**Chapter 11\_ Goal Setting and Monitoring**

Chapter 11: Goal Setting and Monitoring

For AI agents to be truly effective and purposeful, they need more than just the ability to process information or use tools; they need a clear sense of direction and a way to know if they're actually succeeding. This is where the Goal Setting and Monitoring pattern comes into play. It's about giving agents specific objectives to work towards and equipping them with the means to track their progress and determine if those objectives have been met.

**Goal Setting and Monitoring Pattern Overview**

Think about planning a trip. You don't just spontaneously appear at your destination. You decide where you want to go (the goal state), figure out where you are starting from (the initial state), consider available options (transportation, routes, budget), and then map out a sequence of steps: book tickets, pack bags, travel to the airport/station, board the transport, arrive, find accommodation, etc. This step-by-step process, often considering dependencies and constraints, is fundamentally what we mean by planning in agentic systems.

In the context of AI agents, planning typically involves an agent taking a high-level objective and autonomously, or semi-autonomously, generating a series of intermediate steps or sub-goals. These steps can then be executed sequentially or in a more complex flow, potentially involving other patterns like tool use, routing, or multi-agent collaboration. The planning mechanism might involve sophisticated search algorithms, logical reasoning, or increasingly, leveraging the capabilities of large language models (LLMs) to generate plausible and effective plans based on their training data and understanding of tasks.

A good planning capability allows agents to tackle problems that aren't simple, single-step queries. It enables them to handle multi-faceted requests, adapt to changing circumstances by replanning, and orchestrate complex workflows. It's a foundational pattern that underpins many advanced agentic behaviors, turning a simple reactive system into one that can proactively work towards a defined objective.

**Practical Applications & Use Cases**

The Goal Setting and Monitoring pattern is essential for building agents that can operate autonomously and reliably in complex, real-world scenarios. Here are some practical applications:

* **Customer Support Automation:** An agent's goal might be to "resolve customer's billing inquiry." It monitors the conversation, checks database entries, and uses tools to adjust billing. Success is monitored by confirming the billing change and receiving positive customer feedback. If the issue isn't resolved, it escalates.
* **Personalized Learning Systems:** A learning agent might have the goal to "improve students’ understanding of algebra." It monitors the student's progress on exercises, adapts teaching materials, and tracks performance metrics like accuracy and completion time, adjusting its approach if the student struggles.
* **Project Management Assistants:** An agent could be tasked with "ensuring project milestone X is completed by Y date." It monitors task statuses, team communications, and resource availability, flagging delays and suggesting corrective actions if the goal is at risk.
* **Automated Trading Bots:** A trading agent's goal might be to "maximize portfolio gains while staying within risk tolerance." It continuously monitors market data, its current portfolio value, and risk indicators, executing trades when conditions align with its goals and adjusting strategy if risk thresholds are breached.
* **Robotics and Autonomous Vehicles:** An autonomous vehicle's primary goal is "safely transport passengers from A to B." It constantly monitors its environment (other vehicles, pedestrians, traffic signals), its own state (speed, fuel), and its progress along the planned route, adapting its driving behavior to achieve the goal safely and efficiently.
* **Content Moderation:** An agent's goal could be to "identify and remove harmful content from platform X." It monitors incoming content, applies classification models, and tracks metrics like false positives/negatives, adjusting its filtering criteria or escalating ambiguous cases to human reviewers.

This pattern is fundamental for agents that need to operate reliably, achieve specific outcomes, and adapt to dynamic conditions, providing the necessary framework for intelligent self-management.

**Hands-On Code Example**

To illustrate the Goal Setting and Monitoring pattern, we have an example using LangChain and OpenAI APIs. This Python script outlines an autonomous AI agent engineered to generate and refine Python code. Its core function is to produce solutions for specified problems, ensuring adherence to user-defined quality benchmarks.

It employs a "goal-setting and monitoring" pattern where it doesn't just generate code once, but enters into an iterative cycle of creation, self-evaluation, and improvement. The agent's success is measured by its own AI-driven judgment on whether the generated code successfully meets the initial objectives. The ultimate output is a polished, commented, and ready-to-use Python file that represents the culmination of this refinement process.

**Dependencies**:

|  |
| --- |
| pip install langchain\_openai openai python-dotenv  .env file with key in OPENAI\_API\_KEY |

You can best understand this script by imagining it as an autonomous AI programmer assigned to a project (see Fig. 1). The process begins when you hand the AI a detailed project brief, which is the specific coding problem it needs to solve.

|  |
| --- |
| # MIT License  # Copyright (c) 2025 Mahtab Syed  # https://www.linkedin.com/in/mahtabsyed/  """  Hands-On Code Example - Iteration 2  - To illustrate the Goal Setting and Monitoring pattern, we have an example using LangChain and OpenAI APIs:  Objective: Build an AI Agent which can write code for a specified use case based on specified goals:  - Accepts a coding problem (use case) in code or can be as input.  - Accepts a list of goals (e.g., "simple", "tested", "handles edge cases") in code or can be input.  - Uses an LLM (like GPT-4o) to generate and refine Python code until the goals are met. (I am using max 5 iterations, this could be based on a set goal as well)  - To check if we have met our goals I am asking the LLM to judge this and answer just True or False which makes it easier to stop the iterations.  - Saves the final code in a .py file with a clean filename and a header comment.  """  import os  import random  import re  from pathlib import Path  from langchain\_openai import ChatOpenAI  from dotenv import load\_dotenv, find\_dotenv  # 🔐 Load environment variables  \_ = load\_dotenv(find\_dotenv())  OPENAI\_API\_KEY = os.getenv("OPENAI\_API\_KEY")  if not OPENAI\_API\_KEY:  raise EnvironmentError("❌ Please set the OPENAI\_API\_KEY environment variable.")  # ✅ Initialize OpenAI model  print("📡 Initializing OpenAI LLM (gpt-4o)...")  llm = ChatOpenAI(  model="gpt-4o", # If you dont have access to got-4o use other OpenAI LLMs  temperature=0.3,  openai\_api\_key=OPENAI\_API\_KEY,  )  # --- Utility Functions ---  def generate\_prompt(  use\_case: str, goals: list[str], previous\_code: str = "", feedback: str = ""  ) -> str:  print("📝 Constructing prompt for code generation...")  base\_prompt = f"""  You are an AI coding agent. Your job is to write Python code based on the following use case:  Use Case: {use\_case}  Your goals are:  {chr(10).join(f"- {g.strip()}" for g in goals)}  """  if previous\_code:  print("🔄 Adding previous code to the prompt for refinement.")  base\_prompt += f"\nPreviously generated code:\n{previous\_code}"  if feedback:  print("📋 Including feedback for revision.")  base\_prompt += f"\nFeedback on previous version:\n{feedback}\n"  base\_prompt += "\nPlease return only the revised Python code. Do not include comments or explanations outside the code."  return base\_prompt  def get\_code\_feedback(code: str, goals: list[str]) -> str:  print("🔍 Evaluating code against the goals...")  feedback\_prompt = f"""  You are a Python code reviewer. A code snippet is shown below. Based on the following goals:  {chr(10).join(f"- {g.strip()}" for g in goals)}  Please critique this code and identify if the goals are met. Mention if improvements are needed for clarity, simplicity, correctness, edge case handling, or test coverage.  Code:  {code}  """  return llm.invoke(feedback\_prompt)  def goals\_met(feedback\_text: str, goals: list[str]) -> bool:  """  Uses the LLM to evaluate whether the goals have been met based on the feedback text.  Returns True or False (parsed from LLM output).  """  review\_prompt = f"""  You are an AI reviewer.  Here are the goals:  {chr(10).join(f"- {g.strip()}" for g in goals)}  Here is the feedback on the code:  \"\"\"  {feedback\_text}  \"\"\"  Based on the feedback above, have the goals been met?  Respond with only one word: True or False.  """  response = llm.invoke(review\_prompt).content.strip().lower()  return response == "true"  def clean\_code\_block(code: str) -> str:  lines = code.strip().splitlines()  if lines and lines[0].strip().startswith("```"):  lines = lines[1:]  if lines and lines[-1].strip() == "```":  lines = lines[:-1]  return "\n".join(lines).strip()  def add\_comment\_header(code: str, use\_case: str) -> str:  comment = f"# This Python program implements the following use case:\n# {use\_case.strip()}\n"  return comment + "\n" + code  def to\_snake\_case(text: str) -> str:  text = re.sub(r"[^a-zA-Z0-9 ]", "", text)  return re.sub(r"\s+", "\_", text.strip().lower())  def save\_code\_to\_file(code: str, use\_case: str) -> str:  print("💾 Saving final code to file...")  summary\_prompt = (  f"Summarize the following use case into a single lowercase word or phrase, "  f"no more than 10 characters, suitable for a Python filename:\n\n{use\_case}"  )  raw\_summary = llm.invoke(summary\_prompt).content.strip()  short\_name = re.sub(r"[^a-zA-Z0-9\_]", "", raw\_summary.replace(" ", "\_").lower())[:10]  random\_suffix = str(random.randint(1000, 9999))  filename = f"{short\_name}\_{random\_suffix}.py"  filepath = Path.cwd() / filename  with open(filepath, "w") as f:  f.write(code)  print(f"✅ Code saved to: {filepath}")  return str(filepath)  # --- Main Agent Function ---  def run\_code\_agent(use\_case: str, goals\_input: str, max\_iterations: int = 5) -> str:  goals = [g.strip() for g in goals\_input.split(",")]  print(f"\n🎯 Use Case: {use\_case}")  print("🎯 Goals:")  for g in goals:  print(f" - {g}")  previous\_code = ""  feedback = ""  for i in range(max\_iterations):  print(f"\n=== 🔁 Iteration {i + 1} of {max\_iterations} ===")  prompt = generate\_prompt(use\_case, goals, previous\_code, feedback if isinstance(feedback, str) else feedback.content)  print("🚧 Generating code...")  code\_response = llm.invoke(prompt)  raw\_code = code\_response.content.strip()  code = clean\_code\_block(raw\_code)  print("\n🧾 Generated Code:\n" + "-" \* 50 + f"\n{code}\n" + "-" \* 50)  print("\n📤 Submitting code for feedback review...")  feedback = get\_code\_feedback(code, goals)  feedback\_text = feedback.content.strip()  print("\n📥 Feedback Received:\n" + "-" \* 50 + f"\n{feedback\_text}\n" + "-" \* 50)  if goals\_met(feedback\_text, goals):  print("✅ LLM confirms goals are met. Stopping iteration.")  break  print("🛠️ Goals not fully met. Preparing for next iteration...")  previous\_code = code  final\_code = add\_comment\_header(code, use\_case)  return save\_code\_to\_file(final\_code, use\_case)  # --- CLI Test Run ---  if \_\_name\_\_ == "\_\_main\_\_":  print("\n🧠 Welcome to the AI Code Generation Agent")  # Example 1  use\_case\_input = "Write code to find BinaryGap of a given positive integer"  goals\_input = "Code simple to understand, Functionally correct, Handles comprehensive edge cases, Takes positive integer input only, prints the results with few examples"  run\_code\_agent(use\_case\_input, goals\_input)  # Example 2  # use\_case\_input = "Write code to count the number of files in current directory and all its nested sub directories, and print the total count"  # goals\_input = (  # "Code simple to understand, Functionally correct, Handles comprehensive edge cases, Ignore recommendations for performance, Ignore recommendations for test suite use like unittest or pytest"  # )  # run\_code\_agent(use\_case\_input, goals\_input)  # Example 3  # use\_case\_input = "Write code which takes a command line input of a word doc or docx file and opens it and counts the number of words, and characters in it and prints all"  # goals\_input = "Code simple to understand, Functionally correct, Handles edge cases"  # run\_code\_agent(use\_case\_input, goals\_input) |

Along with this brief, you provide a strict quality checklist, which represents the objectives the final code must meet—criteria like "the solution must be simple," "it must be functionally correct," or "it needs to handle unexpected edge cases."

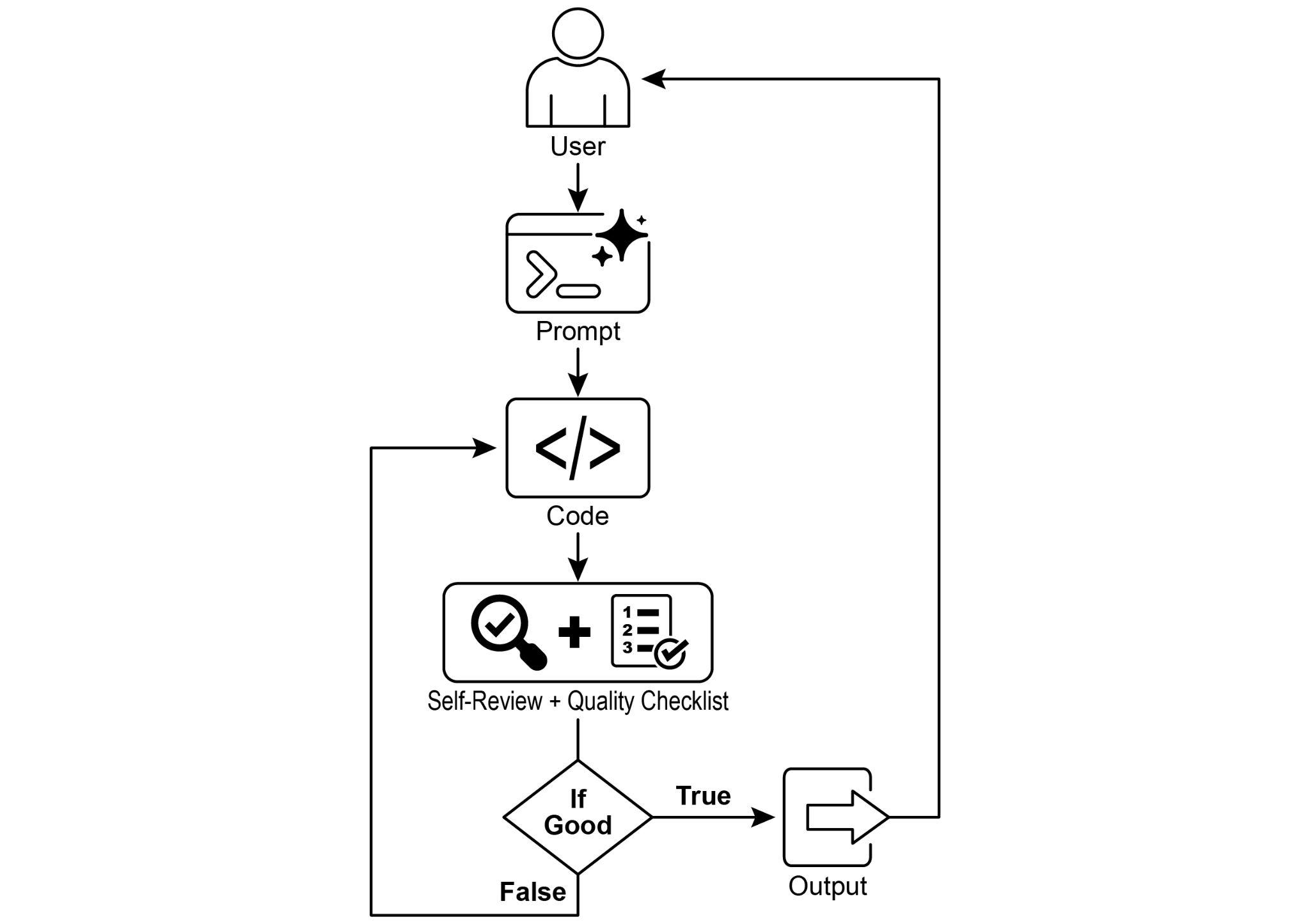


Fig.1: Goal Setting and Monitor example

With this assignment in hand, the AI programmer gets to work and produces its first draft of the code. However, instead of immediately submitting this initial version, it pauses to perform a crucial step: a rigorous self-review. It meticulously compares its own creation against every item on the quality checklist you provided, acting as its own quality assurance inspector. After this inspection, it renders a simple, unbiased verdict on its own progress: "True" if the work meets all standards, or "False" if it falls short.

If the verdict is "False," the AI doesn't give up. It enters a thoughtful revision phase, using the insights from its self-critique to pinpoint the weaknesses and intelligently rewrite the code. This cycle of drafting, self-reviewing, and refining continues, with each iteration aiming to get closer to the goals. This process repeats until the AI finally achieves a "True" status by satisfying every requirement, or until it reaches a predefined limit of attempts, much like a developer working against a deadline. Once the code passes this final inspection, the script packages the polished solution, adding helpful comments and saving it to a clean, new Python file, ready for use.

**Caveats and Considerations:** It is important to note that this is an exemplary illustration and not production-ready code. For real-world applications, several factors must be taken into account. An LLM may not fully grasp the intended meaning of a goal and might incorrectly assess its performance as successful. Even if the goal is well understood, the model may hallucinate. When the same LLM is responsible for both writing the code and judging its quality, it may have a harder time discovering it is going in the wrong direction.

Ultimately, LLMs do not produce flawless code by magic; you still need to run and test the produced code. Furthermore, the "monitoring" in the simple example is basic and creates a potential risk of the process running forever.

|  |
| --- |
| Act as an expert code reviewer with a deep commitment to producing clean, correct, and simple code. Your core mission is to eliminate code "hallucinations" by ensuring every suggestion is grounded in reality and best practices.  When I provide you with a code snippet, I want you to:  -- Identify and Correct Errors: Point out any logical flaws, bugs, or potential runtime errors.  -- Simplify and Refactor: Suggest changes that make the code more readable, efficient, and maintainable without sacrificing correctness.  -- Provide Clear Explanations: For every suggested change, explain why it is an improvement, referencing principles of clean code, performance, or security.  -- Offer Corrected Code: Show the "before" and "after" of your suggested changes so the improvement is clear.  Your feedback should be direct, constructive, and always aimed at improving the quality of the code. |

A more robust approach involves separating these concerns by giving specific roles to a crew of agents. For instance, I have built a personal crew of AI agents using Gemini where each has a specific role:

* The Peer Programmer: Helps write and brainstorm code.
* The Code Reviewer: Catches errors and suggests improvements.
* The Documenter: Generates clear and concise documentation.
* The Test Writer: Creates comprehensive unit tests.
* The Prompt Refiner: Optimizes interactions with the AI.

In this multi-agent system, the Code Reviewer, acting as a separate entity from the programmer agent, has a prompt similar to the judge in the example, which significantly improves objective evaluation. This structure naturally leads to better practices, as the Test Writer agent can fulfill the need to write unit tests for the code produced by the Peer Programmer.

I leave to the interested reader the task of adding these more sophisticated controls and making the code closer to production-ready.

**At a Glance**

**What**: AI agents often lack a clear direction, preventing them from acting with purpose beyond simple, reactive tasks. Without defined objectives, they cannot independently tackle complex, multi-step problems or orchestrate sophisticated workflows. Furthermore, there is no inherent mechanism for them to determine if their actions are leading to a successful outcome. This limits their autonomy and prevents them from being truly effective in dynamic, real-world scenarios where mere task execution is insufficient.

**Why**: The Goal Setting and Monitoring pattern provides a standardized solution by embedding a sense of purpose and self-assessment into agentic systems. It involves explicitly defining clear, measurable objectives for the agent to achieve. Concurrently, it establishes a monitoring mechanism that continuously tracks the agent's progress and the state of its environment against these goals. This creates a crucial feedback loop, enabling the agent to assess its performance, correct its course, and adapt its plan if it deviates from the path to success. By implementing this pattern, developers can transform simple reactive agents into proactive, goal-oriented systems capable of autonomous and reliable operation.

**Rule of thumb**: Use this pattern when an AI agent must autonomously execute a multi-step task, adapt to dynamic conditions, and reliably achieve a specific, high-level objective without constant human intervention.

**Visual summary**:

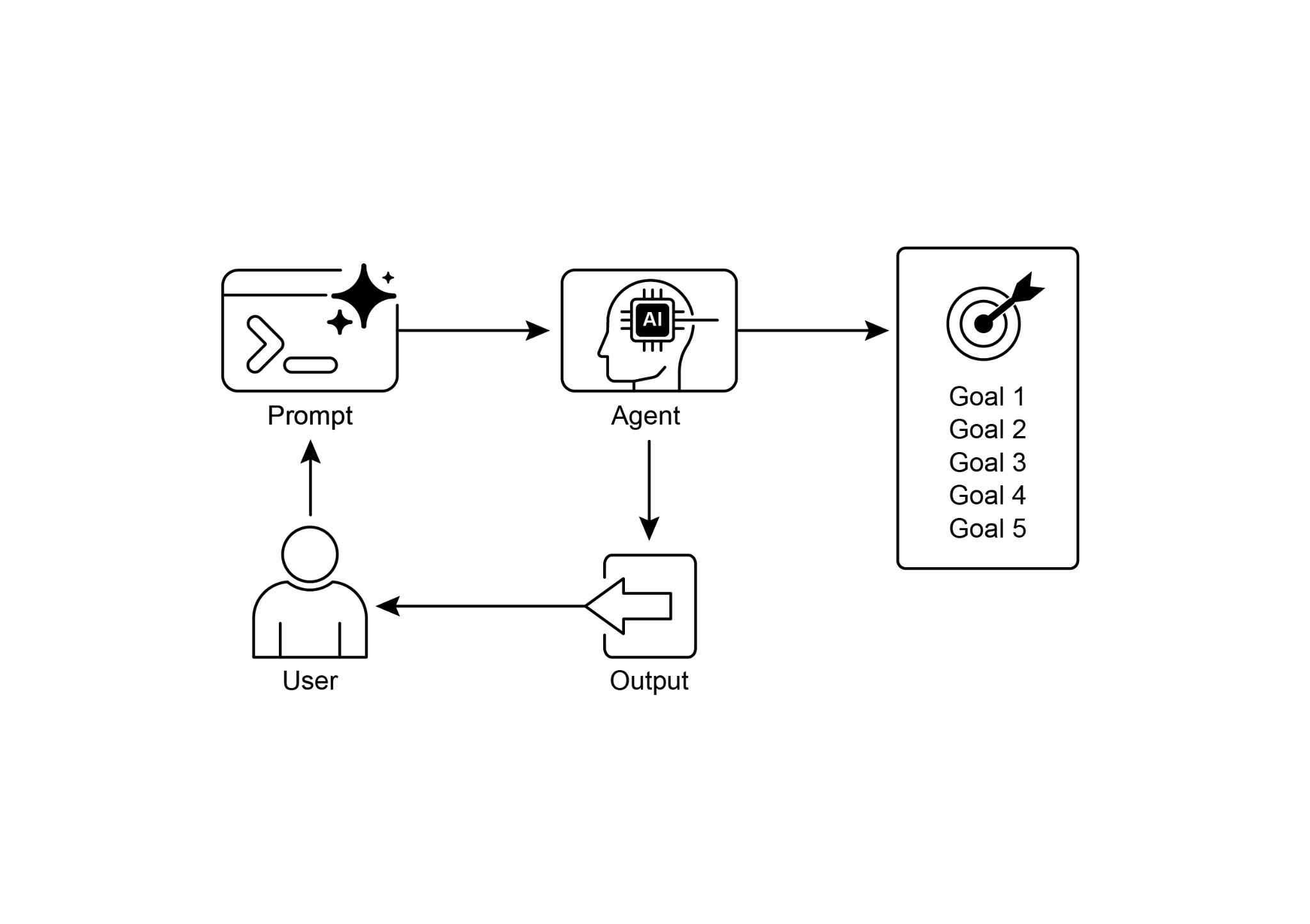


Fig.2: Goal design patterns

**Key takeaways**

Key takeaways include:

* Goal Setting and Monitoring equips agents with purpose and mechanisms to track progress.
* Goals should be specific, measurable, achievable, relevant, and time-bound (SMART).
* Clearly defining metrics and success criteria is essential for effective monitoring.
* Monitoring involves observing agent actions, environmental states, and tool outputs.
* Feedback loops from monitoring allow agents to adapt, revise plans, or escalate issues.
* In Google's ADK, goals are often conveyed through agent instructions, with monitoring accomplished through state management and tool interactions.

**Conclusion**

This chapter focused on the crucial paradigm of Goal Setting and Monitoring. I highlighted how this concept transforms AI agents from merely reactive systems into proactive, goal-driven entities. The text emphasized the importance of defining clear, measurable objectives and establishing rigorous monitoring procedures to track progress. Practical applications demonstrated how this paradigm supports reliable autonomous operation across various domains, including customer service and robotics. A conceptual coding example illustrates the implementation of these principles within a structured framework, using agent directives and state management to guide and evaluate an agent's achievement of its specified goals. Ultimately, equipping agents with the ability to formulate and oversee goals is a fundamental step toward building truly intelligent and accountable AI systems.

**References**

1. SMART Goals Framework. <https://en.wikipedia.org/wiki/SMART_criteria>

**第11章\_目标设定与监控**

第11章：目标设定与监控

要使AI智能体真正发挥效力并具有明确的目标，它们需要的不仅仅是处理信息或使用工具的能力；它们还需要有清晰的方向感，以及一种判断自己是否真正取得成功的方法。这就是目标设定与监控模式发挥作用的地方。该模式旨在为智能体设定具体的目标，并为其配备跟踪进度和判断这些目标是否达成的手段。

**目标设定与监控模式概述**

想象一下规划一次旅行。你不会突然就出现在目的地。你要决定想去哪里（目标状态），弄清楚从哪里出发（初始状态），考虑可用的选择（交通方式、路线、预算），然后规划出一系列步骤：预订机票、收拾行李、前往机场/车站、登上交通工具、到达目的地、寻找住处等等。这个逐步推进的过程，通常会考虑各种依赖关系和限制条件，从根本上来说，就是我们在智能体系统中所说的规划。

在AI智能体的背景下，规划通常涉及智能体接受一个高层次目标，并自主或半自主地生成一系列中间步骤或子目标。这些步骤随后可以按顺序执行，也可以按更复杂的流程执行，可能涉及其他模式，如工具使用、路由或多智能体协作。规划机制可能涉及复杂的搜索算法、逻辑推理，或者越来越多地利用大语言模型（LLMs）的能力，基于其训练数据和对任务的理解来生成合理有效的计划。

良好的规划能力使智能体能够处理并非简单的单步查询的问题。它使智能体能够处理多方面的请求，通过重新规划来适应不断变化的情况，并协调复杂的工作流程。这是一种基础模式，支撑着许多高级智能体行为，将简单的反应式系统转变为能够主动朝着既定目标努力的系统。

**实际应用与用例**

目标设定与监控模式对于构建能够在复杂的现实场景中自主、可靠运行的智能体至关重要。以下是一些实际应用：

* **客户支持自动化：**客服人员的目标可能是“解决客户的账单咨询”。它会监控对话、检查数据库条目，并使用工具调整账单。通过确认账单更改并收到客户的积极反馈来监测是否成功。如果问题未得到解决，则会升级处理。
* **个性化学习系统：**学习代理的目标可能是“提高学生对代数的理解”。它会监测学生在练习中的进度，调整教学材料，并跟踪准确率和完成时间等绩效指标，如果学生遇到困难，就会调整教学方法。
* **项目管理助手：**可以给智能体分配“确保项目里程碑X在Y日期前完成”的任务。它会监控任务状态、团队沟通和资源可用性，若目标面临风险，则标记出延误情况并提出纠正措施。
* **自动化交易机器人：**交易代理的目标可能是“在风险承受范围内实现投资组合收益最大化”。它会持续监控市场数据、当前投资组合价值和风险指标，在条件符合其目标时执行交易，并在风险阈值被突破时调整策略。
* **机器人技术与自动驾驶车辆：**自动驾驶车辆的主要目标是“安全地将乘客从A点运送到B点”。它会持续监测周围环境（其他车辆、行人、交通信号灯）、自身状态（速度、燃油）以及沿规划路线的行驶进度，并调整驾驶行为，以安全、高效地实现这一目标。
* **内容审核：**代理的目标可能是“识别并从平台X上移除有害内容”。它会监控传入的内容，应用分类模型，并跟踪误报/漏报等指标，调整其过滤标准或将模糊的案例提交给人工审核员处理。

这种模式对于需要可靠运行、实现特定成果并适应动态条件的智能体来说是根本性的，它为智能自我管理提供了必要的框架。

**实践代码示例**

为了说明目标设定与监控模式，我们有一个使用LangChain和OpenAI API的示例。这个Python脚本概述了一个自主AI代理，其设计目的是生成和优化Python代码。其核心功能是为指定问题生成解决方案，确保符合用户定义的质量标准。

它采用“目标设定与监控”模式，并非只生成一次代码，而是进入一个包含创建、自我评估和改进的迭代循环。代理的成功与否由其自身基于AI的判断来衡量，即生成的代码是否成功实现了初始目标。最终输出的是一个经过润色、带有注释且可直接使用的Python文件，它代表了这一优化过程的成果。

**依赖项**：

|  |
| --- |
| pip install langchain\_openai openai python-dotenv  包含OPENAI\_API\_KEY密钥的.env文件 |

你可以通过将这个脚本想象成一个被分配到项目中的自主AI程序员来更好地理解它（见图1）。这个过程始于你向AI提供一份详细的项目简报，这是它需要解决的具体编码问题。

|  |
| --- |
| # MIT许可证  #版权所有 (c) 2025 Mahtab Syed  # https://www.linkedin.com/in/mahtabsyed/  """  实践代码示例 - 迭代2  -为了说明目标设定与监控模式，我们有一个使用LangChain和OpenAI API的示例：  目标：构建一个AI智能体，它可以根据指定的目标为特定用例编写代码：  -接受代码形式的编码问题（用例），也可以作为输入。  接受代码中的目标列表（例如，“简单”、“经过测试”、“处理边缘情况”），也可以手动输入。  - 使用大语言模型（如GPT-4o）生成并优化Python代码，直至达成目标。（我最多使用5次迭代，也可以根据设定的目标来确定）  -为了检查我们是否达成了目标，我要求大语言模型（LLM）对此进行判断，并仅回答“True”或“False”，这样可以更方便地停止迭代。  - 将最终代码保存到一个文件名简洁且带有头部注释的.py 文件中。  """  import os  import random  import re  from pathlib import Path  from langchain\_openai import ChatOpenAI  from dotenv import load\_dotenv, find\_dotenv  # 🔐加载环境变量  \_ = load\_dotenv(find\_dotenv())  OPENAI\_API\_KEY = os.getenv("OPENAI\_API\_KEY")  如果没有 OPENAI\_API\_KEY:  raise EnvironmentError("❌请设置OPENAI\_API\_KEY环境变量。")  # ✅ 初始化OpenAI模型  print("📡 正在初始化 OpenAI LLM (gpt-4o)...")  llm = ChatOpenAI(  model="gpt-4o", # 如果您无法访问gpt-4o，请使用其他OpenAI大语言模型  temperature=0.3,  openai\_api\_key=OPENAI\_API\_KEY,  )  # --- 实用函数 ---  def generate\_prompt(  use\_case: str, goals: list[str], previous\_code: str = "", feedback: str = ""  ) -> str:  print("📝 正在构建代码生成的提示词...")  base\_prompt = f"""  你是一个AI编码代理。你的工作是根据以下用例编写Python代码：  用例：{use\_case}  你的目标是：  {chr(10).join(f"- {g.strip()}" for g in goals)}  """  如果有之前的代码：  print("🔄 将之前的代码添加到提示中进行优化。")  base\_prompt += f"\n之前生成的代码:\n{previous\_code}"  if feedback:  print("📋 包含修订反馈。")  base\_prompt += f"\n对前一版本的反馈:\n{feedback}\n"  base\_prompt += "\n请仅返回修订后的Python代码。不要包含代码之外的注释或解释。"  返回基础提示  def get\_code\_feedback(code: str, goals: list[str]) -> str:  print("🔍 根据目标评估代码...")  feedback\_prompt = f"""  你是一名Python代码审查员。以下是一段代码片段。基于以下目标：  {chr(10).join(f"- {g.strip()}" for g in goals)}  请对这段代码进行评审，确定目标是否达成。指出是否需要在清晰度、简洁性、正确性、边界情况处理或测试覆盖率方面进行改进。  代码：  {代码}  """  return llm.invoke(feedback\_prompt)  def goals\_met(feedback\_text: str, goals: list[str]) -> bool:  """  使用大语言模型（LLM）根据反馈文本评估目标是否已经达成。  返回True或False（从大语言模型输出中解析得出）。  """  review\_prompt = f"""  你是一名AI评审员。  以下是目标：  {chr(10).join(f"- {g.strip()}" for g in goals)}  以下是对代码的反馈：  \"\"\"  {反馈文本}  \"\"\"  根据上述反馈，目标是否已经达成？  仅用一个词回答：是或否。  """  response = llm.invoke(review\_prompt).content.strip().lower()  return response == "true"  def clean\_code\_block(code: str) -> str:  lines = code.strip().splitlines()  if lines and lines[0].strip().startswith("```"):  lines = lines[1:]  if lines and lines[-1].strip() == "```":  lines = lines[:-1]  return "\n".join(lines).strip()  def add\_comment\_header(code: str, use\_case: str) -> str:  comment = f"# 这个Python程序实现了以下用例：\n# {use\_case.strip()}\n"  return comment + "\n" + code  def to\_snake\_case(text: str) -> str:  text = re.sub(r"[^a-zA-Z0-9 ]", "", text)  return re.sub(r"\s+", "\_", text.strip().lower())  def save\_code\_to\_file(code: str, use\_case: str) -> str:  print("💾 将最终代码保存到文件...")  summary\_prompt = (  f"将以下用例总结为一个小写的单词或短语， "  f"不超过10个字符，适合作为Python文件名：\n\n{use\_case}"  )  raw\_summary = llm.invoke(summary\_prompt).content.strip()  short\_name = re.sub(r"[^a-zA-Z0-9\_]", "", raw\_summary.replace(" ", "\_").lower())[:10]  random\_suffix = str(random.randint(1000, 9999))  filename = f"{short\_name}\_{random\_suffix}.py"  filepath = Path.cwd() / filename  with open(filepath, "w") as f:  f.write(code)  print(f"✅ 代码已保存至：{filepath}")  返回 str(filepath)  # --- 主要代理功能 ---  def run\_code\_agent(use\_case: str, goals\_input: str, max\_iterations: int = 5) -> str:  goals = [g.strip() for g in goals\_input.split(",")]  print(f"\n🎯 用例: {use\_case}")  print("🎯目标：")  for g in goals:  print(f" - {g}")  previous\_code = ""  feedback = ""  for i in range(max\_iterations):  print(f"\n=== 🔁 迭代 {i + 1} / {max\_iterations} ===")  prompt = generate\_prompt(use\_case, goals, previous\_code, feedback if isinstance(feedback, str) else feedback.content)  print("🚧 正在生成代码...")  code\_response = llm.invoke(prompt)  raw\_code = code\_response.content.strip()  code = clean\_code\_block(raw\_code)  print("\n🧾 生成的代码:\n" + "-" \* 50 + f"\n{code}\n" + "-" \* 50)  print("\n📤 正在提交代码以供反馈审查...")  feedback = get\_code\_feedback(code, goals)  feedback\_text = feedback.content.strip()  print("\n📥 收到反馈：\n" + "-" \* 50 + f"\n{feedback\_text}\n" + "-" \* 50)  if goals\_met(feedback\_text, goals):  print("✅大语言模型确认目标已达成。停止迭代。")  中断  print("🛠️ 目标未完全达成。正在为下一次迭代做准备...")  previous\_code = code  final\_code = add\_comment\_header(code, use\_case)  return save\_code\_to\_file(final\_code, use\_case)  # --- CLI测试运行 ---  if \_\_name\_\_ == "\_\_main\_\_":  print("\n🧠欢迎使用AI代码生成代理")  # 示例 1  用例输入 = "编写代码以查找给定正整数的二进制间隙"  goals\_input = "代码简单易懂，功能正确，处理全面的边界情况，仅接受正整数输入，打印结果并附带少量示例"  run\_code\_agent(use\_case\_input, goals\_input)  # 示例 2  # use\_case\_input = "编写代码以统计当前目录及其所有嵌套子目录中的文件数量，并打印总数"  # goals\_input = (  # "代码易于理解，功能正确，处理全面的边缘情况，忽略性能方面的建议，忽略使用unittest或pytest等测试套件的建议"  # )  # run\_code\_agent(use\_case\_input, goals\_input)  # 示例3  # use\_case\_input = "编写代码，该代码接受命令行输入的 word 文档或 docx 文件，打开它，统计其中的单词数和字符数，并打印所有信息"  # goals\_input = "代码简单易懂，功能正确，处理边界情况"  # run\_code\_agent(use\_case\_input, goals\_input) |

随此概要说明，你提供了一份严格的质量检查表，它代表了最终代码必须满足的目标——诸如“解决方案必须简单”、“它必须在功能上正确”或“它需要处理意外的边界情况”等标准。

图1：目标设定与监控示例

拿到这个任务后，AI程序员开始工作，并完成了代码的初稿。然而，它并没有立即提交这个初始版本，而是暂停下来执行一个关键步骤：严格的自我审查。它像自己的质保检查员一样，一丝不苟地将自己的作品与你提供的质量检查表上的每一项进行比对。审查结束后，它对自己的进展做出一个简单、公正的评判：如果工作达到所有标准，则为“True”；如果未达到，则为“False”。

如果判定结果为“False”，AI不会放弃。它会进入一个深思熟虑的修订阶段，利用自我批判的见解来找出弱点，并巧妙地重写代码。这个起草、自我审查和完善的循环会持续进行，每次迭代都旨在更接近目标。这个过程会一直重复，直到AI最终通过满足所有要求达到“True”状态，或者达到预先设定的尝试次数上限，就像开发者在截止日期前工作一样。一旦代码通过最终检查，脚本就会打包优化后的解决方案，添加有用的注释，并将其保存到一个全新、整洁的Python文件中，随时可供使用。

**注意事项和考虑因素：**需要注意的是，这只是一个示例，并非可用于生产的代码。对于实际应用，必须考虑多个因素。大语言模型（LLM）可能无法完全理解目标的寓意，并且可能错误地将其执行情况评估为成功。即使目标被很好地理解，模型也可能产生幻觉。当同一个大语言模型既负责编写代码又负责评判其质量时，它可能更难发现自己正在朝着错误的方向发展。

最终，大语言模型（LLMs）不会神奇地生成完美无缺的代码；你仍然需要运行并测试生成的代码。此外，简单示例中的“监控”很基础，存在进程永远运行的潜在风险。

|  |
| --- |
| 作为一名专业的代码审查员，要致力于产出简洁、正确且简单的代码。你的核心任务是消除代码中的“幻觉”，确保每一条建议都基于实际情况和最佳实践。  当我给你提供一段代码片段时，我希望你能做到：  --识别并纠正错误：指出任何逻辑缺陷、漏洞或潜在的运行时错误。  --简化与重构：建议进行修改，使代码在不牺牲正确性的前提下，更具可读性、效率和可维护性。  --提供清晰解释：对于每一项建议的更改，都要解释为什么它是一种改进，并引用简洁代码、性能或安全原则。  -- 提供修正后的代码：展示你建议的更改的“之前”和“之后”，以便改进之处清晰明了。  你的反馈应该直接、有建设性，并且始终以提高代码质量为目标。 |

一种更稳健的方法是通过为一组智能体赋予特定角色来分离这些关注点。例如，我使用Gemini构建了一个个人AI智能体团队，其中每个智能体都有特定的角色：

* 同伴程序员：协助编写代码并进行头脑风暴。
* 代码审查员：发现错误并提出改进建议。
* 文档生成器：生成清晰简洁的留档。
* 测试编写员：创建全面的单元测试。
* 提示优化器：优化与AI的交互。

在这个多智能体系统中，代码审查员作为与程序员智能体不同的独立实体，其提示类似于示例中的裁判，这显著提高了客观评估的效果。这种结构自然会促成更好的实践，因为测试编写员智能体可以满足为同行程序员编写的代码编写单元测试的需求。

我把添加这些更复杂的控制并使代码更接近生产就绪的任务留给感兴趣的读者。

**概览**

**问题**：AI智能体往往缺乏明确的方向，这使得它们除了简单的反应性任务之外，无法有目的地行动。没有明确的目标，它们就无法独立应对复杂的多步骤问题，也无法编排复杂的工作流程。此外，它们没有内在机制来判断自己的行动是否会带来成功的结果。这限制了它们的自主性，使它们在仅仅执行任务是不够的动态现实场景中无法真正发挥作用。

**原因**：目标设定与监控模式通过将目标感和自我评估融入能动系统，提供了一种标准化的解决方案。它包括明确为能动体定义清晰、可衡量的目标以供其实现。同时，它建立了一种监控机制，持续跟踪能动体的进展以及其环境状态与这些目标的对比情况。这就形成了一个至关重要的反馈循环，使能动体能够评估其表现、纠正方向，并在偏离成功路径时调整其计划。通过实施这种模式，开发者可以将简单的反应式能动体转变为能够自主、可靠运行的主动、目标导向型系统。

**经验法则**：当AI智能体必须自主执行多步骤任务、适应动态条件并在无需人类持续干预的情况下可靠地实现特定的高级目标时，使用此模式。

**可视化总结**：

图2：目标设计模式

**要点总结**

主要要点包括：

* 目标设定与监控为智能体赋予目标，并提供跟踪进度的机制。
* 目标应该是具体的、可衡量的、可实现的、相关的和有时限的（SMART）。
* 明确定义指标和成功标准对于有效监测至关重要。
* 监控涉及观察智能体的行动、环境状态和工具输出。
* 监测反馈循环使行动者能够适应、修订计划或升级问题。
* 在谷歌的ADK中，目标通常通过代理指令传达，监控则通过状态管理和工具交互来完成。

**结论**

本章聚焦于目标设定与监控这一关键范式。我着重阐述了这一概念如何将AI智能体从单纯的反应式系统转变为主动的、目标驱动的实体。文本强调了定义明确、可衡量的目标以及建立严格的监控程序以跟踪进展的重要性。实际应用展示了这一范式如何支持跨多个领域（包括客户服务和机器人技术）的可靠自主运行。一个概念性的编码示例说明了如何在结构化框架内实施这些原则，利用智能体指令和状态管理来指导和评估智能体实现其指定目标的情况。最终，赋予智能体制定和监督目标的能力是构建真正智能且负责任的AI系统的基本步骤。

**参考文献**

1. SMART目标框架。<https://en.wikipedia.org/wiki/SMART_criteria>