Don't Build Multi-Agents

Frameworks for LLM Agents have been surprisingly disappointing. I want to offer some principles for building agents based on our own trial & error, and explain why some tempting ideas are actually quite bad in practice.

2025-06-12 WALDEN YAN

FOLLOW US ON:

LINKEDIN TWITTER | X

Principles of Context Engineering

We'll work our way up to the following principles:

1. Share context
2. Actions carry implicit decisions

Why think about principles?

HTML was introduced in 1993. In 2013, Facebook released React to the world. It is now 2025 and React (and its descendants) dominates the way developers build sites and apps. Why? Because React is not just a scaffold for writing code. It is a philosophy. By using React, you embrace building applications with a pattern of reactivity and modularity, which people now accept to be a standard requirement, but this was not always obvious to early web developers.

In the age of LLMs and building AI Agents, it feels like we're still playing with raw HTML & CSS and figuring out how to fit these together to make a good experience. No single approach to building agents has become the standard yet, besides some of the absolute basics.

> In some cases, libraries such as https://github.com/openai/swarm by OpenAI and https://github.com/microsoft/autogen by Microsoft actively push concepts which I believe to be the wrong way of building agents. Namely, using multi-agent architectures, and I'll explain why.

That said, if you're new to agent-building, there are lots of resources on how to set up the basic scaffolding [1] [2]. But when it comes to building serious production applications, it's a different story.

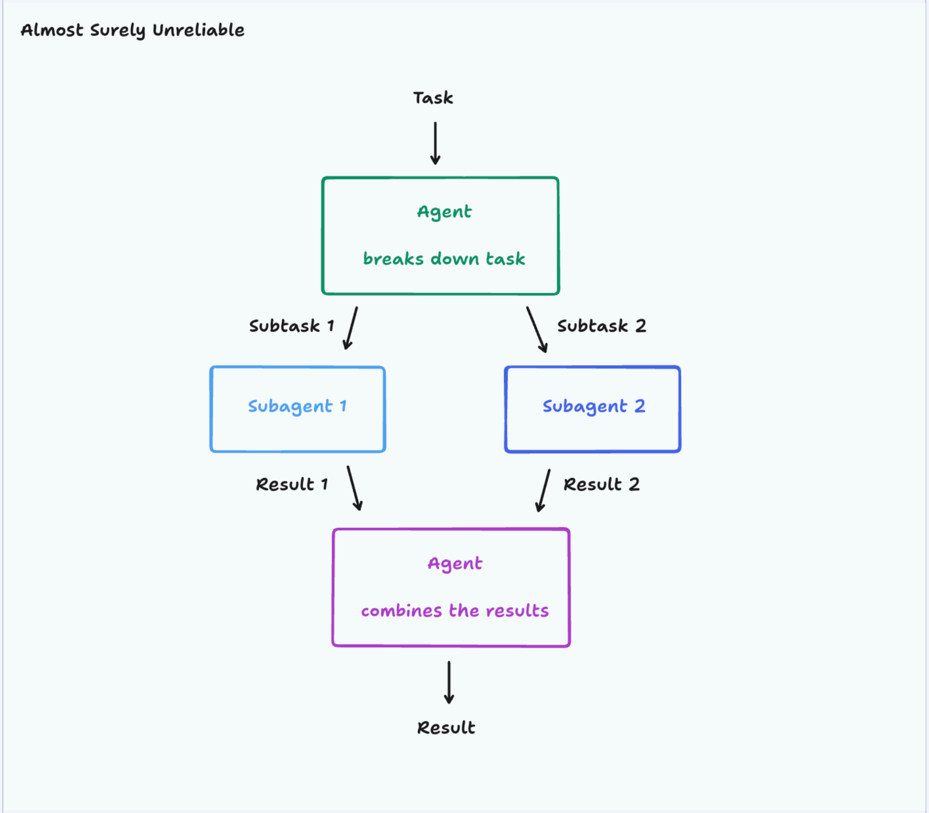
Let's start with reliability. When agents have to actually be reliable while running for long periods of time and maintain coherent conversations, there are certain things you must do to contain the potential for compounding errors. Otherwise, if you're not careful, things fall apart quickly. At the core of reliability is Context Engineering.

Context Engineering

In 2025, the models out there are extremely intelligent. But even the smartest human won't be able to do their job effectively without the context of what they're being asked to do. "Prompt engineering" was coined as a term for the effort needing to write your task in the ideal format for a LLM chatbot. "Context engineering" is the next level of this. It is about doing this automatically in a dynamic system. It takes more nuance and is effectively the #1 job of engineers building AI agents.

Take an example of a common type of agent. This agent

1. breaks its work down into multiple parts
2. starts subagents to work on those parts
3. combines those results in the end



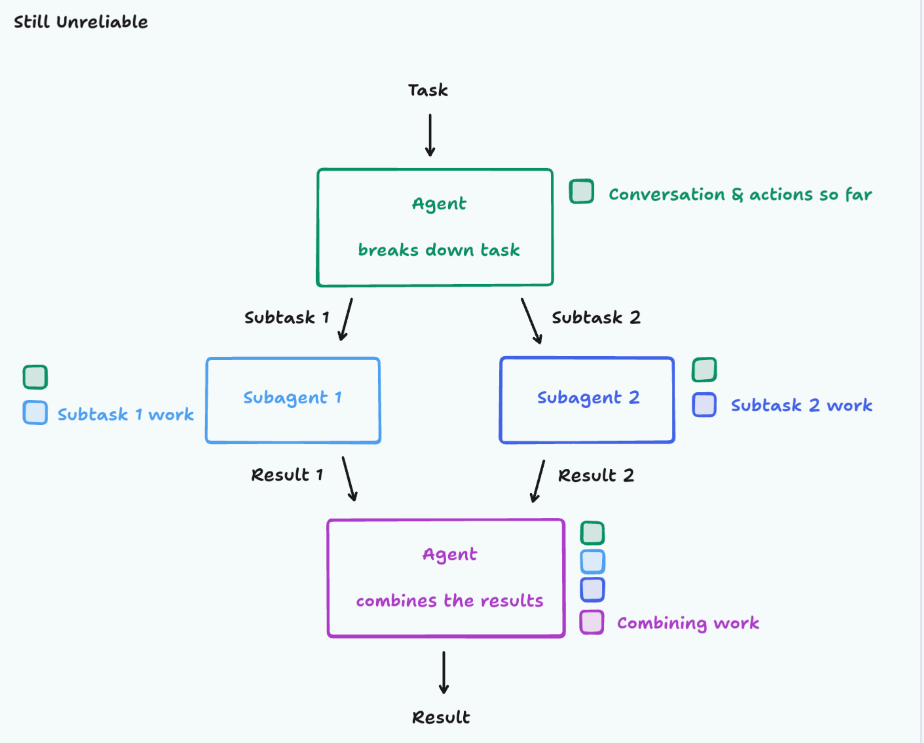
This is a tempting architecture, especially if you work in a domain of tasks with several parallel components to it. However, it is very fragile. The key failure point is this:

> down”.
>
> It turns out subagent 1 actually mistook your subtask and started building a background that looks like Super Mario Bros. Subagent 2 built you a bird, but it doesn’t look like a game asset and it moves nothing like the one in Flappy Bird. Now the final agent is left with the undesirable task of combining these two miscommunications.

This may seem contrived, but most real-world tasks have many layers of nuance that all have the potential to be miscommunicated. You might think that a simple solution would be to just copy over the original task as context to the subagents as well. That way, they don’t misunderstand their subtask. But remember that in a real production system, the conversation is most likely multi-turn, the agent probably had to make some tool calls to decide how to break down the task, and any number of details could have consequences on the interpretation of the task.

> *Principle 1*
> Share context, and share full agent traces, not just individual messages

Let’s take another revision at our agent, this time making sure each agent has the context of the previous agents.



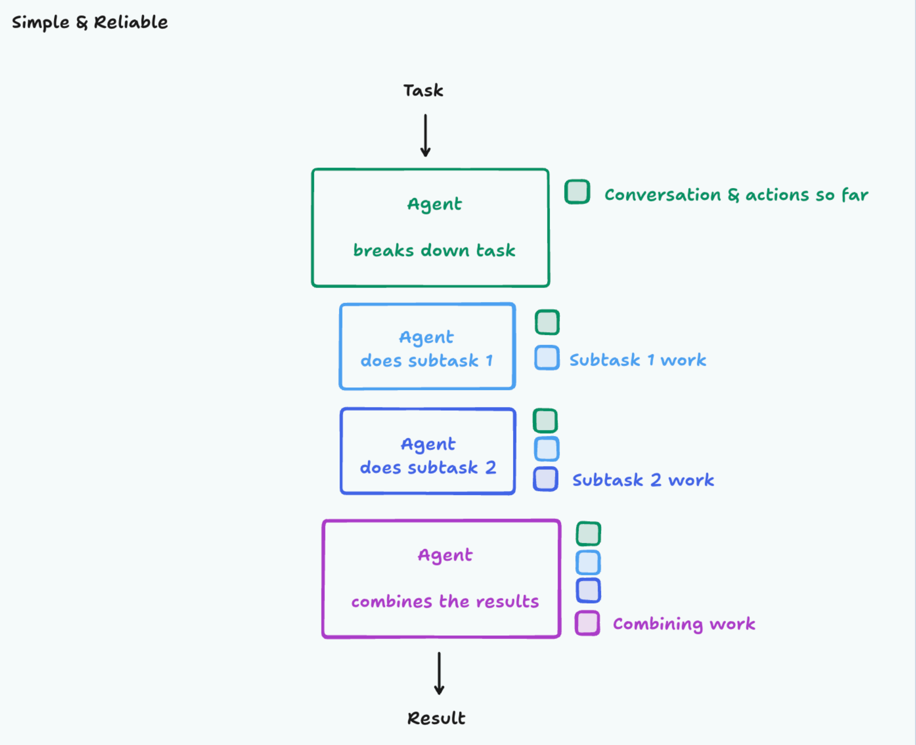
agent 2 cannot not see what the other was doing and so their work ends up being inconsistent with each other.

The actions subagent 1 took and the actions subagent 2 took were based on conflicting assumptions not prescribed upfront.

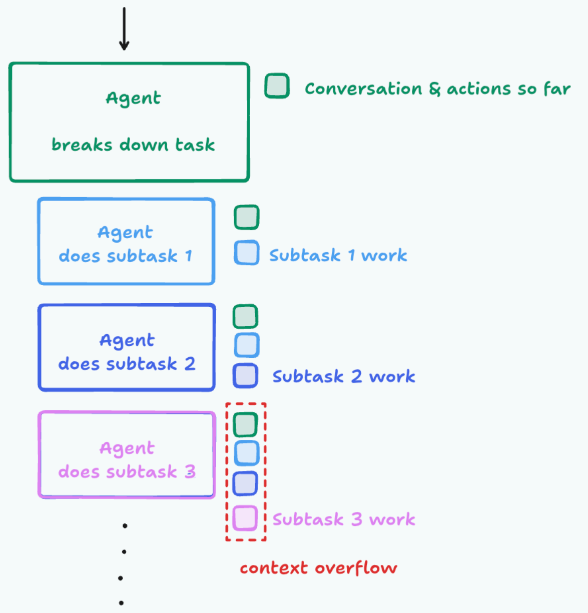
> *Principle 2*
> Actions carry implicit decisions, and conflicting decisions carry bad results

I would argue that Principles 1 & 2 are so critical, and so rarely worth violating, that you should by default rule out any agent architectures that don't abide by then. You might think this is constraining, but there is actually a wide space of different architectures you could still explore for your agent.

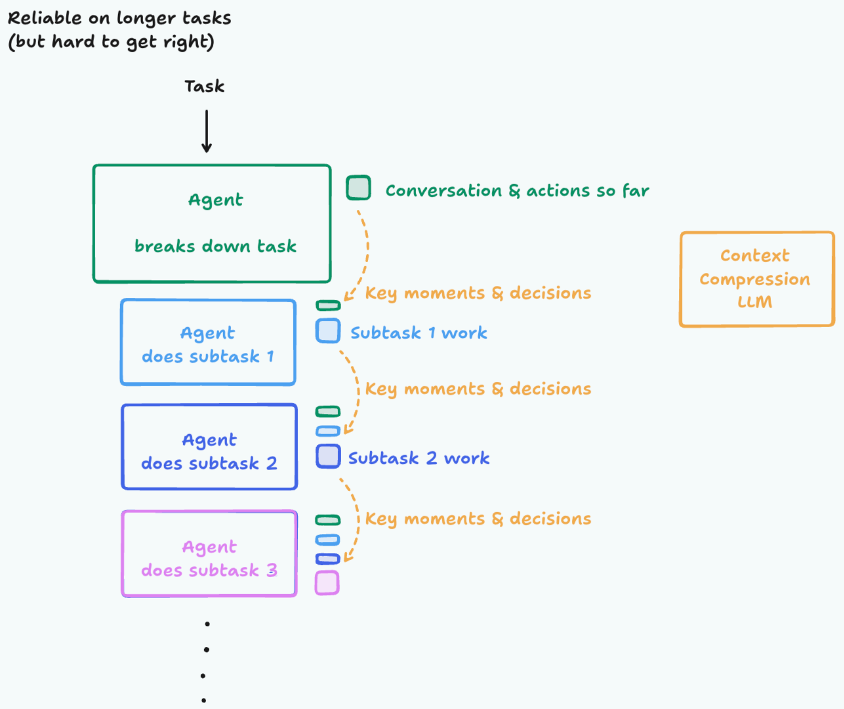
The simplest way to follow the principles is to just use a single-threaded linear agent:



Here, the context is continuous. However, you might run into issues for very large tasks with so many subparts that context windows start to overflow.



To be honest, the simple architecture will get you very far, but for those who have truly long-duration tasks, and are willing to put in the effort, you can do even better. There are several ways you could solve this, but today I will present just one:



In this world, we introduce a new LLM model whose key purpose is to compress a history of actions & conversation into key details, events, and decimodel (this is in fact something we've done at Cognition).

The benefit you get is an agent that is effective at longer contexts. You will still eventually hit a limit though. For the avid reader, I encourage you to think of better ways to manage arbitrarily long contexts. It ends up being quite a deep rabbit hole!

Applying the Principles

If you're an agent-builder, ensure your agent's every action is informed by the context of all relevant decisions made by other parts of the system. Ideally, every action would just see everything else. Unfortunately, this is not always possible due to limited context windows and practical tradeoffs, and you may need to decide what level of complexity you are willing to take on for the level of reliability you aim for.

As you think about architecting your agents to avoid conflicting decision-making, here are some real-world examples to ponder:

Claude Code Subagents
As of June 2025, Claude Code is an example of an agent that spawns sub-tasks. However, it never does work in parallel with the subtask agent, and the subtask agent is usually only tasked with answering a question, not writing any code. Why? The subtask agent lacks context from the main agent that would otherwise be needed to do anything beyond answering a well-defined question. And if they were to run multiple parallel subagents, they might give conflicting responses, resulting in the reliability issues we saw with our earlier examples of agents. The benefit of having a subagent in this case is that all the subagent's investigative work does not need to remain in the history of the main agent, allowing for longer traces before running out of context. The designers of Claude Code took a purposefully simple approach.

Edit Apply Models
In 2024, many models were really bad at editing code. A common practice among coding agents, IDEs, app builders, etc. (including Devin) was to use an "edit apply model." The key idea was that it was actually more reliable to get a small model to rewrite your entire file, given a markdown explanation of the changes you wanted, than to get a large model to output a properly formatted diff. So, builders had the large models output markdown explanations of code edits and then fed these markdown explanations to small models to actually rewrite the files. However, these systems would still be very faulty. Often times, for example, the small model would misinterpret the instructions of the large model and make an incorrect edit due to the most slight ambiguities in the instructions. Today, the edit decision-making and applying are more often done by a single model in one action.

Multi-Agents

If we really want to get parallelism out of our system, you might think to let the decision makers "talk" to each other and work things out.

not quite able to engage in this style of long-context proactive discourse with much more reliability than you would get with a single agent. Humans are quite efficient at communicating our most important knowledge to one another, but this efficiency takes nontrivial intelligence.

Since not long after the launch of ChatGPT, people have been exploring the idea of multiple agents interacting with one another to achieve goals [3][4]. While I'm optimistic about the long-term possibilities of agents collaborating with one another, it is evident that in 2025, running multiple agents in collaboration only results in fragile systems. The decision-making ends up being too dispersed and context isn't able to be shared thoroughly enough between the agents. At the moment, I don't see anyone putting a dedicated effort to solving this difficult cross-agent context-passing problem. I personally think it will come for free as we make our single-threaded agents even better at communicating with humans. When this day comes, it will unlock much greater amounts of parallelism and efficiency.

Toward a More General Theory

These observations on context engineering are just the start to what we might someday consider the standard principles of building agents. And there are many more challenges and techniques not discussed here. At Cognition, agent building is a key frontier we think about. We build our internal tools and frameworks around these principles we repeatedly find ourselves relearning as a way to enforce these ideas. But our theories are likely not perfect, and we expect things to change as the field advances, so some flexibility and humility is required as well.

We welcome you to try our work at app.devin.ai. And if you would enjoy discovering some of these agent-building principles with us, reach out to walden@cognition.ai

HIRE

[DEVIN The AI software engineer]

GET STARTED WITH DEVIN ↗

LEARN ABOUT DEVIN ↗