Agentic Design Patterns Part 1

Four AI agent strategies that improve GPT-4 and GPT-3.5 performance.

Dear friends,

I think AI agent workflows will drive massive AI progress this year — perhaps even more than the next generation of foundation models. This is an important trend, and I urge everyone who works in AI to pay attention to it.

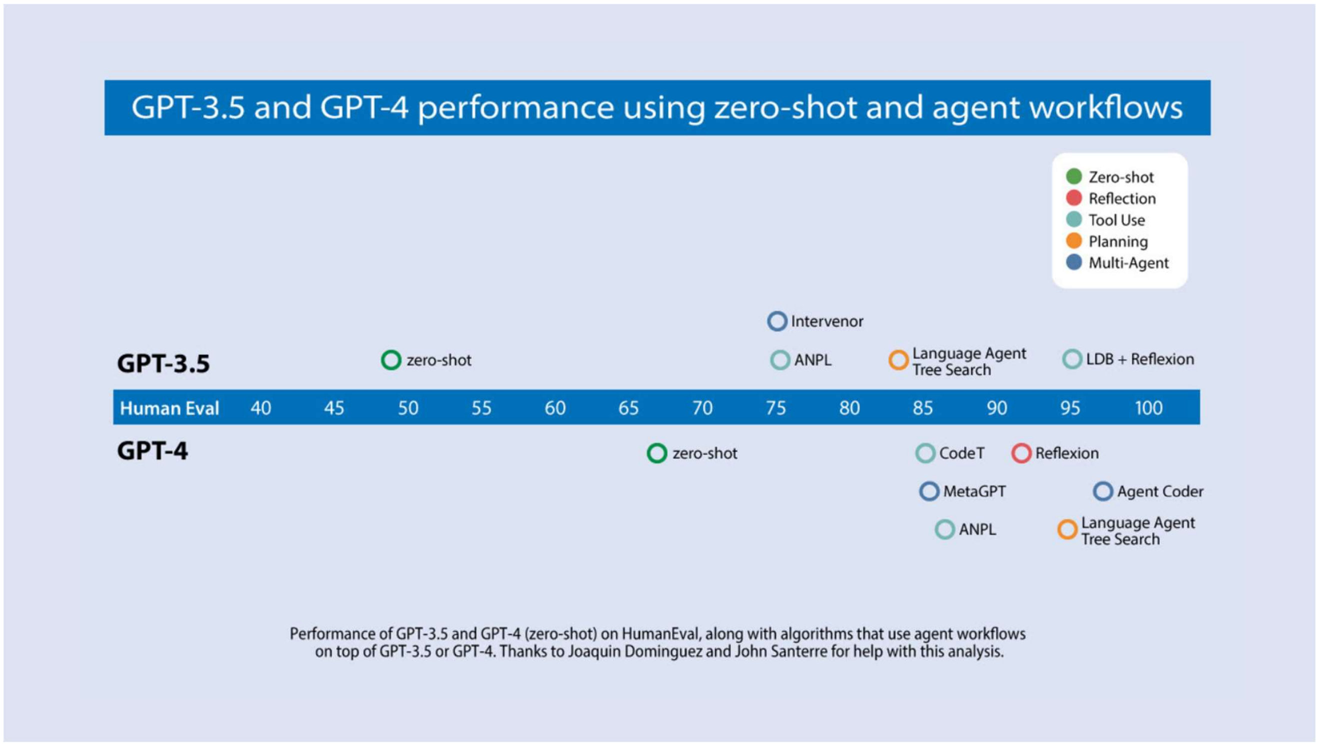
Today, we mostly use LLMs in zero-shot mode, prompting a model to generate final output token by token without revising its work. This is akin to asking someone to compose an essay from start to finish, typing straight through with no backspacing allowed, and expecting a high-quality result. Despite the difficulty, LLMs do amazingly well at this task!

With an agent workflow, however, we can ask the LLM to iterate over a document many times. For example, it might take a sequence of steps such as:

- Plan an outline.
- Decide what, if any, web searches are needed to gather more information.
- Write a first draft.
- Read over the first draft to spot unjustified arguments or extraneous information.
- Revise the draft taking into account any weaknesses spotted.
- And so on.

This iterative process is critical for most human writers to write good text. With AI, such an iterative workflow yields much better results than writing in a single pass.

Devin's splashy demo recently received a lot of social media buzz. My team has been closely following the evolution of AI that writes code. We analyzed results from a number of research teams, focusing on an algorithm's ability to do well on the widely used HumanEval coding benchmark. You can see our findings in the diagram below.



Performance of GPT-3.5 and GPT-4 (zero-shot) on HumanEval, along with algorithms that use agent workflows on top of GPT-3.5 or GPT-4. Thanks to Joaquin Dominguez and John Santerre for help with this analysis.

GPT-3.5 (zero shot) was 48.1% correct. GPT-4 (zero shot) does better at 67.0%. However, the improvement from GPT-3.5 to GPT-4 is dwarfed by incorporating an iterative agent workflow. Indeed, wrapped in an agent loop, GPT-3.5 achieves up to 95.1%.

Open source agent tools and the academic literature on agents are proliferating, making this an exciting time but also a confusing one. To help put this work into perspective, I'd like to share a framework for categorizing design patterns for building agents. My team AI Fund is successfully using these patterns in many applications, and I hope you find them useful.

- **Reflection**: The LLM examines its own work to come up with ways to improve it.
- **Tool Use**: The LLM is given tools such as web search, code execution, or any other function to help it gather information, take action, or process data.
- **Planning**: The LLM comes up with, and executes, a multistep plan to achieve a goal (for example, writing an outline for an essay, then doing online research, then writing a draft, and so on).
- **Multi-agent collaboration**: More than one AI agent work together, splitting up tasks and discussing and debating ideas, to come up with better solutions than a single agent would.

Next week, I'll elaborate on these design patterns and offer suggested readings for each.

Keep learning!

Andrew

Agentic Design Patterns Part 2, Reflection

Large language models can become more effective agents by reflecting on their own behavior.

Dear friends,

Last week, I described four design patterns for AI agentic workflows that I believe will drive significant progress this year: Reflection, Tool Use, Planning and Multi-agent collaboration. Instead of having an LLM generate its final output directly, an agentic workflow prompts the LLM multiple times, giving it opportunities to build step by step to higher-quality output. In this letter, I'd like to discuss Reflection. For a design pattern that's relatively quick to implement, I've seen it lead to surprising performance gains.

You may have had the experience of prompting ChatGPT/Claude/Gemini, receiving unsatisfactory output, delivering critical feedback to help the LLM improve its response, and then getting a better response. What if you automate the step of delivering critical feedback, so the model automatically criticizes its own output and improves its response? This is the crux of Reflection.

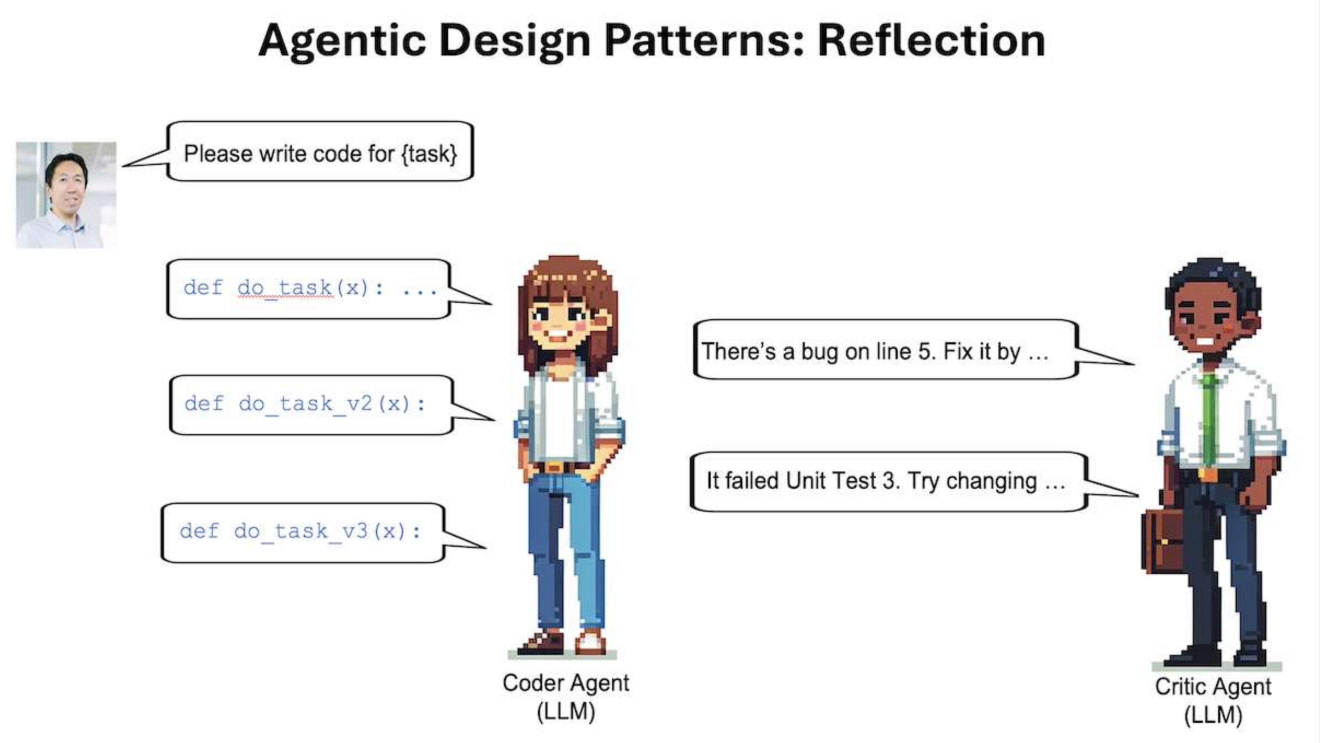
Take the task of asking an LLM to write code. We can prompt it to generate the desired code directly to carry out some task X. After that, we can prompt it to reflect on its own output, perhaps as follows:

Here's code intended for task X: [previously generated code]
Check the code carefully for correctness, style, and efficiency, and give constructive criticism for how to improve it.

Sometimes this causes the LLM to spot problems and come up with constructive suggestions. Next, we can prompt the LLM with context including (i) the previously generated code and the constructive feedback and (ii) ask it to use the feedback to rewrite the code. This can lead to a better response. Repeating the criticism/rewrite process might yield further improvements. This self-reflection process allows the LLM to spot gaps and improve its output on a variety of tasks including producing code, writing text, and answering questions.

And we can go beyond self-reflection by giving the LLM tools that help evaluate its output; for example, running its code through a few unit tests to check whether it generates correct results on test cases or searching the web to double-check text output. Then it can reflect on any errors it found and come up with ideas for improvement.

Further, we can implement Reflection using a multi-agent framework. I've found it convenient to create two different agents, one prompted to generate good outputs and the other prompted to give constructive criticism of the first agent's output. The resulting discussion between the two agents leads to improved responses.


Agentic Design Patterns: Reflection
Please write code for {task}
def do_task(x): ...
def do_task_v2(x):
def do_task_v3(x):
There's a bug on line 5. Fix it by ...
It failed Unit Test 3. Try changing ...
Coder Agent (LLM)
Critic Agent (LLM)

Reflection is a relatively basic type of agentic workflow, but I've been delighted by how much it improved my applications' results in a few cases. I hope you will try it in your own work. If you're interested in learning more about reflection, I recommend these papers:

- "Self-Refine: Iterative Refinement with Self-Feedback," Madaan et al. (2023)
- "Reflexion: Language Agents with Verbal Reinforcement Learning," Shinn et al. (2023)
- "CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing," Gou et al. (2024)

I'll discuss the other agentic design patterns in future letters.

Keep learning!

Andrew

Agentic Design Patterns Part 3, Tool Use

How large language models can act as agents by taking advantage of external tools for search, code execution, productivity, Ad infinitum.

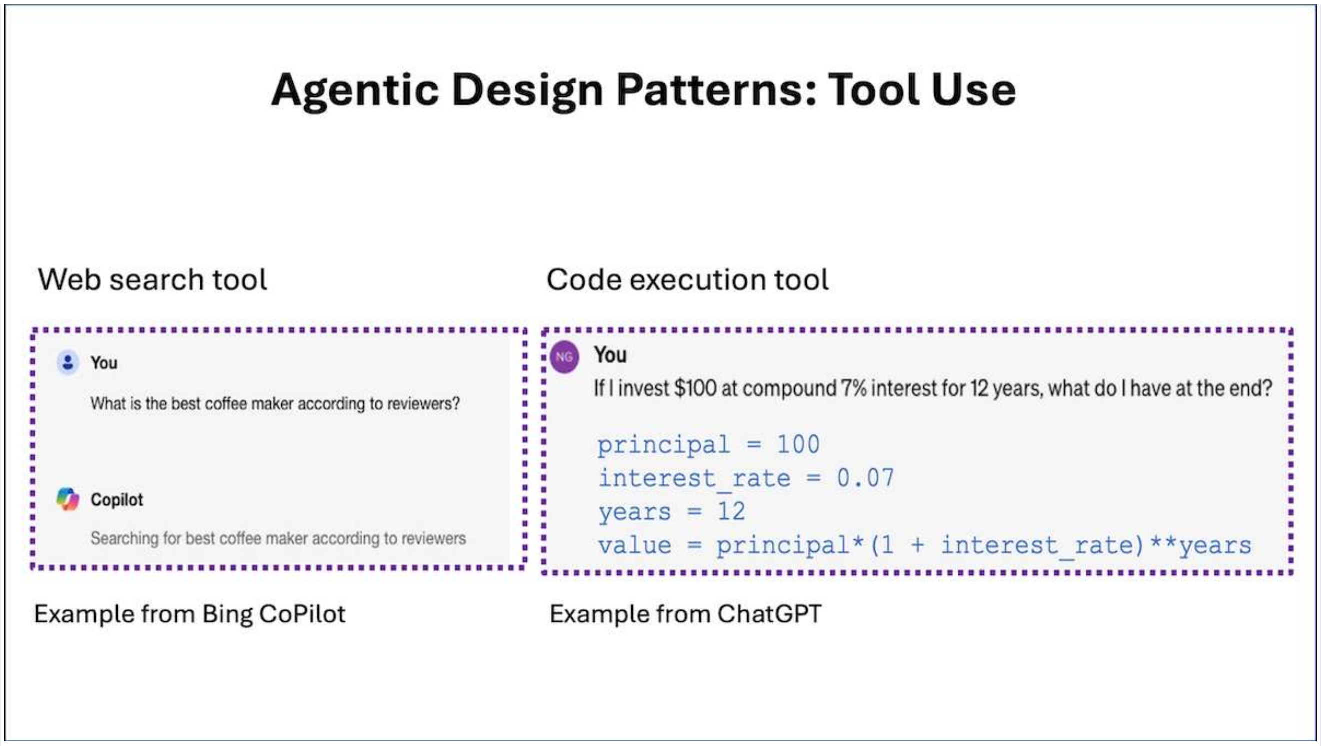
Dear friends,

Tool Use, in which an LLM is given functions it can request to call for gathering information, taking action, or manipulating data, is a key design pattern of AI agentic workflows. You may be familiar with LLM-based systems that can perform a web search or execute code. Indeed, some large, consumer-facing LLMs already incorporate these features. But Tool Use goes well beyond these examples.

If you prompt an online LLM-based chat system, “What is the best coffee maker according to reviewers?”, it might decide to carry out a web search and download one or more web pages to gain context. Early on, LLM developers realized that relying only on a pre-trained transformer to generate output tokens is limiting, and that giving an LLM a tool for web search lets it do much more. With such a tool, an LLM is either fine-tuned or prompted (perhaps with few-shot prompting) to generate a special string like *{tool: web-search, query: "coffee maker reviews"}* to request calling a search engine. (The exact format of the string depends on the implementation.) A post-processing step then looks for strings like these, calls the web search function with the relevant parameters when it finds one, and passes the result back to the LLM as additional input context for further processing.

Similarly, if you ask, “If I invest $100 at compound 7% interest for 12 years, what do I have at the end?”, rather than trying to generate the answer directly using a transformer network — which is unlikely to result in the right answer — the LLM might use a code execution tool to run a Python command to compute 1*00 * (1+0.07)**12* to get the right answer. The LLM might generate a string like this: *{tool: python-interpreter, code: "100 * (1+0.07)**12"}*.

But Tool Use in agentic workflows now goes much further. Developers are using functions to search different sources (web, Wikipedia, arXiv, etc.), to interface with productivity tools (send email, read/write calendar entries, etc.), generate or interpret images, and much more. We can prompt an LLM using context that gives detailed descriptions of many functions. These descriptions might include a text description of what the function does plus details of what arguments the function expects. And we’d expect the LLM to automatically choose the right function to call to do a job. Further, systems are being built in which the LLM has access to hundreds of tools. In such settings, there might be too many functions at your disposal to put all of them into the LLM context, so you might use heuristics to pick the most relevant subset to include in the LLM context at the current step of processing. This technique, which is described in the Gorilla paper cited below, is reminiscent of how, if there is too much text to include as context, retrieval augmented generation (RAG) systems offer heuristics for picking a subset of the text to include.


Agentic Design Patterns: Tool Use

Web search tool

You

What is the best coffee maker according to reviewers?

Copilot

Searching for best coffee maker according to reviewers

Example from Bing CoPilot

Code execution tool

You

If I invest $100 at compound 7% interest for 12 years, what do I have at the end?

```
principal = 100
interest_rate = 0.07
years = 12
value = principal*(1 + interest_rate)**years
```

Example from ChatGPT

Early in the history of LLMs, before widespread availability of large multimodal models (LMMs) like LLaVa, GPT-4V, and Gemini, LLMs could not process images directly, so a lot of work on Tool Use was carried out by the computer vision community. At that time, the only way for an LLM-based system to manipulate an image was by calling a function to, say, carry out object recognition or some other function on it. Since then, practices for Tool Use have exploded. GPT-4's function calling capability, released in the middle of last year, was a significant step toward a general-purpose implementation. Since then, more and more LLMs are being developed to be similarly facile with Tool Use.

If you're interested in learning more about Tool Use, I recommend:

- "Gorilla: Large Language Model Connected with Massive APIs," Patil et al. (2023)
- "MM-REACT: Prompting ChatGPT for Multimodal Reasoning and Action," Yang et al. (2023)
- "Efficient Tool Use with Chain-of-Abstraction Reasoning," Gao et al. (2024)

Both Tool Use and Reflection, which I described in last week's letter, are design patterns that I can get to work fairly reliably on my applications — both are capabilities well worth learning about. In future letters, I'll describe the Planning and Multi-agent collaboration design patterns. They allow AI agents to do much more but are less mature, less predictable — albeit very exciting — technologies.

Keep learning!

Andrew

Agentic Design Patterns Part 4, Planning

Large language models can drive powerful agents to execute complex tasks if you ask them to plan the steps before they act.

Dear friends,

Planning is a key agentic AI design pattern in which we use a large language model (LLM) to autonomously decide on what sequence of steps to execute to accomplish a larger task. For example, if we ask an agent to do online research on a given topic, we might use an LLM to break down the objective into smaller subtasks, such as researching specific subtopics, synthesizing findings, and compiling a report.

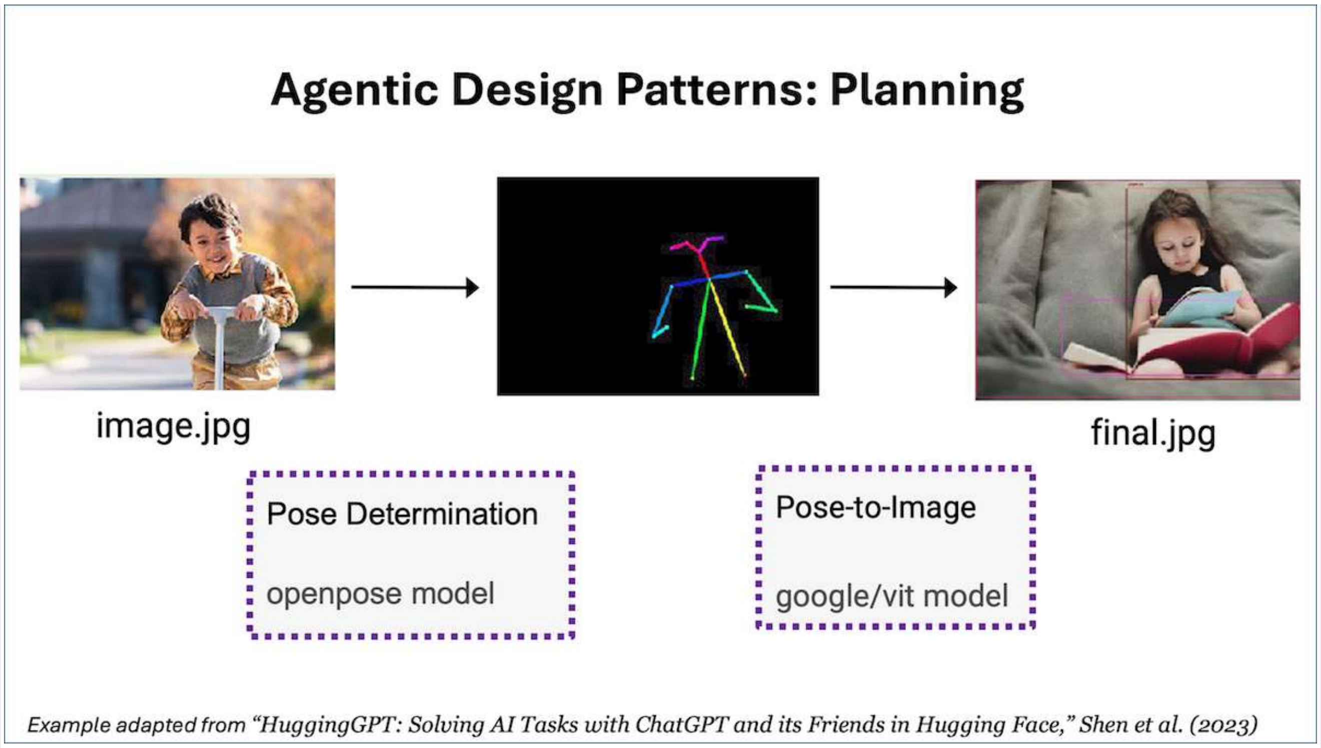
Many people had a “ChatGPT moment” shortly after ChatGPT was released, when they played with it and were surprised that it significantly exceeded their expectation of what AI can do. If you have not yet had a similar “AI Agentic moment,” I hope you will soon. I had one several months ago, when I presented a live demo of a research agent I had implemented that had access to various online search tools.

I had tested this agent multiple times privately, during which it consistently used a web search tool to gather information and wrote up a summary. During the live demo, though, the web search API unexpectedly returned with a rate limiting error. I thought my demo was about to fail publicly, and I dreaded what was to come next. To my surprise, the agent pivoted deftly to a Wikipedia search tool — which I had forgotten I’d given it — and completed the task using Wikipedia instead of web search.

This was an AI Agentic moment of surprise for me. I think many people who haven’t experienced such a moment yet will do so in the coming months. It’s a beautiful thing when you see an agent autonomously decide to do things in ways that you had not anticipated, and succeed as a result!

Many tasks can’t be done in a single step or with a single tool invocation, but an agent can decide what steps to take. For example, to simplify an example from the HuggingGPT paper (cited below), if you want an agent to consider a picture of a boy and draw a picture of a girl in the same pose, the task might be decomposed into two distinct steps: (i) detect the pose in the picture of the boy and (ii) render a picture of a girl in the detected pose. An LLM might be fine-tuned or prompted (with few-shot prompting) to specify a plan by outputting a string like "*{tool: pose-detection, input: image.jpg, output: temp1 } {tool: pose-to-image, input: temp1, output: final.jpg}*".

This structured output, which specifies two steps to take, then triggers software to invoke a pose detection tool followed by a pose-to-image tool to complete the task. (This example is for illustrative purposes only; HuggingGPT uses a different format.)


Agentic Design Patterns: Planning
image.jpg
final.jpg
Pose Determination
openpose model
Pose-to-Image
google/vit model
Example adapted from "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face," Shen et al. (2023)

Admittedly, many agentic workflows do not need planning. For example, you might have an agent reflect on, and improve, its output a fixed number of times. In this case, the sequence of steps the agent takes is fixed and deterministic. But for complex tasks in which you aren't able to specify a decomposition of the task into a set of steps ahead of time, Planning allows the agent to decide dynamically what steps to take.

On one hand, Planning is a very powerful capability; on the other, it leads to less predictable results. In my experience, while I can get the agentic design patterns of Reflection and Tool Use to work reliably and improve my applications' performance, Planning is a less mature technology, and I find it hard to predict in advance what it will do. But the field continues to evolve rapidly, and I'm confident that Planning abilities will improve quickly.

If you're interested in learning more about Planning with LLMs, I recommend:

- "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Wei et al. (2022)
- "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face," Shen et al. (2023)
- "Understanding the planning of LLM agents: A survey," by Huang et al. (2024)

Keep learning!

Andrew

Agentic Design Patterns Part 5, Multi-Agent Collaboration

Prompting an LLM to play different roles for different parts of a complex task summons a team of AI agents that can do the job more effectively.

Dear friends,

Multi-agent collaboration is the last of the four key AI agentic design patterns that I've described in recent letters. Given a complex task like writing software, a multi-agent approach would break down the task into subtasks to be executed by different roles — such as a software engineer, product manager, designer, QA (quality assurance) engineer, and so on — and have different agents accomplish different subtasks.

Different agents might be built by prompting one LLM (or, if you prefer, multiple LLMs) to carry out different tasks. For example, to build a software engineer agent, we might prompt the LLM: "You are an expert in writing clear, efficient code. Write code to perform the task . . .."

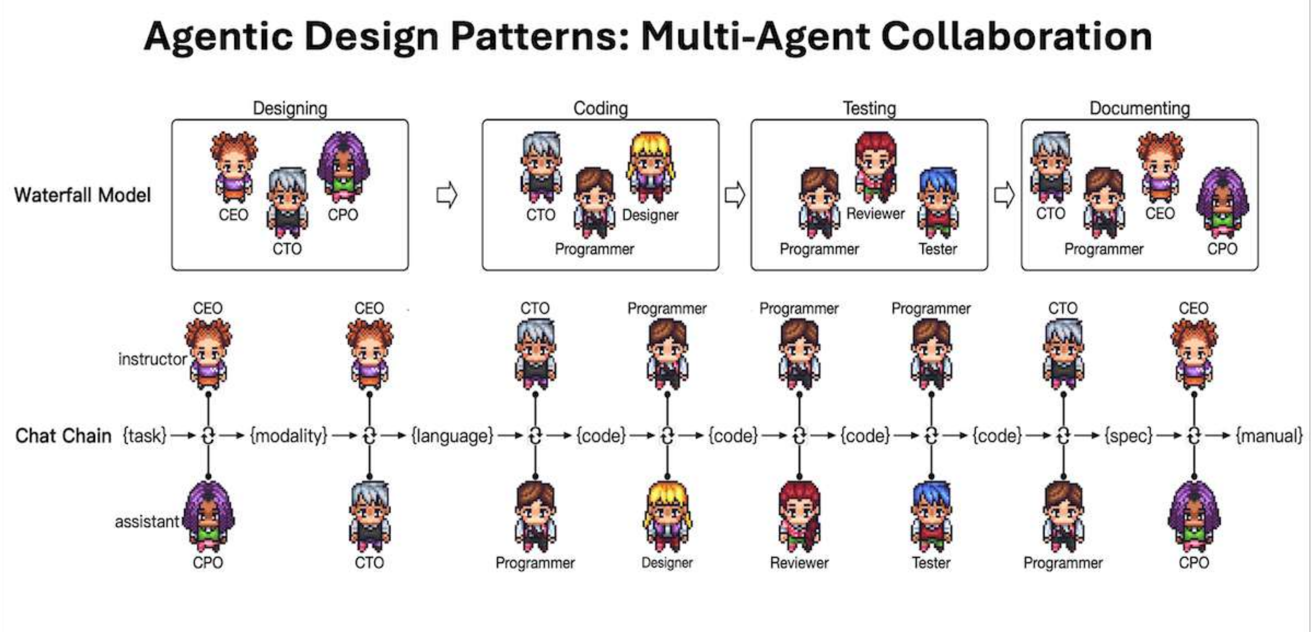
It might seem counterintuitive that, although we are making multiple calls to the same LLM, we apply the programming abstraction of using multiple agents. I'd like to offer a few reasons:

- It works! Many teams are getting good results with this method, and there's nothing like results! Further, ablation studies (for example, in the AutoGen paper cited below) show that multiple agents give superior performance to a single agent.
- Even though some LLMs today can accept very long input contexts (for instance, Gemini 1.5 Pro accepts 1 million tokens), their ability to truly understand long, complex inputs is mixed. An agentic workflow in which the LLM is prompted to focus on one thing at a time can give better performance. By telling it when it should play software engineer, we can also specify what is important in that role's subtask. For example, the prompt above emphasized clear, efficient code as opposed to, say, scalable and highly secure code. By decomposing the overall task into subtasks, we can optimize the subtasks better.
- Perhaps most important, the multi-agent design pattern gives us, as developers, a framework for breaking down complex tasks into subtasks. When writing code to run on a single CPU, we often break our program up into different processes or threads. This is a useful abstraction that lets us decompose a task, like implementing a web browser, into subtasks that are easier to code. I find thinking through multi-agent roles to be a useful abstraction as well.

In many companies, managers routinely decide what roles to hire, and then how to split complex projects — like writing a large piece of software or preparing a research report — into smaller tasks to assign to employees with different specialties. Using multiple agents is analogous. Each agent implements its own workflow, has its own memory (itself a rapidly evolving area in agentic technology: how can an agent remember enough of its past interactions to perform better on upcoming ones?), and may ask other agents for help. Agents can also engage in Planning and Tool Use. This results in a cacophony of LLM calls and message passing between agents that can result in very complex workflows.

While managing people is hard, it's a sufficiently familiar idea that it gives us a mental framework for how to "hire" and assign tasks to our AI agents. Fortunately, the damage from mismanaging an AI agent is much lower than that from mismanaging humans!

Emerging frameworks like AutoGen, Crew AI, and LangGraph, provide rich ways to build multi-agent solutions to problems. If you're interested in playing with a fun multi-agent system, also check out ChatDev, an open source implementation of a set of agents that run a virtual software company. I encourage you to check out their GitHub repo and perhaps clone the repo and run the system yourself. While it may not always produce what you want, you might be amazed at how well it does.



Proposed ChatDev architecture. Image adapted from "Communicative Agents for Software Development," Qian et al. (2023).

Like the design pattern of Planning, I find the output quality of multi-agent collaboration hard to predict, especially when allowing agents to interact freely and providing them with multiple tools. The more mature patterns of Reflection and Tool Use are more reliable. I hope you enjoy playing with these agentic design patterns and that they produce amazing results for you!

If you're interested in learning more, I recommend:

- "Communicative Agents for Software Development," Qian et al. (2023) (the ChatDev paper)
- "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," Wu et al. (2023)
- "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework," Hong et al. (2023)

Keep learning!

Andrew