LangChain

Plan-and-Execute Agents

Plan and execute agents promise faster, cheaper, and more performant task execution over previous agent designs. Learn how to build 3 types of planning agents in LangGraph in this post.

BY LANGCHAIN 5 MIN READ FEB 13, 2024

Links

Plan-and-execute (**Python**, **JS**)

LLMCompiler (**Python**)

ReWOO (**Python**)

Youtube

Subscribe

We're releasing three agent architectures in LangGraph showcasing the "plan-and-execute" style agent design. These agents promise a number of improvements over traditional Reasoning and Action (ReAct)-style agents.

⏰ First of all, they can execute multi-step workflow ***faster**,* since the larger agent doesn't need to be consulted after each action. Each sub-task can be performed without an additional LLM call (or with a call to a lighter-weight LLM).

💸 Second, they offer **cost savings** over ReAct agents. If LLM calls are used for sub-tasks, they typically can be made to smaller, domain-specific models. The larger model then is only called for (re-)planning steps and to generate the final response.

🏆 Third, they can **perform better** overall (in terms of task completions rate and quality) by forcing the planner to explicitly "think through" all the steps required to accomplish the entire task. Generating the full reasoning steps is a tried-and-true prompting technique to improve outcomes. Subdividing the problem also permits more focused task execution.

Background

Over the past year, language model-powered agents and state machines have emerged as a promising design pattern for creating flexible and effective ai-powered products.

At their core, agents use LLMs as general-purpose problem-solvers, connecting them with external resources to answer questions or accomplish tasks.

LLM agents typically have the following main steps:

1. Propose action: the LLM generates text to respond directly to a user or to pass to a function.
2. Execute action: your code invokes other software to do things like query a database or call an API.
3. Observe: react to the response of the tool call by either calling another function or responding to the user.

The **ReAct** agent is a great prototypical design for this, as it prompts the language model using a repeated thought, act, observation loop:

```
Thought: I should call Search() to see the current score of the gam
Act: Search("What is the current score of game X?")
Observation: The current score is 24-21
... (repeat N times)
```

A typical ReAct-style agent trajectory.

This takes advantage of **Chain-of-thought** prompting to make a single action choice per step. While this can be effect for simple tasks, it has a couple main downsides:

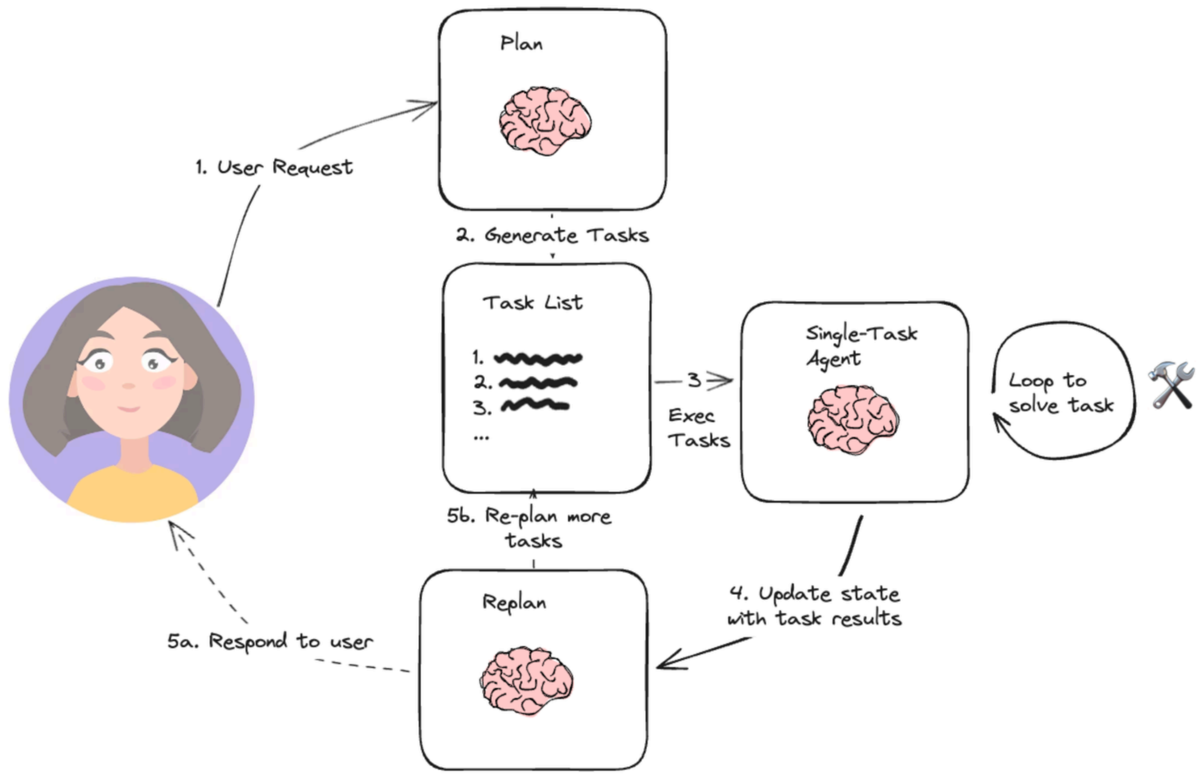
1. It requires an LLM call for each tool invocation.
2. The LLM only plans for 1 sub-problem at a time. This may lead to sub-optimal trajectories, since it isn't forced to "reason" about the whole task.

One way to overcome these two shortcomings is through an explicit planning step. Below are two such designs we have implemented in LangGraph.

Plan-And-Execute

🔗 **Python Link**

🔗 **JS Link**



Plan-and-execute Agent

Based loosely on Wang, et. al.'s paper on **Plan-and-Solve Prompting**, and Yohei Nakajima's **BabyAGI** project, this simple architecture is emblematic of the planning agent architecture. It consists of two basic components:

1. A **planner**, which prompts an LLM to generate a multi-step plan to complete a large task.
2. **Executor**(s), which accept the user query and a step in the plan and invoke 1 or more tools to complete that task.

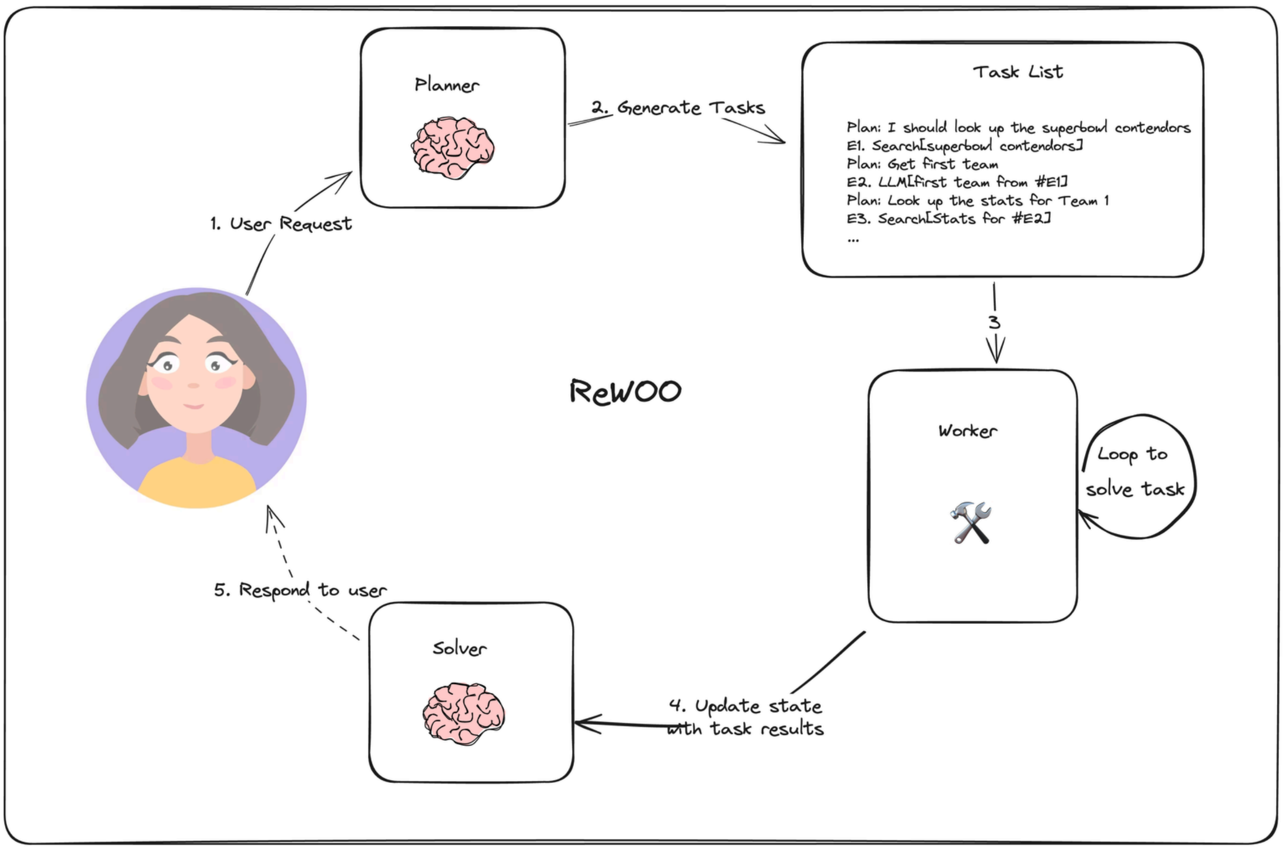
Once execution is completed, the agent is called again with a re-planning prompt, letting it decide whether to finish with a response or whether to generate a follow-up plan (if the first plan didn't have the desired effect).

This agent design lets us avoid having to call the large planner LLM for each tool invocation. It still is restricted by serial tool calling and uses an LLM for each task since it doesn't support variable assignment.

Reasoning WithOut Observations

🔗 Python Link

In **ReWOO**, Xu, et. al, propose an agent that removes the need to always use an LLM for each task while still allowing tasks to depend on previous task results. They do so by permitting variable assignment in the planner's output. Below is a diagram of the agent design.



```
Plan: I should look up the superbowl contendors
E1. Search[superbowl contendors]
Plan: Get first team
E2. LLM[first team from #E1]
Plan: Look up the stats for Team 1
E3. Search[Stats for #E2]
...
```

ReWOO Agent

Its **planner** generates a plan list consisting of interleaving "Plan" (reasoning) and "E#" lines. As an example, given the user query "What are the stats for the quarterbacks of the super bowl contenders this year", the planner may generate the following plan:

```
Plan: I need to know the teams playing in the superbowl this year
E1: Search[Who is competing in the superbowl?]
Plan: I need to know the quarterbacks for each team
E2: LLM[Quarterback for the first team of #E1]
Plan: I need to know the quarterbacks for each team
E3: LLM[Quarter back for the second team of #E1]
Plan: I need to look up stats for the first quarterback
E4: Search[Stats for #E2]
Plan: I need to look up stats for the second quarterback
E5: Search[Stats for #E3]
```

Notice how the planner can reference previous outputs using syntax like `#E2` . This means it can execute a task list without having to re-plan every time.

The **worker** node loops through each task and assigns the task output to the corresponding variable. It also replaces variables with their results when calling subsequent calls.

Finally, the **Solver** integrates all these outputs into a final answer.

This agent design can be more effective than a naive plan-and-execute agent since each task can have only the required context (its input and variable values).

It still relies on sequential task execution, however, which can create a longer runtime.

LLMCompiler

🔗 **Python Link**

LLMCompiler Agent

The **LLMCompiler**, by **Kim, et. al.,** is an agent architecture designed to further increase the **speed** of task execution beyond the plan-and-execute and ReWOO agents described above, and even beyond OpenAI's parallel tool calling.

The LLMCompiler has the following main components:

1. **Planner**: streams a DAG of tasks. Each task contains a tool, arguments, and list of dependencies.
2. **Task Fetching Unit** schedules and executes the tasks. This accepts a stream of tasks. This unit schedules tasks once their dependencies are met. Since many tools involve other calls to search engines or LLMs, the

extra parallelism can grant a significant speed boost (the paper claims 3.6x).

3. **Joiner**: dynamically replan or finish based on the entire graph history (including task execution results) is an LLM step that decides whether to respond with the final answer or whether to pass the progress back to the (re-)planning agent to continue work.

The key runtime-boosting ideas here are:

> **Planner** outputs are ***streamed;*** the output parser eagerly yields task parameters and their dependencies.
>
> The **task fetching unit** receives the parsed task stream and schedules tasks once all their dependencies are satisfied.
>
> Task arguments can be *variables,* which are the outputs of previous tasks in the DAG. For instance, the model can call `search("${1}")` to search for queries generated by the output of task 1. This lets the agent work even faster than the "embarrassingly parallel" tool calling in OpenAI.

By formatting tasks as a DAG, the agent can save precious time while invoking tools, leading to an overall better user experience.

Conclusion

These three agent architectures are prototypical of the "plan-and-execute" design pattern, which separates an LLM-powered "planner" from the tool execution runtime. If your application requires multiple tool invocations or API calls, these types of approaches can reduce the time it takes to return a final result and help you save costs by reducing the frequency of calls to more powerful LLMs.