核文本。它定义了两个LlmAgent实例：生成器和审核器。生成器代理旨在针对给定主题创建初始草稿段落。它被要求撰写简短且信息丰富的内容，并将其输出保存到状态键draft\_text中。审核器代理充当生成器生成文本的事实核查员。它被要求从draft\_text中读取文本并验证其事实准确性。审核器的输出是一个包含两个键的结构化字典：status和reasoning。status表示文本是“ACCURATE”还是“INACCURATE”，而reasoning则为该状态提供解释。这个字典被保存到状态键review\_output中。创建了一个名为review\_pipeline的SequentialAgent来管理两个代理的执行顺序。它确保生成器先运行，然后是审核器。整体执行流程是生成器生成文本，然后将其保存到状态中。随后，审核器从状态中读取此文本，进行事实核查，并将其调查结果（状态和推理）保存回状态中。这个管道允许使用单独的代理进行结构化的内容生产和审核过程。**注意：**对于有兴趣的人，还可以使用ADK的LoopAgent进行替代实现。

在结束之前，重要的是要考虑到，虽然反思模式显著提高了输出质量，但也伴随着重要的权衡。迭代过程虽然强大，但可能导致更高的成本和延迟，因为每个细化循环都可能需要新的大语言模型调用，这使得它对于时间敏感的应用来说并非最优选择。此外，该模式对内存的需求很大； 每一次迭代，对话历史都会扩展，包括初始输出、评论和后续的改进。

**概览**

**问题**：智能体的初始输出往往不尽如人意，存在不准确、不完整或无法满足复杂要求的问题。基本的智能体工作流程缺乏让智能体识别并修正自身错误的内置机制。解决办法是让智能体自行评估其工作，或者更可靠的做法是引入一个独立的逻辑智能体作为审查者，从而防止初始响应无论质量如何都成为最终结果。

**原因：**反思模式通过引入自我修正和完善机制提供了解决方案。它建立了一个反馈循环，其中“生产者”代理生成一个输出，然后“批评者”代理（或生产者本身）根据预定义的标准对其进行评估。然后，这种评估被用于生成一个改进版本。这种生成、评估和完善的迭代过程逐步提高了最终结果的质量，从而产生更准确、连贯和可靠的结果。

**经验法则：**当最终输出的质量、准确性和细节比速度和成本更重要时，使用反思模式。它对于生成润色后的长篇内容、编写和调试代码以及制定详细计划等任务特别有效。当任务需要高度客观性或需要专业评估（而通才生成代理可能会遗漏）时，应使用单独的评估代理。

**可视化总结**

图 1：反射设计模式，自反射

图2：反思设计模式、生产者和批判代理

**要点总结**

* 反思模式的主要优势在于其能够迭代地自我纠正和完善输出，从而显著提高质量、准确性，并更好地遵循复杂指令。
* 它涉及执行、评估/批判和改进的反馈循环。反思对于需要高质量、准确或细致输出的任务至关重要。
* 一种强大的实现方式是生产者 - 评判者模型，其中一个独立的智能体（或被赋予特定角色的实体）会对初始输出进行评估。这种关注点分离的方式增强了客观性，并允许提供更专业、结构化的反馈。
* 然而，这些优势是以增加延迟和计算成本为代价的，同时还伴随着超出模型上下文窗口或被 API 服务限流的更高风险。
* 虽然完整的迭代反思通常需要有状态的工作流（如LangGraph），但单个反思步骤可以在LangChain中使用LCEL实现，以传递输出进行批判和后续改进。
* 谷歌ADK可以通过顺序工作流程促进反思，在该流程中，一个智能体的输出会受到另一个智能体的批评，从而允许后续的细化步骤。
* 这种模式使智能体能够进行自我修正，并随着时间的推移提升其性能。

**结论**

反思模式为智能体工作流程中的自我修正提供了关键机制，使迭代改进能够超越单次执行。这是通过创建一个循环来实现的，在这个循环中，系统生成输出，根据特定标准对其进行评估，然后利用该评估结果产生优化后的结果。这种评估可以由智能体自身进行（自我反思），或者，通常更有效的做法是，由一个独立的评判智能体进行，这代表了该模式中的一个关键架构选择。

虽然一个完全自主的多步骤反思过程需要一个强大的状态管理架构，但其核心原则在单个生成-批判-改进循环中得到了有效体现。作为一种控制结构，反思可以与其他基础模式相结合，以构建更强大、功能更复杂的智能体系统。

**参考文献**

以下是一些关于反射模式及相关概念的进一步阅读资源：

1. 通过强化学习训练语言模型进行自我纠错，<https://arxiv.org/abs/2409.12917>
2. LangChain表达式语言（LCEL）文档：<https://python.langchain.com/docs/introduction/>
3. LangGraph文档：<https://www.langchain.com/langgraph>
4. Google代理开发者套件（ADK）文档（多代理系统）：<https://google.github.io/adk-docs/agents/multi-agents/>