Apr 27, 2025

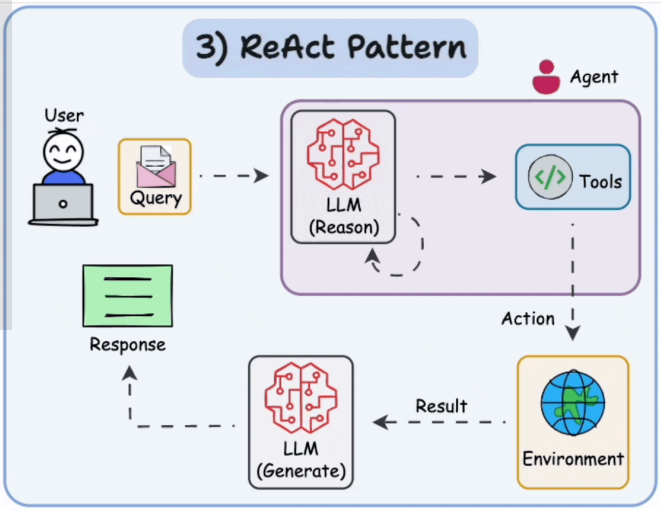
Implementing Multi-agent Agentic Pattern From Scratch

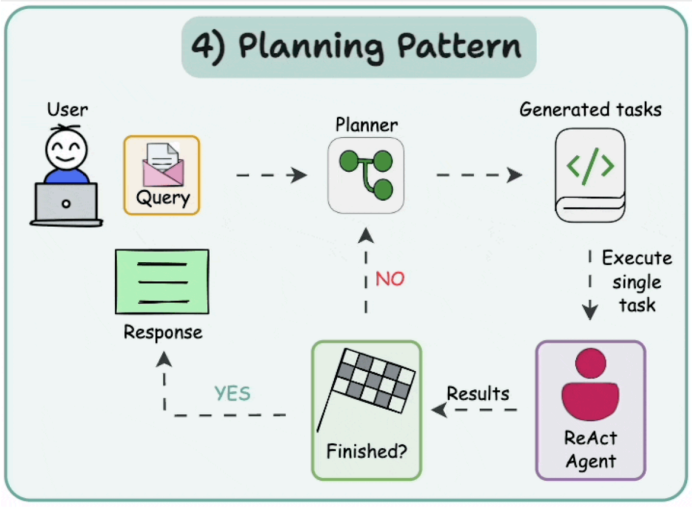
AI Agents Crash Course—Part 12 (with implementation).

Avi Chawla, Akshay Pachaar

Introduction

In this crash course series, we've already implemented two powerful patterns that power modern AI agents: the ReAct (Reasoning + Acting) loop and the Planning pattern for structured task decomposition.





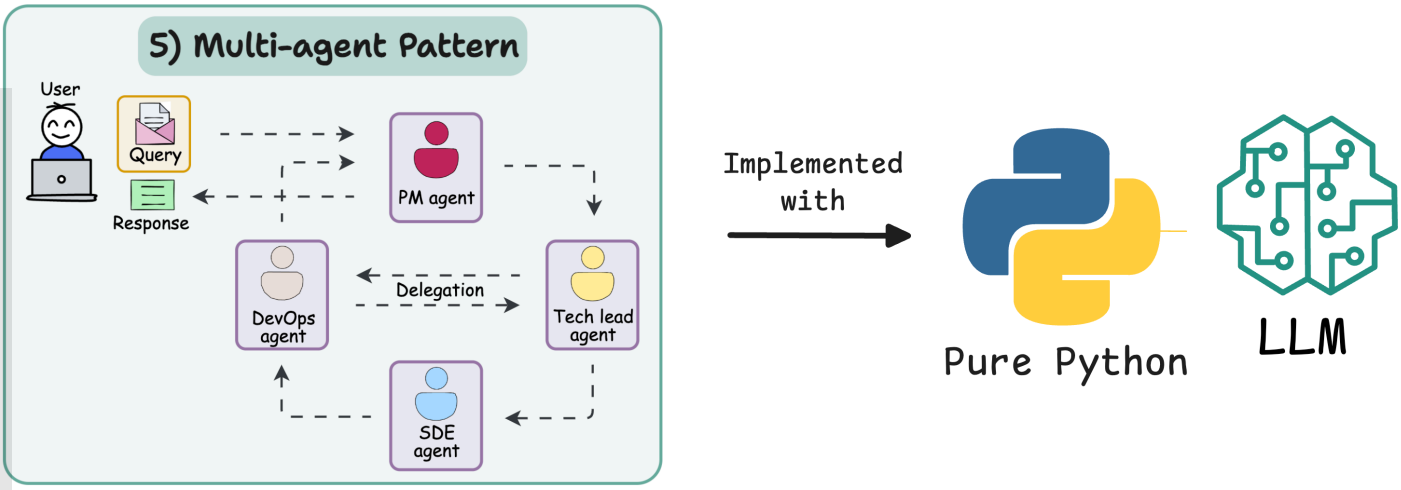
But so far, we've focused on individual agents, each reasoning and acting in isolation.

What if you want multiple agents working together?

What if each agent had a specific goal, role, and toolset, and they had to collaborate to solve a broader task?

That's where the multi-agent pattern comes in.

In this article, we'll build a multi-agent framework from scratch, without relying on high-level orchestration libraries like LangChain or CrewAI.



Instead, we'll break the entire logic down into three simple and logical building blocks:

- An `Agent` class that thinks, acts, and maintains its own reasoning loop.
- A `Tool` class that agents can invoke to interact with the outside world.
- A `Crew` class that combines multiple agents and coordinates their workflows.

By doing so, we gain full control over the agent's behavior, making it easier to optimize and troubleshoot.

We'll use OpenAI, but if you prefer to do it with Ollama locally, an open-source tool for running LLMs locally, with a model like Llama3 to power the agent, you can do that as well.

Along the way, we'll explain the multi-agent pattern, design an agent loop that interleaves collaboration and tool usage, and implement multiple tools that the

agent can invoke.

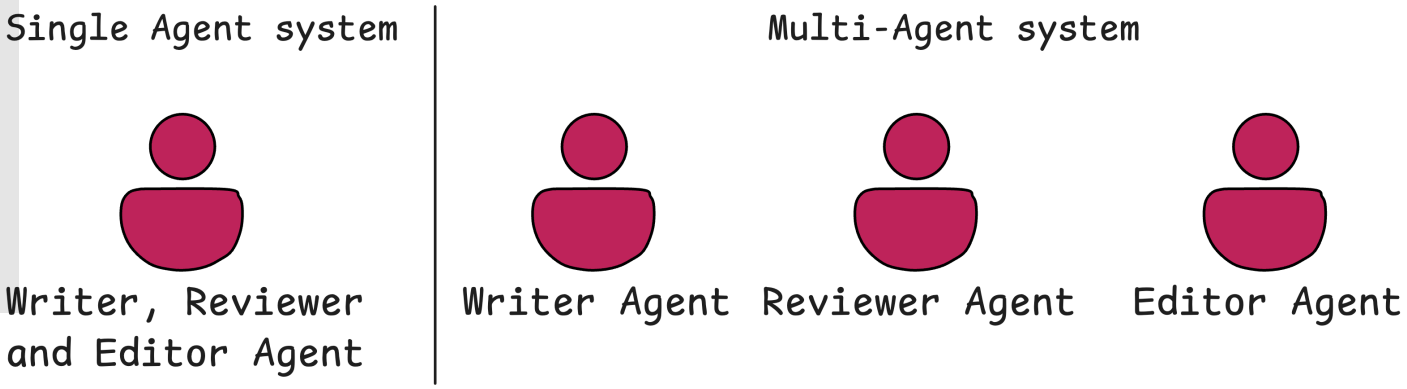
By the end, you'll walk away with a fully working multi-agent system built using plain Python and an LLM backend—one where agents reason independently, call tools, and pass results to each other in a controlled pipeline.

Let's get started.

Why Multi-Agent?

As we have seen before, a single large language model (LLM) agent is very capable, but there's a limit to what one agent can handle alone, especially for complex, multi-step tasks.

A multi-agent approach (using a team of specialized agents) has emerged as a powerful way to tackle intricate workflows.

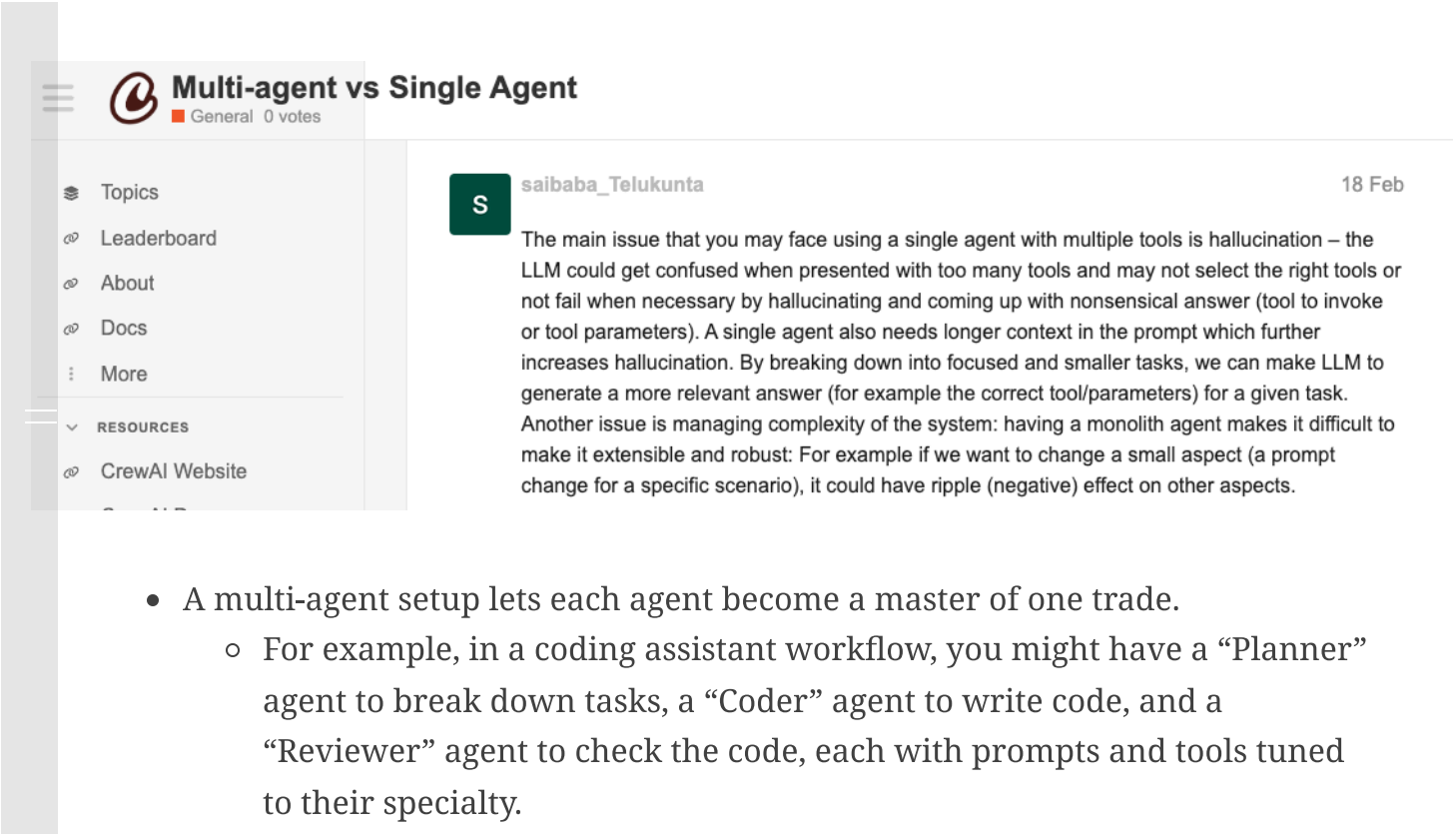


By breaking a big problem into parts and assigning each part to a dedicated agent, we gain several benefits over a monolithic agent.

For instance:

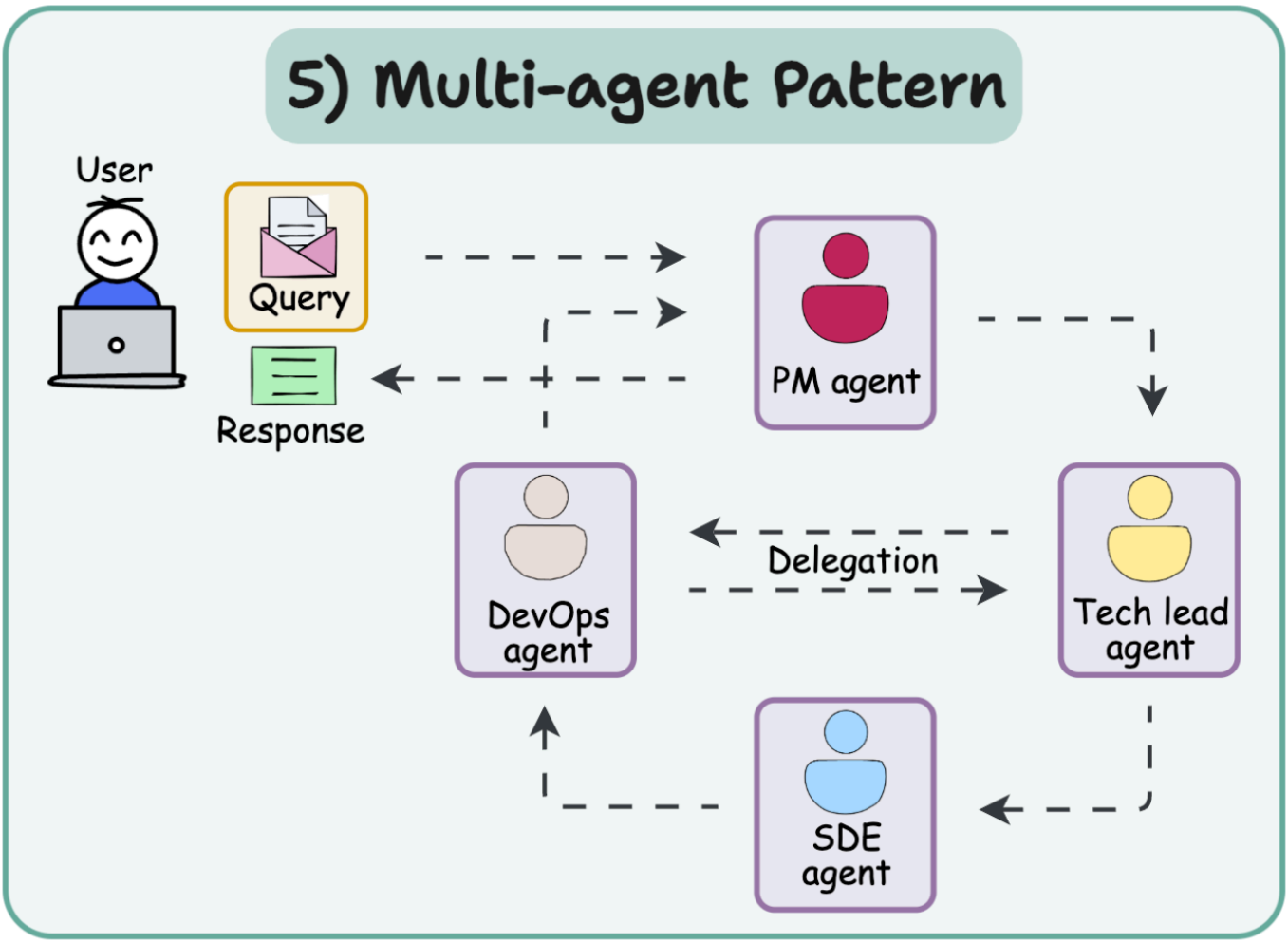
- Each agent focuses on a specific subtask or role, making the overall system easier to develop and debug.

    - Changes to one agent's prompt or logic won't derail the entire system, since other agents are isolated in their own roles (although, there, one's output may be used by another).
    - This separation of concerns means we can update or replace one agent (say, improve the "research" agent) without unintended side effects on others, much like swapping out one specialist on a team without retraining everyone else.
    - It also reduces complexity: a single agent trying to juggle all tools and instructions in one huge prompt can become confused or hallucinate instructions, whereas multiple focused agents stay on track:



- A multi-agent setup lets each agent become a master of one trade.
    - For example, in a coding assistant workflow, you might have a "Planner" agent to break down tasks, a "Coder" agent to write code, and a "Reviewer" agent to check the code, each with prompts and tools tuned to their specialty.
    - Because each agent is specialized, they can employ domain-specific reasoning or use dedicated tools most effectively.
    - This distributed problem-solving means complex jobs (like planning a vacation) can be split up: one agent checks the weather, another finds hotels, another maps routes, and so on.
    - Together, the specialists solve the whole problem more efficiently than one generalist agent.
- Having multiple agents inherently creates a clear structure of who does what, which makes the system's reasoning process more transparent.

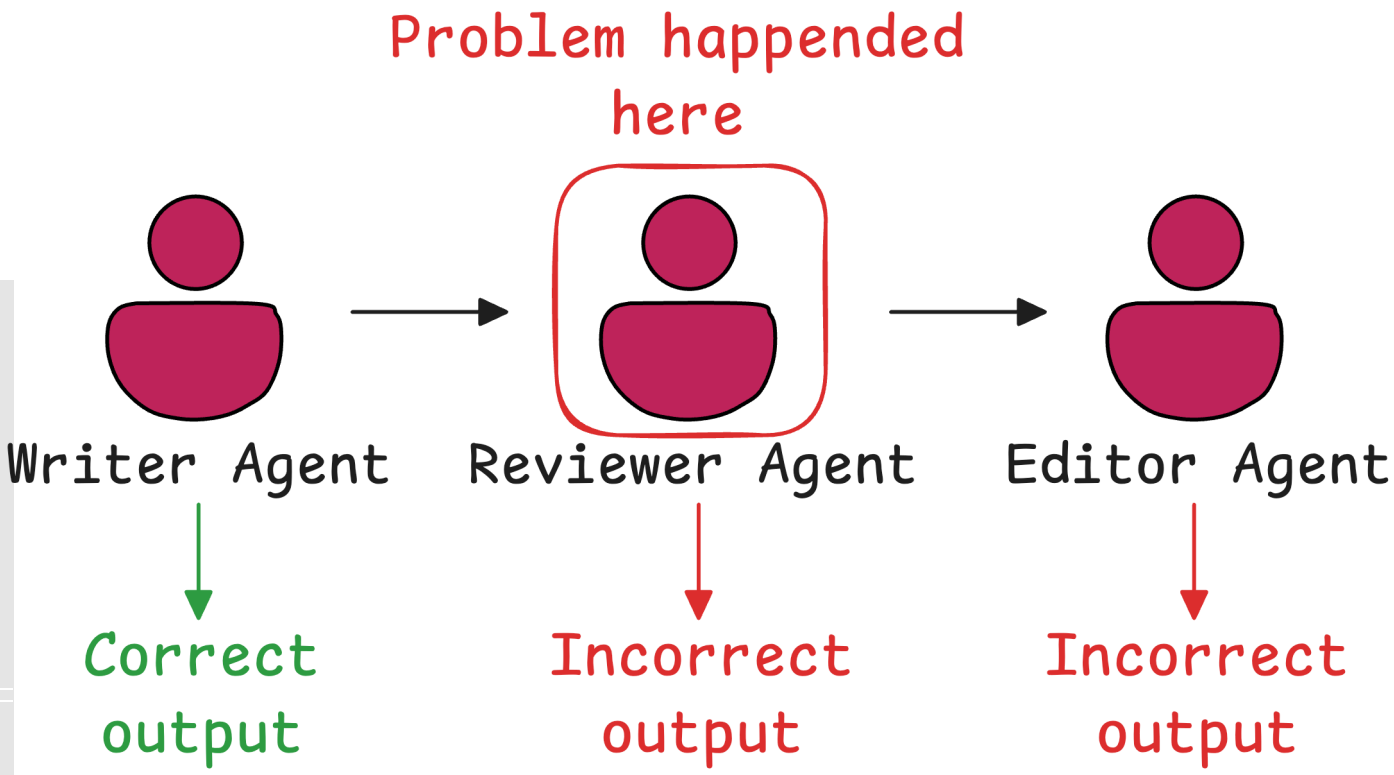
- Each agent typically “thinks out loud” (via chain-of-thought prompts) and produces an intermediate result that other agents or the developer can inspect.
- In fact, multi-agent workflows often allow or even encourage agents to cross-verify each other’s outputs, catching mistakes and providing justification for decisions.



- For instance, one agent can generate a plan and another agent can critique or refine it, producing a traceable dialogue.
- This yields an audit trail of reasoning, improving explainability since you can see the rationale from each specialist rather than a single opaque giant model answer.
- Recent AI systems have used this approach to provide detailed rationales; one real-world platform reported its multi-agent setup gave

defensible, nuanced recommendations with clear explanations for each decision.

- Moving on, another benefit is that when something goes wrong in a multi-agent pipeline, it's easier to pinpoint the issue.



  - Because tasks are segmented, you can tell which agent (and which step) produced a bad output or got stuck (using the verbose output or other logging procedures).
  - This is much harder with one mega-agent doing everything.
  - Developers can examine the intermediate outputs (e.g. the search results from the Researcher agent, or the draft summary from the Writer agent) and identify where a flaw occurred.
  - Also, you wouldn't give one person too many tasks at once and expect perfect results. The same applies to agents.
  - Dividing the work not only boosts quality, it lets you trace errors to the responsible agent quickly and fix that part of the process without overhauling the rest.
- Lastly, the multi-agent pattern mirrors how human teams operate, making AI workflows more natural to design.

- In organizations, people specialize in roles – you have analysts, planners, builders, reviewers, and they collaborate to accomplish something complex.
- Similarly, in a multi-agent AI system, each agent is like a team member with a clear job, and a coordinator ensures they work in concert.
- This mapping makes it conceptually easier to design AI solutions for complex problems by thinking in terms of "Which agents (roles) would my team need?"
- For example, an AI travel concierge service can be designed as a crew of agents: one for looking up attractions, one for checking logistics (flights, weather), one for optimizing the itinerary, etc., all working together on the user's request.
- Such a division not only feels intuitive but has been shown to handle the complexity better than a single all-purpose agent.

In summary, it must be clear that using multiple agents introduces modularity, specialization, and clarity into AI reasoning.

It can make AI systems more robust and interpretable by having dedicated mini-brains tackling each part of a problem and then sharing their findings.

Recent research and industry experiments consistently find that multi-agent collaboration can solve more intricate tasks and provide better transparency than a lone AI agent, much like a well-coordinated team outperforms a single overworked individual.

Multi-agent internal details

As discussed earlier, a multi-agent pattern structures an AI workflow as a team of agents that work together, each with a well-defined role.

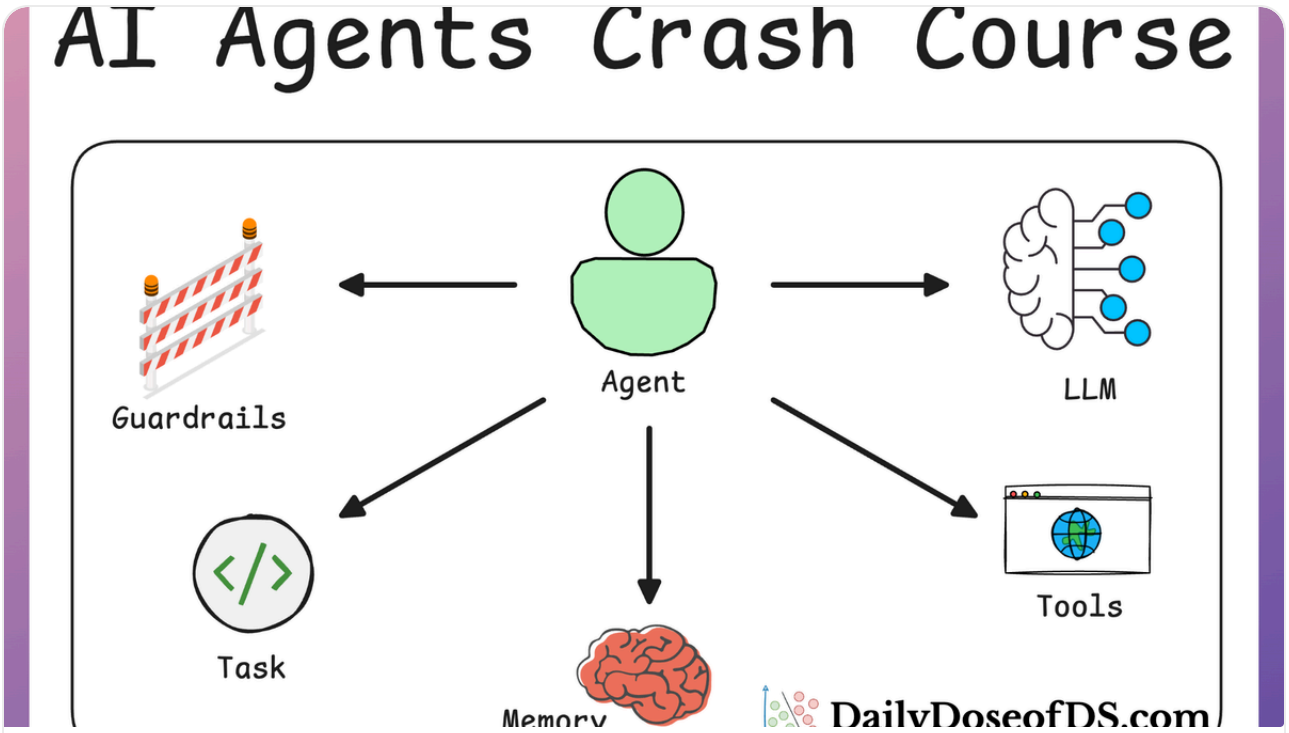
Instead of one agent handling a task from start to finish, you have multiple agents in a pipeline, where each agent takes on one part of the task and then hands off the result to the next agent.

This design is often (**not always**) coordinated by an orchestrator that makes sure the agents run in the right sequence and share information.

The idea is analogous to an assembly line or a relay team: Agent A does step 1, then Agent B uses A's output to do step 2, and so on, until the goal is achieved.

Each agent is autonomous in how it thinks/acts for its sub-problem, but they collaborate through the defined workflow to solve the bigger problem together.

Technically speaking, each agent in a multi-agent pattern still follows the reasoning+acting loop internally (often implemented via the ReAct paradigm, which we have already seen in the ReAct pattern issue).



AI Agents Crash Course—Part 10 (With Implementation)

A deep dive into ReAct patterns and implementing it from scratch.

Daily Dose of Data Science • Avi Chawla

That is, an agent will receive some input (or context), think about what to do (generate a "Thought"), then act by either producing an output or calling a Tool, observing the result, and so on.

The crucial difference is that an agent's scope is limited to its assigned role.

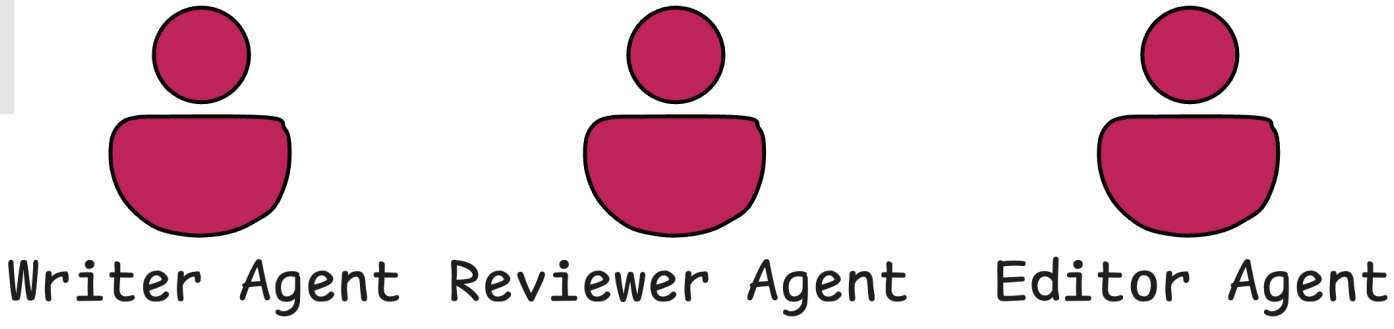
It has a narrower focus, a smaller set of tools, and a specific goal in the chain. For example, an agent tasked with "database lookup" will primarily reason about how to query the database and act by executing that query, rather than also worrying about how to present the final answer to the user, which would be another agent's job.

By constraining an agent's responsibilities, we make its prompts simpler and more targeted, which often leads to more reliable behavior.

System overview

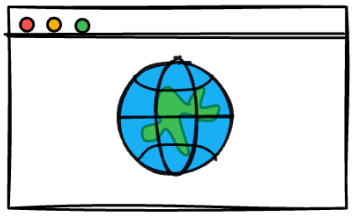
To clarify the pieces of this architecture, let's break down the main components involved in a multi-agent pipeline:

- Agent:

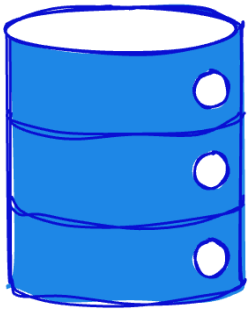


    - In this context, an Agent is an autonomous AI unit (usually an LLM with a prompt) that can perceive inputs, reason (via chain-of-thought), and perform actions to complete a subtask.

    - Each agent is typically configured with a specific role and has access to only the tools or information it needs for that role.
    - For instance, you might have a `ResearchAgent` that can use a web search tool, and a separate `SummaryAgent` that can use a text generation tool.
    - The agent will loop through thinking ("Thought…") and acting ("Action…") until it produces an outcome for its part of the job.
    - Because it's focused, it can follow a strict prompt format or protocol (often defined by the framework) that keeps its behavior safe and on-task.
- Tool:



Web search tool



Database tool

    - A Tool is any external capability that an agent can invoke to help it act on the world or fetch information.
    - Tools could be things like a web search API, a calculator, a database query interface, an email-sending function, etc.
    - In the multi-agent pattern, tools are the specific actions available to agents. Each agent usually has a limited toolbox relevant to its role.
    - For example, the ResearchAgent might have a "Search the web" tool, while the SummaryAgent might have a "Lookup knowledge base" tool.
    - The ReAct loop of the agent involves choosing a tool and providing inputs to it (e.g. a search query), then reading the tool's output (observation) to inform its next thought.

    - By structuring tool use this way, we ensure the agent's reasoning and actions are transparent and can be logged (we see what query it searched for, what result came back, etc.).
    - Notably, frameworks like LangChain or CrewAI define tool interfaces and force agents to adhere to a format (e.g. always output a JSON for tool inputs) to keep this interaction reliable.
- Crew
    - The Crew is essentially the orchestrator, a component (or class, in code) that sets up the multi-agent workflow and manages the execution flow.
    - If each agent is a team member, the Crew is the team manager that knows the overall plan.
    - In a sequential pipeline, the Crew will pass the initial input to the first agent, take that agent's output and feed it to the next agent, and so forth, until the final agent produces the answer.
    - This Crew defines the order of agents (and sometimes the specific task each agent performs on the data).
    - For example, using the CrewAI library, one can instantiate a `Crew(agents=[Agent1, Agent2, ...], tasks=[Task1, Task2, ...], process=Process.sequential)` and then call `crew.kickoff()` to run through the agents in sequence.
    - The Crew handles coordination details like ensuring each agent gets the right input, and it can collect or consolidate outputs.
    - In more complex setups, the coordinator might also implement logic for looping (repeating agents or steps until a condition is met) or branching (choosing which agent to invoke based on some criteria), but the basic idea is that the Crew oversees the agents so they work together smoothly.
    - In some architectures, the coordinator is itself an agent (often called a "manager" or "supervisor" agent) that has the job of delegating tasks to other specialist agents.
    - Whether it's an explicit class or a manager agent, this coordinating role is what makes the group of agents function as a unified system rather than a bunch of isolated bots.

In a multi-agent pipeline, information typically flows from one agent to the next.

For instance, consider a simple pipeline where Agent A gathers data, Agent B analyzes it, and Agent C writes a report.

The Crew would give Agent A the initial query; Agent A uses its tools and returns, say, some raw data; that data is passed to Agent B, which produces an analysis; then Agent C gets the analysis and produces the final report.

Each agent only deals with the input relevant to its task and doesn't worry about the overall mission beyond its part.

This sequential hand-off is what we mean by the multi-agent pattern: it's a design where multiple narrow AI agents are chained together, each one's output feeding into the next's input until the goal is achieved.

Implementing Multi-agent system from scratch

So far, we've seen why multi-agent setups offer better modularity, clarity, and interpretability compared to single-agent systems.

Now, let's get our hands dirty and implement a multi-agent system from scratch, without relying on any external orchestration libraries like LangChain or CrewAI.

As discussed earlier, our system will be built using three core abstractions:

- Tool class:
    - The Tool class wraps around real functions (like search_news, summarize_text, or calculate_stats) and exposes them in a way that agents can discover, validate inputs against, and invoke dynamically. Each tool includes:
        - A parsed function signature,

        - Automatic input validation and type coercion,
        - A standardized calling interface.
    - This allows agents to reason about what tools are available and how to use them, without hardcoding function calls.
- Agent class:
    - The **Agent** class represents an individual AI agent that can:
        - Think through a task (via the ReAct loop),
        - Invoke tools to act on the environment,
        - Pass outputs downstream to dependent agents.
    - Each agent has a distinct backstory, task, output expectation, and toolset, making it highly modular and role-specific.
    - Agents can also define dependencies: e.g., Agent A must run before Agent B. Internally, each agent is powered by a `ReactAgent` wrapper that handles its reasoning loop using a language model.
- Crew class:
    - The Crew class is the orchestrator that glues everything together.
    - It manages:
        - Agent registration,
        - Dependency resolution using topological sorting,
        - Ordered execution of agents based on dependencies.

In the next sections, we'll walk through each class in detail, understand how they work, and see how they come together to form a working multi-agent system.

Let's start with the Tool class.

You can download the code below:

multi-agent-implementation

multi-agent-implementation.zip • 29 KB

Tool implementation

In any agent system, tools are what give your agents the power to take action.

They let your LLM do things it couldn't otherwise do on its own, like fetch data, run calculations, or trigger APIs.

But in order for an agent to call a tool safely and correctly, the tool must:

- Describe its inputs and expected types.
- Validate arguments before calling the real function.
- Provide a consistent interface, regardless of the underlying implementation.

Here's what we want to support:

notebook.ipynb

```python
@tool
def lookup_weather(city: str, units: str = "Celsius") -> str:
    """Gets the weather in a city."""
```

Once decorated with `@tool`, this function becomes:

- Introspectable: the agent can see it takes two arguments and returns a string,
- Callable: the agent can invoke it with dynamic keyword arguments,

- Validated: the system will check inputs and apply defaults or type casting if needed.

Let's build exactly that.

Below is the complete implementation of a `Tool` abstraction that wraps any Python function and makes it callable by an AI agent.

We'll build this using just a few standard libraries:

notebook.ipynb

```
import inspect
import json
import logging
from typing import Any, Callable, Dict, Optional, Type
```

Let's say you have a function like this:

notebook.ipynb

```
def lookup_weather(city: str, units: str = "Celsius") → str:
    """Gets the weather in a city."""
```

We want to extract its metadata, its name, input parameters, default values, and return type, so that an agent can understand how to use it.

That's where the `FunctionSignature` class comes in:

notebook.ipynb

```
class FunctionSignature:
    def __init__(self, func: Callable) -> None:
        sig = inspect.signature(func)
        self.name = func.__name__  # e.g., "lookup_weather"
        self.description = (func.__doc__ or "").strip() or None # docstring
        self.parameters = {}
```

We use Python's built-in `inspect` module to grab the signature of the function (i.e., the argument names, types, defaults, return type, docstring, etc.). The docstring specifically will help the Agent figure out which tool would be an ideal tool to invoke from all the available tools.

Let us quickly see what the `inspect` module does on the `lookup_weather` dummy function defined earlier:

notebook.ipynb

```
import inspect
def lookup_weather(city: str, units: str = "Celsius") -> str:

    """Gets the weather in a city."""
    ...

sig = inspect.signature(lookup_weather)

for param_name, param in sig.parameters.items():
    print(param_name, " -- ", param)

"city  --  city: str"
"units  --  units: str = 'Celsius'"
```

This tells us the name of the parameter and its type.

The loop below parses all parameters:

notebook.ipynb

```
for param_name, param in sig.parameters.items():
    if param_name == 'self':
        continue

    info = {}
    if param.annotation is not inspect._empty:
        annot = param.annotation

        if hasattr(annot, '__name__'):
            info['type'] = annot.__name__
        else:
            info['type'] = str(annot)

    if param.default is not inspect._empty:
        info['default'] = param.default

    self.parameters[param_name] = info
```

If a parameter has a type hint, like `units: str`, we store `"str"` as its type. If it has a default, like `units="Celsius"`, we record that too.

Finally, we do the same for the return type, if one exists.

notebook.ipynb

```
        self.return_type: Optional[str] = None

        if sig.return_annotation is not inspect._empty:
            ret = sig.return_annotation

            self.return_type = ret.__name__ if hasattr(
                ret, '__name__') else str(ret)
```

At the end, we can convert this whole thing to a Python `dict` or JSON string via:

notebook.ipynb

```
    def to_dict(self) -> Dict[str, Any]:
        return {
            "name": self.name,
            "description": self.description,
            "parameters": self.parameters,
            "return_type": self.return_type,
        }

    def to_json(self) -> str:
        return json.dumps(self.to_dict())
```

This allows Agents (or developers) to inspect the tool like so:

notebook.ipynb

```
{
  "name": "lookup_weather",
  "description": "Gets the weather in a city.",
  "parameters": {
    "city": { "type": "str" },
    "units": { "type": "str", "default": "Celsius" }
  },
  "return_type": "str"
}
```

We did this because Agents don't read Python code. But they do understand structured JSON. By giving them a tool's name, description, and argument schema in JSON, we help them figure out which tool to use and how.

Let's quickly inspect the output of the above implementation by running the `lookup_weather` dummy function through it:

notebook.ipynb

```
sig = FunctionSignature(lookup_weather)
```

Now printing the instance-level attributes we created, we get:

notebook.ipynb

```
>>> sig.parameters
{'city': {'type': 'str'}, 'units': {'type': 'str', 'default': 'Celsius'}}

>>> sig.return_type
'str'

>>> sig.description, sig.name
('Gets the weather in a city.', 'lookup_weather')
```

Now, imagine an agent wants to call `lookup_weather(city="London", units="C")`. To do so, it will prepare a function call using the instructions we will specify in the ReAct prompt.

Thus, during this function call and going back to the earlier example, we need to make sure:

- `city` is a string.
- `units` is a string.
- If the agent omits `units`, we use the default.

That's what the `ArgumentValidator` does:

notebook.ipynb

```
class ArgumentValidator:

    """
    Validates and coerces arguments based on a FunctionSignature.
    """
```

We go through each parameter in the signature and:

- Check if the argument was provided.
- If yes: Convert it to the right type (e.g., str, int).
- If not: Use a default if one is available.
- If all else fails: raise an error.

So essentially, using the `FunctionSignature`, we know what a function expects. But now, we need to validate whatever the agent sends in.

Agents can make mistakes. They might pass "5" instead of 5, or forget a required argument. This class catches that:

notebook.ipynb

```
class ArgumentValidator:

    """
    Validates and coerces arguments based on a FunctionSignature.
    """

    def __init__(self):
        self._type_map = {"int": int,
                          "float": float,
                          "str": str,
                          "bool": bool,
                          "list": list,
                          "dict": dict}
```

In the above code, we map string-type names (like `"int"`) to real Python types (`int`).

Next, the `validate()` function checks each input:

notebook.ipynb

```
def validate(self, args: Dict[str, Any], signature: FunctionSignature) → Dict[str, Any]:
    validated: Dict[str, Any] = {}

    # for all expected parameters in the function signature
    for name, meta in signature.parameters.items():

        # check if it exists in the function call prepared by the Agent
        if name in args:

            # if yes, get the expected data type
            val = args[name]
            expected = meta.get('type')

            if expected and expected in self._type_map:

                # convert to target data type if not already
                target_type = self._type_map[expected]
                if not isinstance(val, target_type):
                    try:
                        val = target_type(val)  # coerce type
                    except Exception as e:
                        raise TypeError(f"Argument '{name}' must be of type {expected}")

            # store paramter in validate parameters
            validated[name] = val

        # if parameter does not exist in function call, see if it has a default value
        else:
            if 'default' in meta:
                validated[name] = meta['default']
            else:
                raise KeyError(f"Missing required argument: '{name}'")

    return validated
```

This may look a bit intimidating, but all it's doing is:

- If a required argument is missing → throw an error.
- If an argument is the wrong type → try to convert it.
- If an argument is missing but has a default → use the default.

This gives us safe, predictable function calls, even when the agent provides inconsistent input types.

To put it another way, this lets us sanitize the agent's input before calling the real function.

Finally, we have an actual object the agent interacts with—the Tool class:

notebook.ipynb

```
class Tool:
    """
    Wraps a callable for standardized signature
    introspection and invocation.
    """

    def __init__(self,
                 function: Callable,
                 signature: FunctionSignature,
                 validator: ArgumentValidator) → None:

        self._fn = function
        self.signature = signature
        self._validator = validator
```

When an agent uses a tool, it will simply call the `Tool` object, passing keyword arguments:

Here's what happens internally:

1. Input is validated and type-checked,
2. The wrapped function is called with the clean arguments,

3. The result is returned back to the agent.

The `info()` method just returns the tool's metadata in JSON:

To make this API developer-friendly, we define a decorator:

Now you can do:

And get a full-featured Tool object that:

- Carries all metadata,
- Validates its input,
- Can be invoked safely by an agent.

Let's look at it in action on the `multiply` method. After decorating the method, we can do all this:

- View its metadata (name, parameters, return type):

- Get a JSON summary for logging or inspection:

Use it like a callable with validation:

If you forget a parameter or pass the wrong type and it can’t be coerced, it raises an error:

Overall, with this Tool abstraction in place, we now have:

- A structured way to expose capabilities to agents,
- Reliable runtime checks to avoid bad inputs,
- A clean, developer-friendly interface (`@tool`) to wrap your functions.

This makes it easy for agents to both reason about and use tools programmatically.

In the next section, we'll look at how agents use these tools and how each agent is structured to reason, act, and pass context.

Let's move on to the Agent class.

Agent implementation

With the `Tool` class in place, the next step is to create our Agent, which is the main actor in our multi-agent system.

This class represents an intelligent unit that can:

- Reason using a language model,
- Act by invoking tools (via the ReAct loop),
- Pass results to other agents in a pipeline,
- Use context from upstream agents to enrich its own decisions.

It's a powerful but complex class, so let's walk through it step-by-step.

We begin with a few critical imports:

This includes:

- `Crew`: to register the agent with a crew coordinator.
- `Tool`: for tool-based reasoning.
- `MessageHistory`, `create_message`, `TagParser`: utility classes to manage LLM messages and parse structured tags from outputs.
- `litellm`: used to call LLMs like GPT-4o through a clean, minimal interface.

Next, we define the ReAct prompt template used when the agent has tools:

The `REACT_PROMPT` is the instruction manual we hand to the LLM-powered agent.

It tells the model:

- How to think (generate intermediate reasoning),

- When to act (call a tool),
- How to wrap its responses (so we can parse and route them),
- And most importantly, to follow a predictable format so we can automate the loop.

Let's break it down section by section.

> 💡 1. Your workflow is by running a ReAct (Reasoning + Action) loop with the following steps:
> - Thought
> - Action
> - Observation

This sets the foundation: we're not asking the model to answer immediately.

Instead, we want a step-by-step reasoning process:

- Thought: What the agent is thinking.
- Action: What tool the agent wants to use.
- Observation: What result it got back from that tool.

This loop continues until the agent is ready to finalize its response.

Right below the prompt, we'll inject the full list of tools (via `Tool.info()`), so the model knows:

> 💡 2. You have access to function signatures within <tools></tools> tags.

- Each tool's name,
- The types of arguments it expects,
- And what it does (via the docstring).

Next, even if a function has defaults (like `units="Celsius"`), we ask the model to explicitly pass all arguments. This keeps behavior consistent and easier to debug.

> 💡 3. Feel free to invoke one or more of these functions to address the user's request. Do not presume default values—always use the signatures provided.

Moving on, we mention this:

> 💡 4. If a tool/function is available for some task you MUST use it, without fail.

This reinforces the key idea: the agent shouldn't answer questions directly if there's a tool available to help. It must:

- Think about the problem,
- Choose the right tool,
- Call it,
- Then form the final answer based on that.

Next up,we have this:

> 💡 5. You are someone who only knows how to write, for any sort of data, see what tools are available and how to use them.

This is clever: it limits the model's capabilities, nudging it to delegate any data fetching or computation to tools.

In other words:

- If it wants to know something → use a tool.

- If it wants to calculate → use a tool.
- It can only "write" based on what it observes.

This leads to structured behavior and minimal overreach.

Next, we ask it to follow exact input format:

> 💡 6. Pay close attention to each function's types property and supply arguments exactly as a Python dictionary.

This ensures every tool call looks like...

...instead of some vague natural language. This makes it parsable and guarantees compatibility with our tool executor.

Next, we ask it to use tags to wrap observations and responses:

> 💡 7. After successful tool call, give <observation> result from tool call </observation> and <response> final output </response>.

This allows us to clearly extract:

- The tool result using `<observation>` tags,
- The final answer using `<response>` tags.

You can think of these as programmatic anchors that we extract using `TagParser` (a regex pattern match class).

Next, we give it the tool call format:

> 💡 8. For every function call, return a JSON object wrapped in <tool_call> </tool_call> tags, using this pattern:
>
> <tool_call>
> {"name": <function-name>,"arguments": <args-dict>, "id": <sequential-id>}
> </tool_call>

This is critical: tool calls must follow this exact format so our code can parse them cleanly, validate the arguments, and run the function dynamically.

The `id` lets us map tool results back to their call, useful if we ever support multiple calls in a single turn.

We also add scope to dynamically add tools to this prompt:

> 💡 Here are the available tools/actions/functions to assist you:
> <tools>
> %s
> </tools>

We'll inject available tools into this template so the agent knows exactly:

- What tools it can use.
- The expected JSON format for calling them.
- How to structure thoughts, actions, observations, and responses.

Lastly, we have a final note:

💡 Note: If the user's question doesn't require a tool, then DO NOT use tool and answer directly inside <response> tags.

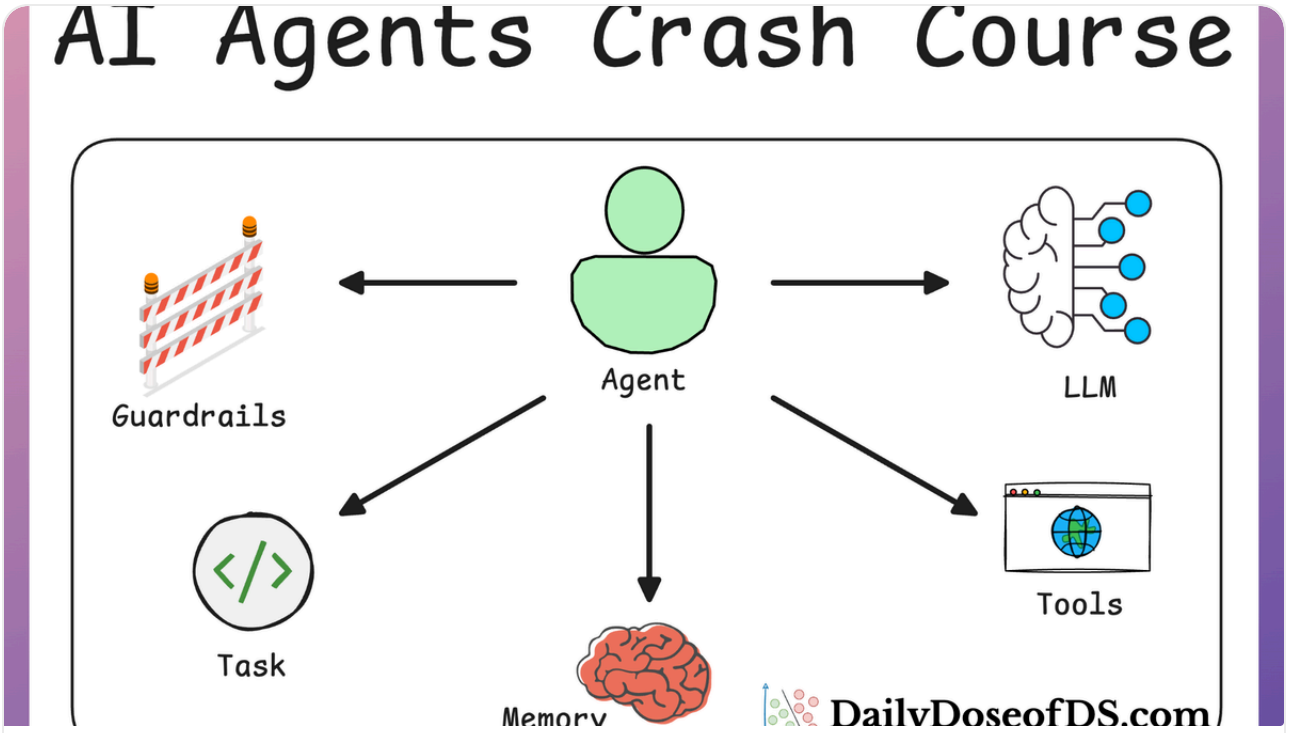
This ensures the agent doesn't over-engineer answers. If the task is something like "Write a sentence about AI," it can skip tools entirely and just respond.

To summarize, this prompt becomes the Agent's system instruction when tools are involved.

Also, this structured prompt:

- Forces the model to think → act → observe → respond.
- Encourages deterministic tool calls using `<tool_call>` tags.
- Injects tool signatures in `<tools>` for inspection.
- Helps us parse outputs reliably by structuring the model's behavior.

This is key to building ReAct agents that can reliably call tools and return final answers with justification. We did something similar when we implemented the ReAct pattern from scratch:



Moving on, let's look at the `Agent` class constructor:

Each agent is initialized with:

- A `name` (for logging and dependency wiring),
- A `backstory` (sets the system prompt, helps frame the agent's personality or role),
- A `task_description` (what it's supposed to do),
- An optional `expected_output_format` (helps shape final outputs),
- A list of tools (if any),
- And the name of the LLM model to use.

We load the OpenAI key and configure LiteLLM:

This lets the agent make calls using the selected LLM backend.

We also store all inputs as attributes:

Tools will be passed when the Agent will be defined.

We build a dictionary of tool names → tool objects:

This allows fast lookup during tool calls.

Next, we set up tag parsers to extract `<thought>`, `<tool_call>`, and `<response>` blocks from model outputs:

These make structured parsing easy during the reasoning loop.

Agents may depend on other agents. These attributes track that:

Next, we implement a function to generate LLM outputs. This function sends a list of chat messages to the LLM:

It’s called repeatedly in the reasoning loop to generate the next step of the conversation.

We define two simple methods for wiring agents and defining which Agent runs after which other Agent:

For instance:

Both mean: A runs before B.

Since some Agents depend on others, we need a method to receive context. This method lets upstream agents pass results to this Agent:

These messages will be shown to the model in its context. So essentially, every agent appends its final output to its dependents' context block.

Moving on, before an agent can reason, it needs a clean prompt:

This prompt includes:

- What to do (`<task_description>`), which will be specified during Agent declaration.
- How to format the output,
- What has already been done (from dependencies).

Moreover, we also need to render the full React prompt. We include available tool signatures in the system prompt:

This is inserted into the prompt so the model knows how to call which tools, with the exact signature.

When the agent emits `<tool_call>` tags, we extract and execute them:

For instance:

The above will trigger:

Finally, we have the full execution loop for the agent, which runs the actual reasoning loop:

In case A, if Agent has tools → use ReAct loop:

Then we iterate:

If `<response>` is found, we're done.

Otherwise, we parse:

If tool calls are found, we invoke them and insert the results back into the prompt:

This cycle continues until a `<response>` is returned or we hit the round limit.

In case B, if Agent has no tools → plain LLM prompt:

The final step is to share the output with downstream Agents:

In the above code, once the agent produces an output (response tag found!), it's passed to all the agents that depend on it.

Below, let's look at a tool-using Agent in action:

This produces the following output:

Perfect!

Overall, this Agent class is your fully-featured AI worker:

- ReAct-based when tools are involved,
- Simple LLM when not,
- Automatically handles context, tool invocation, and chaining.

It encapsulates independent, modular reasoning, which is the essence of how powerful multi-agent systems scale.

Next up, we'll wire all of this together using the Crew class, which manages the overall pipeline and executes agents in the right order.

Crew implementation

We've seen how agents can reason, act, and pass context. But how do we manage multiple agents? How do we ensure they run in the right order? And how do we connect them together automatically?

That's the job of the Crew.

The `Crew` class acts like a workflow manager for your agents.

You can think of it as:

- A container that holds all your agents,
- A scheduler that figures out the right order to run them,
- And a controller that actually executes them.

Let's walk through the code, step by step.

We import what we need: `deque` for queueing agents, `Optional` and `Any` for type hints.

Next, here's the Crew class overview:

This docstring says it all. The Crew:

- Holds your agents,
- Knows their dependencies,
- Sorts them topologically,
- And runs them in the correct sequence.

Let’s dive into each part.

First, we have a class-level variable used to track the currently active `Crew`:

So to elaborate, whenever you create a `Crew` with:

That crew becomes “active.” This allows individual agents to self-register without you needing to manually add them.

Next, we initialize a private list to hold all the agents that belong to this crew:

Moving on, we have the context manager enter and exit methods:

This is what enables the nice Pythonic syntax:

By setting the `Crew._active` class variable on enter and clearing it on exit, we make sure agents can detect the active crew and add themselves to it.

Next, we have Agent registration. This is how Agents add themselves to the crew.

This `register()` method checks if there is an active crew. If so, it calls `add()` to formally attach the Agent to the Crew.

If no crew is active, it returns.

The above register method invokes the `add` method, which is implemented below:

This method verifies that the object:

- Has a `.name` attribute,
- Has a `.run()` method.

This ensures only valid agents are added. Then it appends the agent to the crew's list.

Now comes the most important part, which is sorting agents based on who depends on whom.

Let's say:

- Agent A must run before Agent B,
- And Agent B must run before Agent C.

This method ensures that the agents run in the right order: A → B → C.

This is a classic topological sort problem and we begin by assigning each agent an "in-degree", which is the number of agents it depends on:

Next, we create a queue for Agents with no dependencies:

These are agents that can run immediately, which represent the “starting points” of the graph.

Now, from the existing Queue, we go agent by agent, reduce the dependency count of downstream agents, and queue them once they’re ready to run.

This is a classic topological sort used in compilers, build systems, and dependency resolution.

If we can’t sort all agents, a cycle exists, which means something like A depends on B and B depends on A. This is illegal and needs to be fixed.

Done!

Finally, we run all the Agents:

This function:

- Sorts agents by dependency order,
- Calls `.run()` on each one,
- Prints clean trace messages,
- Catches and logs any error in the pipeline.

Let's bring it all together and walk through a simple, minimal multi-agent pipeline using the system we just built.

We'll define two tools:

1. One that doubles a number.
2. One that squares a number.

Then we'll create two agents:

- Agent A: doubles the number 4.
- Agent B: takes the result from Agent A and squares it.

First, we define the tools:

These are regular Python functions, decorated with `@tool`, which:

- Expose metadata (name, input types, docstring),
- Automatically validate inputs,
- Can be called by the agent using structured prompts.

Next, we create the Agents:

Each agent has:

- A name (used for identification),
- A backstory (used as the system prompt),
- A task description (what the agent is supposed to do),
- A list of tools it is allowed to use.

Here:

- Agent A will double 4.
- Agent B will square the output of Agent A (i.e., square 8).

Moving on, we define the dependency:

This is equivalent to: Agent B depends on Agent A.

It ensures Agent A's output is passed to Agent B as context.

Finally, we run a Crew:

We open a `Crew` context to:

- Allow agents to auto-register,
- Maintain one clean execution scope.

Then we run all agents using `run_all()`, which:

1. Sorts them topologically (A → B),
2. Calls `.run()` on each agent in order,
3. Prints the output.

This produces the following output:

Let's break that down:

- The first agent doubles 4 → outputs 8.

- The second agent receives this value as part of its context, squares it → outputs 64.

And this all happens automatically, because:

- The agents are aware of their dependencies,
- The Crew takes care of ordering and context passing,
- The tools are validated and executed as structured function calls.

This example may be simple, but it demonstrates a complete multi-agent orchestration flow:

- Dynamic reasoning + tool use (via ReAct),
- Structured chaining of specialized agents,
- Clear, interpretable steps with reproducible outputs.

And the beauty is that you can now scale this pattern up to 3, 5, or 20 agents, each with unique tools, tasks, and dependencies.

Conclusion

In this tutorial, we implemented a fully functional multi-agent system from scratch, using nothing but Python, a local LLM via LiteLLM, and a few well-scoped abstractions.

We covered:

- Why the multi-agent pattern matters in real-world AI systems, especially when you want modularity, role specialization, and interpretability.
- How this approach builds on top of ReAct and Planning, but extends the paradigm to teams of agents collaborating toward a shared goal.
- How to define a Tool abstraction that lets agents introspect and call functions dynamically, with type-safe validation and structured invocation.

- How to build an Agent class that can reason, act, use tools, and pass outputs downstream—using a clean, prompt-driven ReAct loop.
- How to wire multiple agents together with the Crew class, automatically resolving dependencies and running agents in the correct topological order.
- How to simulate a full pipeline, with clear task delegation (Agent A → Agent B), structured tool usage, and context propagation between agents.

All without touching a single orchestration library like LangChain, CrewAI, AutoGen, or LlamaIndex.

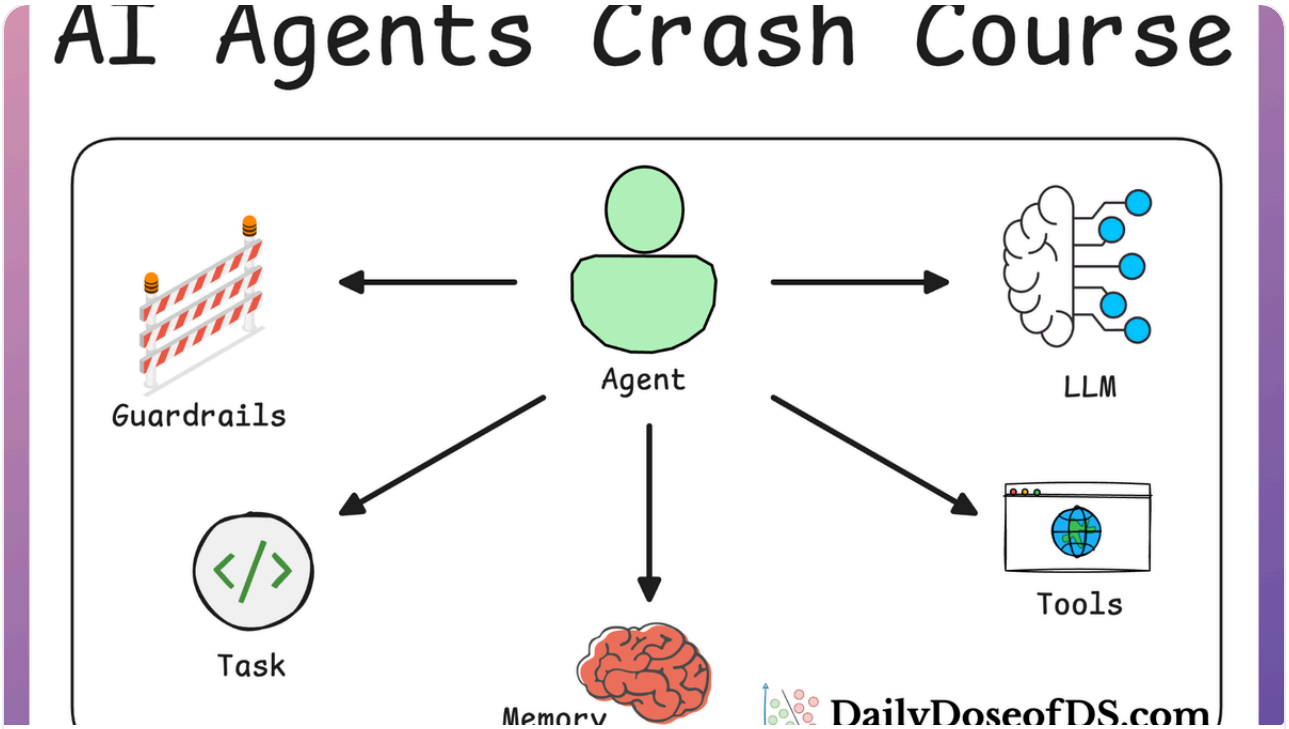
This kind of low-level yet transparent architecture is perfect for:

- Writing research agents that gather, evaluate, and summarize documents.
- Building content generation workflows where each agent handles a phase (e.g., idea → draft → polish).
- Constructing sequential chains for data extraction, transformation, and reporting.
- Teaching and debugging agentic reasoning without black-box frameworks.

Yes, this is still a simple, synchronous setup, and the tools are wired via decorators and function calls, but that's the point. It helps you see under the hood and understand how these larger frameworks are working.

And once you're fluent with these patterns, scaling them becomes a matter of design, not guesswork. You can confidently move toward agent routing, dynamic task delegation, or even distributed agent scheduling.

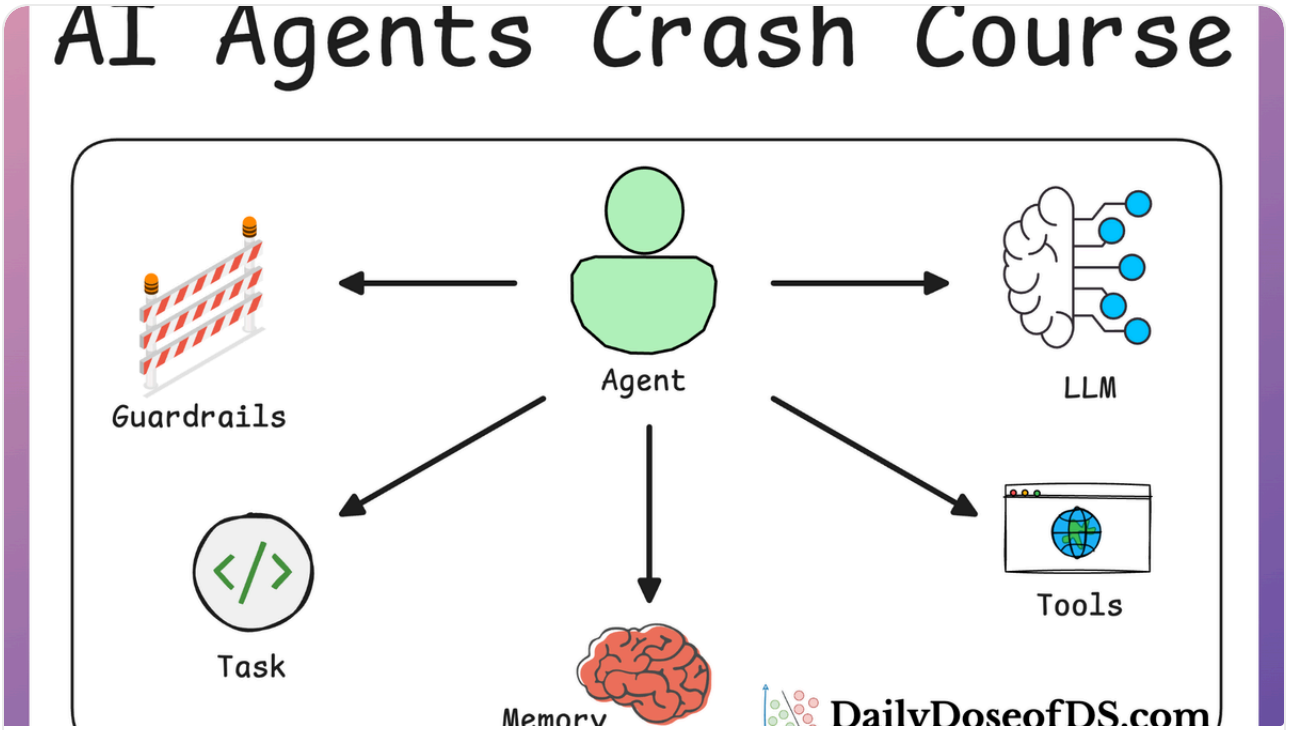
Do not forget to read the from scratch implementation of the ReAct and Planning patterns we covered earlier:



AI Agents Crash Course—Part 10 (With Implementation)

A deep dive into ReAct patterns and implementing it from scratch.

Daily Dose of Data Science • Avi Chawla

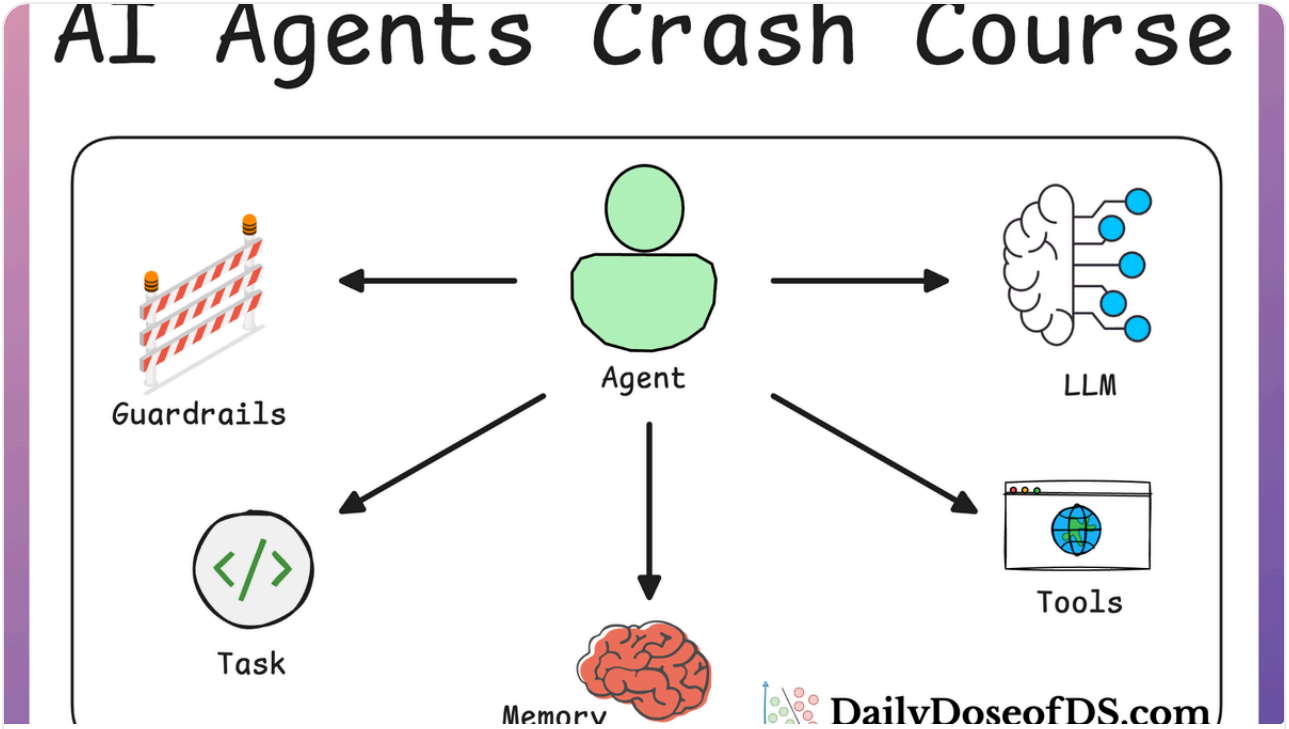


AI Agents Crash Course—Part 11 (With Implementation)

A deep dive into Planning patterns and implementing it from scratch.

Daily Dose of Data Science • Avi Chawla

Also, read the next part of this crash course here:



10 Practical Steps to Improve Agentic Systems (Part A)

AI Agents Crash Course—Part 13 (with implementation).

Daily Dose of Data Science • Avi Chawla

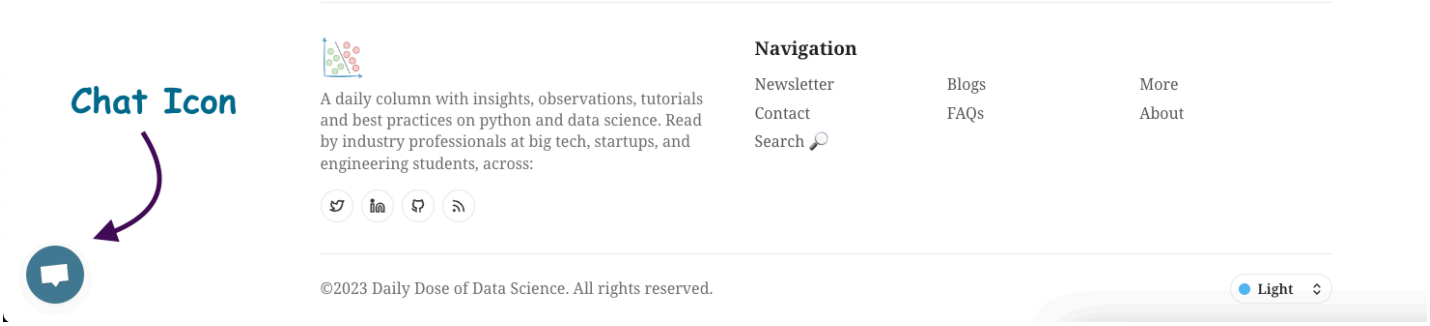
As always, thanks for reading!

Any questions?

Feel free to post them in the comments.

Or

If you wish to connect privately, feel free to initiate a chat here:



Connect via chat

Agents LLMs AI Agent Crash Course

Share this article

Read next

MCP Jun 15, 2025 • 22 min read

The Full MCP Blueprint: Building a Full-Fledged MCP Workflow using Tools, Resources, and Prompts

Model context protocol crash course—Part 4.

Avi Chawla, Akshay Pachaar

Agents Jun 8, 2025 • 20 min read

The Full MCP Blueprint: Building a Custom MCP Client from Scratch

Model context protocol crash course—Part 3.

Avi Chawla, Akshay Pachaar

Agents Jun 1, 2025 • 22 min read

The Full MCP Blueprint: Background, Foundations, Architecture, and Practical Usage (Part B)

Model context protocol crash course—Part 2.

Avi Chawla, Akshay Pachaar

A daily column with insights, observations, tutorials and best practices on python and data science. Read by industry professionals at big tech, startups, and engineering students, across:

Navigation

Sponsor Newsletter More Contact