Agentic Design Patterns You Must Know in 2025

Thumbnail

Designing a well-performing AI agent can be a tricky task, especially because the Agentic AI field is relatively new and agent architectures are constantly evolving which make it hard for non experts to create production level AI Agents.

During the last year, I have built multiple AI agents and tried numerous frameworks for that purpose. And based on the knowledge and experience I gathered, I wanted to share with you the most important design patterns that will help you build useful AI agents.

TL;DR

Designing production-ready AI agents is hard because the field is new and evolving fast. A strong agent must be robust, efficient, autonomous, and usable. To achieve this, AI engineers rely on design patterns, reusable architectural strategies that solve common challenges and speed up development.

The key agentic design patterns you should know are:

- **Prompt Chaining:** Break big tasks into smaller steps for reliability and control.

- **Routing & Parallelization:** Dynamically choose paths or run tasks simultaneously to boost adaptability and speed.
- **Reflection:** Let agents critique and refine their own outputs for higher quality.
- **Tool Use:** Extend agents beyond static knowledge by calling external APIs, databases, or functions.
- **Planning:** Decompose complex goals into structured subtasks with flexible orchestration.
- **Multi-Agent Collaboration:** Coordinate specialized agents to solve complex, multi-domain problems.

Mastering these patterns gives you a **practical toolkit** for building advanced, real-world AI agents.

What makes a good AI agent?

There are many components that comes into play when creating or designing an AI agent, from choosing the suitable LLMs to defining its core capabilities like planning, tool usage, and memory, all within a robust safety framework.

However, no matter how you design an Agent, it should meet several key criteria to ensure it performs effectively in its intended environment. These criteria are well-known in other area of development, and they still hold for AI agents, forming the foundation of a trustworthy and useful system. Below are 4 primary criteria for a well-designed agent:

Good Quality Criteria

1. **Robustness:**
   A robust agent functions reliably in unpredictable environments, such as network failures or incomplete data. It handles errors, unexpected inputs, and adverse conditions without failing or producing incorrect results.
   For instance, a chatbot with agentic capabilities should be able to manage ambiguous queries or timeouts.
   Here are few steps you can take to assure agent robustness:
   - Establish strict input validation and robust error-handling routines.
   - Integrate redundancy and fallback mechanisms to maintain functionality.
   - Perform extensive testing across diverse conditions, including edge cases.

2. **Efficiency:**
   Efficiency describes the ability to perform tasks using minimal computational resources (e.g., CPU, memory, energy) and time while maintaining high performance.
   It is one of the most important criteria for a production-level agent, as it ensures high performance while enabling reduced operational costs, improved responsiveness, and better scalability in resource-constrained environments (e.g. Edge devices). Efficiency can be ensured through several methods, such as:
   - Optimized algorithms and data structures.
   - Caching frequently used data.
   - Minimizing redundant computations or network calls.

3. **Autonomy:**
   This criteria is critical for building agentic systems because it enables them to operate independently and make real-time decisions without constant human oversight.
   This is essential for practical applications such as a advanced smart home agent that autonomously manages lighting and temperature based on user habits. The foundation for agent autonomy is a design that incorporates advanced techniques, including:
   - Well-defined decision-making algorithms or AI models.
   - Sensors or data inputs for situational awareness.
   - Clear and robust multi-agent collaboration.

4. **Usability:**
   Ultimately, a good agent needs to be usable, meaning it should feel intuitive and easy to work with, whether you're a person or another system.
   This is crucial because a highly usable agent minimizes the learning curve and

enhances user satisfaction, which is particularly important for human-facing applications like chatbots. Usability is achieved through several key practices:
- Designing intuitive interfaces (UIs or/and APIs).
- Implementing clear feedback mechanisms, such as helpful error messages and confirmations, to keep users informed.
- Conducting regular user testing to iteratively refine and improve the interaction flows.

What are design patterns & Why do we need them?

In the context of AI Engineering, effective agent development requires more than assembling components, it demands an understanding of proven design patterns. These patterns provide reusable solutions to the common challenges of building intelligent agents.

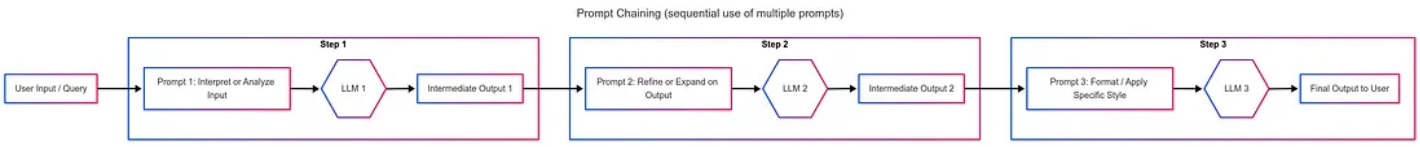
While we can design agents without patterns, they are indispensable for efficient development. They boost productivity by preventing us from reinventing the wheel. Here's few reasons why they're essential:

1. **Standardizes Solutions**: They ensure consistency across projects, which is especially valuable in large teams or long-term projects.
2. **Saves Time**: Instead of reinventing solutions for recurring issues (e.g., managing object creation or communication), patterns offer tested templates, speeding up development.
3. **Improves Code Quality:** They promote best practices, leading to cleaner, more maintainable, and scalable code.
4. **Enhances Collaboration:** Patterns create a shared vocabulary (e.g. "Retrieval Augmented Generation", "Human-In-The-Loop"), making it easier for developers to communicate and understand designs.

Design patterns you must know as and AI Engineer

There are several design patterns that are proven to be useful. Below are the ones that I used in my work and that I think are essential for an AI engineer to master.

Prompt Chaining Pattern



Prompt chaining pattern flowchart

Don't force your LLM to handle everything in one massive prompt! This all-at-once approach causes instructions to be ignored, context to shift, and errors to spread.

The solution is the Prompt Chaining Pattern. Whether you're creating an agentic Information ingestion pipeline, or and advanced multi-modal reasoning agent, prompt chaining is what you need to use to make the agent deal successfully with complex tasks.

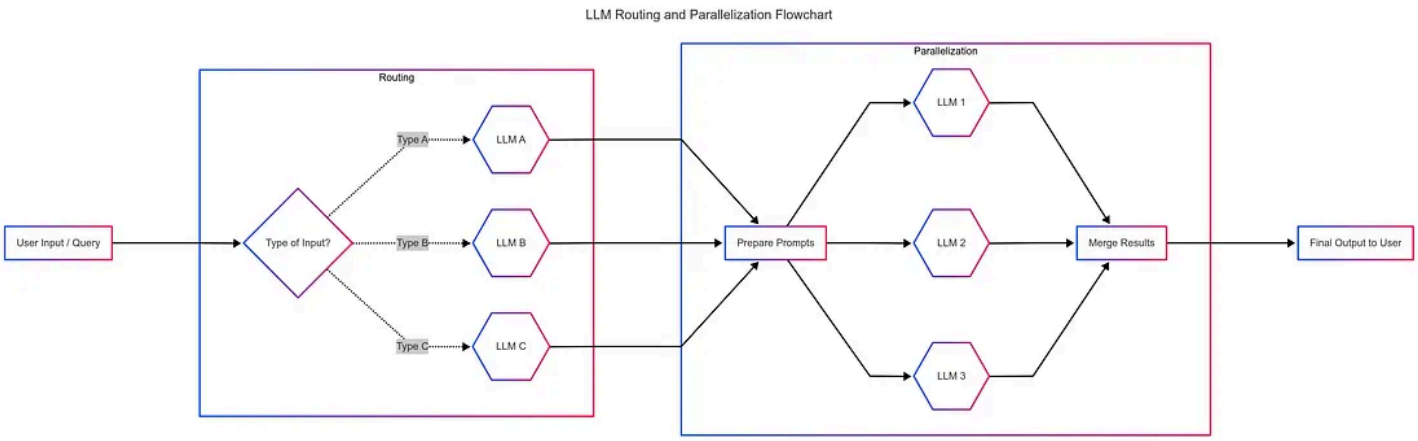
It works by using a divide-and-conquer strategy: Breaking a complex task into a sequence of smaller, manageable sub-problems, where the output of one prompt becomes the input for the next, establishing a reliable workflow. This method also allows us to use multiple specialized LLMs for each sub-task, and that would drastically improve the final result.

Benefits of using Chain Prompts:

- **Granular Control:** Each step is simpler, allowing you to assign a specific "role" (e.g., *Analyst*, *Drafting Expert*) for higher accuracy.
- **Enhanced Reliability:** Sequential decomposition prevents errors from compounding.
- **Scalability:** It's the foundational technique for building sophisticated **AI agents** capable of multi-step reasoning and integrating external tools.

Example: Instead of one failed prompt for "summarize a report, find data, and draft an email," you use a chain: Summarize => Extract Data => Compose Email. The final result is more accurate and easier to debug.

Routing and Parallelization Patterns



Routing & Parallelization patterns flowchart

Routing

While the most basic agent you could design would definitely follow a sequential pattern, we all know, building any effective solution for a real-world problem requires complexity and the capacity for dynamic adaptation. That's why **routing** is so essential. Routing introduces the conditional logic that shifts your agent from a predetermined path to one where it dynamically evaluates specific criteria (like the environment state or user input) to select from a set of potential subsequent actions. Think of it as the agent's internal traffic controller.

For instance, a sophisticated customer service agent might:

- Analyze the user's initial query.
- Route it based on intent:
- If "check order status," route to a database retrieval tool.
- If "technical support," route to a specialized troubleshooting agent.
- If "unclear," route to a clarification sub-agent.

This dynamic decision-making can be implemented using methods such as:

- **LLM-based Routing**: The language model itself determines the next step by outputting a category identifier.
- **Embedding-based Routing**: The query's vector embedding is compared to those representing different functional routes, selecting the most semantically similar match.

- **Rule-based Routing**: Uses deterministic if-else logic based on extracted keywords or patterns.

Parallelization

Complementing routing is parallelization, a technique focused on efficiency by executing multiple independent components concurrently rather than sequentially. Instead of waiting for one step to finish before the next one starts, parallel execution allows independent tasks to run *at the same time*.

Example of a research agent: With a sequential approach, it would first search for Source A, then summarize it. Only after that's done would it search for Source B, and then summarize that.

A parallel approach would:

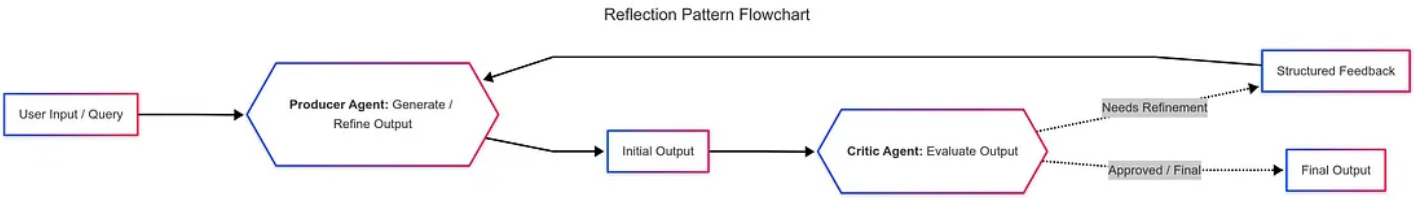
1. Search for Source A and Source B simultaneously.
2. Once both searches complete, summarize Source A and Source B simultaneously.

This pattern is vital for improving responsiveness and significantly reducing overall execution time, especially in tasks involving multiple lookups or interactions with external services (like APIs or databases) that introduce latency.

Key Use Cases for Parallelization:

- **Comprehensive Information Gathering:** A travel agent can simultaneously query different external services to secure flight prices, check hotel room availability, and research local activities, delivering a complete itinerary much faster.
- **Accelerated Content Assembly:** For a marketing agent composing an email, it can generate the subject line, draft the main body text, and select the accompanying image concurrently, streamlining the content creation pipeline.
- **Rapid Input Validation:** An agent validating user submission can run checks for correct email formatting, phone number validity, and address verification against a database all at once, providing immediate, comprehensive feedback.

Reflection Pattern



Reflection design pattern flowchart

We've explored fundamental agentic patterns for execution and efficiency so far, but what happens when an agent's first attempt is flawed? This is where the Reflection pattern takes over.

Reflection is the agent's ability to critique its own work and use that critique for iterative improvement. Having used this pattern across many projects, I've come to believe it is one of the most important methods available for dramatically boosting agent quality and results. This design pattern establishes a vital self-correction loop:

1. An initial LLM generates an output.
2. A second step, the reflector or evaluator, critiques this output against requirements.
3. This feedback is then used by the generator to produce a refined output.

The most effective implementation is the Producer-Critic model:

- **Producer Agent:** Focuses only on generating the initial content.
- **Critic Agent:** Given a separate, distinct persona (e.g., "meticulous fact-checker"), its sole job is to ruthlessly evaluate the Producer's work and suggest structured improvements.

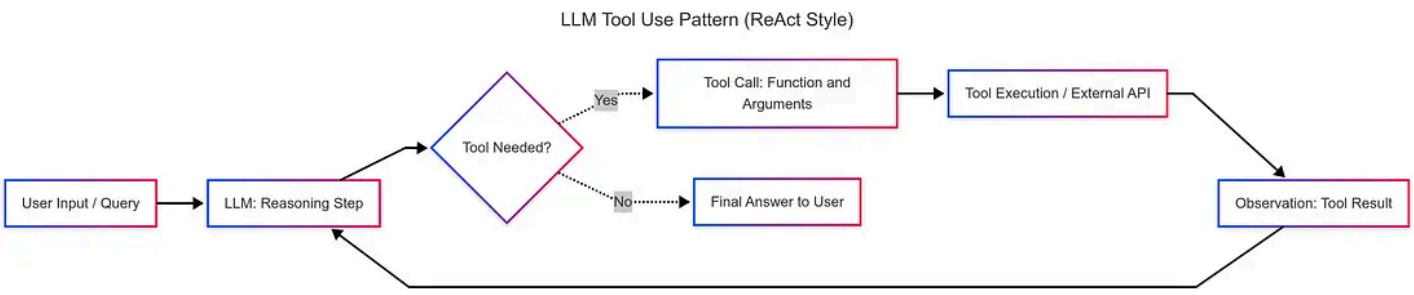
This separation prevents cognitive bias, ensuring the agent is truly self-aware and adaptive.
When combined with the agent's memory, reflection allows the system to learn from past critiques and consistently produce high-quality results.

Below are few reflection-based scenarios where the final output quality, accuracy, and adherence to complex instructions are critical:

- **Refining Creative Content:** An agent writing a marketing email can generate a draft, critique it for flow, tone, and clarity, and then rewrite it based on the critique, ensuring the final message is polished and effective.
- **Robust Code Generation:** Agents can write initial code, run static analysis or simulated tests, identify errors or inefficiencies, and then modify the code based on the findings — leading to more functional and reliable software.
- **Accurate Information Synthesis:** When summarizing a lengthy report, the agent can generate a draft, compare it against the original document's key points, and refine the summary to ensure it is comprehensive and factually precise.

When combined with the agent's memory, reflection allows the system to learn from past critiques and consistently produce high-quality results.

Tool Use Pattern



Tool Use design pattern flowchart

The Tool Use pattern, often implemented using Function Calling, is fundamental because it breaks the limits of an LLM's fixed training data, allowing it to interact with the outside world.

It enables an agent to invoke external APIs, databases, or even delegate tasks to other specialized agents. The process is a critical reasoning loop:

1. **Tool Definition:** External capabilities are described to the LLM.
2. **LLM Decision:** The LLM receives the user's request and generates a structured output (like JSON) that specifies the Tool to call and the precise arguments.
3. **Execution & Result:** The agentic framework runs the tool and returns the output to the LLM.

4. **Final Response:** The LLM uses this new information as context to formulate the final answer.

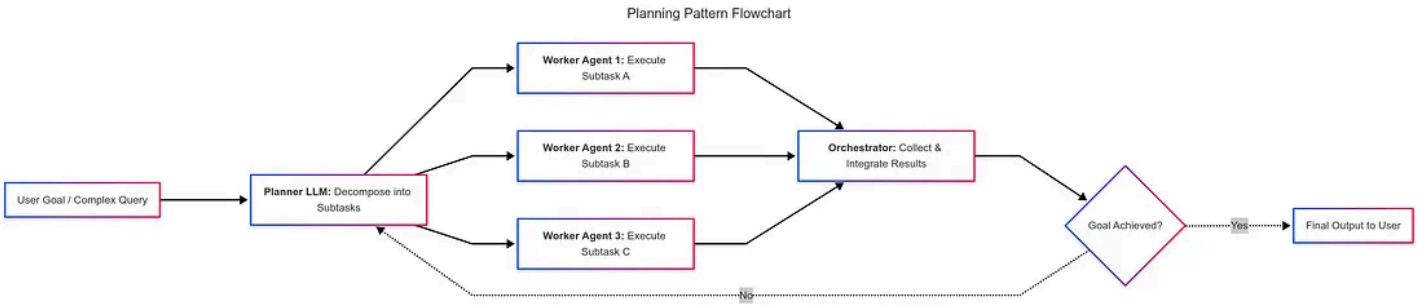
I firmly believe that Tool Calling capabilities are non-negotiable for modern agents. An agent unable to interact with external services is essentially a traditional workflow, not what we recognize as a true agent today. Simply put, Tool Calling is the cornerstone of designing any effective, externally aware agentic system.

The practical applications of the Tool Use pattern are virtually limitless, extending the agent's utility into nearly every domain.

- **Information Retrieval:** An agent can query a weather API for real-time conditions or check e-commerce inventory via a database.
- **Calculations & Analysis:** Agents can use a financial calculator to analyze stock data or execute code snippets for complex tasks.
- **Real-World Actions:** They can send emails or trigger external communications.

In fact, most advanced agents available online today, especially those specializing in fields like coding, data analysis, and complex problem-solving, rely heavily on this pattern. **Tool Calling is what transforms a language model into an actively functional agent.**

Planning Pattern



Planning design pattern flowchart

The Planning pattern is the agent's strategy engine, designed to solve complex problems that require multi-step reasoning. Unlike Routing, which selects a single next action, Planning transforms a high-level goal into a dynamic, structured list of subtasks.

The process relies on orchestrating specialized components:

- A **Planner LLM** breaks the complex request into an initial sequence of steps.
- **Worker Agents** (often using Tool Use) are delegated to execute these subtasks, potentially in parallel.
- An **Orchestrator** synthesizes the results, reflects on goal achievement, and initiates a re-planning step if needed.

The real power of this pattern is its adaptability. If the preferred venue is booked or a constraint changes, the agent doesn't fail; it registers the new information, re-evaluates its options, and formulates a new plan.

When Should You Use a Planner Pattern?

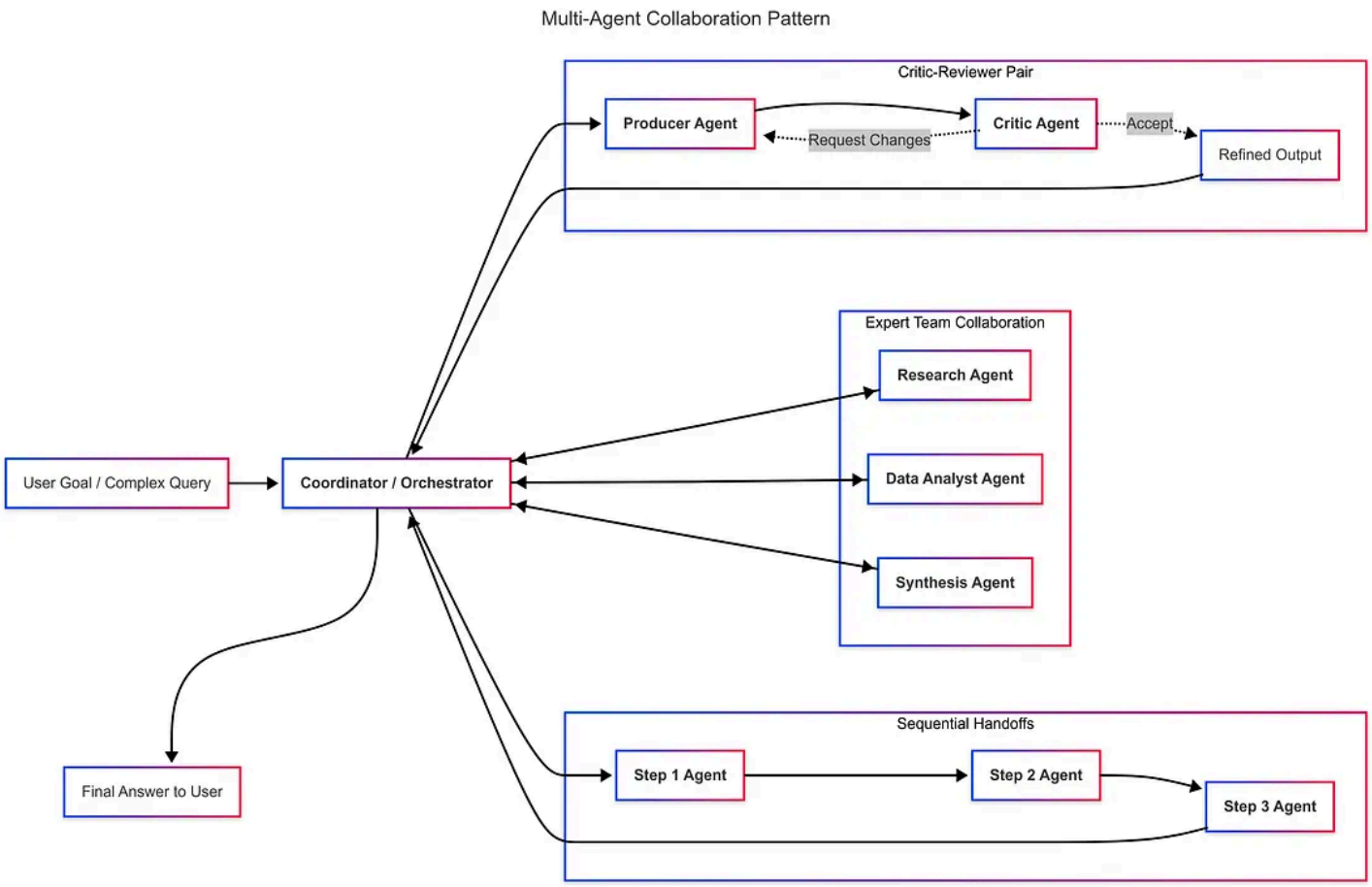
Planning is essential when the "how" needs to be discovered rather than known in advance.

- **Procedural Task Automation:** Decomposing complex business processes, like new employee onboarding, into a logical sequence of sub-tasks, managing all dependencies.
- **Structured Information Synthesis:** Generating a complex output, such as a research report, by planning distinct phases for data gathering, structuring, and iterative refinement.
- **Autonomous Navigation:** For systems (physical or virtual) that must generate an optimized sequence of actions to move from an initial state to a goal state while adhering to constraints.

Here are some key applications for the Planning pattern:

- **Software Development:** Decomposing a goal like "build a new feature" into a sequence of dependent subtasks: design, coding, testing, and documentation.
- **Comprehensive Research:** Systematically planning the workflow for a major report, including literature searches, data extraction, analysis, and structured report writing.
- **Multi-Modal Workflows:** Structuring tasks that involve different data types, such as planning steps for image generation, text analysis, and data integration.

Multi-Agent Pattern



Multi-agent design pattern flowchart

While a single agent works for simple problems, complex, multi-domain tasks need the Multi-Agent Collaboration pattern.
This approach creates an ensemble of specialized agents, smartly dividing a high-level goal into smaller sub-problems. The idea is that the collective performance will always surpass the capabilities of any single agent.

The system relies on various forms of synergy:

- **Expert Teams:** Agents with distinct roles (like a **Research Agent**, **Data Analyst**, and **Synthesis Agent**) work together on a single query.
- **Sequential Handoffs:** An agent completes its task and passes the output to the next specialized agent in the pipeline.
- **Critic-Reviewer:** One agent generates the work (e.g., a plan or code), and a second agent critiques it for correctness and quality.

This pattern offers great modularity and robustness, but it depends entirely on standardized communication protocols for agents to coordinate actions.

Frameworks like LangChain, LangGraph, and Crew AI are specifically designed to make implementing these advanced agentic patterns much easier.
They provide the necessary architecture for defining agent roles, managing communication, and orchestrating complex workflows. I'll share the practical code examples for implementing all the discussed design patterns in the next blog post.
Also, if you're interested in learning how to use LangGraph to build agents and how to implement some of the design patterns, feel free to check my LangGraph Crash Course.

Conclusion

These five patterns (Reflection, Tool Use, Planning, Routing, and Multi-Agent Collaboration**)** represent the critical architectural building blocks for constructing robust, high-performing AI agents today.
By mastering these methods, you acquire the essential toolkit necessary to tackle the vast majority of complex, real-world agentic challenges effectively.
While the field of AI engineering offers a few more specialized design patterns for niche use cases, these core five provide the bedrock you need right now. We'll be sure to explore those additional patterns in a future post!