Apr 20, 2025

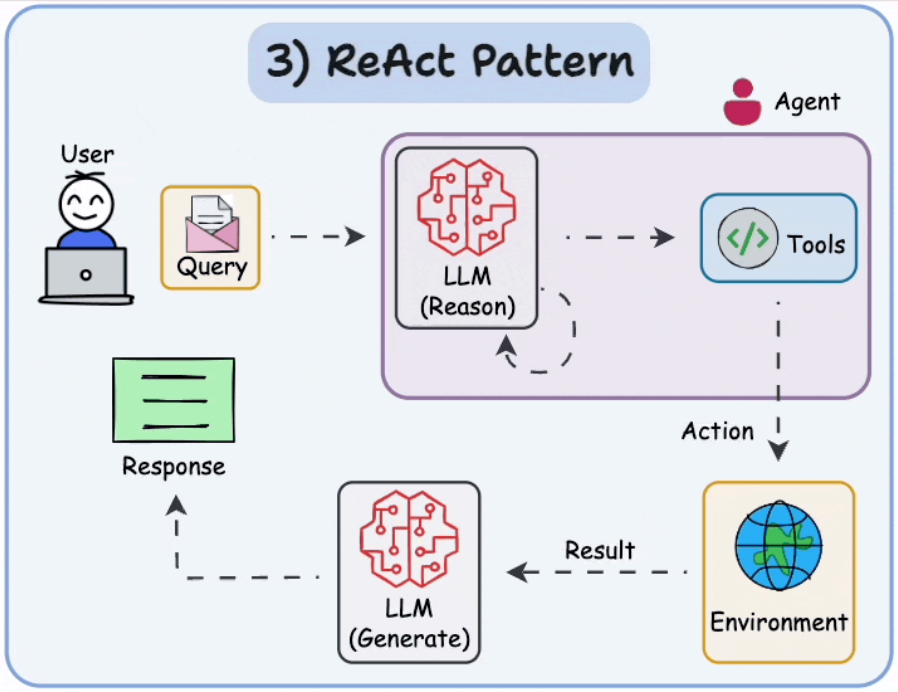
Implementing Planning Agentic Pattern From Scratch

AI Agents Crash Course—Part 11 (with implementation).

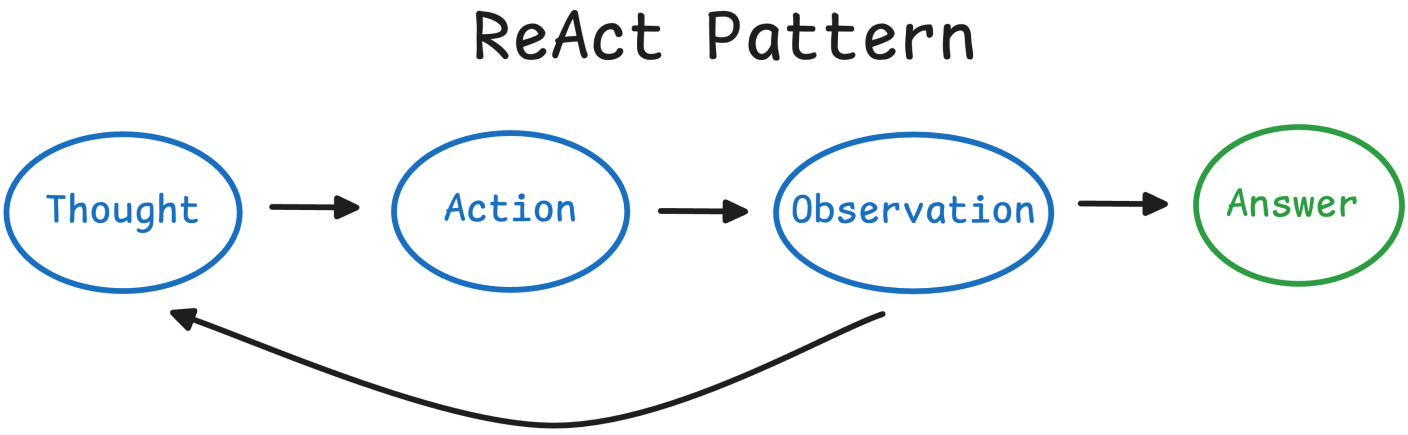
Avi Chawla, Akshay Pachaar

Introduction

In the previous part of this crash course, we explored the ReAct pattern, a dynamic approach where an AI agent interleaves reasoning and acting steps to solve a problem.

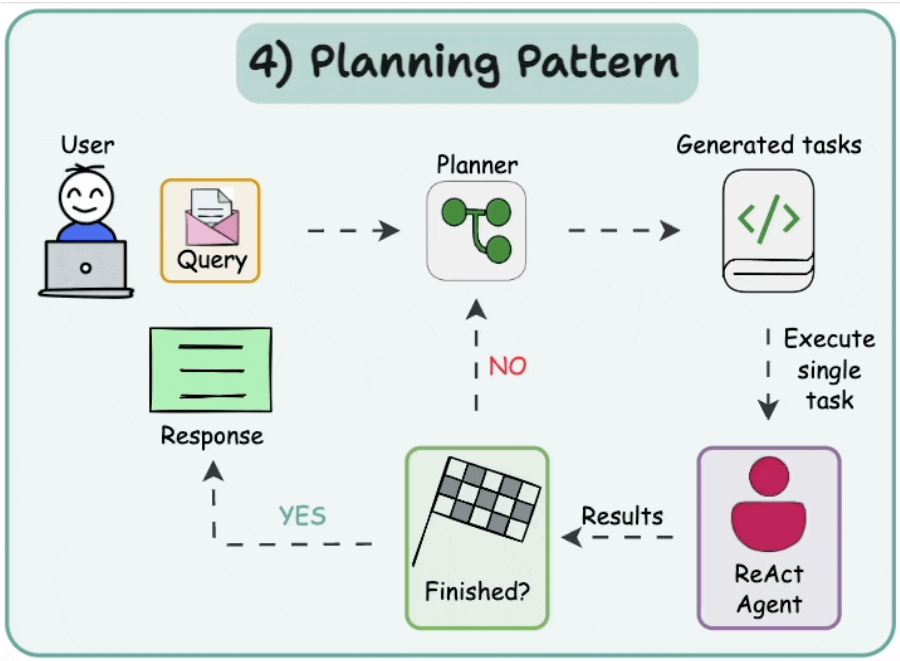


The ReAct agent thinks step-by-step (producing Thoughts) and takes actions (using tools) in a loop, adjusting its strategy as new information comes in.



This reactive approach (Reasoning + Acting) enables an agent to adapt on the fly, combining chain-of-thought reasoning with external tool use to handle complex queries.

However, ReAct isn't the only way to organize an agent's thinking. Another powerful approach is the Planning pattern, which is the focus of this part.



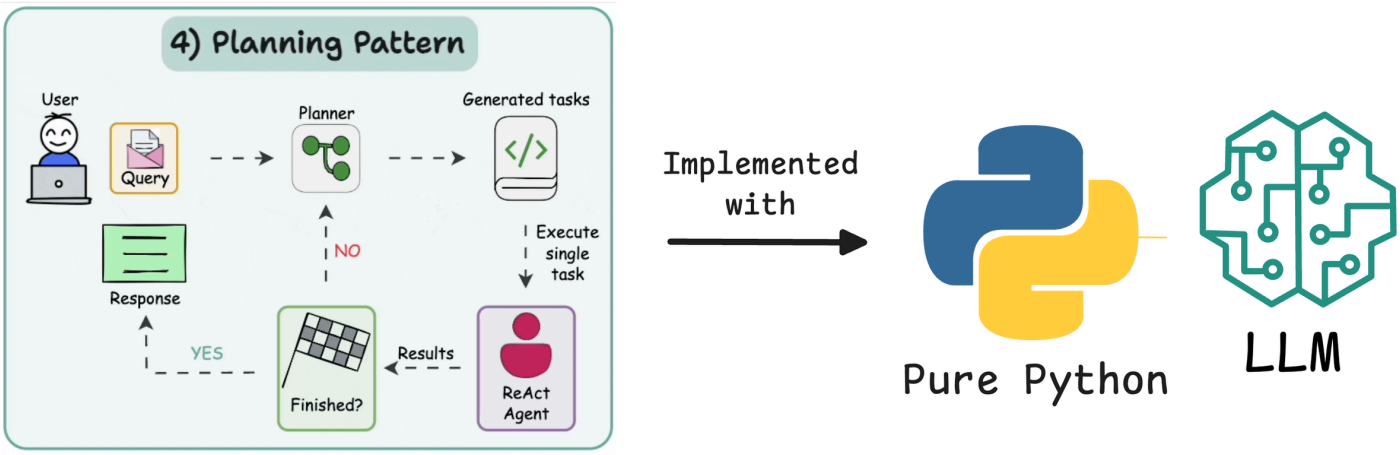
The Planning pattern has the agent plan out a solution strategy before executing any actions.

Instead of diving directly into step-by-step reasoning and tool use as ReAct does, a Planning agent will first take a step back and formulate a high-level plan or "roadmap" for the task.

Only after this plan is laid out does the agent proceed to carry out each step in the plan (using tools or other means), and finally, it produces the answer.

This approach enhances an LLM agent's ability to handle complex tasks and decisions by first laying out a plan and then acting on it.

In this article, we'll demystify that process by building a Planning agent from scratch using only Python and an LLM.



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 Planning pattern, design an agent loop that interleaves planning and tool usage, and implement multiple tools that the agent can invoke.

Moreover, we'll discuss why planning matters in AI agents, how the Planning pattern works in practice, what recent research says about its benefits (and challenges), and how it compares to other patterns like ReAct. By the end, you'll understand when and how to use Planning in your own AI agent applications.

The goal is to help you understand both the theory and implementation of the Planning pattern in AI Agents.

By the end, you'll have a working agent and a clear picture of how frameworks like CrewAI leverage Planning internally.

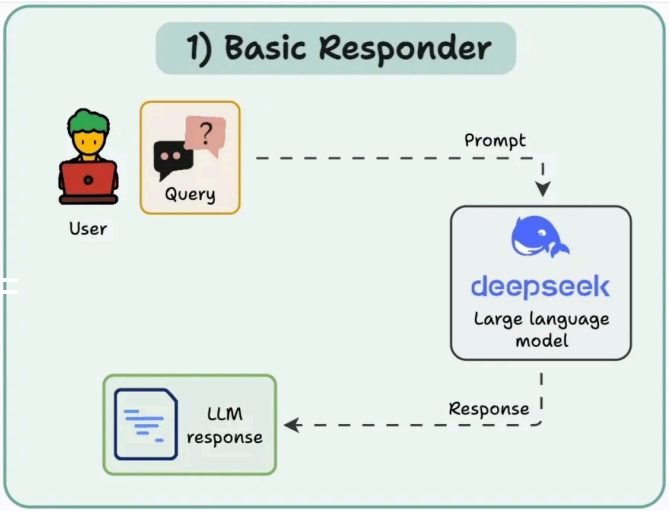
Let's begin!

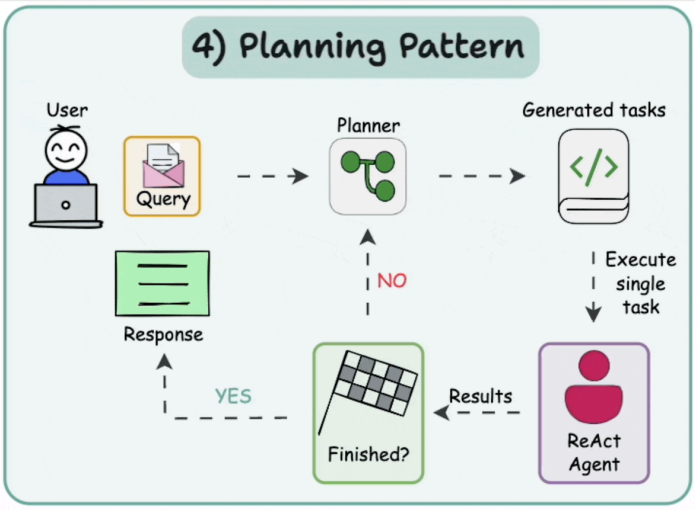
Why planning?

Why do we need an explicit planning phase for an AI agent?

The short answer is that some problems are too complex to tackle one step at a time without a global strategy.

The ReAct paradigm excels at reactive decision-making. The agent looks at the current state, decides on one action, executes it, then observes the result and repeats.





This works well for straightforward or highly uncertain tasks, but it can fall short for multi-step problems that benefit from foresight.

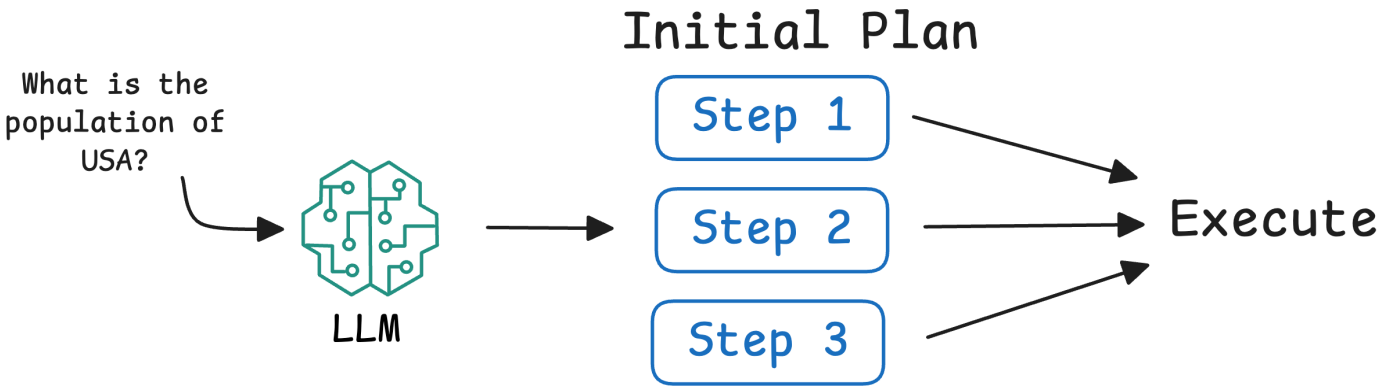
Consider an intuitive example: finding a pen versus making a cup of coffee.

If you ask an agent to simply get a pen from your desk, a reactive approach is fine. It can try one location, see if the pen is there, and if not, try the next.

In contrast, if you ask for a flat white coffee, the agent must handle several sub-tasks (boil water, grind beans, froth milk, etc.) and possibly adjust the plan as conditions change.

So put it into perspective, if ReAct is more suitable for tasks like getting a pen from the desk, then Plan & Solve is better suited for tasks like making a cup of flat white.

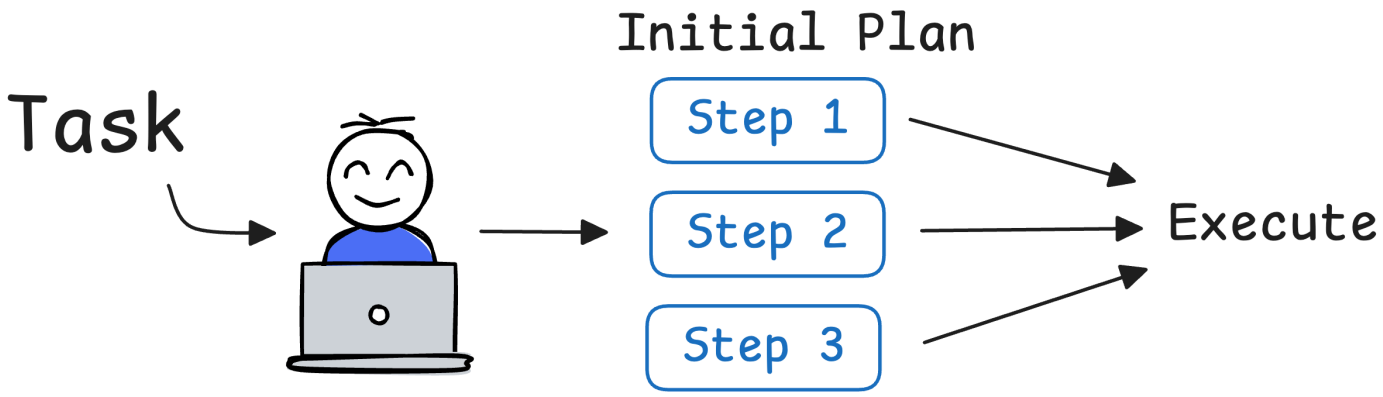
You need to plan, and the plan might change during the process (for example, if you open the fridge and find no milk, you would add 'buy milk' as a step.



In other words, the more elaborate the task, the more an agent needs a game plan, much like a human would outline an approach before tackling a big project.

For instance, think of how you might tackle a complex question yourself: often you'll outline a plan or break the problem into sub-tasks before doing any detailed work.

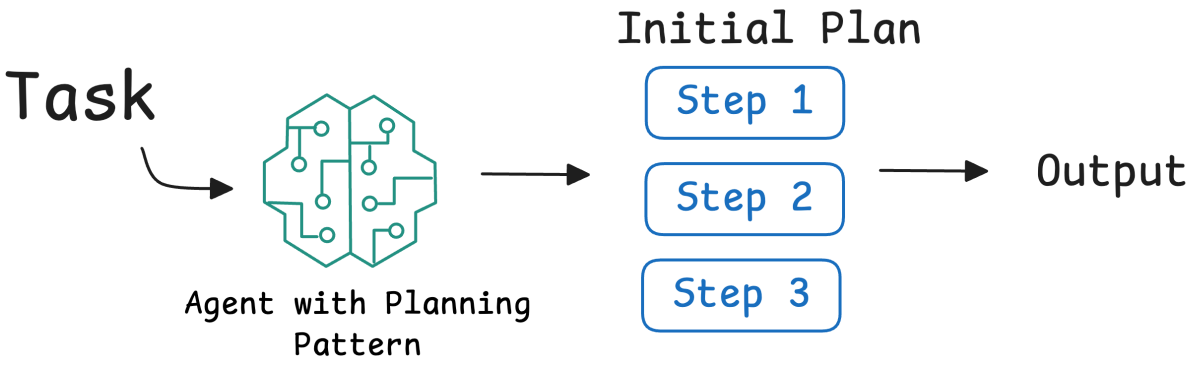
By subdividing the task and outlining objectives, the agent gains a global view of the problemby subdividing the task and outlining objectives.

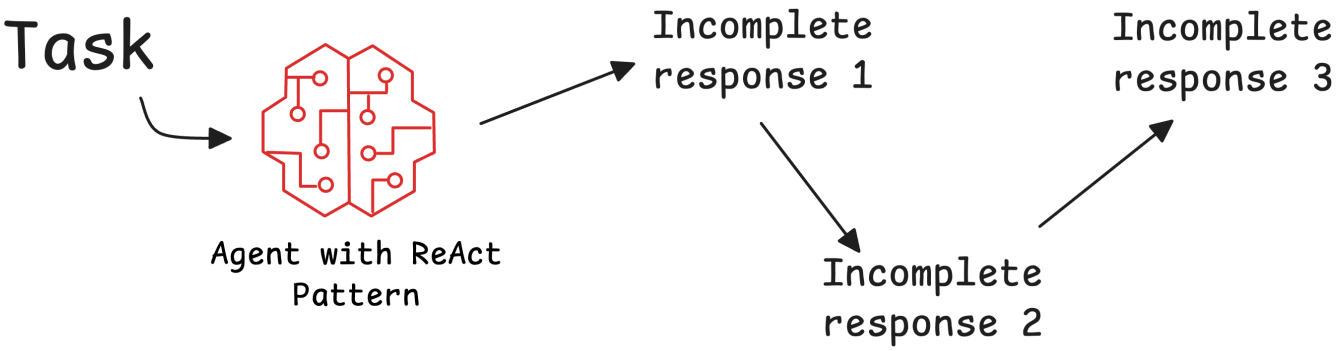


This can make its approach more strategic and efficient.

The Planning pattern is especially useful when the problem involves multiple distinct steps or pieces of information that need to be gathered and combined.

In such cases, a ReAct agent might still eventually find the answer, but it could meander or repeat work along the way because it decides actions one at a time.





A Planning agent, on the other hand, will attempt to identify all the necessary steps up front and is less likely to get sidetracked.

Let's compare the two patterns in a nutshell, then dive into the literature (research) review about Planning pattern and finally the implementation.

- ReAct:
  - Ideology → "Think, act, observe, repeat."
  - The agent decides on the next action based on the current context, which means it can adapt if something unexpected comes up.
  - This is great for open-ended or uncertain environments, but the agent might not always take the most direct route to the solution since it's not

planning far ahead.

- Planning:
    - Ideology → “Think ahead, then execute.”
    - The agent first reasons about the overall problem and sketches out a plan (a sequence of steps) before doing anything. This ensures a coherent strategy from start to finish.
    - It shines when the path to the solution can be anticipated in advance (for example, answering a question that clearly breaks down into a few sub-questions).
    - Planning can improve reliability on complex tasks by reducing the chances of forgetting a sub-task or going in circles.
    - However, if the initial plan is flawed or if something unexpected occurs, a Planning agent might need to revise its plan..

In summary, the Planning pattern matters because it encourages structured problem solving.

It’s best used when you have tasks that can be broken into clear sub-tasks or when you want the agent to think globally about a problem before getting its “hands dirty” with tools or calculations.

Many real-world scenarios benefit from this. Let’s look at a few common scenarios:

- Writing a report:
    - This involves researching information, creating an outline, writing and editing sections, and compiling everything.
    - Without planning, an AI might jump into writing and easily go off-topic or miss key parts.
    - A Planning agent would first break down the task (e.g. outline sections, gather facts for each section, draft each section), ensuring it covers everything in a logical order.
- Planning a trip
    - Booking a vacation has multiple steps (decide on destination, find flights, reserve hotels, plan activities, etc.).

    - A reactive agent might handle one step at a time but could lose track of the overall itinerary.
    - With a plan, the agent can enumerate all sub-tasks and tackle them methodically, keeping the final goal in focus.
- Creating an outline or project plan
    - This is essentially planning as the end goal.
    - It requires identifying all components of a task upfront. An agent that explicitly plans will perform better here than one that improvises step-by-step.

Another reason planning helps is to avoid missing important steps.

Researchers have observed that even when prompting an LLM to reason stepwise (chain-of-thought), it may skip critical steps or produce a flawed solution path.

For example, standard zero-shot chain-of-thought often suffers from “missing-step errors”:

> 💡 On a side note: Zero-shot chain-of-thought (CoT) refers to a prompting technique where you ask the LLM to reason step-by-step without giving it any examples beforehand. For instance, if you prompt it with “What’s 17 × 4? Let’s think step by step,” it will try to show its reasoning even though it wasn’t shown how. This can be powerful—but also brittle. Without examples to guide it, the model might skip steps, make leaps, or hallucinate logic, especially on more complex tasks.

Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, Ee-Peng Lim

Abstract

Large language models (LLMs) have recently been shown to deliver impressive performance in various NLP tasks. To tackle multi-step reasoning tasks, Few-shot chain-of-thought (CoT) prompting includes a few manually crafted step-by-step reasoning demonstrations which enable LLMs to explicitly generate reasoning steps and improve their reasoning task accuracy. To eliminate the manual efforts, Zero-shot-CoT concatenates the target problem statement with *"Let's think step by step"* as an input prompt to LLMs. Despite the success of Zero-shot-CoT, it still suffers from three pitfalls: calculation errors, missing-step errors, and semantic misunderstanding errors. To address the missing-step errors, we propose Plan-and-Solve (PS) Prompting. It consists of two components: first, devising a plan to divide the entire task into smaller subtasks, and then carrying out the subtasks according to the plan. To address the calculation errors and improve the quality of generated reasoning steps, we extend PS prompting with more detailed instructions and derive PS+ prompting. We evaluate our proposed prompting strategy on ten datasets across three reasoning problems. The experimental results over GPT-3 show that our proposed zero-shot prompting consistently outperforms Zero-shot-CoT across all datasets by a large margin, is comparable to or exceeds Zero-shot-Program-of-Thought Prompting, and has comparable performance with 8-shot CoT prompting on the math reasoning problem. The code can be found at https://github.com/AGI-Edgerunners/Plan-and-Solve-Prompting.

By contrast, if we ask the model to devise a plan first and then execute it, we force it to think through the entire solution path, reducing the chance of skipping steps.

Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, Ee-Peng Lim

Abstract

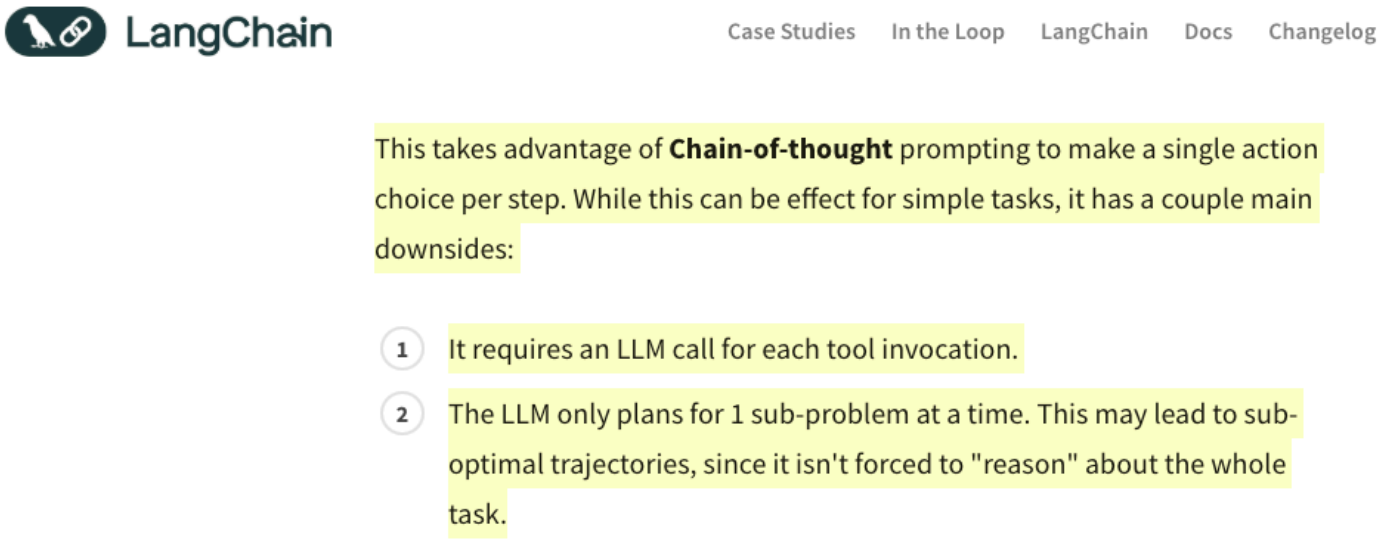
Large language models (LLMs) have recently been shown to deliver impressive performance in various NLP tasks. To tackle multi-step reasoning tasks, Few-shot chain-of-thought (CoT) prompting includes a few manually crafted step-by-step reasoning demonstrations which enable LLMs to explicitly generate reasoning steps and improve their reasoning task accuracy. To eliminate the manual efforts, Zero-shot-CoT concatenates the target problem statement with *"Let's think step by step"* as an input prompt to LLMs. Despite the success of Zero-shot-CoT, it still suffers from three pitfalls: calculation errors, missing-step errors, and semantic misunderstanding errors. To address the missing-step errors, we propose Plan-and-Solve (PS) Prompting. It consists of two components: first, devising a plan to divide the entire task into smaller subtasks, and then carrying out the subtasks according to the plan. To address the calculation errors and improve the quality of generated reasoning steps, we extend PS prompting with more detailed instructions and derive PS+ prompting. We evaluate our proposed prompting strategy on ten datasets across three reasoning problems. The experimental results over GPT-3 show that our proposed zero-shot prompting consistently outperforms Zero-shot-CoT across all datasets by a large margin, is comparable to or exceeds Zero-shot-Program-of-Thought Prompting, and has comparable performance with 8-shot CoT prompting on the math reasoning problem. The code can be found at https://github.com/AGI-Edgerunners/Plan-and-Solve-Prompting.

PDF
Cite
Search
Video
Fix data

In essence, planning imposes a structure that improves thoroughness.

Finally, planning can lead to more optimal decision sequences.

A ReAct agent, by design, only plans one action at a time and doesn't necessarily look ahead beyond the immediate next step. This can yield suboptimal trajectories for complex tasks.

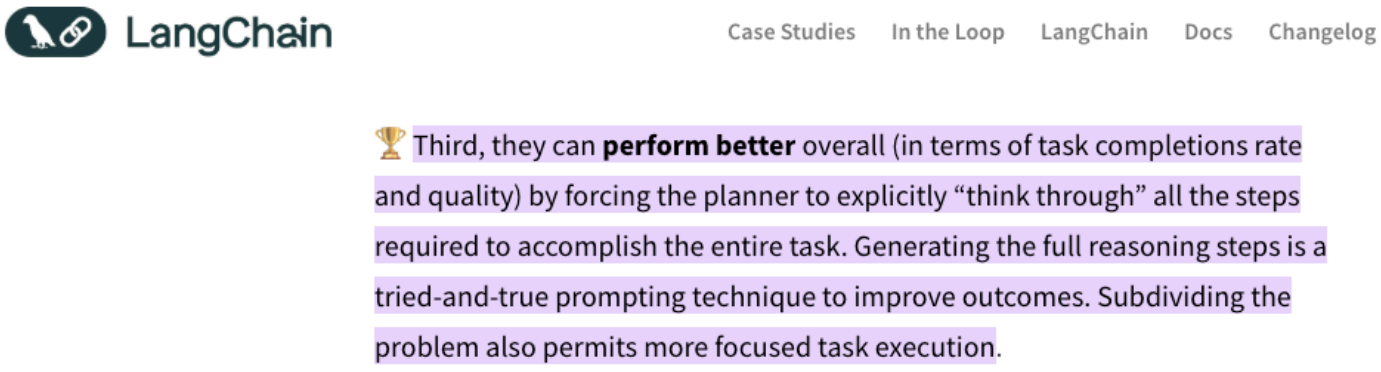


It's like walking through a maze with no map: you decide each turn as you go.

A planning agent, on the other hand, sketches a rough map of the maze first.

This doesn't guarantee a perfect route, but it encourages global reasoning about the problem.

Empirically, forcing the model to articulate a full solution strategy often leads to better outcomes:



Subdividing a problem into sub-tasks also permits more focused execution of each part , as mentioned above.

In summary, planning is beneficial when:

- The problem naturally breaks into sub-tasks or stages.
- The agent might otherwise forget requirements or context over many steps.
- A greedy step-by-step approach might get stuck or wander off-course.
- You want the agent's solution path to be interpretable and verifiable (having an explicit plan gives a blueprint).

ReAct gets us a long way, but when task complexity grows, a plan can be the agent's compass.

Formalizing the Planning Pattern

How does the Planning pattern actually work?

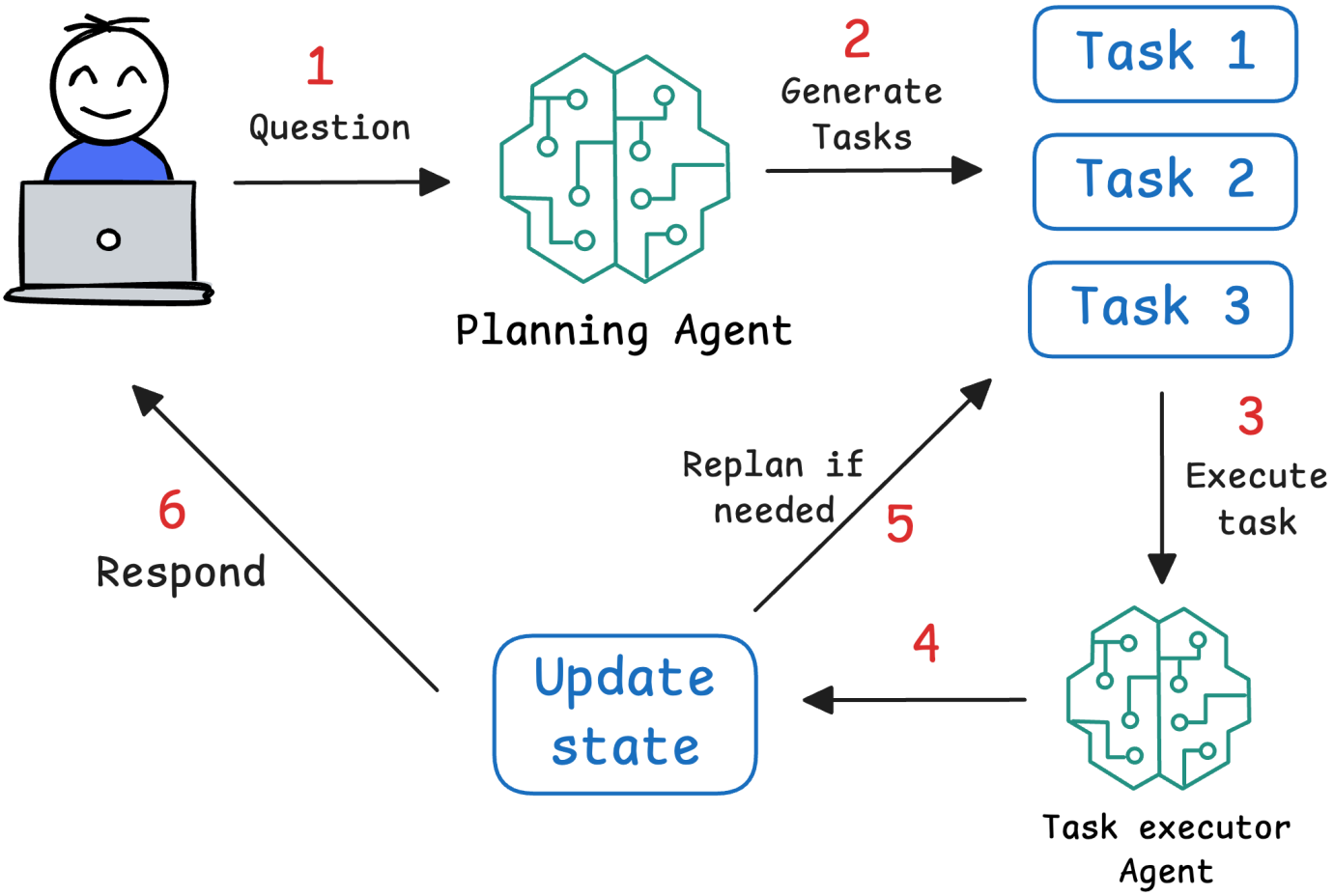
At a high level, a Planning agent introduces a two-phase approach to problem solving—**first devise a plan, then execute it.**

This "plan first, execute later" strategy cleanly separates what to do from how to do it.

In practice, the agent will spend some time thinking ahead (perhaps using the LLM to outline steps in natural language), and only then carry out those steps one by one .

By structuring the agent's loop this way, we gain a more organized and often more efficient workflow.

Here's an illustrative workflow for the Planning pattern.



- First, the agent (LLM “Planner”) analyzes the user request and generates a list of sub-tasks (a plan).
- Next, a single-task executor agent (which could be the same LLM or a different one) tackles each sub-task in sequence, possibly using tools (indicated by the wrench icon) to complete them.
- After each step, the agent updates its state with results. If new information requires it, the agent can revise the plan (re-planning) before proceeding.
- Finally, once all steps are done, the agent produces a response to the user. This separation of concerns allows clear long-term planning (left boxes) combined with iterative action (right box).

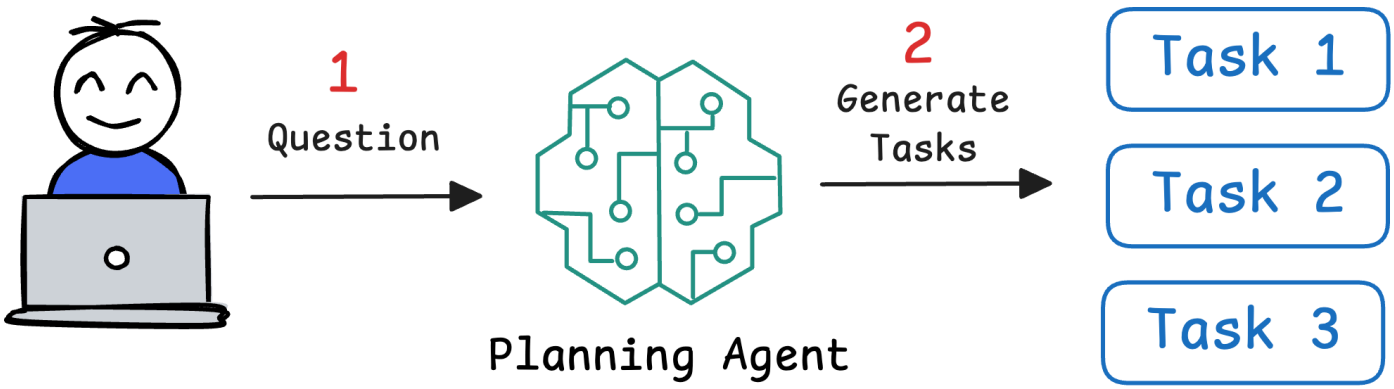
More formally, we can break down the Planning pattern into distinct stages or components:

#1) Planning Phase

The agent examines the user's query or goal and formulates a high-level plan.

This often means decomposing the problem into subgoals or sub-tasks.

The LLM might be prompted to output an ordered list of steps needed to solve the problem.



For example, given a question, the agent (as a planner) might respond with: "1. Do X; 2. Find Y; 3. Then calculate Z."

Here, the agent is not yet solving anything – it's creating a roadmap.

It analyzes the objective and decides on a strategy before diving into execution.

The plan can be in plain language or a structured format (a list, JSON, etc.), as long as subsequent steps can interpret it.

Internally, this phase is often a single prompt/response turn with the LLM acting as a "planner."

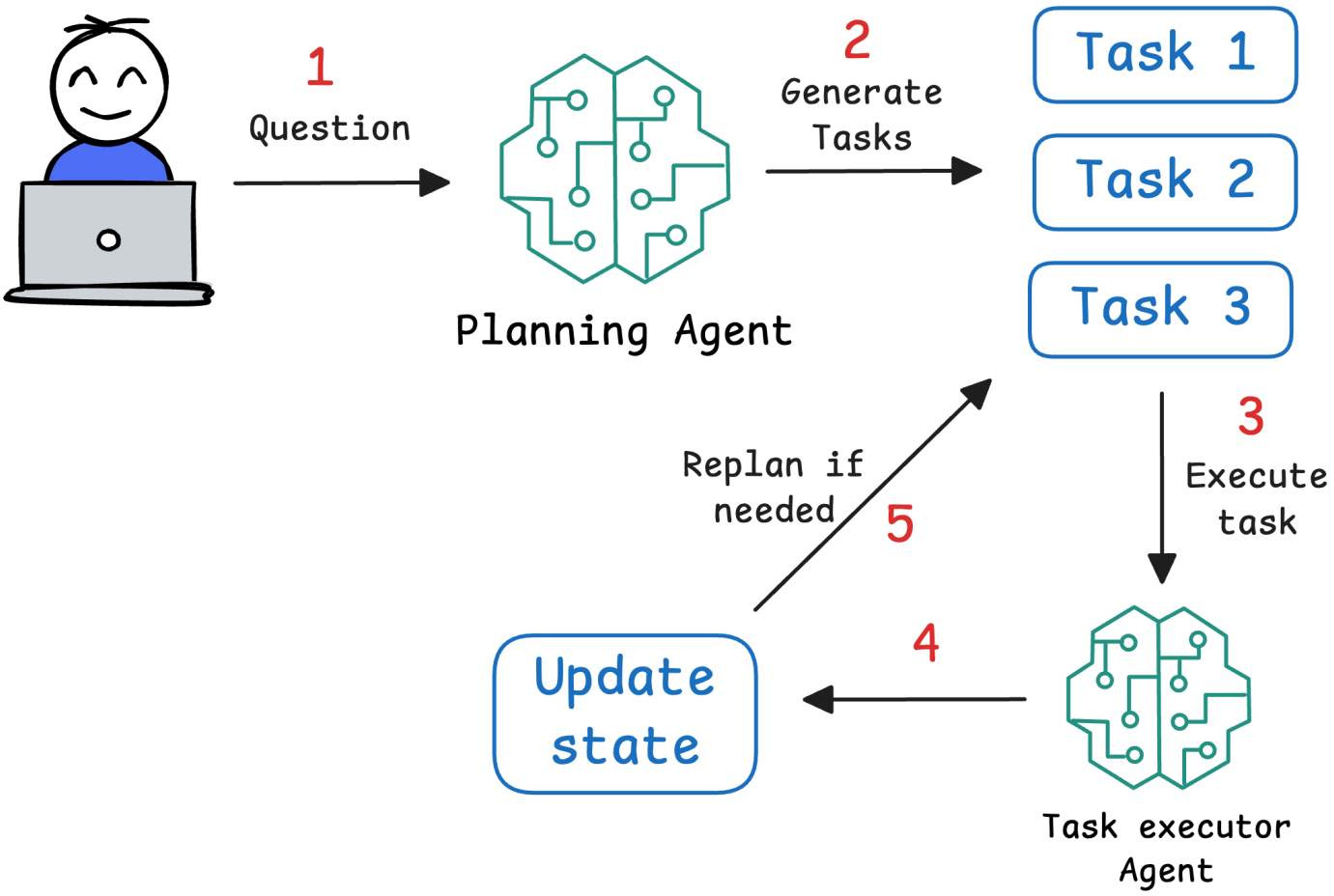
#2) Execution Phase

Once a plan is in hand, the agent enters a loop to execute each sub-task iteratively.

It takes the first step from the plan, carries it out, and obtains a result (Observation). Then it moves to the next step, and so on.

During execution, the agent might call external tools or APIs as needed for each sub-task (just as a ReAct agent would during its reasoning process).

Importantly, the agent uses the plan as a guide for what to do next, rather than coming up with new actions from scratch.



This continues until all planned steps are completed.

Throughout this phase, the agent aggregates results or intermediate findings in its working memory or scratchpad.

For instance, if step 2 required a web search, the retrieved info is stored for later use in step 5. Each iteration can be thought of as the agent focusing on a single sub-problem (hence “single-task agent” in the diagram) guided by the plan.

#3) Aggregation and Response

After executing the necessary steps, the agent collects all the relevant outputs and composes the final answer or result.

This might be as simple as taking the last step’s outcome as the answer, or as involved as synthesizing information from multiple sub-tasks.

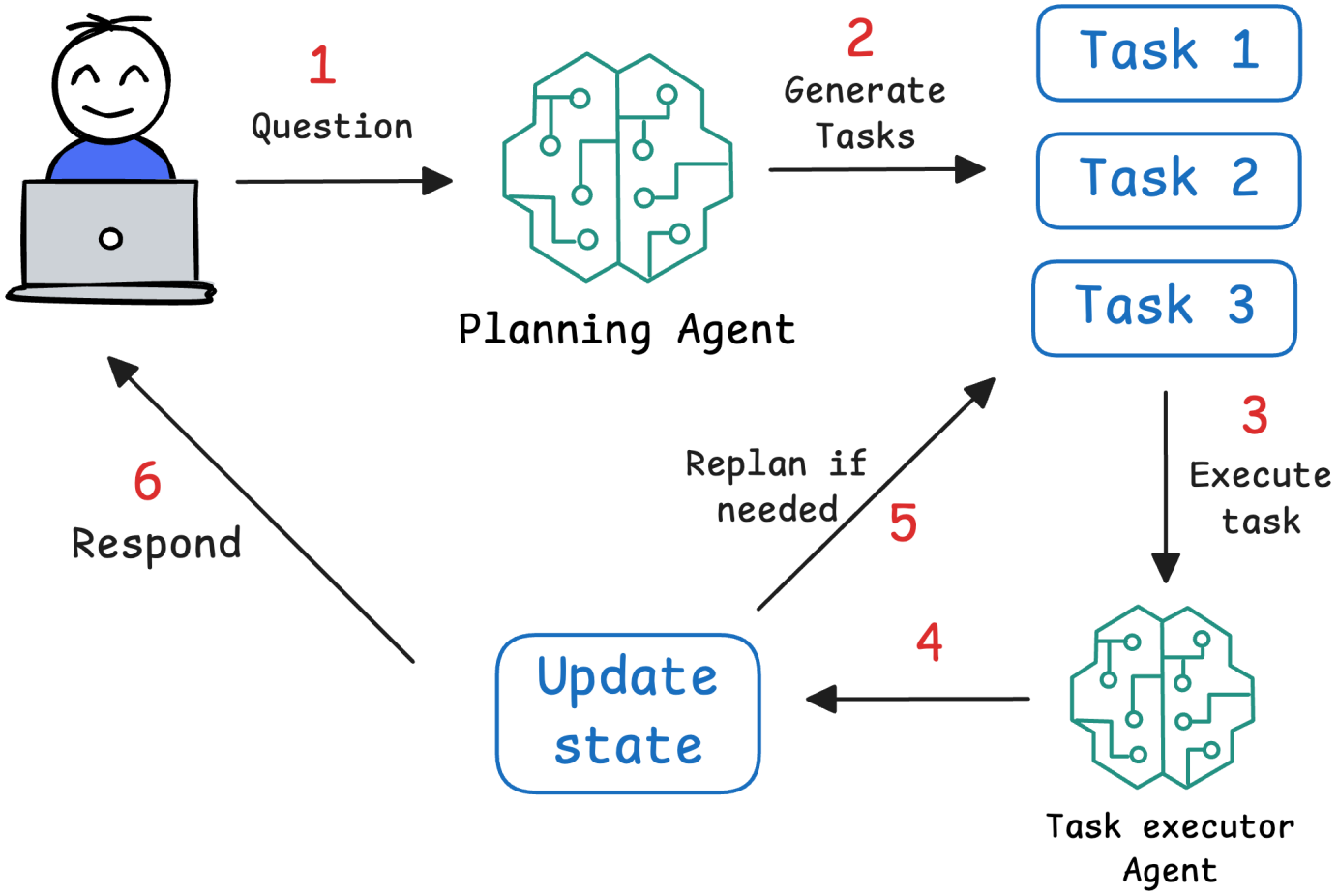
Because the agent followed a structured plan, it’s straightforward at this stage to aggregate the outcomes (they’re often already organized by step).

The final response is then presented to the user (or passed to whatever system is consuming the agent’s result).

An optional but important aspect of the Planning pattern is feedback and iteration.

Unlike a rigid traditional program, an AI agent can detect when something didn’t go as expected and adjust.

During execution, if a sub-task fails or yields an unexpected result, the agent can decide to revisit the plan.



In our workflow diagram, after step 4 the agent could decide to insert a new step or modify future steps (this corresponds to the “Replan” box, labeled 5).

For example, if step 3 “Find information on Y” came up empty, the agent might pause and re-plan the remaining steps to find an alternative route.

Ideally, the agent will then continue execution with the revised plan. In practice, this can be implemented by prompting the LLM (in planner mode) again, giving it the partial progress and asking for an updated plan.

This ensures the agent isn’t stuck following a bad initial plan when circumstances change.

To ground this in a concrete example, imagine an agent asked: “How many prime numbers are there between 1,000 and 1,100, and what are they?” A ReAct agent might start calculating or searching immediately in a loop.

A Planning agent, however, might first output a plan like:

```
"""
Plan:
1. Identify the range (1000 to 1100).
2. Find all prime numbers in this range.
3. Count those primes.
4. Return the count and list of primes.
"""
```

Then it would execute step 2 (maybe by calling a prime-checking tool for each number or an API to fetch primes in range), store the list, execute step 3 (count the list length), and finally output the formatted answer for step 4.

The benefit is that the agent (and the developer observing it) knows exactly what the intermediate objectives are at each point, and it won't forget to output both the count and the list because those were in the plan.

In summary, the Planning pattern means an agent will "think before it acts" – it dedicates part of its reasoning to lay out a solution path, then follows that path.

This approach echoes the formal Plan-and-Solve strategies proposed in research, which consist of "first, devising a plan to divide the entire task into smaller subtasks, and then carrying out the subtasks according to the plan."

Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, Ee-Peng Lim

Abstract

Large language models (LLMs) have recently been shown to deliver impressive performance in various NLP tasks. To tackle multi-step reasoning tasks, Few-shot chain-of-thought (CoT) prompting includes a few manually crafted step-by-step reasoning demonstrations which enable LLMs to explicitly generate reasoning steps and improve their reasoning task accuracy. To eliminate the manual efforts, Zero-shot-CoT concatenates the target problem statement with *"Let's think step by step"* as an input prompt to LLMs. Despite the success of Zero-shot-CoT, it still suffers from three pitfalls: calculation errors, missing-step errors, and semantic misunderstanding errors. To address the missing-step errors, we propose Plan-and-Solve (PS) Prompting. It consists of two components: first, devising a plan to divide the entire task into smaller subtasks, and then carrying out the subtasks according to the plan. To address the calculation errors and improve the quality of generated reasoning steps, we extend PS prompting with more detailed instructions and derive PS+ prompting. We evaluate our proposed prompting strategy on ten datasets across three reasoning problems. The experimental results over GPT-3 show that our proposed zero-shot prompting consistently outperforms Zero-shot-CoT across all datasets by a large margin, is comparable to or exceeds Zero-shot-Program-of-Thought Prompting, and has comparable performance with 8-shot CoT prompting on the math reasoning problem. The code can be found at https://github.com/AGI-Edgerunners/Plan-and-Solve-Prompting.

By structuring the agent's workflow into a plan phase and an execution phase, we gain better organization, clarity, and often better results for complex tasks.

Literature Insights

The idea of separating planning and execution in AI agents has gained significant attention in recent research and developer communities.

A number of research papers have explored the benefits (and challenges) of the Planning pattern and related techniques.

Here we highlight a few notable insights from the literature:

#1) Plan-and-Solve Prompting

This paper introduced Plan-and-Solve (PS) prompting, which is essentially the same two-step idea we described: first have the model outline a plan, then have it solve according to that plan:

Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, Ee-Peng Lim

Abstract

Large language models (LLMs) have recently been shown to deliver impressive performance in various NLP tasks. To tackle multi-step reasoning tasks, Few-shot chain-of-thought (CoT) prompting includes a few manually crafted step-by-step reasoning demonstrations which enable LLMs to explicitly generate reasoning steps and improve their reasoning task accuracy. To eliminate the manual efforts, Zero-shot-CoT concatenates the target problem statement with *"Let's think step by step"* as an input prompt to LLMs. Despite the success of Zero-shot-CoT, it still suffers from three pitfalls: calculation errors, missing-step errors, and semantic misunderstanding errors. To address the missing-step errors, we propose Plan-and-Solve (PS) Prompting. It consists of two components: first, devising a plan to divide the entire task into smaller subtasks, and then carrying out the subtasks according to the plan. To address the calculation errors and improve the quality of generated reasoning steps, we extend PS prompting with more detailed instructions and derive PS+ prompting. We evaluate our proposed prompting strategy on ten datasets across three reasoning problems. The experimental results over GPT-3 show that our proposed zero-shot prompting consistently outperforms Zero-shot-CoT across all datasets by a large margin, is comparable to or exceeds Zero-shot-Program-of-Thought Prompting, and has comparable performance with 8-shot CoT prompting on the math reasoning problem. The code can be found at https://github.com/AGI-Edgerunners/Plan-and-Solve-Prompting.

Their motivation was to fix errors that LLMs made in zero-shot chain-of-thought reasoning.

By forcing the LLM to explicitly list out the solution steps before executing, they found the model was much less likely to skip necessary steps.

The results were impressive – across a variety of reasoning tasks, zero-shot Plan-and-Solve "consistently outperforms Zero-Shot-CoT… by a large margin," even matching the accuracy of some few-shot methods (where few examples were given).

In other words, planning gave a big boost in problem-solving quality without any extra training; it's purely a prompting change. This validates the intuition that an upfront plan helps guide the model to better solutions.

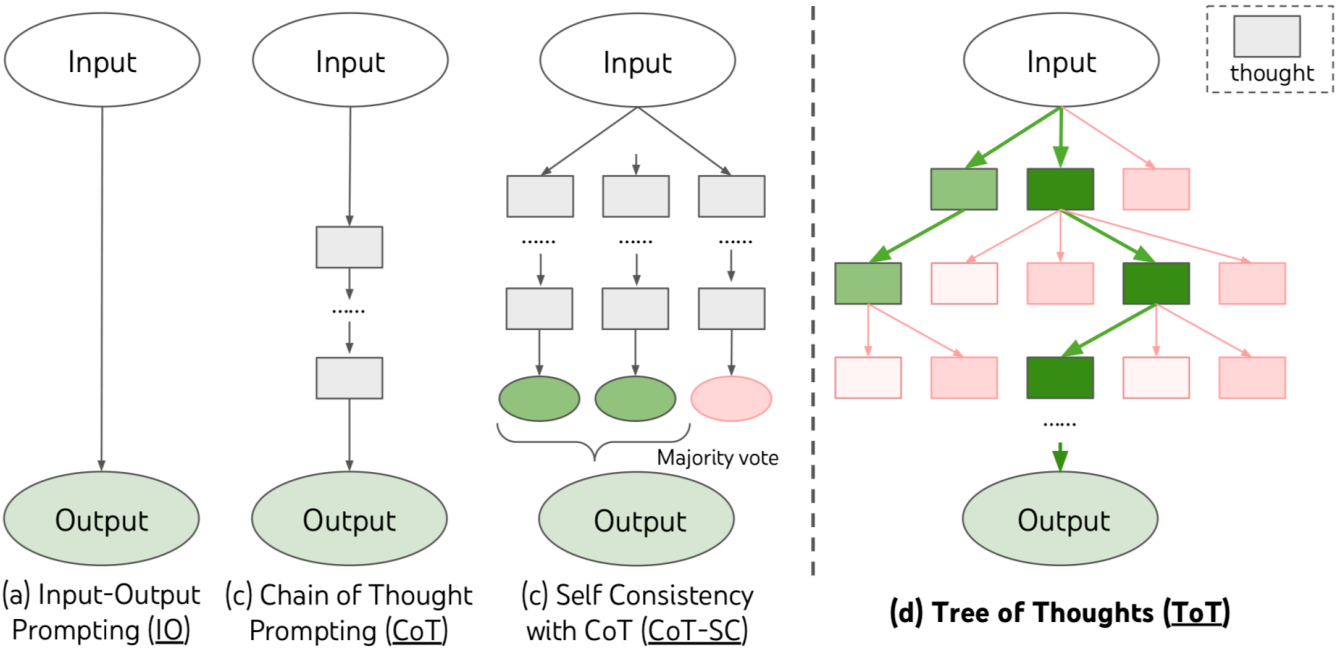
#2) Tree-of-Thought (ToT)

One limitation of a linear plan is that it's fixed since the agent commits to a single path.

What if the agent could explore many possible plans or chains of thought and choose the best?

Tree-of-Thought prompting, introduced in 2023 (and highlighted by researchers at Google DeepMind), expands on the planning idea by allowing branching and backtracking.

Instead of one linear sequence, the agent generates a tree of possible moves or steps, evaluating outcomes along the way:



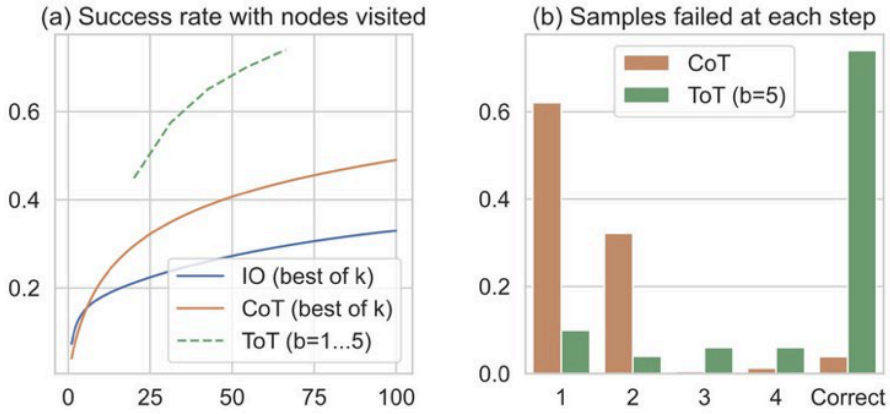
It's analogous to how a human might mentally try different approaches to a puzzle and abandon the ones that look unpromising.

The agent can thus consider multiple reasoning paths and iteratively deepen the most promising ones, even revisiting earlier decisions if needed.

This approach dramatically improved performance on certain problems requiring non-trivial planning or search.

For example, on a puzzle task (finding 24 using four numbers), GPT-4 with normal chain-of-thought solved only 4% of cases, whereas with Tree-of-Thought prompting it solved 74%:

Method	Success
IO prompt	7.3%
CoT prompt	4.0%
CoT-SC (k=100)	9.0%
ToT (ours) (b=1)	45%
ToT (ours) (b=5)	**74%**
IO + Refine (k=10)	27%
IO (best of 100)	33%
CoT (best of 100)	49%



That is a huge jump, underscoring how powerful deliberate planning and search can be when harnessed properly.

The trade-off, of course, is that exploring a tree of possibilities is more computationally intensive, but it's a promising direction for complex decision-making tasks.

#3) Reflexion and Self-Reflection

Planning can guide an agent from the start, but what about learning from mistakes made along the way?

The Reflexion approach doesn't replace ReAct or Planning patterns, but rather augments them with a self-correcting ability:

README MIT license

Original Reflexion Implementation

This is outdated: the main repository is located at https://github.com/noahshinn/reflexion

Code for the approach first proposed in Reflexion: an autonomous agent with dynamic memory and self-reflection

Note

Reflexion is not a replacement for ReAct or any other decision-guiding approach! It is a simple retry technique that can be used enhance other approaches.

Check out the original blog post here.

Check out an interesting type-inference implementation OpenTau that uses a variation of Reflexion to play a 2-player game with type-checker.

If you have any questions, please contact noahrshinn@gmail.com

Cite

```
@article{shinn2023reflexion,
  title={Reflexion: an autonomous agent with dynamic memory and self-reflection},
  author={Shinn, Noah and Labash, Beck and Gopinath, Ashwin},
  journal={arXiv preprint arXiv:2303.11366},
  year={2023}
}
```

A Reflexion-enabled agent has a form of dynamic memory and can critically examine its own outputs or actions after the fact:

Reflexion: an autonomous agent with dynamic memory and self-reflection

Noah Shinn, Beck Labash, Ashwin Gopinath

Recent advancements in decision-making large language model (LLM) agents have demonstrated impressive performance across various benchmarks. However, these state-of-the-art approaches typically necessitate internal model fine-tuning, external model fine-tuning, or policy optimization over a defined state space. Implementing these methods can prove challenging due to the scarcity of high-quality training data or the lack of well-defined state space. Moreover, these agents do not possess certain qualities inherent to human decision-making processes, specifically the ability to learn from mistakes. Self-reflection allows humans to efficiently solve novel problems through a process of trial and error. Building on recent research, we propose Reflexion, an approach that endows an agent with dynamic memory and self-reflection capabilities to enhance its existing reasoning trace and task-specific action choice abilities. To achieve full automation, we introduce a straightforward yet effective heuristic that enables the agent to pinpoint hallucination instances, avoid repetition in action sequences, and, in some environments, construct an internal memory map of the given environment. To assess our approach, we evaluate the agent's ability to complete decision-making tasks in AlfWorld environments and knowledge-intensive, search-based question-and-answer tasks in HotPotQA environments. We observe success rates of 97% and 51%, respectively, and provide a discussion on the emergent property of self-reflection.

In practice, the agent generates a solution (using ReAct or plan-execute), then a secondary process has the agent reflect: “Did I make an error? Does the answer make sense? If not, why not, and what should I do differently?”

The agent can then retry the task with the benefit of this self-feedback, effectively re-planning or adjusting its strategy on the next attempt.

This is like giving the agent a chance to learn from a failed attempt without human intervention.

Reflexion addresses issues like hallucinations or logical errors by having the model itself pinpoint hallucination instances and avoid repeating the same mistakes.

For example, if the agent wrongly assumed something in step 2, the reflection phase would catch that and the agent could modify its plan for the next run.

Experiments with Reflexion showed extremely high success rates on certain decision-making tasks (solving 97% of challenges in a virtual game environment), though the improvement on knowledge questions was more modest:

Reflexion: an autonomous agent with dynamic memory and self-reflection

Noah Shinn, Beck Labash, Ashwin Gopinath

Recent advancements in decision-making large language model (LLM) agents have demonstrated impressive performance across various benchmarks. However, these state-of-the-art approaches typically necessitate internal model fine-tuning, external model fine-tuning, or policy optimization over a defined state space. Implementing these methods can prove challenging due to the scarcity of high-quality training data or the lack of well-defined state space. Moreover, these agents do not possess certain qualities inherent to human decision-making processes, specifically the ability to learn from mistakes. Self-reflection allows humans to efficiently solve novel problems through a process of trial and error. Building on recent research, we propose Reflexion, an approach that endows an agent with dynamic memory and self-reflection capabilities to enhance its existing reasoning trace and task-specific action choice abilities. To achieve full automation, we introduce a straightforward yet effective heuristic that enables the agent to pinpoint hallucination instances, avoid repetition in action sequences, and, in some environments, construct an internal memory map of the given environment. To assess our approach, we evaluate the agent's ability to complete decision-making tasks in AlfWorld environments and knowledge-intensive, search-based question-and-answer tasks in HotPotQA environments. We observe success rates of 97% and 51%, respectively, and provide a discussion on the emergent property of self-reflection.

The key takeaway is that Reflexion introduces a feedback loop on top of planning/execution, making the agent more robust by iteratively refining its approach.

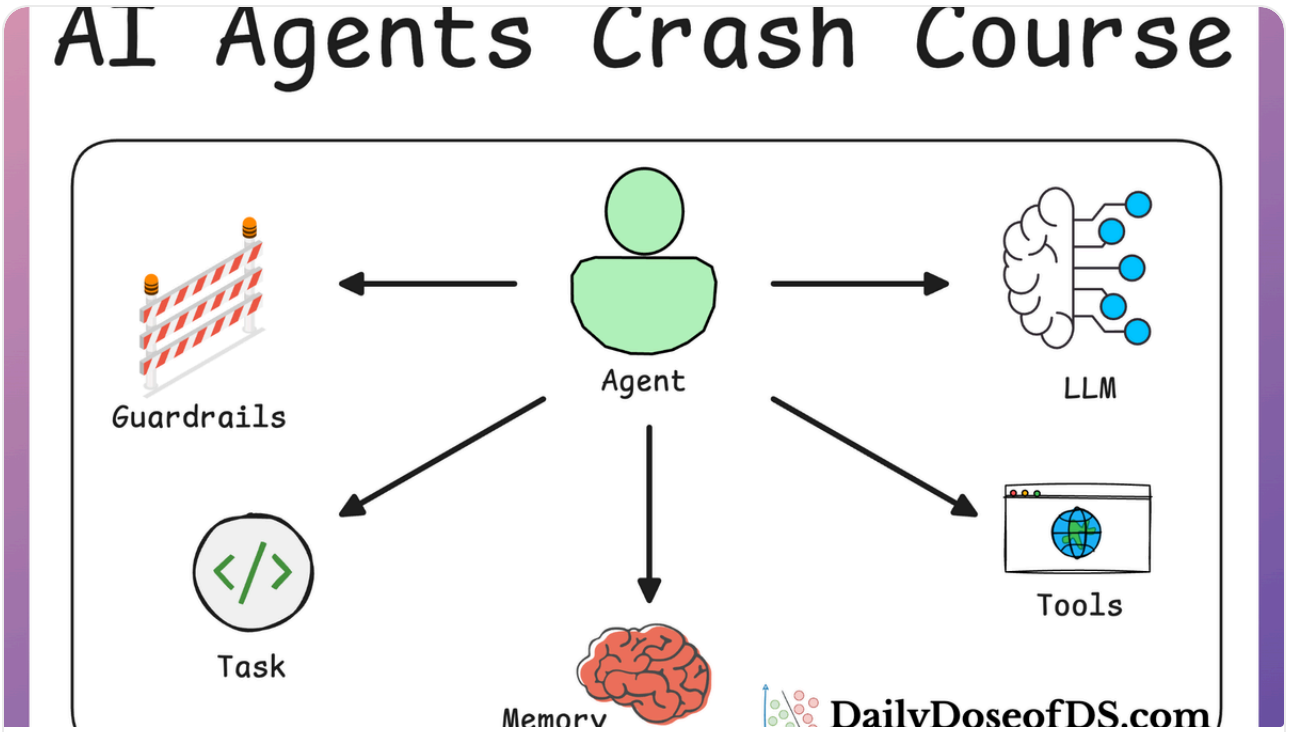
It showcases an important point: planning need not be a single-shot affair. An agent can plan, execute, then re-plan based on errors, approaching a kind of trial-and-error learning process.

#4) Function Calling and Structured Actions (OpenAI)

Another development relevant to the Planning pattern is OpenAI's introduction of function calling in their APIs.

While not a “planning method” per se, function calling allows developers to define tools or actions the model can use in a structured way (the model outputs a JSON object calling a function).

We also learned about it in Part 2 of the AI Agents Crash Course:



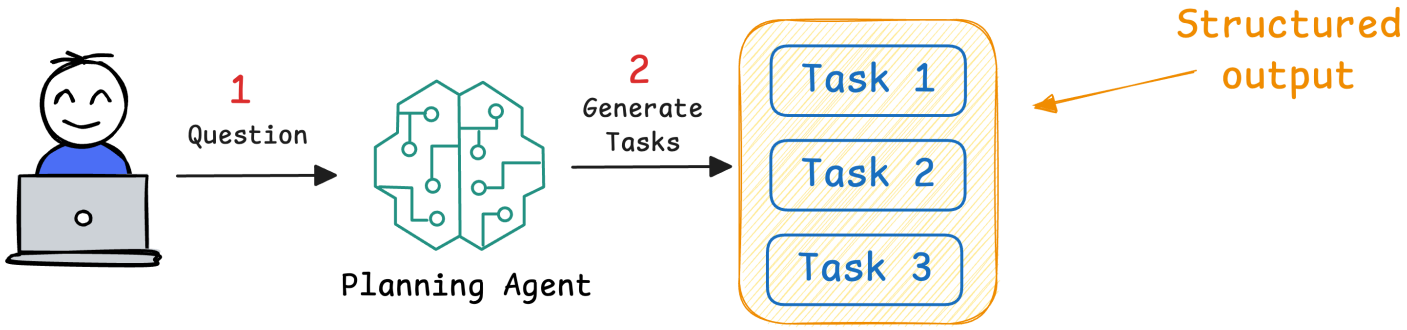
AI Agents Crash Course—Part 2 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

This can be harnessed to implement planning and tool use more reliably.

For instance, one could define a function like `plan_steps(tasks: list)` and another for actual actions; the model could first be forced to call `plan_steps` and return a list of tasks, then sequentially call execution functions.



Function calling ensures that the agent's intended actions are constrained to developer-defined functions, mitigating free-form hallucinations and making the interaction more deterministic.

Essentially, it gives a way to enforce the Planning pattern via the API contract since the model might internally decide to plan, but when it outputs the plan, it does so in a predefined format that the system can parse and follow.

OpenAI's function calling was quickly adopted in agent frameworks to better handle multi-step tool usage.

It guarantees well-structured output for each action, which is useful when an agent must coordinate multiple steps (the system can parse the plan and feed each part back for execution in turn).

While function calling alone doesn't make an agent smarter, it does make it easier to implement patterns like ReAct and Plan-and-Execute cleanly, because the model can explicitly signal its next action or plan in a machine-readable way rather than relying on brittle prompt parsing.

In summary, OpenAI's function-calling feature provides infrastructure to support the Planning pattern, giving developers more control over how the plan and actions are represented and executed.

These literature insights all point to a common theme: structured reasoning improves performance.

Whether it's splitting planning from execution, exploring a tree of possibilities, reflecting on mistakes, or enforcing structure through an API, the goal is to overcome the limitations of a naive step-by-step approach.

Planning in various forms has been shown to make AI agents more reliable, accurate, and interpretable on complex tasks.

Of course, each technique comes with costs – more computation, more prompts, or more engineering overhead – and researchers continue to refine when and how these patterns should be applied for best effect.

That said, now that we understand what the Planning pattern is and when it's useful, let's build a Planning agent from scratch to see it in action!

We'll follow a similar approach as in Part 10: designing a clever prompt, implementing the agent logic in Python, and then walking through an example query step by step.

Planning Implementation from Scratch

Below, we shall implement a Planning Agent in two ways:

- Manually executing each step for better clarity.
- Without manual intervention, to fully automate the planning and execution steps involved in a Planning Agent.

You can download the code below:

notebook

Notebook for planning pattern.

notebook.ipynb • 12 KB

Let's look at the manual process first!

#1) Planning with manual execution

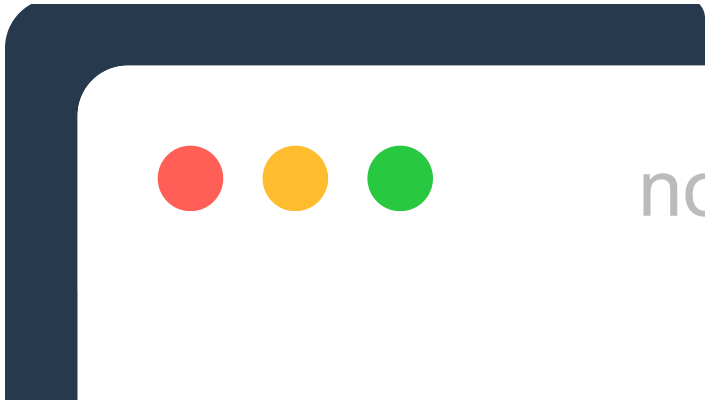
In this section, we'll implement a lightweight Planning-style agent from scratch, without using any orchestration framework like CrewAI or LangChain.

We'll manually simulate each round of the agent's planning and execution stages, exactly as a Planning Agent is meant to function.

By running the logic cell-by-cell, we will gain full visibility and control over the thinking process, allowing us to debug and validate the agent's behavior at each step.

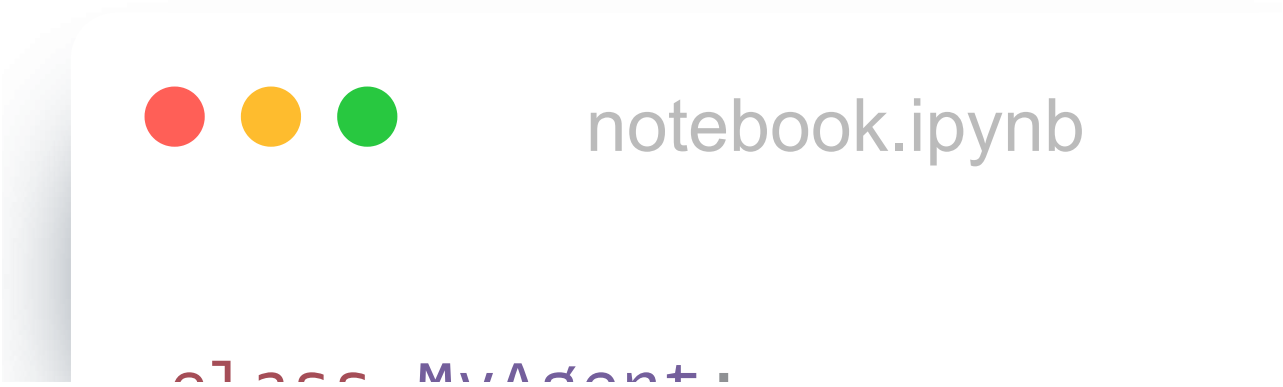
> 💡 Make sure you have created a `.env` file with the `OPENAI_API_KEY` specified in it. It will make things much easier and faster for you.

To begin, we load the environment variables (like your LLM API key) and import completion from LiteLLM (also install it first– `pip install litellm`), a lightweight wrapper to query LLMs like OpenAI or local models via Ollama.



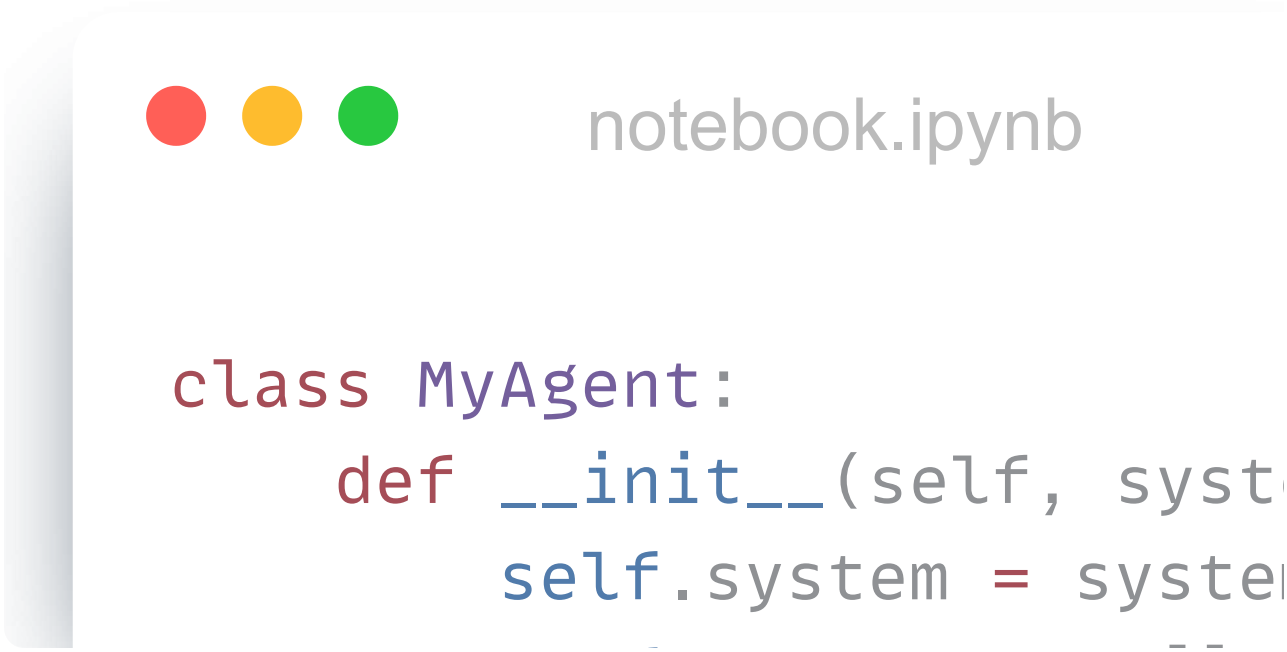
Next, we define a minimal Agent class, which wraps around a conversational LLM and keeps track of its full message history, allowing it to reason step-by-step, access system prompts, remember prior inputs and outputs, and produce multi-turn interactions.

Here's what it looks like:



- `system` (str): This is the system prompt that sets the personality and behavioral constraints for the agent. If passed, it becomes the very first message in the conversation just like in OpenAI Chat APIs.
- `self.messages`: This list acts as the conversation memory. Every interaction, whether it's user input or assistant output is appended to this list. This history is crucial for LLMs to behave coherently across multiple turns.
- If `system` is provided, it's added to the message list using the special `"role": "system"` identifier (we learned about this in **Part 5 of RAG crash course**). This ensures that every completion that follows is conditioned on the system instructions.

Next, we define a `complete` method in this class:



This is the core interface you'll use to interact with your agent.

- If a `message` is passed:
  - It gets appended as a `"user"` message to `self.messages`.
  - This simulates the human asking a question or giving instructions.
- Then, `self.invoke()` is called (which we will define shortly). This method sends the full conversation history to the LLM.
- The model's reply (stored in `result`) is then appended to `self.messages` as an `"assistant"` role.
- Finally, the reply is returned to the caller.

This method does three things in one call:

1. Records the user input.
2. Gets the model's reply.
3. Updates the message history for future turns.

Finally, we have the `invoke` method below:

This method handles the actual API call to your LLM provider, in this case, via LiteLLM, using the `"openai/gpt-4o"` model.

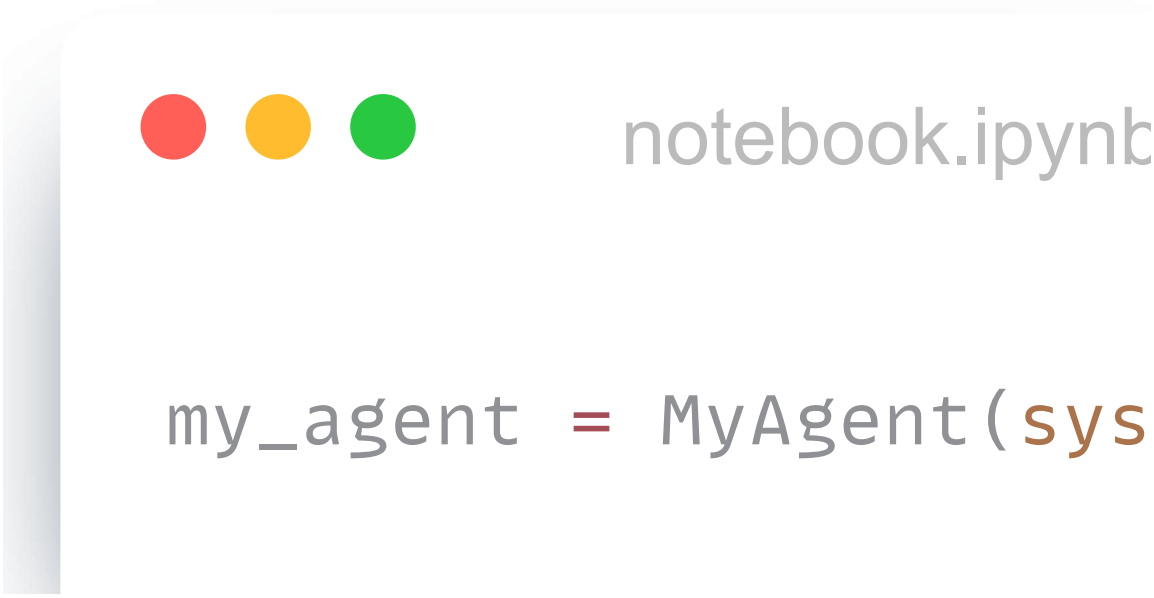
- `completion()` is a wrapper around the chat completion API. It receives the entire message history and returns a response. If needed, you can also change the model to a local LLM served via Ollama as follows:

OpenAI Anthropic xAI VertexAI NVIDIA HuggingFace Azure OpenAI Ollama Openrouter

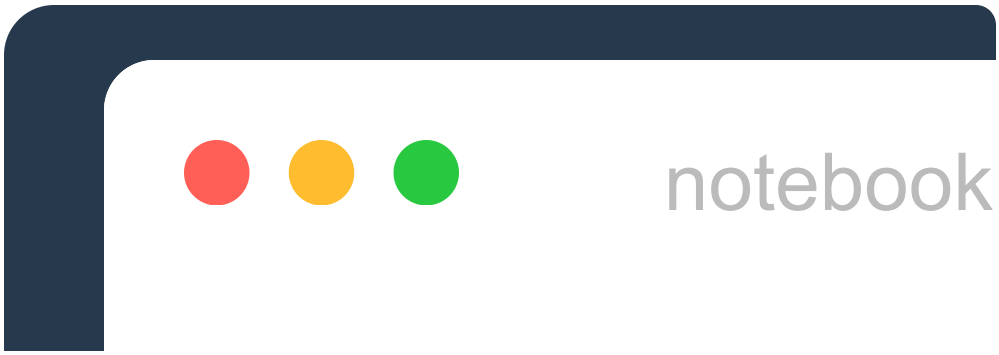
```
from litellm import completion

response = completion(
            model="ollama/llama2",
            messages = [{ "content": "Hello, how are you?","role": "user"}],
            api_base="http://localhost:11434"
)
```

- We assume `completion()` returns a structure similar to OpenAI's format: a list of choices, where each choice has a `.message.content` field.
- We extract and return that content which is the assistant's next response.

As a test, we can quickly run a simple interaction below:



At this stage, if we ask it about the previous message, we get the correct output, which shows the assistant has visibility on the previous context:



It correctly remembers and reflects!

Now that our conversational class is set up, we come to the most interesting part, which is defining a Planning-style prompt.

Before an LLM can behave like an agent, it needs clear instructions, not just on what to answer, but how to go about answering. That's exactly what this `system_prompt` does, which is defined below:

notebook.ipynb

```
system_prompt = """
You are a smart planning agent.
You act in iterations and do JUST ONE

1) "Plan" to plan the steps needed to
2) "Execute" to execute the planned s
3) "Observation" to get the output of
4) "Collect" to just collect the resu
5) "Answer" to answer the user's ques

So to summarize, to answer a question
- Think through the entire solution f
- Then execute each step in order by
- Collect all the individual results.
- Finally, answer the user's question

Here are the tools available to you:

math:
Use this to evaluate math expressions
Example: math: (125000000 + 14000000

lookup_population:
```

```
lookup_population:
Use this to get the population of a c
Example: lookup_population: Japan

You must first output a PLAN and ther
At the end, output the FINAL ANSWER.

---

Here's a sample run for your referenc

Question: What is the population of J

<Iteration 1>
Plan:
1. Use lookup_population on Japan.
2. Use lookup_population on India.
3. Use math to add the two populatior
</Iteration 1>

<Iteration 2>
Execute:
Step 1: lookup_population: Japan
</Iteration 2>
```

This isn't just a prompt. It's a behavioral protocol which defines what structure the agent should follow, how it should plan and execute, and when it should stop.

Let's break it down line by line.

> 👉 You are a smart planning agent. You act in iterations and do JUST ONE thing in a single iteration:

This sets the role clearly. You are not a general assistant or a one-shot solver. You are a planning agent that solves problems methodically by thinking ahead. The second line is even more important: do JUST ONE thing in a single iteration.

This restriction enforces stepwise behavior—planning, executing, observing, collecting, and finally answering—one at a time.

This avoids the agent jumping ahead or trying to solve the whole problem in one go (a common issue with LLMs). It encourages discipline and structure.

👉 1) "Plan" to plan the steps needed to answer the question.
2) "Execute" to execute the planned steps, one step at a time.
3) "Observation" to get the output of the execution.
4) "Collect" to just collect the result of all the steps.
5) "Answer" to answer the user's question using the collected results.

This gives the LLM an exact loop to follow. Each iteration must begin with one of these phases. Here's what each does:

- Plan: Before doing anything, the model must output a clear ordered list of steps needed to answer the question. This is where task decomposition happens. The idea is to map the entire task upfront, much like creating a checklist.
- Execute: Now, take one step from the plan and run it. This is where the model simulates calling a tool (like lookup_population) and outputs the step being executed.
- Observation: After the tool call, the agent gets the result. This step explicitly marks the observation. For instance, “Observation: 125000000”. This enforces structured thinking and allows the next step to use this data.
- Collect: Once all steps are executed and their results are observed, the agent must summarize all the gathered data. This is like consolidating your scratchpad before writing the final answer.
- Answer: Finally, the model uses the collected results to produce the final answer in natural language.

This entire loop mimics how thoughtful humans work: plan → do → note → summarize → answer. It forces the model to approach problems more deliberately.

💡 So to summarize, to answer a question, you will:
- Think through the entire solution first, listing each step clearly before taking an action.

- Then execute each step in order by calling one of the available tools.
- Collect all the individual results.
- Finally, answer the user's question using the collected results.

This paragraph recaps the iterative process in natural language. It’s like giving a checklist to a junior analyst:

- Think.
- Do the tasks.
- Write everything down.
- Then give the final answer.

💡 Here are the tools available to you:

math:
Use this to evaluate math expressions using Python syntax.
Example: math: (125000000 + 1400000000)

lookup_population:
Use this to get the population of a country.
Example: lookup_population: Japan

This is the tool instruction segment, a core part of agentic reasoning. It gives the model controlled affordances. In other words, here’s what you are allowed to do.

These tools act as the "actions" the model can perform during execution. Providing clear syntax examples ensures the model outputs them in a predictable format, which is crucial if you're building a tool parser.

> 💡 You must first output a PLAN and then execute each step, showing the result after each one.

This line enforces the sequence. The agent should never jump into execution without first giving a PLAN. This ensures every execution is grounded in a broader context.

Next, we give it a sample run.

> 💡 Here's a sample run for your reference:
>
> Question: What is the population of Japan plus the population of India?

> 💡 <Iteration 1>
> Plan:
> 1. Use lookup_population on Japan.
> 2. Use lookup_population on India.
> 3. Use math to add the two populations.
> </Iteration 1>

Here, the agent is given a query: “What is the population of Japan plus the population of India?”

In response, it lists three ordered steps while looking at the tools it has access to:

1. Get Japan’s population.
2. Get India’s population.

3. Add the two.

This step forces the model to map the problem before jumping into action, which is a key difference from ReAct.

💡 <Iteration 2>
Execute:
Step 1: lookup_population: Japan
</Iteration 2>

The model now picks the first step from its plan and outputs the tool call. Note: It does not yet execute the tool or produce the result, it just declares what it is doing.

💡 <Iteration 3>
Observation: 125000000
</Iteration 3>

After this, the model gets the tool output (we'll understand how, but for now, assume it did), and the agent logs it as an observation. This updates the working memory of the agent and will be used in the next step.

💡 <Iteration 4>
Execute:
Step 2: lookup_population: India
</Iteration 4>

The agent is now executing the second step from its original plan to find the population of India.

- In the earlier iterations, it already executed `lookup_population: Japan` and recorded its output.
- Now, it continues with the next task in the plan which is a new API/tool call.

💡 <Iteration 5>
Observation: 1400000000
</Iteration 5>

The external system (our Python environment) executes the tool call and feeds the actual result back to the agent.

- This updates the agent's context: now it knows India's population is 1.4 billion.
- The Observation becomes part of the agent's working memory for future computations.

💡 <Iteration 6>
Execute:
Step 3: math: (125000000 + 1400000000)
</Iteration 6>

Now that the agent has both values (Japan and India populations), it executes the final step of its plan:

- It uses the `math` tool to add the two numbers.
- The syntax is specific, structured, and deterministic, exactly how we'd want it for parsing and execution.

This is the final tool interaction before moving to summarization. It's crucial that the model uses the exact syntax (`math: (...)`) as defined in the system prompt to keep things predictable and machine-readable.

💡 <Iteration 7>
Observation: 1525000000
</Iteration 7>

The math tool is executed, and the system feeds back the result (1,525,000,000), the total population of Japan and India combined.

- This is the final numeric result the agent needs.
- With this, it has everything required to synthesize the final answer.

💡 <Iteration 8>
Collect:
Step 1: Japan population: 125000000
Step 2: India population: 1400000000
Step 3: Total population: 1525000000
</Iteration 8>

After all Observations are gathered, the model summarizes everything in the Collect phase. This mimics a scratchpad of everything it has learned.

This is not yet the final answer, just an internal consolidation.

💡 <Iteration 9>
Answer:
The total population of Japan and India is approximately 1.525 billion.
</Iteration 9>

Now, the model outputs the final answer using the collected results. It also uses friendly formatting (1.525 billion instead of 1525000000), showing its reasoning capabilities post-computation.

To summarize, our prompt:

- Teaches the model to plan first, rather than act impulsively.
- Breaks the loop into observable and parsable steps.
- Encourages disciplined use of tools.
- Makes the model's internal state transparent at every step.

You could plug this into any execution engine that parses the plan, routes tool calls, and handles Observations, and it would make debugging and monitoring far easier.

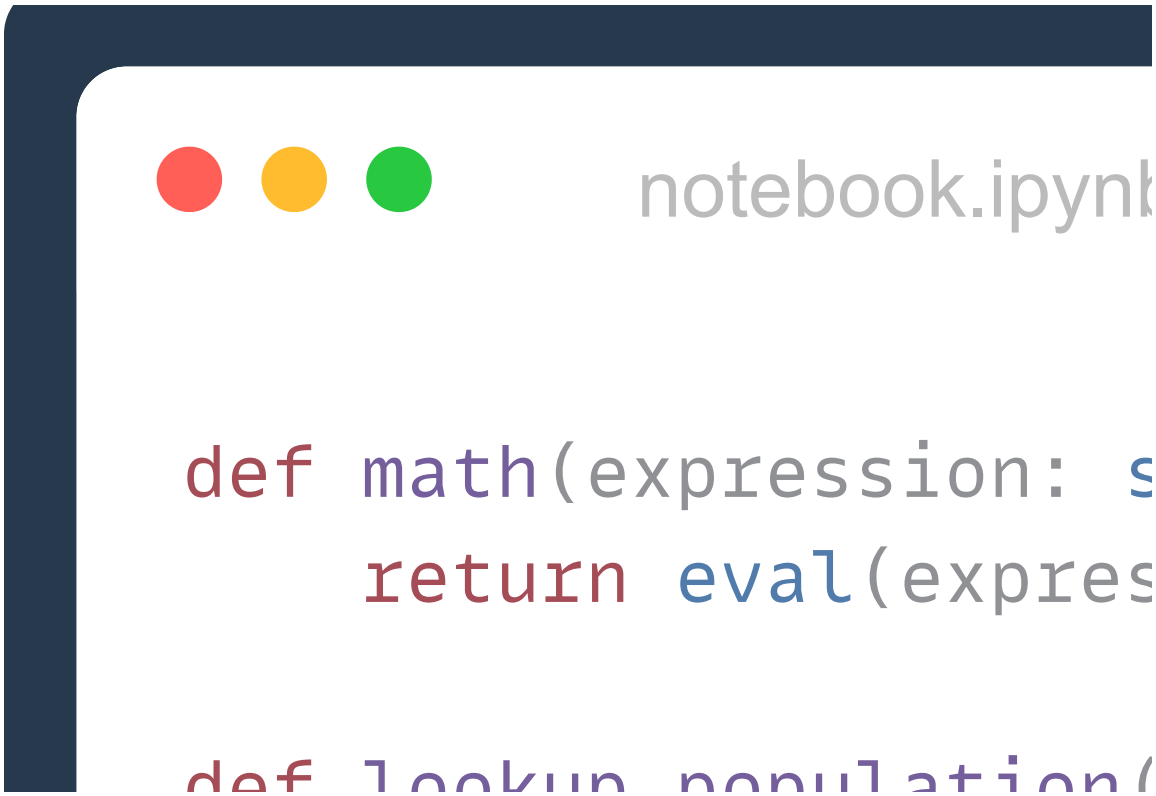
In fact, the design almost mimics the agent pattern popularized by frameworks like CrewAI where agents define a sequence of actions and execute them with tool invocations and context updates.

notebook.ipynb

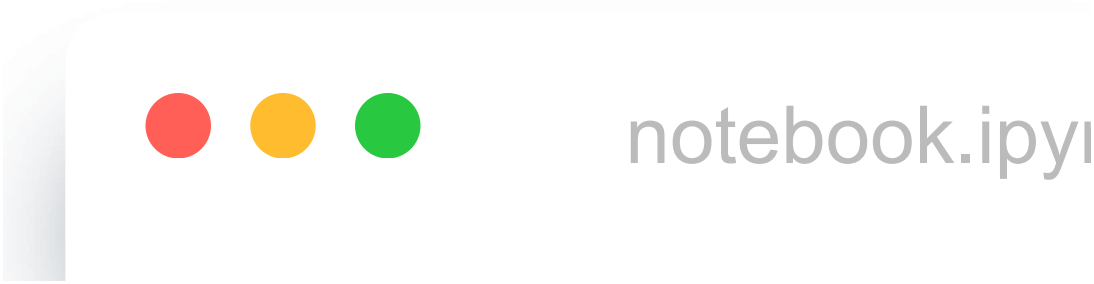
```
system_prompt = """
You are a smart planning agent.
You act in iterations and do JUST ONE

1) "Plan" to plan the steps needed to
2) "Execute" to execute the planned s
3) "Observation" to get the output of
4) "Collect" to just collect the resu
5) "Answer" to answer the user's ques

So to summarize, to answer a question
- Think through the entire solution f
- Then execute each step in order by
- Collect all the individual results.
- Finally, answer the user's question

Here are the tools available to you:

math:
Use this to evaluate math expressions
Example: math: (125000000 + 14000000

lookup_population:
```

```
lookup_population:
Use this to get the population of a c
Example: lookup_population: Japan

You must first output a PLAN and ther
At the end, output the FINAL ANSWER.

---

Here's a sample run for your referenc
```

Now that the prompt is defined, we implement the tools



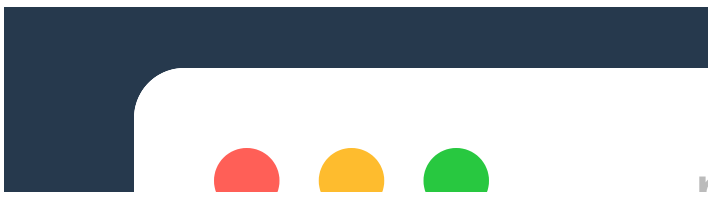
Finally, we begin a manual Planning Agent session:



This produces the following output:

👉 <Iteration 1>
Plan:
1. Use lookup_population on Japan.
2. Use lookup_population on India.
3. Use math to add the two populations.
</Iteration 1>

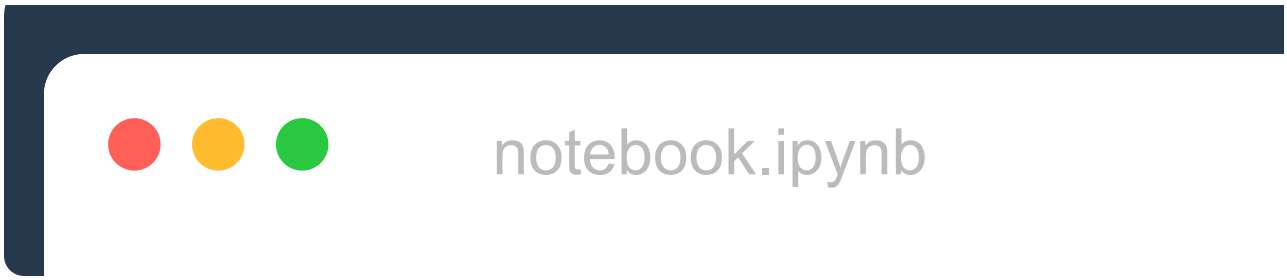
We, as a user, don't have any input to give at this stage so we just invoke the `complete()` method again:



This produces the following output:

👉 <Iteration 2>
Execute:
Step 1: lookup_population: Japan
</Iteration 2>

Now it has reached an execution step, where the `lookup_population` the method must be invoked with the parameter `Japan`, and it needs to get the tool output in the form of an observation. Here, let's intervene and provide it with the observation in the appropriate format:



This produces the following output:

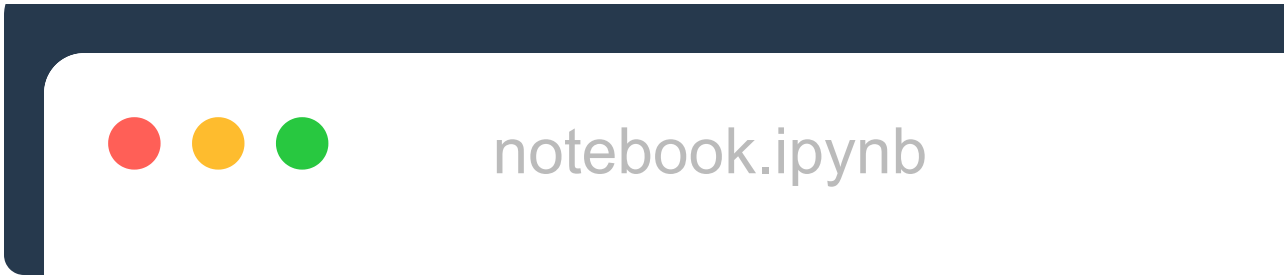
👉 <Iteration 4>
Execute:
Step 2: lookup_population: India
</Iteration 4>

Now it has again reached an execution step, where the `lookup_population` the method must be invoked with the parameter `India` instead, and it needs to get the tool output in the form of an observation. So let's again intervene and provide it with the observation in the appropriate format:

This produces the following output:

👉 <Iteration 6>
Execute:
Step 3: math: (125000000 + 1400000000)
</Iteration 6>

At this stage, it needs to get the tool output in the form of an observation. Here, let's intervene and provide it with the observation:

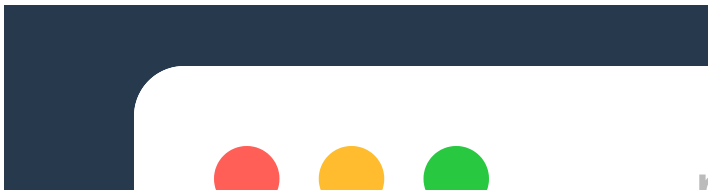


This produces the following output:

👉 <Iteration 8>
Collect:
- Step 1: Japan population: 125000000
- Step 2: India population: 1400000000
- Step 3: Total population: 1525000000
</Iteration 8>

Since it has now reached the collect stage, it shows that the entire plan the Agent laid out earlier has been executed successfully, and it is ready to produce an answer next.

Thus, we let it continue its execution:



We get the following output:

👉 Iteration 9>
Answer:
The total population of Japan and India is approximately 1.525 billion.
</Iteration 9>

Great!!

With this process:

- The LLM thought about what steps to take.
- It chose the functions to invoke.
- We manually injected tool outputs like real-world observations.
- Once it had executed all the steps, it collected the entire information.
- It looped until it had enough information to generate a final answer.

This gives us an explicit understanding of how planning comes together in Planning-style agents.

In the next part, we'll fully automate this, no manual calls required, and build a full controller that simulates this entire loop programmatically.

#2) Planning without manual execution

Now that we have understood how the above Planning execution went, we can easily automate that to remove the interventions we did earlier.

In this section, we'll create a controller function that:

- Sends an initial question to the agent,
- Understands its plans.
- Automatically runs external tools when asked,
- Feeds back observations to the agent,
- And stops the loop once a final answer is found.

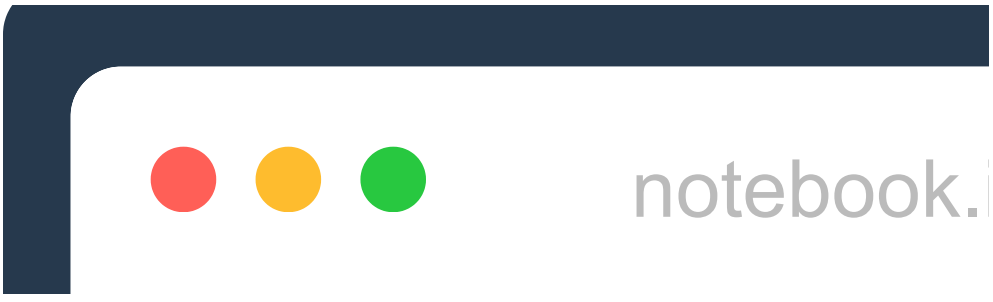
This is the entire code that does this:

notebook.ipynb

```
import re

def agent_loop(query, system_prompt

    my_agent = PlanningAgent(system

    available_tools = {"math": math
                       "lookup_popu

    current_prompt = query

    while "ANSWER" not in current_p
```

Let's break down the full loop.

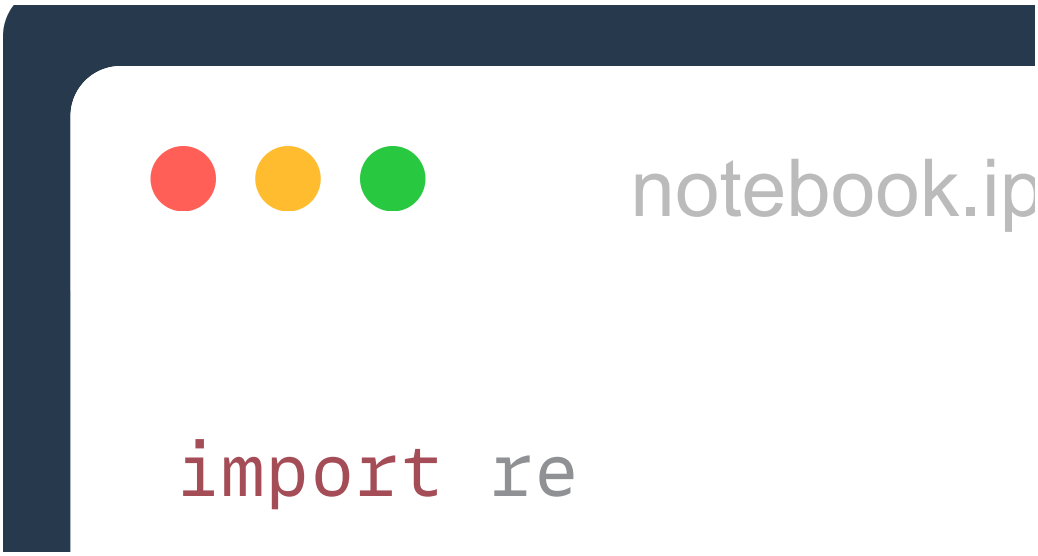
We begin by defining the `agent_loop()` function:



It takes:

- `query`: the user's natural language question.
- `system_prompt`: the same Planning system prompt we explored earlier (defining the behavior loop).

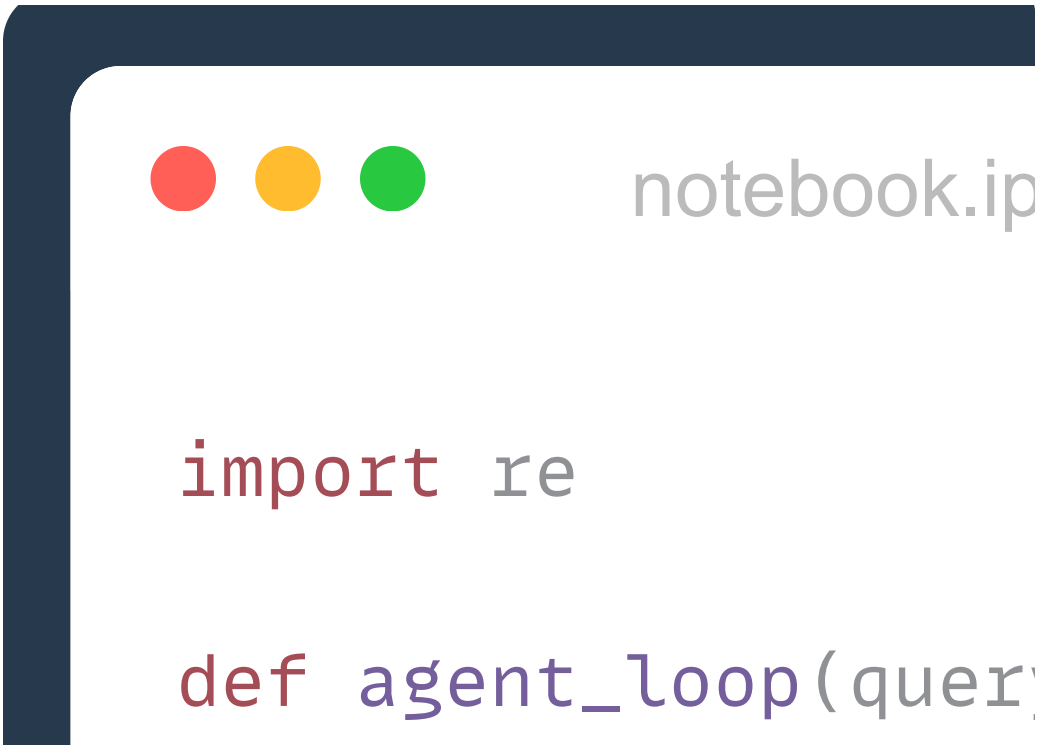
Next, inside this function, we initialize the Agent and available tools:



- Create a new `MyAgent` instance, using the structured Planning prompt.
- Define the dictionary of callable tools available to the agent. These names must match exactly what the agent uses in its `Action:` lines.

Moving on, we defined a state variable `current_prompt` to store the next message to be sent to the LLM.

Next, we run the reasoning loop, which continues until the agent produces a final answer. The answer is expected to be marked with `Answer:` based on our prompt design:



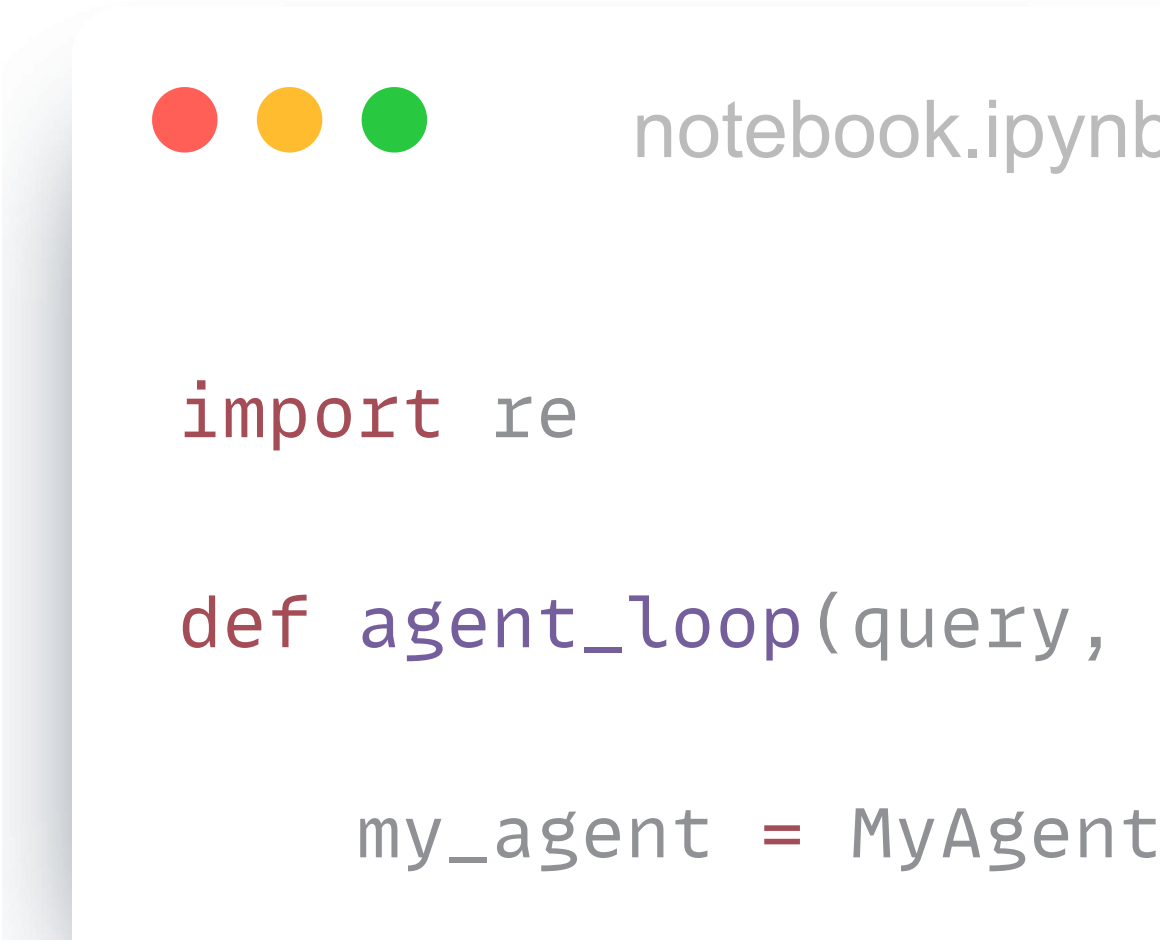
Next, we feed the `current_prompt` into the agent.

The `current_prompt` could be:

- The initial user query,
- A blank string to let the agent continue reasoning,
- An observation from a tool.

We then print the agent's output so we can inspect each iteration.

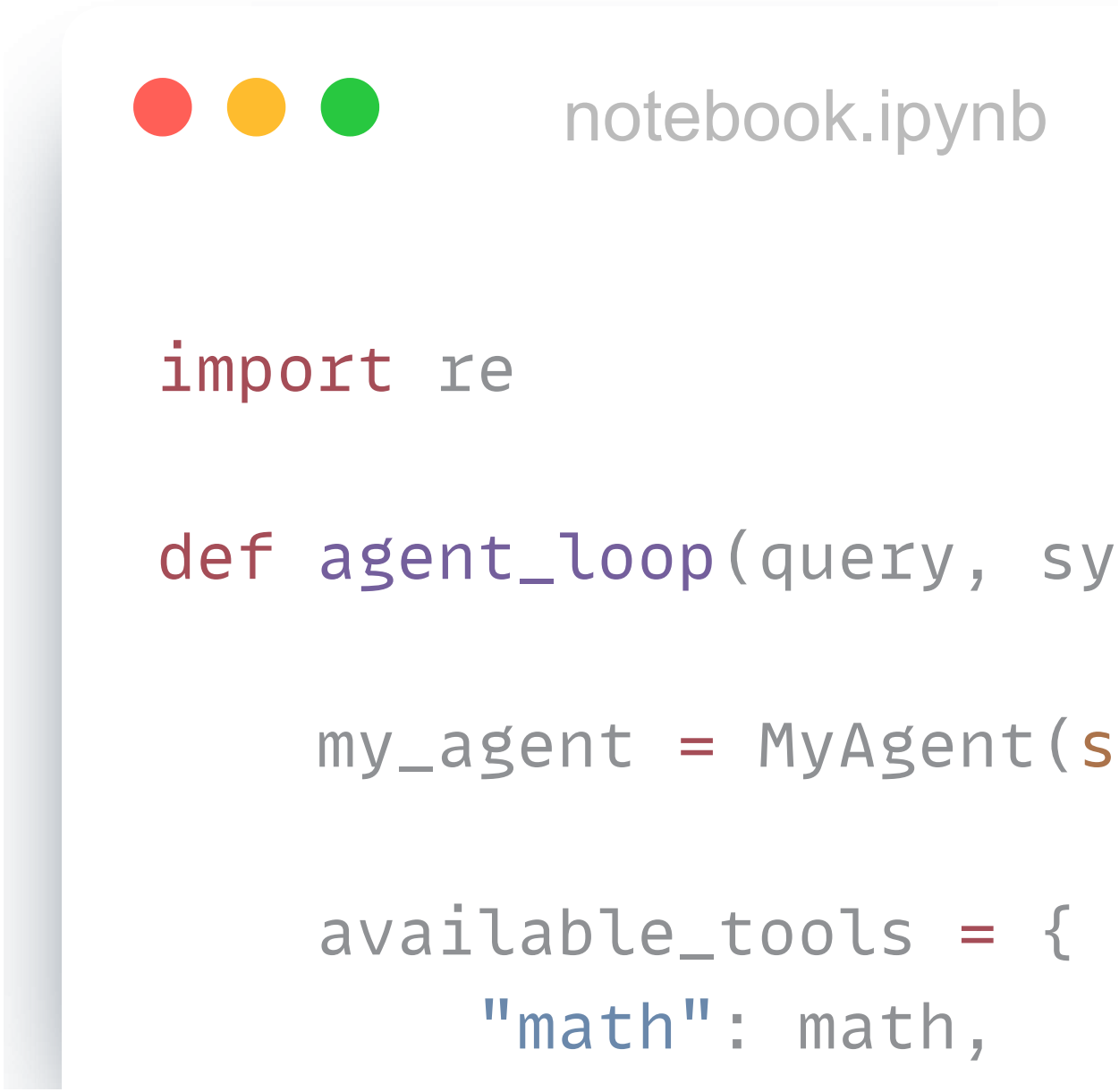
Next, if the agent produces a final answer, we break the loop.



In another case, if the response includes a `Plan:` or `Collect:` line, we:

- Set `current_prompt` to an empty string to continue to the next stage (Execute or Answer as per what we mentioned in the prompt earlier).

However, if the response contains `Execute:`, we must provide it with an input. Moreover, we also need to pull out the function name and parameter. To do this, we use a regex to extract the tool name and its argument.



For example, in: `Step 2: lookup_population: India`, the regex pulls out:

- `lookup_population` as the tool.
- `India` as the argument.

Moving on, we execute the tool and capture the observation:

notebook.ipynb

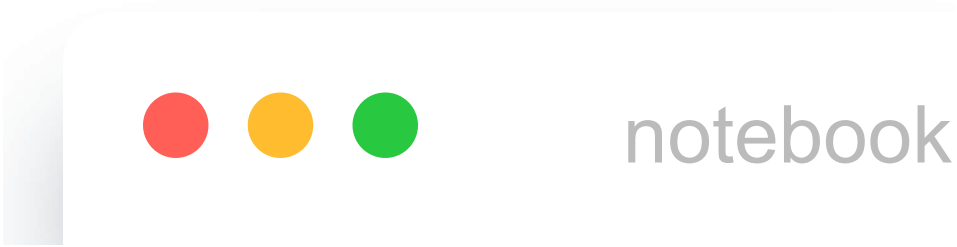
```
import re

def agent_loop(query, system_pr

    my_agent = MyAgent(system=s

    available_tools = {
        "math": math,
        "lookup_population": lo
    }

    current_prompt = query

    while "ANSWER" not in
```

- If the tool name is valid, we call it like a Python function and capture the result.
- We format the output into `Observation: ...` so the agent can use it in the next step.
- If the tool doesn't exist, we ask the agent to retry.

This mimics tool execution + response injection.

Done!

Now we can run this function as follows:



This produces the following output, which is indeed correct:

```
agent_loop("What is the population of Japan plus the population of India?", system_prompt)
✓ 6.0s
<Iteration 1>
Plan:
1. Use lookup_population on Japan.
2. Use lookup_population on India.
3. Use math to add the two populations.
</Iteration 1>
<Iteration 2>
Execute:
Step 1: lookup_population: Japan
</Iteration 2>
<Iteration 3>
Execute:
Step 2: lookup_population: India
</Iteration 3>
<Iteration 4>
Execute:
Step 3: math: (125000000 + 1400000000)
</Iteration 4>
<Iteration 5>
Collect:
- Step 1: Japan population: 125000000
- Step 2: India population: 1400000000
- Step 3: Total population: 1525000000
</Iteration 5>

<Iteration 6>
Answer:
The total population of Japan and India is approximately 1.525 billion.
</Iteration 6>
```

You now have a fully working Planning pattern without needing any external framework.

Of course, in this implementation, we're using regex matching and hardcoded conditionals to parse the agent's actions and route them to the correct tools.

This approach works well for a tightly controlled setup like this demo. However, it's brittle:

- If the agent slightly deviates from the expected format (e.g., adds extra whitespace, uses different casing, or mislabels an action), the regex could fail to match.

- We're also assuming that the agent will never call a tool that doesn't exist, and that all tools will succeed silently.

In a production-grade system, you'd want to:

- Add more robust parsing (e.g., structured prompts with JSON outputs or function calling).
- Include tool validation, retries, and exception handling.
- Use guardrails or output formatters to constrain what the LLM is allowed to emit.

But for the purpose of understanding how Planning-style loops work under the hood, this is a clean and minimal place to start. It gives you complete transparency into what's happening at each stage of the agent's reasoning and execution process.

This loop demonstrates how a simple agent can plan and execute, all powered by your own Python + local LLM stack.

Conclusion

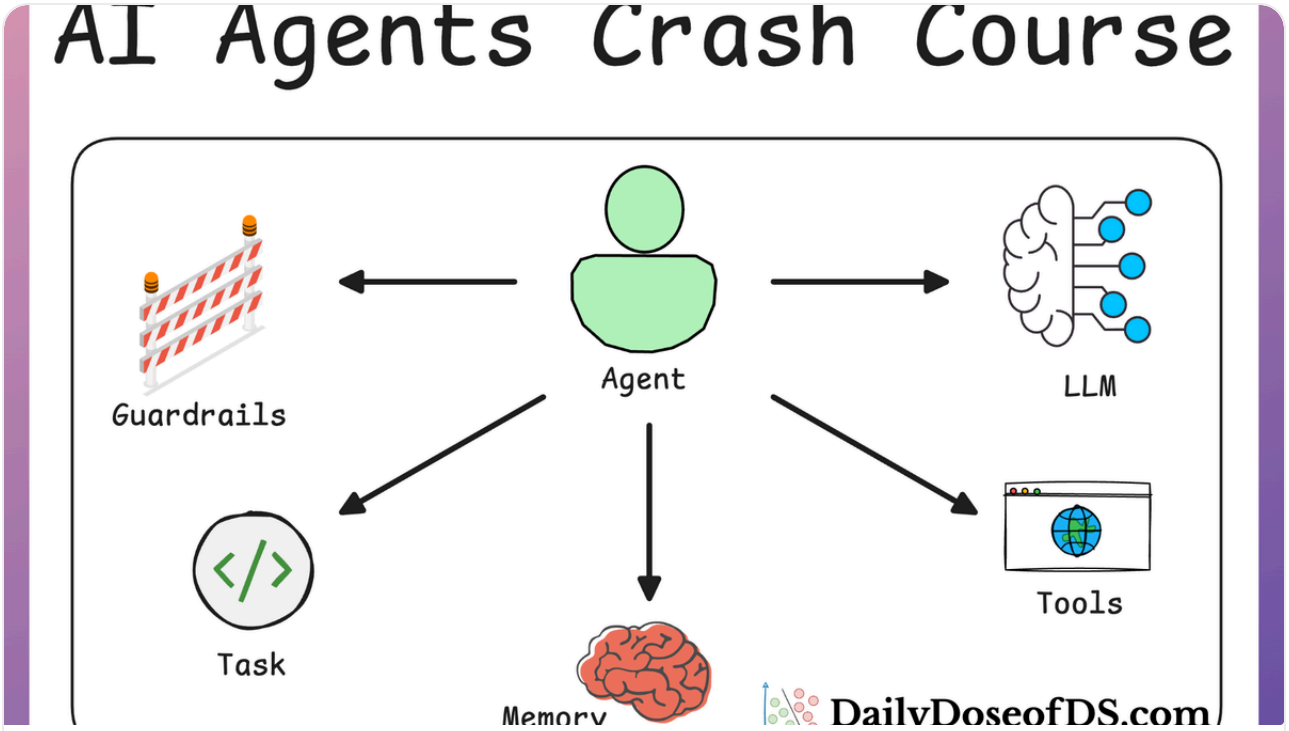
In this tutorial, we explored the Planning pattern in depth, and built a Planning-style agent completely from scratch using Python and a local LLM via the `litellm` library.

We covered:

- Why Planning is a critical pattern for agentic workflows, especially when problems benefit from decomposing into smaller sub-tasks.
- How this pattern differs from reactive patterns like ReAct, and why Planning is better suited for situations where strategic sequencing of actions is required.
- How to write a system prompt that instructs the LLM to:
    - Think ahead before taking action,

    - Execute tasks one by one in order,
    - Collect and observe intermediate results,
    - And only then, synthesize a final answer.
- How to implement a custom `PlanningAgent` that stores conversational history, drives the loop, and ensures deterministic LLM interactions.
- How to structure each phase of the loop (Plan → Execute → Observation → Collect → Answer) using clear, tool-callable syntax.
- How to walk through each iteration manually to debug and inspect the agent's reasoning, tool calls, and observations.

And just like in our ReAct implementation, we did all of this without any orchestration framework like CrewAI, LangChain, or LlamaIndex.



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

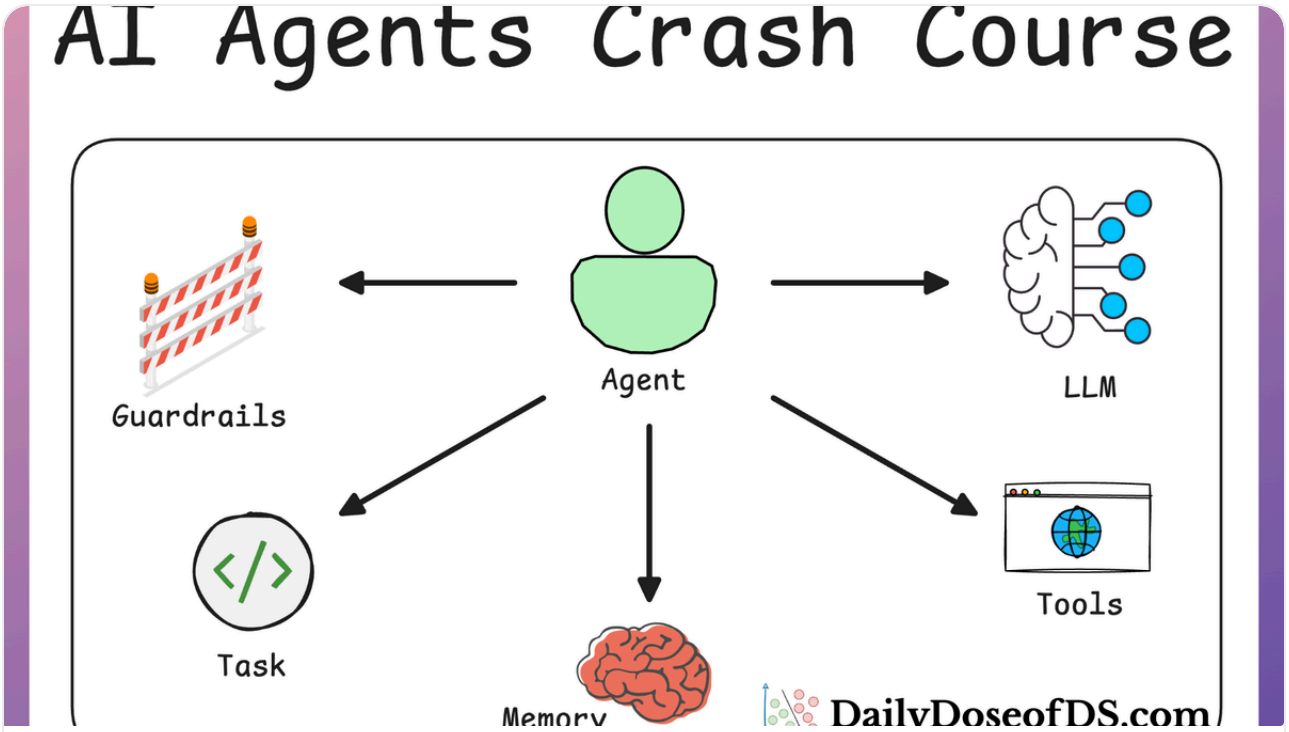
This kind of lightweight but transparent design is perfect for:

- Research assistants that break down papers into summaries
- Web agents that crawl and aggregate content across sites
- Retrieval agents that structure multi-hop queries
- Any autonomous system that benefits from planning first, then acting

Yes, our approach still uses hardcoded prompts and brittle regex for tool parsing but that's also the point: it teaches you what these frameworks are doing behind the scenes.

Once you're fluent with these patterns, you'll have a much stronger grasp of when to use Planning over ReAct, and how to extend these agents into more complex agentic pipelines either standalone or integrated into production workflows.

Read the next part of this crash course here:



Implementing Multi-agent Agentic Pattern From Scratch

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

Daily Dose of Data Science • Avi Chawla

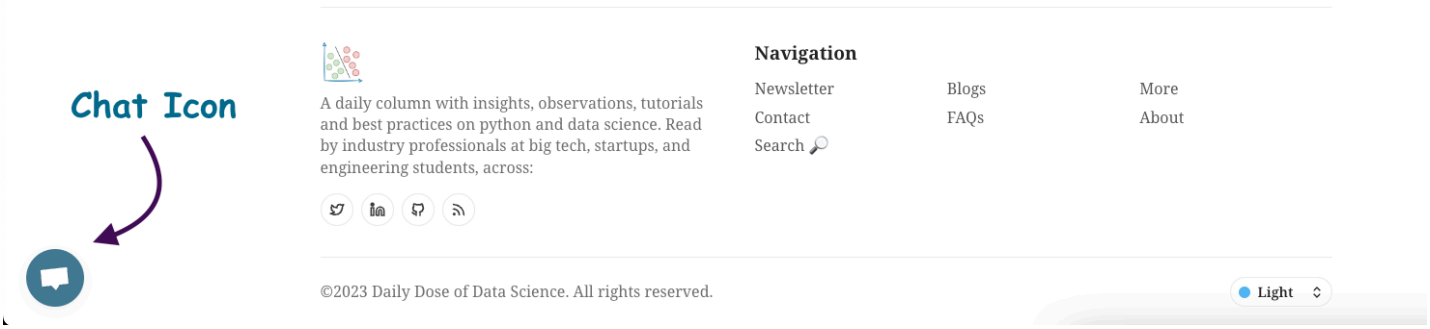
As always, thanks for reading!

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

Navigation