# Building effective agents

2024年12月20日

Over the past year, we've worked with dozens of teams building large language model (LLM) agents across industries. Consistently, the most successful implementations weren't using complex frameworks or specialized libraries. Instead, they were building with simple, composable patterns.

In this post, we share what we’ve learned from working with our customers and building agents ourselves, and give practical advice for developers on building effective agents.

## What are agents?

"Agent" can be defined in several ways. Some customers define agents as fully autonomous systems that operate independently over extended periods, using various tools to accomplish complex tasks. Others use the term to describe more prescriptive implementations that follow predefined workflows. At Anthropic, we categorize all these variations as **agentic systems**, but draw an important architectural distinction between **workflows**and**agents**:

* **Workflows** are systems where LLMs and tools are orchestrated through predefined code paths.
* **Agents**, on the other hand, are systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

Below, we will explore both types of agentic systems in detail. In Appendix 1 (“Agents in Practice”), we describe two domains where customers have found particular value in using these kinds of systems.

## When (and when not) to use agents

When building applications with LLMs, we recommend finding the simplest solution possible, and only increasing complexity when needed. This might mean not building agentic systems at all. Agentic systems often trade latency and cost for better task performance, and you should consider when this tradeoff makes sense.

When more complexity is warranted, workflows offer predictability and consistency for well-defined tasks, whereas agents are the better option when flexibility and model-driven decision-making are needed at scale. For many applications, however, optimizing single LLM calls with retrieval and in-context examples is usually enough.

## When and how to use frameworks

There are many frameworks that make agentic systems easier to implement, including:

* [LangGraph](https://langchain-ai.github.io/langgraph/) from LangChain;
* Amazon Bedrock's [AI Agent framework](https://aws.amazon.com/bedrock/agents/);
* [Rivet](https://rivet.ironcladapp.com/), a drag and drop GUI LLM workflow builder; and
* [Vellum](https://www.vellum.ai/), another GUI tool for building and testing complex workflows.

These frameworks make it easy to get started by simplifying standard low-level tasks like calling LLMs, defining and parsing tools, and chaining calls together. However, they often create extra layers of abstraction that can obscure the underlying prompts ​​and responses, making them harder to debug. They can also make it tempting to add complexity when a simpler setup would suffice.

We suggest that developers start by using LLM APIs directly: many patterns can be implemented in a few lines of code. If you do use a framework, ensure you understand the underlying code. Incorrect assumptions about what's under the hood are a common source of customer error.

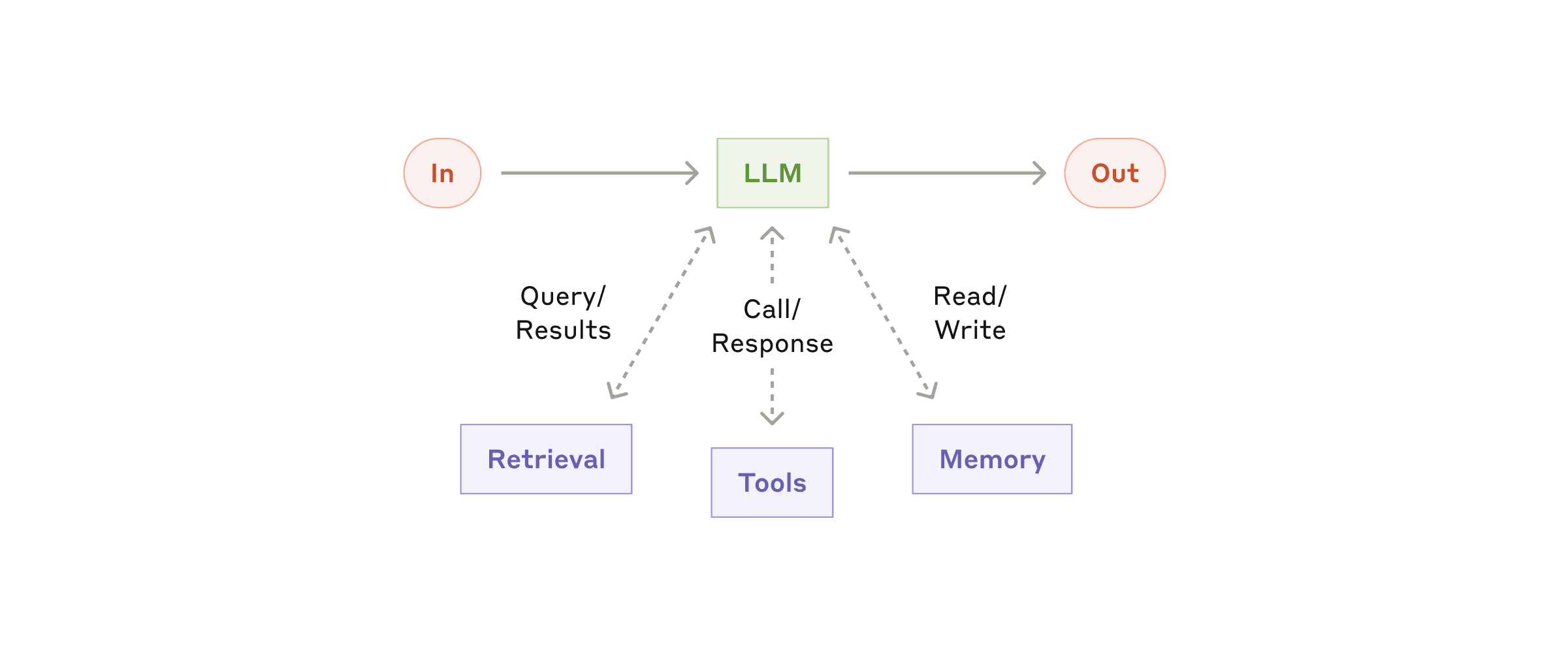
See our [cookbook](https://github.com/anthropics/anthropic-cookbook/tree/main/patterns/agents) for some sample implementations.

## Building blocks, workflows, and agents

In this section, we’ll explore the common patterns for agentic systems we’ve seen in production. We'll start with our foundational building block—the augmented LLM—and progressively increase complexity, from simple compositional workflows to autonomous agents.

### Building block: The augmented LLM

The basic building block of agentic systems is an LLM enhanced with augmentations such as retrieval, tools, and memory. Our current models can actively use these capabilities—generating their own search queries, selecting appropriate tools, and determining what information to retain.

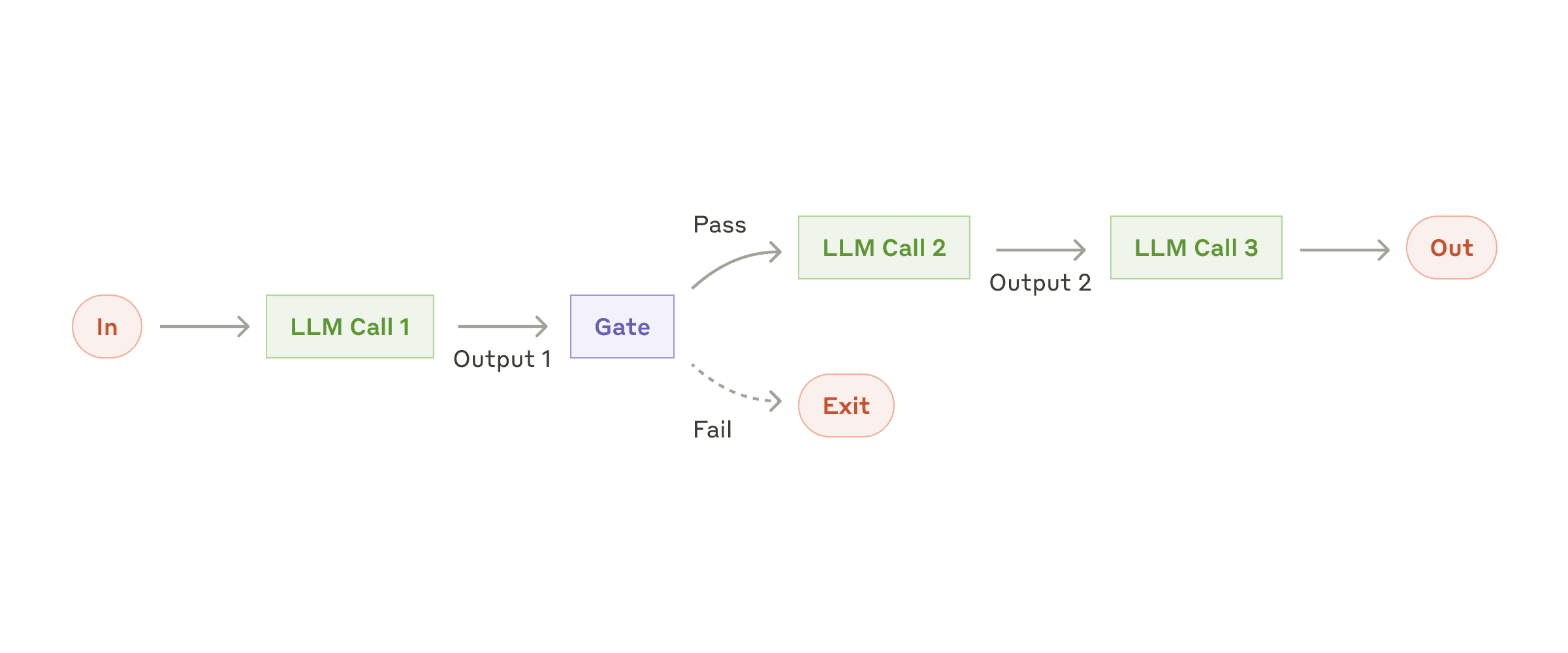
The augmented LLM

We recommend focusing on two key aspects of the implementation: tailoring these capabilities to your specific use case and ensuring they provide an easy, well-documented interface for your LLM. While there are many ways to implement these augmentations, one approach is through our recently released [Model Context Protocol](https://www.anthropic.com/news/model-context-protocol), which allows developers to integrate with a growing ecosystem of third-party tools with a simple [client implementation](https://modelcontextprotocol.io/tutorials/building-a-client" \l "building-mcp-clients).

For the remainder of this post, we'll assume each LLM call has access to these augmented capabilities.

### Workflow: Prompt chaining

Prompt chaining decomposes a task into a sequence of steps, where each LLM call processes the output of the previous one. You can add programmatic checks (see "gate” in the diagram below) on any intermediate steps to ensure that the process is still on track.

The prompt chaining workflow

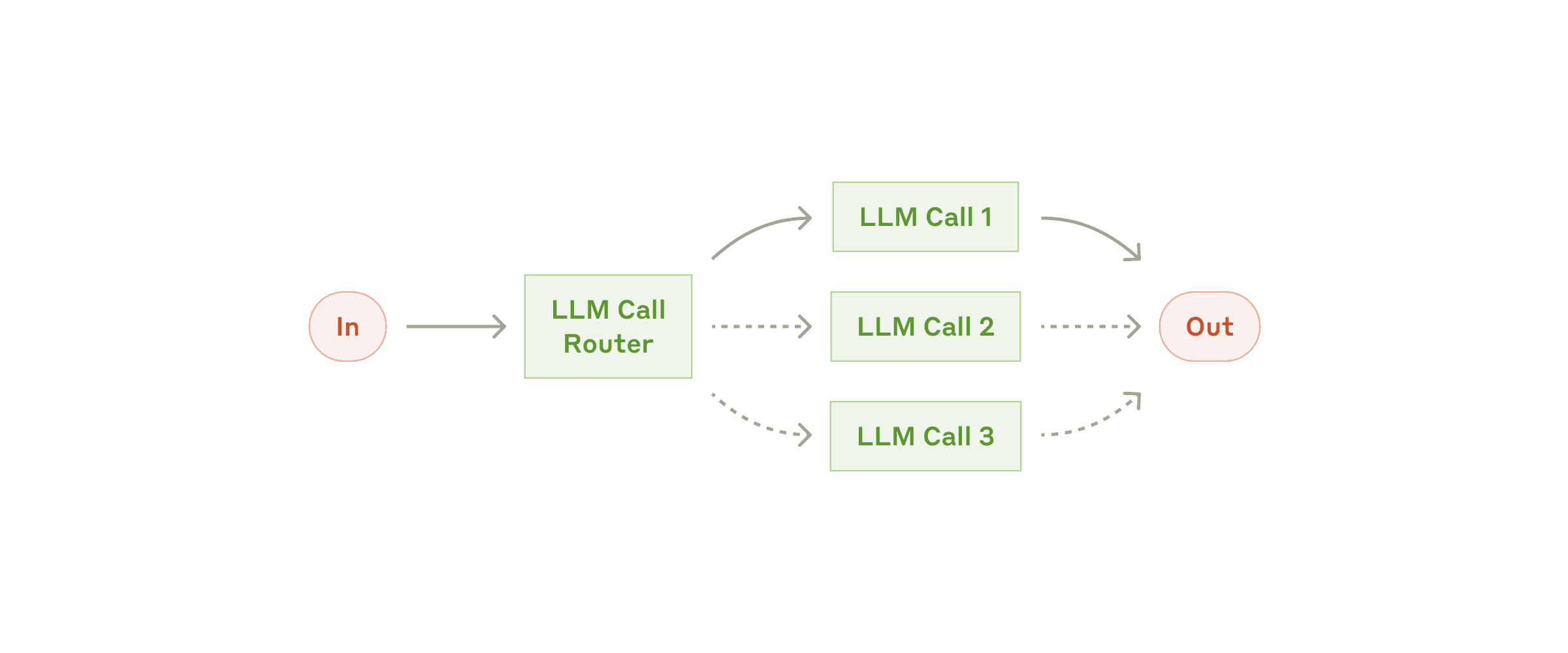
**When to use this workflow:** This workflow is ideal for situations where the task can be easily and cleanly decomposed into fixed subtasks. The main goal is to trade off latency for higher accuracy, by making each LLM call an easier task.

**Examples where prompt chaining is useful:**

* Generating Marketing copy, then translating it into a different language.
* Writing an outline of a document, checking that the outline meets certain criteria, then writing the document based on the outline.

### Workflow: Routing

Routing classifies an input and directs it to a specialized followup task. This workflow allows for separation of concerns, and building more specialized prompts. Without this workflow, optimizing for one kind of input can hurt performance on other inputs.

The routing workflow

**When to use this workflow:** Routing works well for complex tasks where there are distinct categories that are better handled separately, and where classification can be handled accurately, either by an LLM or a more traditional classification model/algorithm.

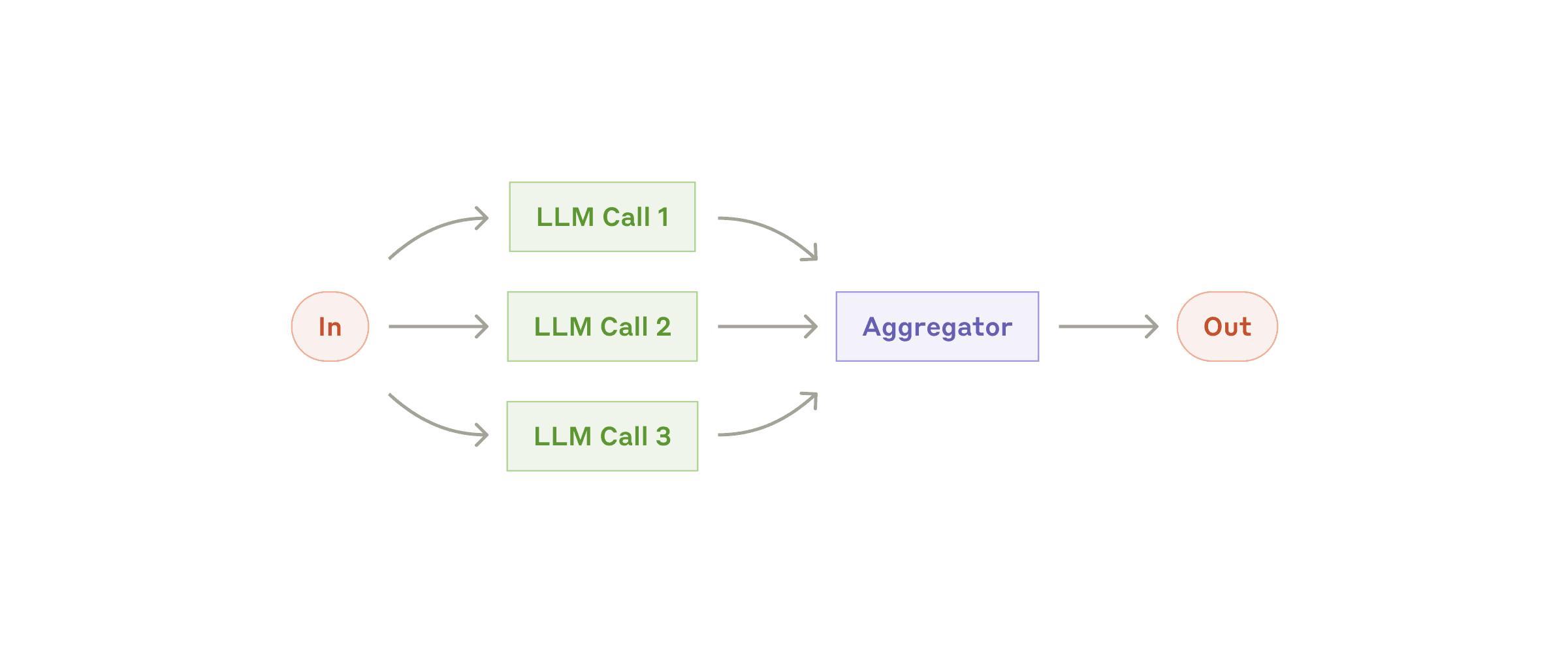
**Examples where routing is useful:**

* Directing different types of customer service queries (general questions, refund requests, technical support) into different downstream processes, prompts, and tools.
* Routing easy/common questions to smaller models like Claude 3.5 Haiku and hard/unusual questions to more capable models like Claude 3.5 Sonnet to optimize cost and speed.

### Workflow: Parallelization

LLMs can sometimes work simultaneously on a task and have their outputs aggregated programmatically. This workflow, parallelization, manifests in two key variations:

* **Sectioning**: Breaking a task into independent subtasks run in parallel.
* **Voting:** Running the same task multiple times to get diverse outputs.

The parallelization workflow

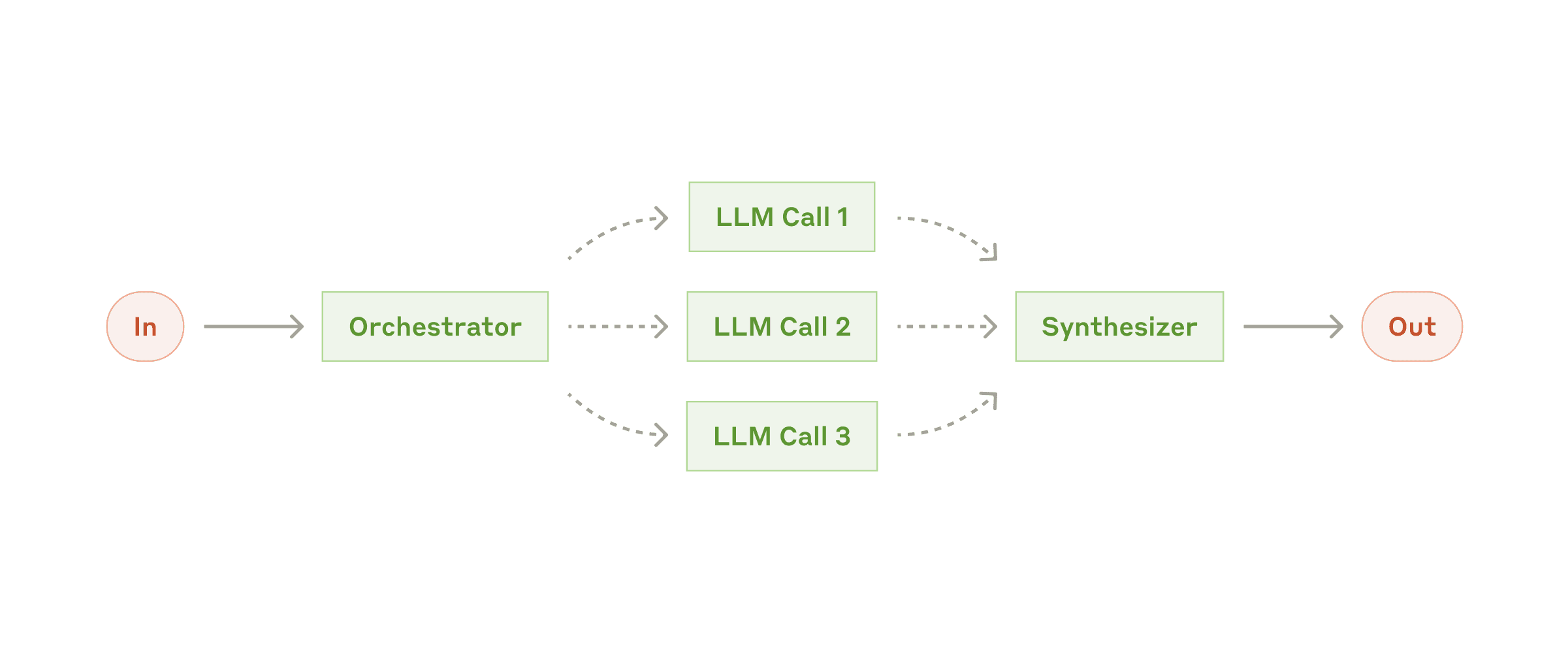
**When to use this workflow:** Parallelization is effective when the divided subtasks can be parallelized for speed, or when multiple perspectives or attempts are needed for higher confidence results. For complex tasks with multiple considerations, LLMs generally perform better when each consideration is handled by a separate LLM call, allowing focused attention on each specific aspect.

**Examples where parallelization is useful:**

* **Sectioning**:
  + Implementing guardrails where one model instance processes user queries while another screens them for inappropriate content or requests. This tends to perform better than having the same LLM call handle both guardrails and the core response.
  + Automating evals for evaluating LLM performance, where each LLM call evaluates a different aspect of the model’s performance on a given prompt.
* **Voting**:
  + Reviewing a piece of code for vulnerabilities, where several different prompts review and flag the code if they find a problem.
  + Evaluating whether a given piece of content is inappropriate, with multiple prompts evaluating different aspects or requiring different vote thresholds to balance false positives and negatives.

### Workflow: Orchestrator-workers

In the orchestrator-workers workflow, a central LLM dynamically breaks down tasks, delegates them to worker LLMs, and synthesizes their results.

The orchestrator-workers workflow

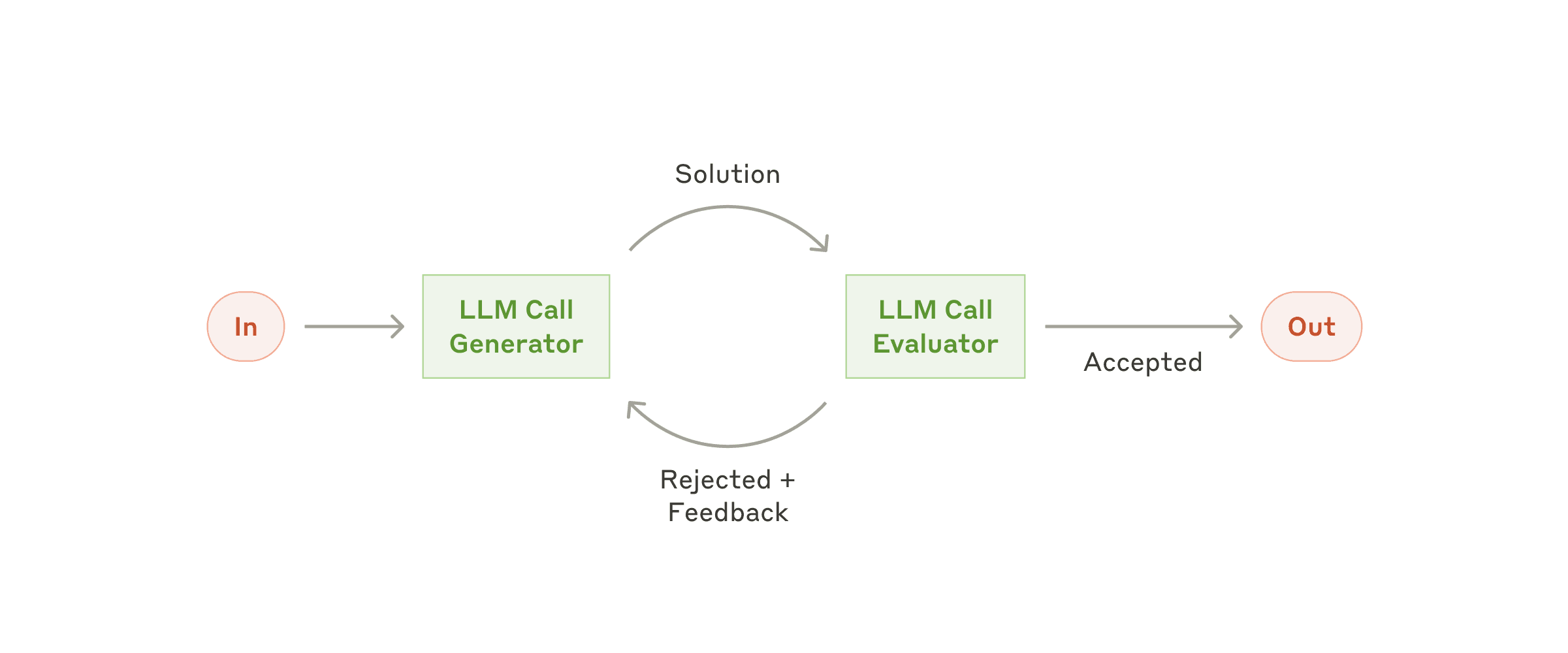
**When to use this workflow:** This workflow is well-suited for complex tasks where you can’t predict the subtasks needed (in coding, for example, the number of files that need to be changed and the nature of the change in each file likely depend on the task). Whereas it’s topographically similar, the key difference from parallelization is its flexibility—subtasks aren't pre-defined, but determined by the orchestrator based on the specific input.

**Example where orchestrator-workers is useful:**

* Coding products that make complex changes to multiple files each time.
* Search tasks that involve gathering and analyzing information from multiple sources for possible relevant information.

### Workflow: Evaluator-optimizer

In the evaluator-optimizer workflow, one LLM call generates a response while another provides evaluation and feedback in a loop.

The evaluator-optimizer workflow

**When to use this workflow:** This workflow is particularly effective when we have clear evaluation criteria, and when iterative refinement provides measurable value. The two signs of good fit are, first, that LLM responses can be demonstrably improved when a human articulates their feedback; and second, that the LLM can provide such feedback. This is analogous to the iterative writing process a human writer might go through when producing a polished document.

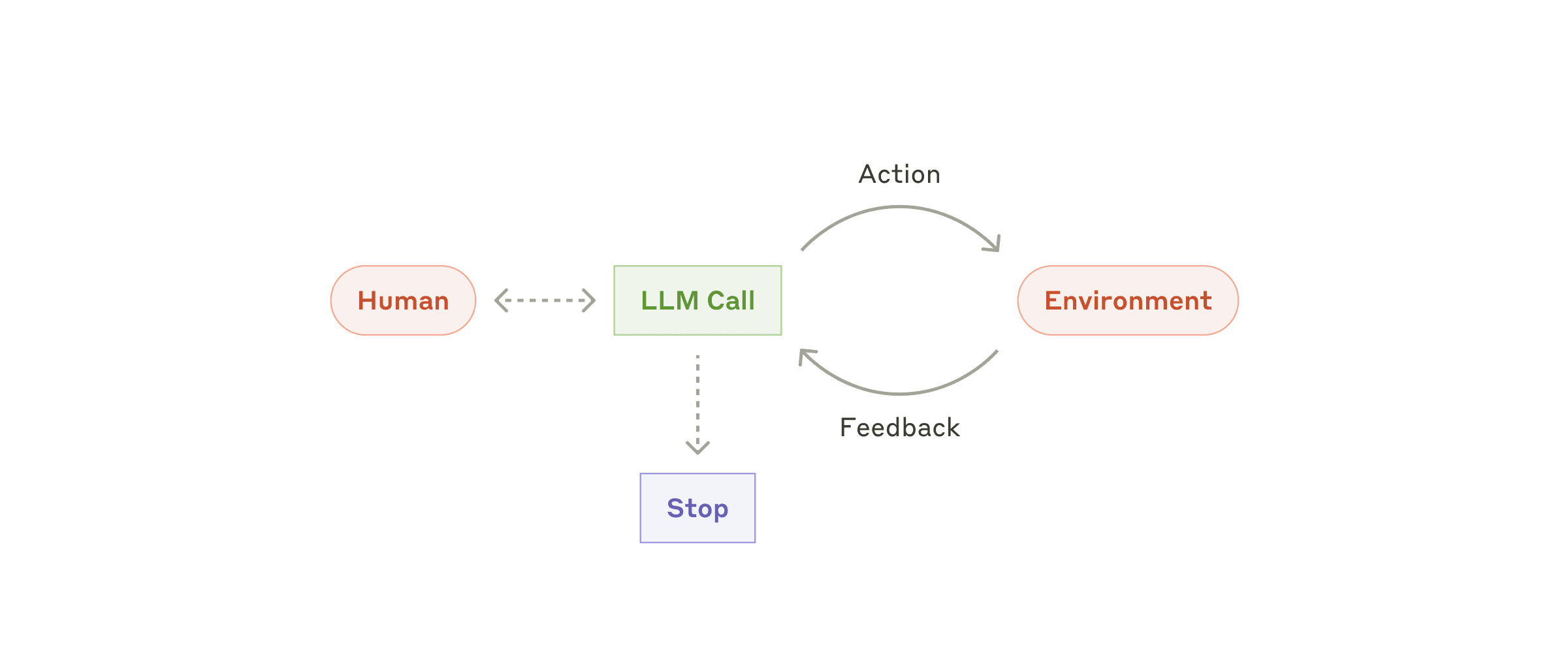
**Examples where evaluator-optimizer is useful:**

* Literary translation where there are nuances that the translator LLM might not capture initially, but where an evaluator LLM can provide useful critiques.
* Complex search tasks that require multiple rounds of searching and analysis to gather comprehensive information, where the evaluator decides whether further searches are warranted.

### Agents

Agents are emerging in production as LLMs mature in key capabilities—understanding complex inputs, engaging in reasoning and planning, using tools reliably, and recovering from errors. Agents begin their work with either a command from, or interactive discussion with, the human user. Once the task is clear, agents plan and operate independently, potentially returning to the human for further information or judgement. During execution, it's crucial for the agents to gain “ground truth” from the environment at each step (such as tool call results or code execution) to assess its progress. Agents can then pause for human feedback at checkpoints or when encountering blockers. The task often terminates upon completion, but it’s also common to include stopping conditions (such as a maximum number of iterations) to maintain control.

Agents can handle sophisticated tasks, but their implementation is often straightforward. They are typically just LLMs using tools based on environmental feedback in a loop. It is therefore crucial to design toolsets and their documentation clearly and thoughtfully. We expand on best practices for tool development in Appendix 2 ("Prompt Engineering your Tools").

Autonomous agent

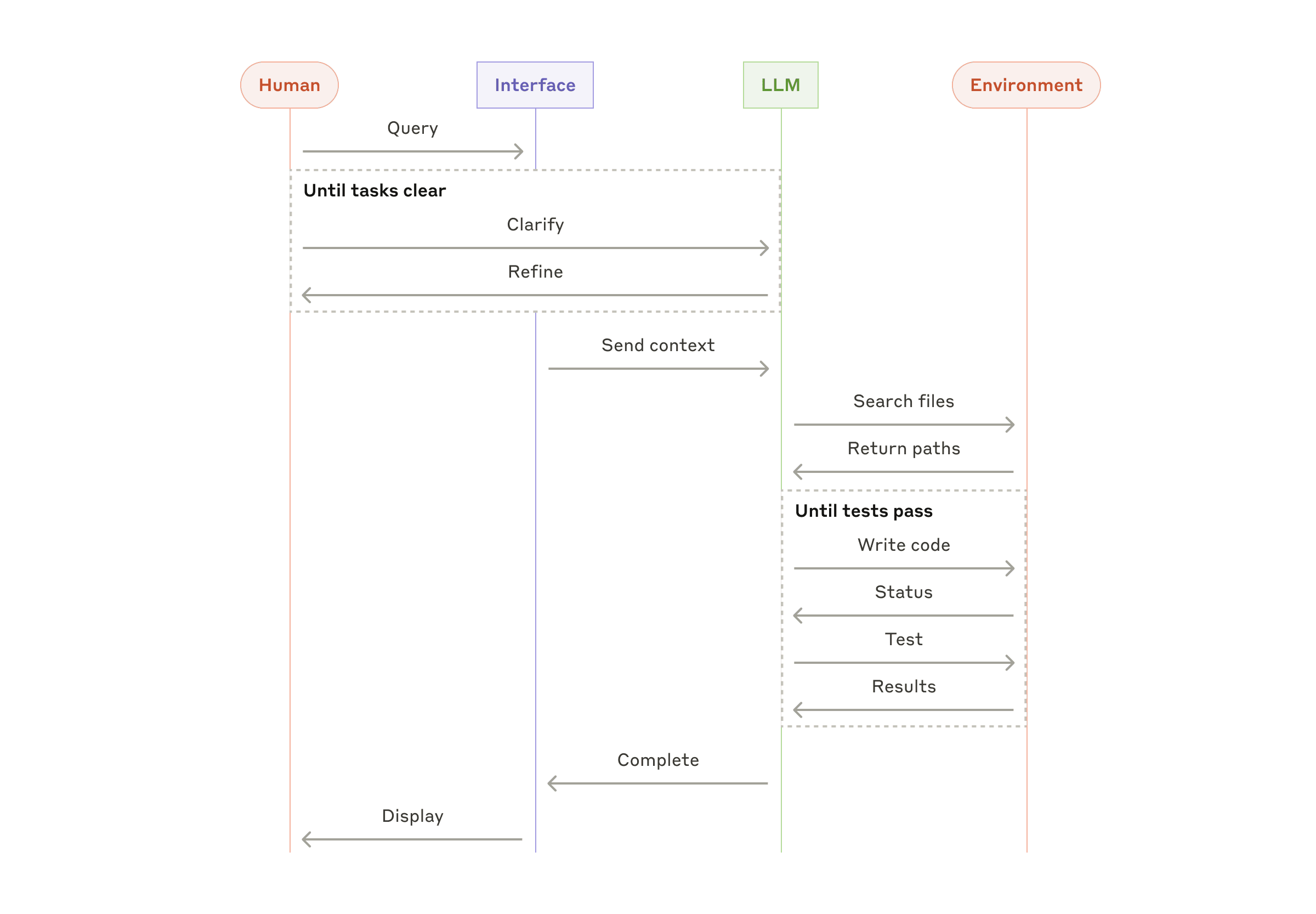
**When to use agents:** Agents can be used for open-ended problems where it’s difficult or impossible to predict the required number of steps, and where you can’t hardcode a fixed path. The LLM will potentially operate for many turns, and you must have some level of trust in its decision-making. Agents' autonomy makes them ideal for scaling tasks in trusted environments.

The autonomous nature of agents means higher costs, and the potential for compounding errors. We recommend extensive testing in sandboxed environments, along with the appropriate guardrails.

**Examples where agents are useful:**

The following examples are from our own implementations:

* A coding Agent to resolve [SWE-bench tasks](https://www.anthropic.com/research/swe-bench-sonnet), which involve edits to many files based on a task description;
* Our [“computer use” reference implementation](https://github.com/anthropics/anthropic-quickstarts/tree/main/computer-use-demo), where Claude uses a computer to accomplish tasks.

High-level flow of a coding agent

## Combining and customizing these patterns

These building blocks aren't prescriptive. They're common patterns that developers can shape and combine to fit different use cases. The key to success, as with any LLM features, is measuring performance and iterating on implementations. To repeat: you should consider adding complexity only when it demonstrably improves outcomes.

## Summary

Success in the LLM space isn't about building the most sophisticated system. It's about building the right system for your needs. Start with simple prompts, optimize them with comprehensive evaluation, and add multi-step agentic systems only when simpler solutions fall short.

When implementing agents, we try to follow three core principles:

1. Maintain **simplicity** in your agent's design.
2. Prioritize **transparency** by explicitly showing the agent’s planning steps.
3. Carefully craft your agent-computer interface (ACI) through thorough tool **documentation and testing**.

Frameworks can help you get started quickly, but don't hesitate to reduce abstraction layers and build with basic components as you move to production. By following these principles, you can create agents that are not only powerful but also reliable, maintainable, and trusted by their users.

### Acknowledgements

Written by Erik Schluntz and Barry Zhang. This work draws upon our experiences building agents at Anthropic and the valuable insights shared by our customers, for which we're deeply grateful.

## Appendix 1: Agents in practice

Our work with customers has revealed two particularly promising applications for AI agents that demonstrate the practical value of the patterns discussed above. Both applications illustrate how agents add the most value for tasks that require both conversation and action, have clear success criteria, enable feedback loops, and integrate meaningful human oversight.

### A. Customer support

Customer support combines familiar chatbot interfaces with enhanced capabilities through tool integration. This is a natural fit for more open-ended agents because:

* Support interactions naturally follow a conversation flow while requiring access to external information and actions;
* Tools can be integrated to pull customer data, order history, and knowledge base articles;
* Actions such as issuing refunds or updating tickets can be handled programmatically; and
* Success can be clearly measured through user-defined resolutions.

Several companies have demonstrated the viability of this approach through usage-based pricing models that charge only for successful resolutions, showing confidence in their agents' effectiveness.

### B. Coding agents

The software development space has shown remarkable potential for LLM features, with capabilities evolving from code completion to autonomous problem-solving. Agents are particularly effective because:

* Code solutions are verifiable through automated tests;
* Agents can iterate on solutions using test results as feedback;
* The problem space is well-defined and structured; and
* Output quality can be measured objectively.

In our own implementation, agents can now solve real GitHub issues in the [SWE-bench Verified](https://www.anthropic.com/research/swe-bench-sonnet) benchmark based on the pull request description alone. However, whereas automated testing helps verify functionality, human review remains crucial for ensuring solutions align with broader system requirements.

## Appendix 2: Prompt engineering your tools

No matter which agentic system you're building, tools will likely be an important part of your agent. [Tools](https://www.anthropic.com/news/tool-use-ga) enable Claude to interact with external services and APIs by specifying their exact structure and definition in our API. When Claude responds, it will include a [tool use block](https://docs.anthropic.com/en/docs/build-with-claude/tool-use" \l "example-api-response-with-a-tool-use-content-block) in the API response if it plans to invoke a tool. Tool definitions and specifications should be given just as much prompt engineering attention as your overall prompts. In this brief appendix, we describe how to prompt engineer your tools.

There are often several ways to specify the same action. For instance, you can specify a file edit by writing a diff, or by rewriting the entire file. For structured output, you can return code inside markdown or inside JSON. In software engineering, differences like these are cosmetic and can be converted losslessly from one to the other. However, some formats are much more difficult for an LLM to write than others. Writing a diff requires knowing how many lines are changing in the chunk header before the new code is written. Writing code inside JSON (compared to markdown) requires extra escaping of newlines and quotes.

Our suggestions for deciding on tool formats are the following:

* Give the model enough tokens to "think" before it writes itself into a corner.
* Keep the format close to what the model has seen naturally occurring in text on the internet.
* Make sure there's no formatting "overhead" such as having to keep an accurate count of thousands of lines of code, or string-escaping any code it writes.

One rule of thumb is to think about how much effort goes into human-computer interfaces (HCI), and plan to invest just as much effort in creating good agent-computer interfaces (ACI). Here are some thoughts on how to do so:

* Put yourself in the model's shoes. Is it obvious how to use this tool, based on the description and parameters, or would you need to think carefully about it? If so, then it’s probably also true for the model. A good tool definition often includes example usage, edge cases, input format requirements, and clear boundaries from other tools.
* How can you change parameter names or descriptions to make things more obvious? Think of this as writing a great docstring for a junior developer on your team. This is especially important when using many similar tools.
* Test how the model uses your tools: Run many example inputs in our [workbench](https://console.anthropic.com/workbench) to see what mistakes the model makes, and iterate.
* [Poka-yoke](https://en.wikipedia.org/wiki/Poka-yoke) your tools. Change the arguments so that it is harder to make mistakes.

While building our agent for [SWE-bench](https://www.anthropic.com/research/swe-bench-sonnet), we actually spent more time optimizing our tools than the overall prompt. For example, we found that the model would make mistakes with tools using relative filepaths after the agent had moved out of the root directory. To fix this, we changed the tool to always require absolute filepaths—and we found that the model used this method flawlessly.

# Building effective agents

2024年12月20日

Over the past year, we've worked with dozens of teams building large language model (LLM) agents across industries. Consistently, the most successful implementations weren't using complex frameworks or specialized libraries. Instead, they were building with simple, composable patterns.

In this post, we share what we’ve learned from working with our customers and building agents ourselves, and give practical advice for developers on building effective agents.

## What are agents?

"Agent" can be defined in several ways. Some customers define agents as fully autonomous systems that operate independently over extended periods, using various tools to accomplish complex tasks. Others use the term to describe more prescriptive implementations that follow predefined workflows. At Anthropic, we categorize all these variations as **agentic systems**, but draw an important architectural distinction between **workflows**and**agents**:

* **Workflows** are systems where LLMs and tools are orchestrated through predefined code paths.
* **Agents**, on the other hand, are systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

Below, we will explore both types of agentic systems in detail. In Appendix 1 (“Agents in Practice”), we describe two domains where customers have found particular value in using these kinds of systems.

## When (and when not) to use agents

When building applications with LLMs, we recommend finding the simplest solution possible, and only increasing complexity when needed. This might mean not building agentic systems at all. Agentic systems often trade latency and cost for better task performance, and you should consider when this tradeoff makes sense.

When more complexity is warranted, workflows offer predictability and consistency for well-defined tasks, whereas agents are the better option when flexibility and model-driven decision-making are needed at scale. For many applications, however, optimizing single LLM calls with retrieval and in-context examples is usually enough.

## When and how to use frameworks

There are many frameworks that make agentic systems easier to implement, including:

* [LangGraph](https://langchain-ai.github.io/langgraph/) from LangChain;
* Amazon Bedrock's [AI Agent framework](https://aws.amazon.com/bedrock/agents/);
* [Rivet](https://rivet.ironcladapp.com/), a drag and drop GUI LLM workflow builder; and
* [Vellum](https://www.vellum.ai/), another GUI tool for building and testing complex workflows.

These frameworks make it easy to get started by simplifying standard low-level tasks like calling LLMs, defining and parsing tools, and chaining calls together. However, they often create extra layers of abstraction that can obscure the underlying prompts ​​and responses, making them harder to debug. They can also make it tempting to add complexity when a simpler setup would suffice.

We suggest that developers start by using LLM APIs directly: many patterns can be implemented in a few lines of code. If you do use a framework, ensure you understand the underlying code. Incorrect assumptions about what's under the hood are a common source of customer error.

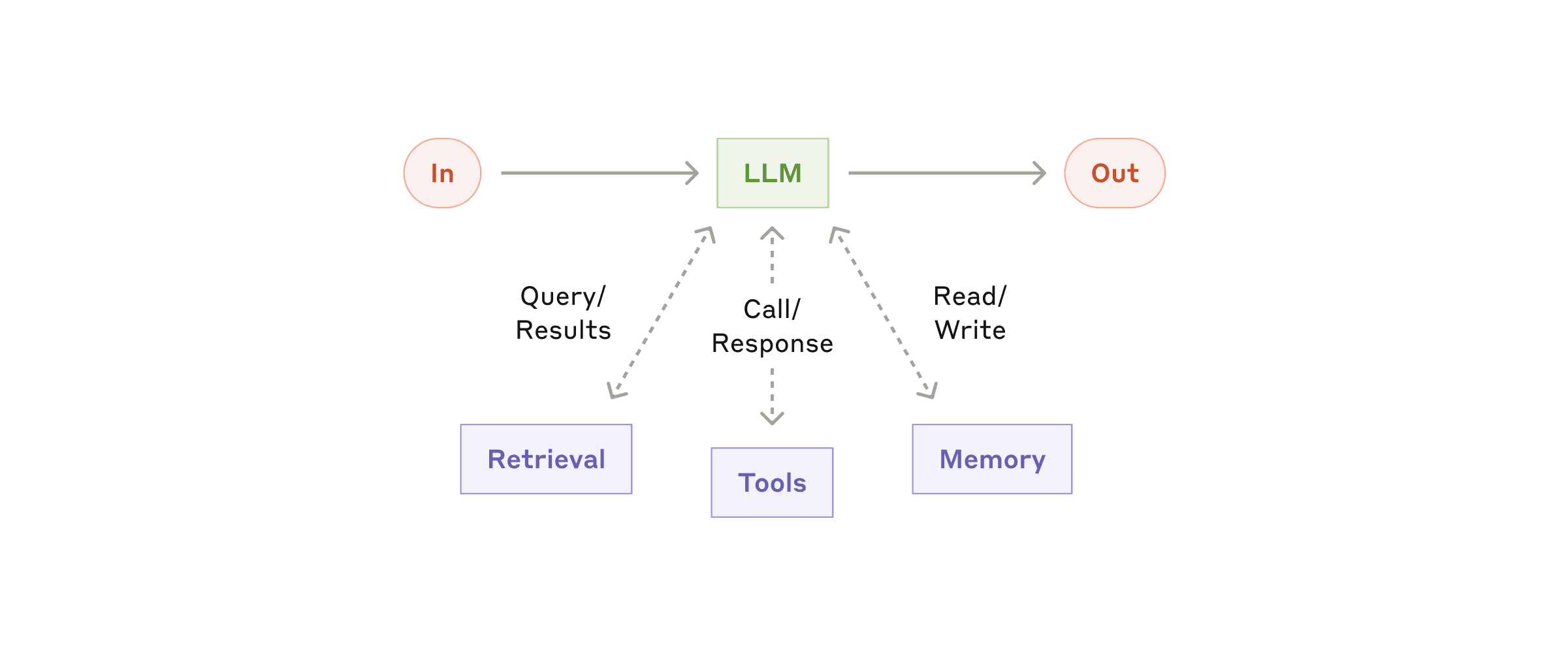
See our [cookbook](https://github.com/anthropics/anthropic-cookbook/tree/main/patterns/agents) for some sample implementations.

## Building blocks, workflows, and agents

In this section, we’ll explore the common patterns for agentic systems we’ve seen in production. We'll start with our foundational building block—the augmented LLM—and progressively increase complexity, from simple compositional workflows to autonomous agents.

### Building block: The augmented LLM

The basic building block of agentic systems is an LLM enhanced with augmentations such as retrieval, tools, and memory. Our current models can actively use these capabilities—generating their own search queries, selecting appropriate tools, and determining what information to retain.

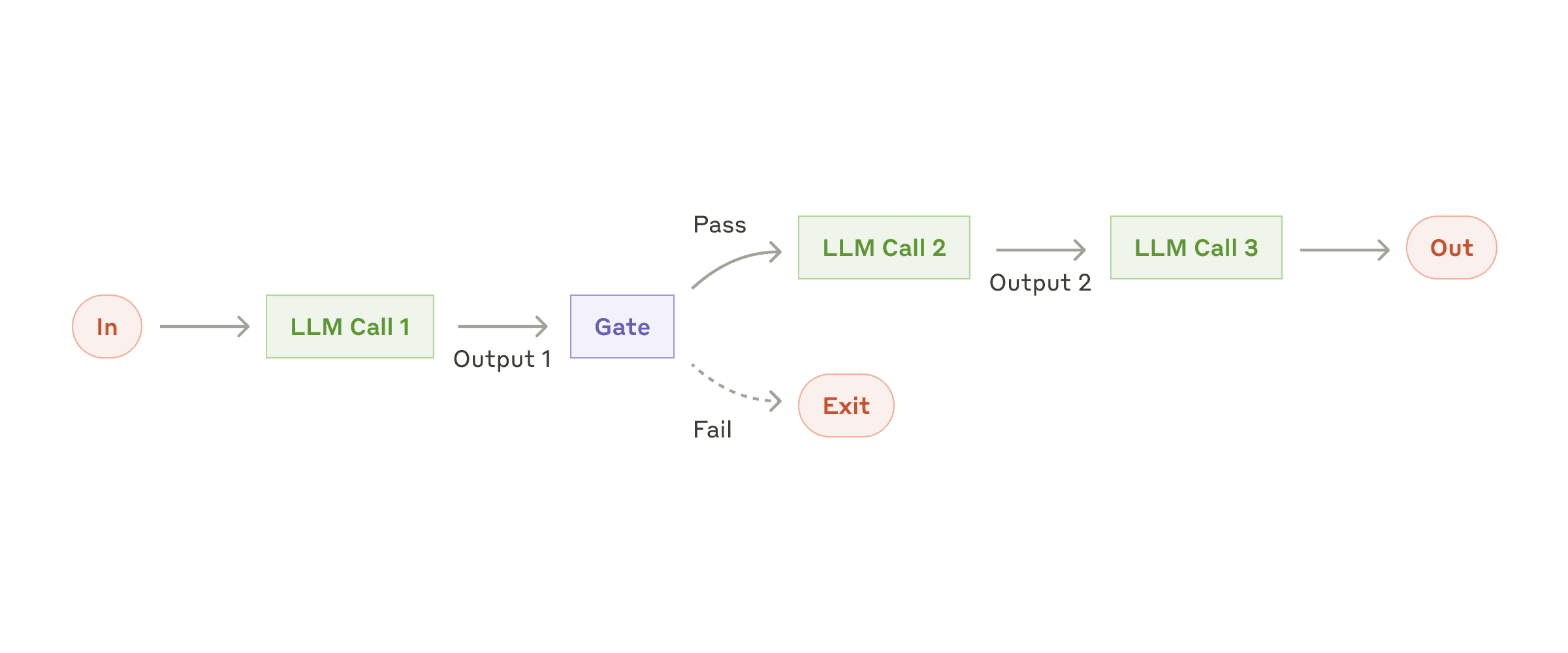
The augmented LLM

We recommend focusing on two key aspects of the implementation: tailoring these capabilities to your specific use case and ensuring they provide an easy, well-documented interface for your LLM. While there are many ways to implement these augmentations, one approach is through our recently released [Model Context Protocol](https://www.anthropic.com/news/model-context-protocol), which allows developers to integrate with a growing ecosystem of third-party tools with a simple [client implementation](https://modelcontextprotocol.io/tutorials/building-a-client" \l "building-mcp-clients).

For the remainder of this post, we'll assume each LLM call has access to these augmented capabilities.

### Workflow: Prompt chaining

Prompt chaining decomposes a task into a sequence of steps, where each LLM call processes the output of the previous one. You can add programmatic checks (see "gate” in the diagram below) on any intermediate steps to ensure that the process is still on track.

The prompt chaining workflow

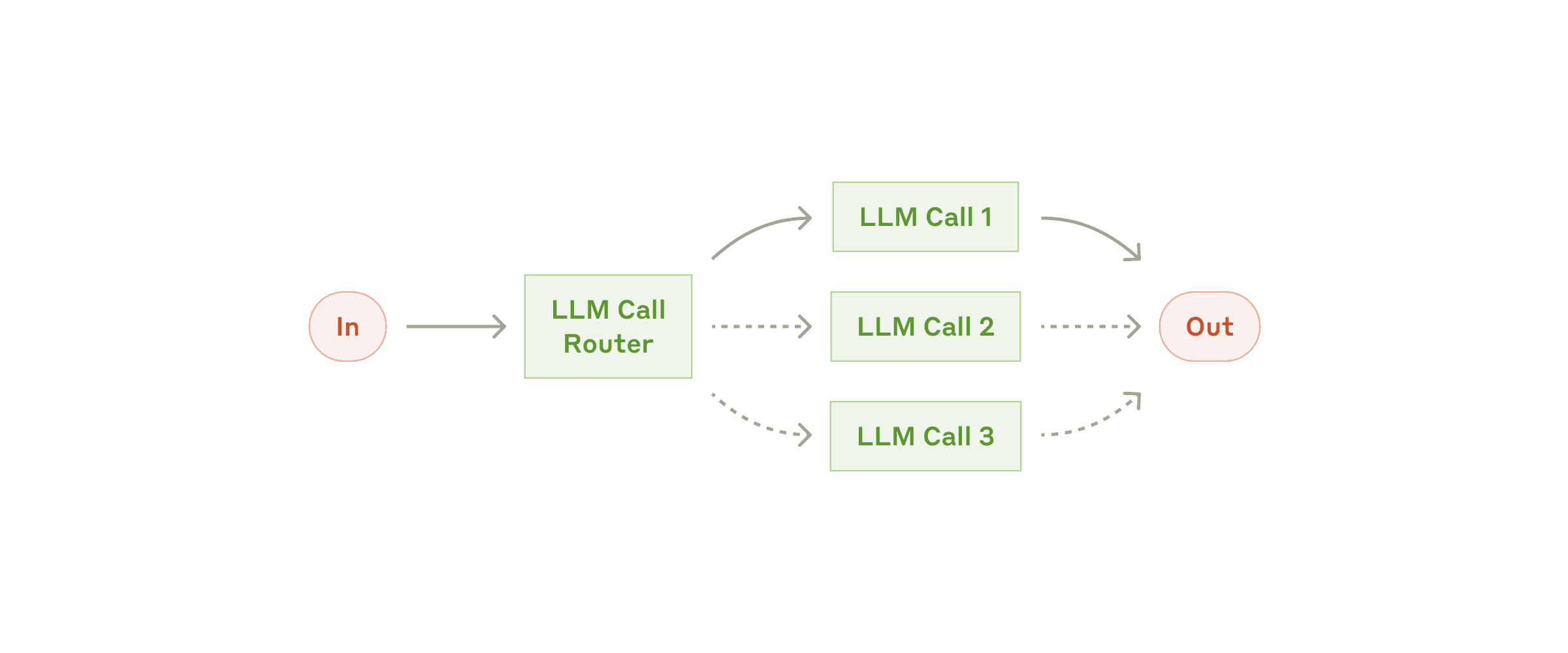
**When to use this workflow:** This workflow is ideal for situations where the task can be easily and cleanly decomposed into fixed subtasks. The main goal is to trade off latency for higher accuracy, by making each LLM call an easier task.

**Examples where prompt chaining is useful:**

* Generating Marketing copy, then translating it into a different language.
* Writing an outline of a document, checking that the outline meets certain criteria, then writing the document based on the outline.

### Workflow: Routing

Routing classifies an input and directs it to a specialized followup task. This workflow allows for separation of concerns, and building more specialized prompts. Without this workflow, optimizing for one kind of input can hurt performance on other inputs.

The routing workflow

**When to use this workflow:** Routing works well for complex tasks where there are distinct categories that are better handled separately, and where classification can be handled accurately, either by an LLM or a more traditional classification model/algorithm.

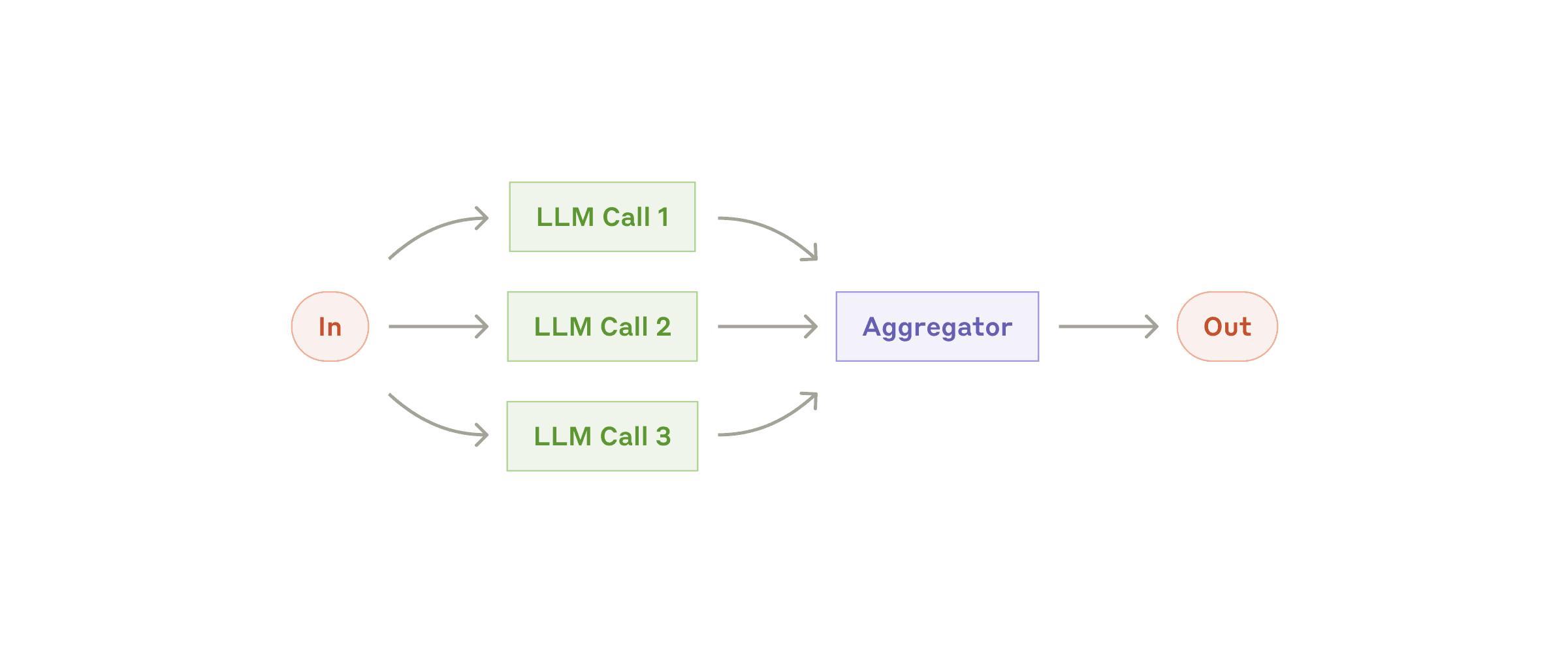
**Examples where routing is useful:**

* Directing different types of customer service queries (general questions, refund requests, technical support) into different downstream processes, prompts, and tools.
* Routing easy/common questions to smaller models like Claude 3.5 Haiku and hard/unusual questions to more capable models like Claude 3.5 Sonnet to optimize cost and speed.

### Workflow: Parallelization

LLMs can sometimes work simultaneously on a task and have their outputs aggregated programmatically. This workflow, parallelization, manifests in two key variations:

* **Sectioning**: Breaking a task into independent subtasks run in parallel.
* **Voting:** Running the same task multiple times to get diverse outputs.

The parallelization workflow

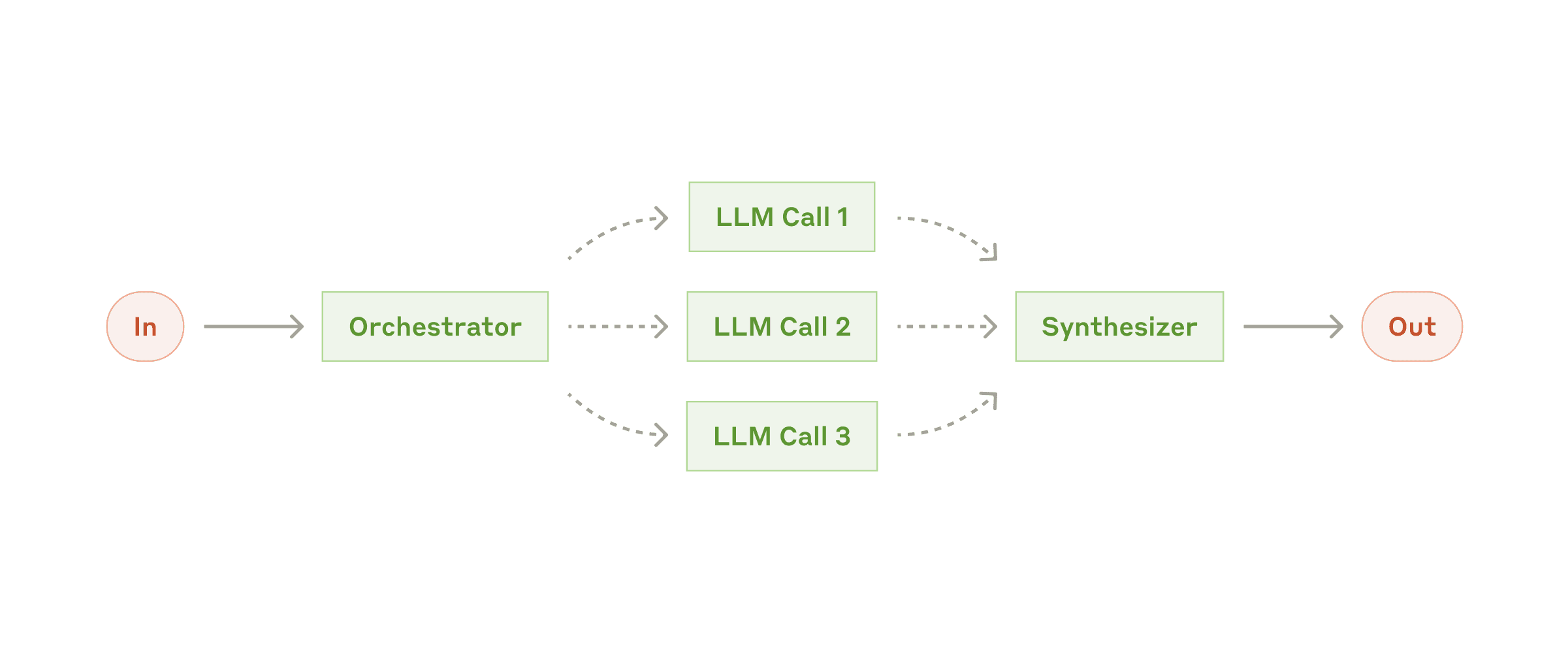
**When to use this workflow:** Parallelization is effective when the divided subtasks can be parallelized for speed, or when multiple perspectives or attempts are needed for higher confidence results. For complex tasks with multiple considerations, LLMs generally perform better when each consideration is handled by a separate LLM call, allowing focused attention on each specific aspect.

**Examples where parallelization is useful:**

* **Sectioning**:
  + Implementing guardrails where one model instance processes user queries while another screens them for inappropriate content or requests. This tends to perform better than having the same LLM call handle both guardrails and the core response.
  + Automating evals for evaluating LLM performance, where each LLM call evaluates a different aspect of the model’s performance on a given prompt.
* **Voting**:
  + Reviewing a piece of code for vulnerabilities, where several different prompts review and flag the code if they find a problem.
  + Evaluating whether a given piece of content is inappropriate, with multiple prompts evaluating different aspects or requiring different vote thresholds to balance false positives and negatives.

### Workflow: Orchestrator-workers

In the orchestrator-workers workflow, a central LLM dynamically breaks down tasks, delegates them to worker LLMs, and synthesizes their results.

The orchestrator-workers workflow

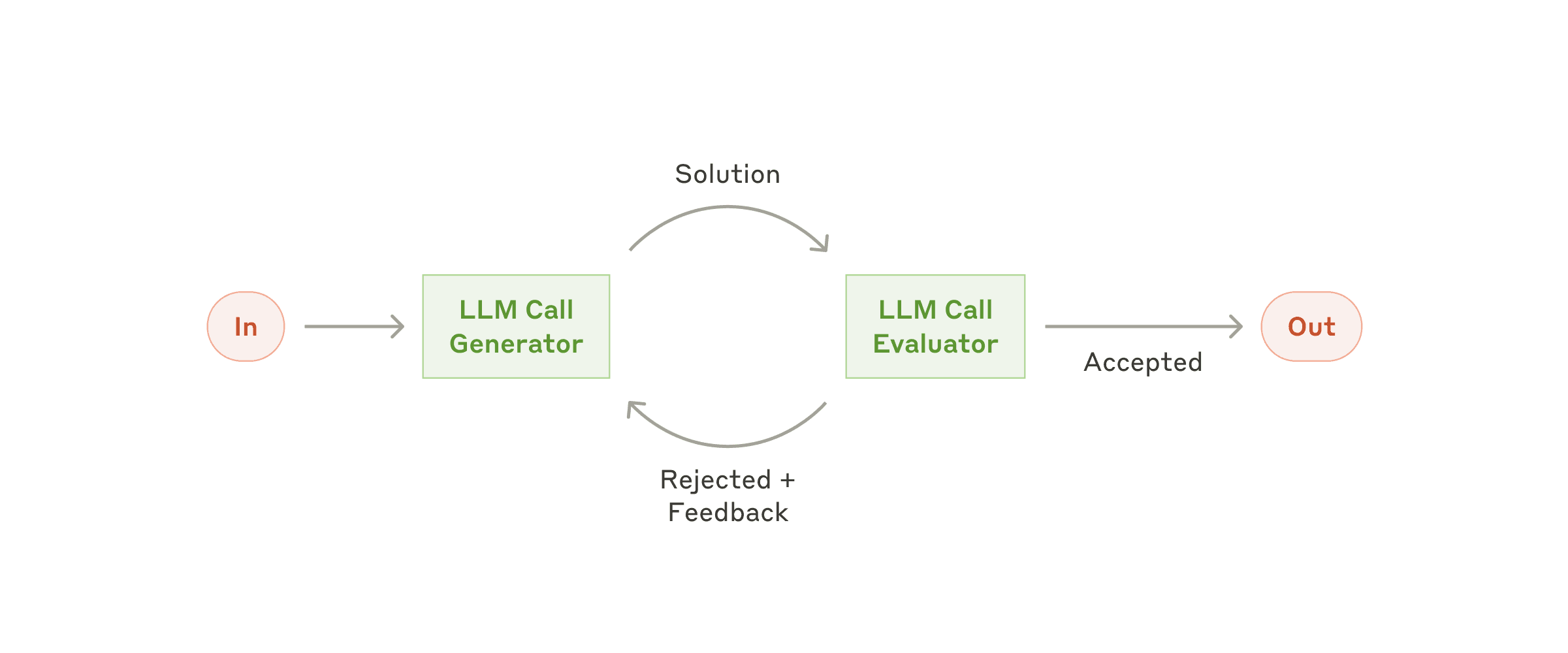
**When to use this workflow:** This workflow is well-suited for complex tasks where you can’t predict the subtasks needed (in coding, for example, the number of files that need to be changed and the nature of the change in each file likely depend on the task). Whereas it’s topographically similar, the key difference from parallelization is its flexibility—subtasks aren't pre-defined, but determined by the orchestrator based on the specific input.

**Example where orchestrator-workers is useful:**

* Coding products that make complex changes to multiple files each time.
* Search tasks that involve gathering and analyzing information from multiple sources for possible relevant information.

### Workflow: Evaluator-optimizer

In the evaluator-optimizer workflow, one LLM call generates a response while another provides evaluation and feedback in a loop.

The evaluator-optimizer workflow

**When to use this workflow:** This workflow is particularly effective when we have clear evaluation criteria, and when iterative refinement provides measurable value. The two signs of good fit are, first, that LLM responses can be demonstrably improved when a human articulates their feedback; and second, that the LLM can provide such feedback. This is analogous to the iterative writing process a human writer might go through when producing a polished document.

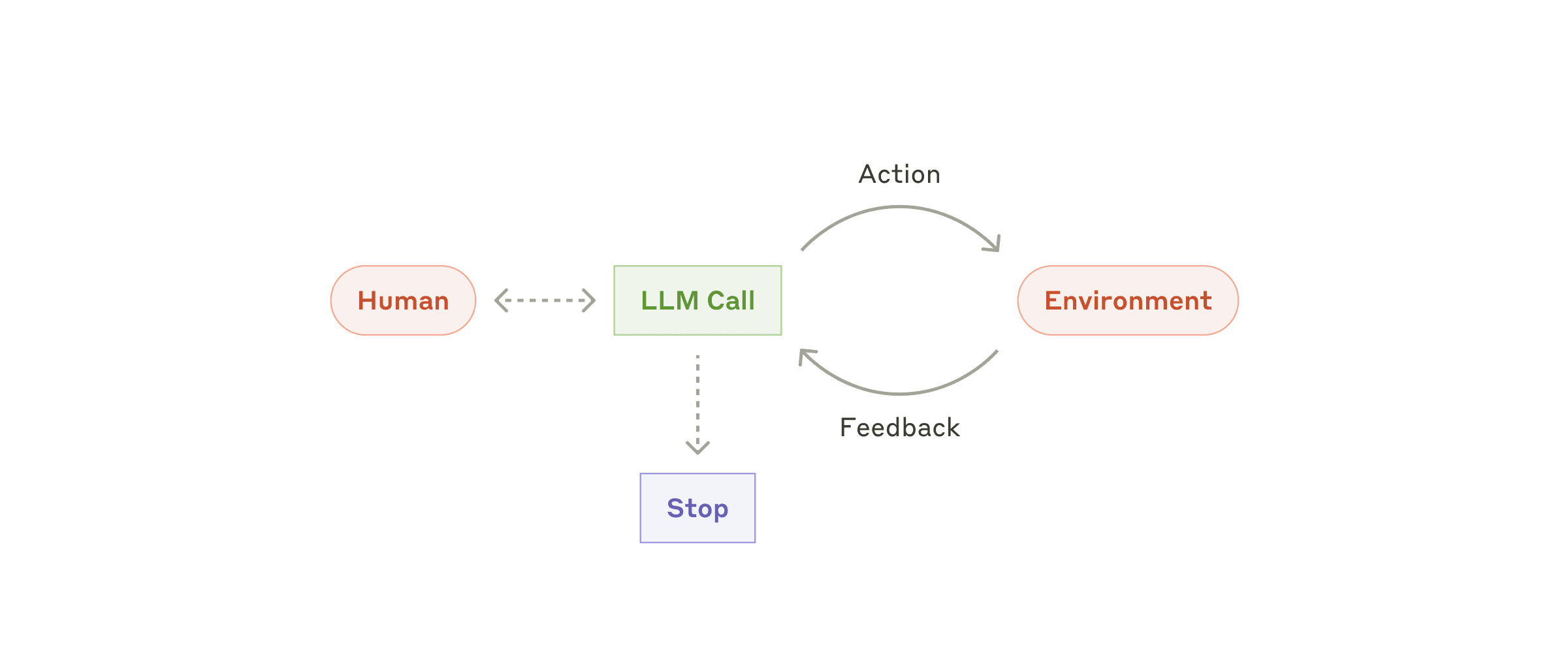
**Examples where evaluator-optimizer is useful:**

* Literary translation where there are nuances that the translator LLM might not capture initially, but where an evaluator LLM can provide useful critiques.
* Complex search tasks that require multiple rounds of searching and analysis to gather comprehensive information, where the evaluator decides whether further searches are warranted.

### Agents

Agents are emerging in production as LLMs mature in key capabilities—understanding complex inputs, engaging in reasoning and planning, using tools reliably, and recovering from errors. Agents begin their work with either a command from, or interactive discussion with, the human user. Once the task is clear, agents plan and operate independently, potentially returning to the human for further information or judgement. During execution, it's crucial for the agents to gain “ground truth” from the environment at each step (such as tool call results or code execution) to assess its progress. Agents can then pause for human feedback at checkpoints or when encountering blockers. The task often terminates upon completion, but it’s also common to include stopping conditions (such as a maximum number of iterations) to maintain control.

Agents can handle sophisticated tasks, but their implementation is often straightforward. They are typically just LLMs using tools based on environmental feedback in a loop. It is therefore crucial to design toolsets and their documentation clearly and thoughtfully. We expand on best practices for tool development in Appendix 2 ("Prompt Engineering your Tools").

Autonomous agent

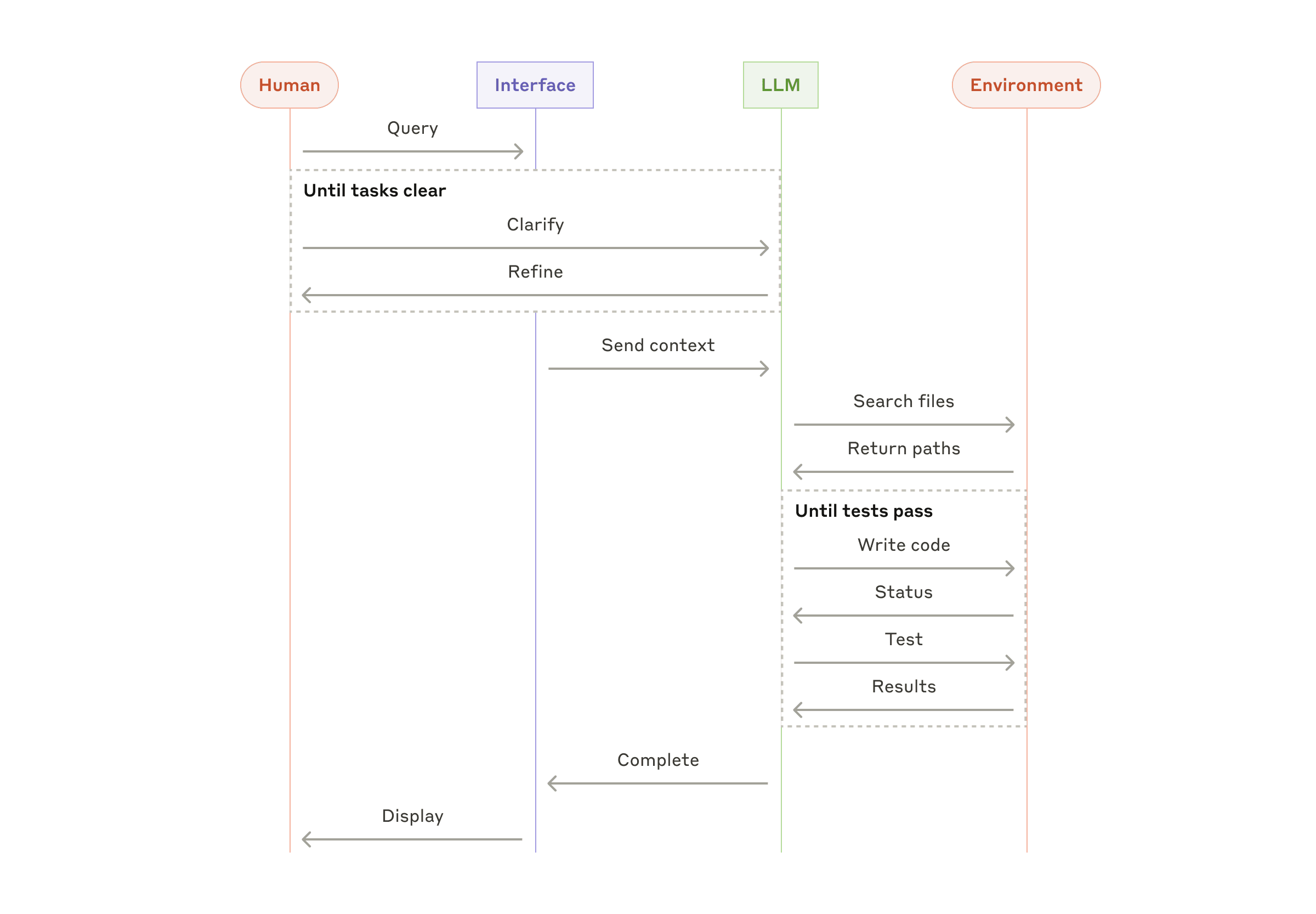
**When to use agents:** Agents can be used for open-ended problems where it’s difficult or impossible to predict the required number of steps, and where you can’t hardcode a fixed path. The LLM will potentially operate for many turns, and you must have some level of trust in its decision-making. Agents' autonomy makes them ideal for scaling tasks in trusted environments.

The autonomous nature of agents means higher costs, and the potential for compounding errors. We recommend extensive testing in sandboxed environments, along with the appropriate guardrails.

**Examples where agents are useful:**

The following examples are from our own implementations:

* A coding Agent to resolve [SWE-bench tasks](https://www.anthropic.com/research/swe-bench-sonnet), which involve edits to many files based on a task description;
* Our [“computer use” reference implementation](https://github.com/anthropics/anthropic-quickstarts/tree/main/computer-use-demo), where Claude uses a computer to accomplish tasks.

High-level flow of a coding agent

## Combining and customizing these patterns

These building blocks aren't prescriptive. They're common patterns that developers can shape and combine to fit different use cases. The key to success, as with any LLM features, is measuring performance and iterating on implementations. To repeat: you should consider adding complexity only when it demonstrably improves outcomes.

## Summary

Success in the LLM space isn't about building the most sophisticated system. It's about building the right system for your needs. Start with simple prompts, optimize them with comprehensive evaluation, and add multi-step agentic systems only when simpler solutions fall short.

When implementing agents, we try to follow three core principles:

1. Maintain **simplicity** in your agent's design.
2. Prioritize **transparency** by explicitly showing the agent’s planning steps.
3. Carefully craft your agent-computer interface (ACI) through thorough tool **documentation and testing**.

Frameworks can help you get started quickly, but don't hesitate to reduce abstraction layers and build with basic components as you move to production. By following these principles, you can create agents that are not only powerful but also reliable, maintainable, and trusted by their users.

### Acknowledgements

Written by Erik Schluntz and Barry Zhang. This work draws upon our experiences building agents at Anthropic and the valuable insights shared by our customers, for which we're deeply grateful.

## Appendix 1: Agents in practice

Our work with customers has revealed two particularly promising applications for AI agents that demonstrate the practical value of the patterns discussed above. Both applications illustrate how agents add the most value for tasks that require both conversation and action, have clear success criteria, enable feedback loops, and integrate meaningful human oversight.

### A. Customer support

Customer support combines familiar chatbot interfaces with enhanced capabilities through tool integration. This is a natural fit for more open-ended agents because:

* Support interactions naturally follow a conversation flow while requiring access to external information and actions;
* Tools can be integrated to pull customer data, order history, and knowledge base articles;
* Actions such as issuing refunds or updating tickets can be handled programmatically; and
* Success can be clearly measured through user-defined resolutions.

Several companies have demonstrated the viability of this approach through usage-based pricing models that charge only for successful resolutions, showing confidence in their agents' effectiveness.

### B. Coding agents

The software development space has shown remarkable potential for LLM features, with capabilities evolving from code completion to autonomous problem-solving. Agents are particularly effective because:

* Code solutions are verifiable through automated tests;
* Agents can iterate on solutions using test results as feedback;
* The problem space is well-defined and structured; and
* Output quality can be measured objectively.

In our own implementation, agents can now solve real GitHub issues in the [SWE-bench Verified](https://www.anthropic.com/research/swe-bench-sonnet) benchmark based on the pull request description alone. However, whereas automated testing helps verify functionality, human review remains crucial for ensuring solutions align with broader system requirements.

## Appendix 2: Prompt engineering your tools

No matter which agentic system you're building, tools will likely be an important part of your agent. [Tools](https://www.anthropic.com/news/tool-use-ga) enable Claude to interact with external services and APIs by specifying their exact structure and definition in our API. When Claude responds, it will include a [tool use block](https://docs.anthropic.com/en/docs/build-with-claude/tool-use" \l "example-api-response-with-a-tool-use-content-block) in the API response if it plans to invoke a tool. Tool definitions and specifications should be given just as much prompt engineering attention as your overall prompts. In this brief appendix, we describe how to prompt engineer your tools.

There are often several ways to specify the same action. For instance, you can specify a file edit by writing a diff, or by rewriting the entire file. For structured output, you can return code inside markdown or inside JSON. In software engineering, differences like these are cosmetic and can be converted losslessly from one to the other. However, some formats are much more difficult for an LLM to write than others. Writing a diff requires knowing how many lines are changing in the chunk header before the new code is written. Writing code inside JSON (compared to markdown) requires extra escaping of newlines and quotes.

Our suggestions for deciding on tool formats are the following:

* Give the model enough tokens to "think" before it writes itself into a corner.
* Keep the format close to what the model has seen naturally occurring in text on the internet.
* Make sure there's no formatting "overhead" such as having to keep an accurate count of thousands of lines of code, or string-escaping any code it writes.

One rule of thumb is to think about how much effort goes into human-computer interfaces (HCI), and plan to invest just as much effort in creating good agent-computer interfaces (ACI). Here are some thoughts on how to do so:

* Put yourself in the model's shoes. Is it obvious how to use this tool, based on the description and parameters, or would you need to think carefully about it? If so, then it’s probably also true for the model. A good tool definition often includes example usage, edge cases, input format requirements, and clear boundaries from other tools.
* How can you change parameter names or descriptions to make things more obvious? Think of this as writing a great docstring for a junior developer on your team. This is especially important when using many similar tools.
* Test how the model uses your tools: Run many example inputs in our [workbench](https://console.anthropic.com/workbench) to see what mistakes the model makes, and iterate.
* [Poka-yoke](https://en.wikipedia.org/wiki/Poka-yoke) your tools. Change the arguments so that it is harder to make mistakes.

While building our agent for [SWE-bench](https://www.anthropic.com/research/swe-bench-sonnet), we actually spent more time optimizing our tools than the overall prompt. For example, we found that the model would make mistakes with tools using relative filepaths after the agent had moved out of the root directory. To fix this, we changed the tool to always require absolute filepaths—and we found that the model used this method flawlessly.

这几天刷推很明显的感觉到英文技术社区对中国AI产业的进步速度处于一种半震动半懵逼的状态，应激来源主要是两个，一个是宇树（Unitree）的轮足式机器狗B2-W，另一个是开源MoE模型DeepSeek-V3。

宇树在早年基本上属于是波士顿动力的跟班，产品形态完全照猫画虎，商业上瞄准的也是低配平替生态位，没有太大的吸引力，但从B系列型号开始，宇树的机器狗就在灵活性上可以和波士顿动力平起平坐了。

B2-W的意外在于切换了技术线，用运动性更高但平衡性同时也更难的动轮方案取代了B2还在沿用四足方案，然后在一年时间里完成了能在户外环境里跋山涉水的训练，很多美国人在视频底下说这一定是CGI的画面，不知道是真串还是心态炸了。

波士顿在机器狗身上也曾短暂用过动轮方案，或者说它测过的方案远比宇树要多——公司成立时长摆在那里——但是作为行业先驱，它连保持一家美国公司的实体都办不到了。

现代汽车2020年以打折价从软银手里买了波士顿动力，正值软银账面巨亏需要回血，而软银当初又是在2017年从Google那里买到手的，Google为什么卖呢，因为觉得太烧钱了，亏不起。

这理由就很离谱，美国的风险资本系统对于亏损的容忍度本来就是全球最高的，没有之一，对于前沿性的研究，砸钱画饼是再寻常不过了的——看这两年硅谷在AI上的投入产出比就知道了——但波士顿动力何以在独一档的地位上被当成不良资产卖来卖去？

那头房间里的大象，美国的科技行业普遍都装作看不到：美国人，如今的美国人，从投行到企业，从CEO到程序员，从纽约到湾区，对制造业的厌弃已经成为本能了。

A16Z的合伙人马克·安德森2011年在「华尔街日报」写了那篇流传甚广的代表作「软件吞噬世界」，大概意思是，边际成本极低的软件公司注定接管一切水草繁盛之地，和这种可以提供指数级增长的生意比起来，其他的行业都不够看。

并不是说马克·安德森的表达有问题，后面这十几年来的现实走向，也确实在证明这条攫取规模化利润的回报是最高的，但美国人的路径依赖到最后必然带来一整代人丧失制造能力的结果。

这里说的丧失制造能力，并不是说丧失制造兴趣或是热情，我前段时间拜访了深圳一家逆向海淘公司，业务就是把华强北的电子配件做成可索引的结构化目录，然后提供从采购到验货再到发包的全流程服务，最大的买方就是美国的DIY市场和高校学生，他们之所以要不远万里的等上几个星期委托中国人来买东西，就是因为在诺大的美国本土，根本找不到供应链。

然后那些学生也只有在读书时才有真正尝试制造某些东西的机会，到了要去大公司里上班领薪后，再也没人愿意把手弄脏了。

但软件终究不能脱离硬件运行，哪怕硬件生产的附加值再不够看，基于采集一手物理数据的入口，制造商腰板硬起来后去做全套解决方案，只取决于能不能组建好的工程师团队，反过来却不一样，制造订单长期外包出去，它就变成产业链配套回不来了。

所以像是多旋翼无人机和四足机器狗这类新兴科技萌芽的原型机一般都还是产自有着试错资本的欧美，也就是所谓「从零到一」的过程，而在「从一到十」的落地阶段，中国的追赶成果就会开始密集呈现，进入「从十到百」的量产之后，中国的供应链成本直接杀死比赛。

波士顿动力的机器人最早在网上爆火的时候，Google X的负责人在内部备忘录里说他已经和媒体沟通了，希望不要让视频和Google扯上太大关系，是不是很迷惑，这么牛逼的事情，你作为母公司非但不高兴，还想躲起来，现在你们懂得这种顾虑从何而来了，就是觉得贵为软件巨头的Google去卷袖子干制造的活儿太卑贱了呗。

当然美国也还有马斯克这样的建设者（Builder），但你要知道马斯克的故事之所以动人，是因为他这样的人现在是极度稀缺的，而且长期以来不受主流科技业界待见，完全是靠逆常识的成就——造汽车，造火箭，造隧道，这都是硅谷唯恐避之不及的事情——去一步步打脸打出来的名声。

如果说宇树是在硬件上引起了一波怀疑现实的热度，那么DeepSeek则在软件的原生地盘，把大模型厂商都给硬控住了。

在微软、Meta、Google都在奔着10万卡集群去做大模型训练时，DeepSeek在2000个GPU上，花了不到600万美金和2个月的时间，就实现了对齐GPT-4o和Claude 3.5 Sonnet的测试结果。

DeepSeek-V2在半年前就火过一波，但那会儿的叙事还相对符合旧版本的预期：中国AI公司推出了低成本的开源模型，想要成为行业里的价格屠夫，中国人就擅长做这种便宜耐用的东西，只要不去和顶级产品比较，能用是肯定的。

但V3则完全不同了，它把成本降了10倍以上，同时质量却能比肩t1阵营，关键还是开源的，相关推文的评论区全是「中国人咋做到的？」

虽然但是，后发的大模型可以通过知识蒸馏等手段实现性价比更高的训练——类似你学习牛顿三定律的速度降低的斜率也在有利于追赶者，肯定比牛顿本人琢磨出定律的速度要快——成本，但匪夷所思的效率提升，是很难用已知训练方法来归纳的，它一定是是在底层架构上做了不同于其他巨头的创新。

另一个角度更有意思，如果针对中国的AI芯片禁售政策最后产生的后果，是让中国的大模型公司不得不在算力受限的约束下实现了效率更高的解决方案，这种适得其反的剧情就太讽刺了。

DeepSeek的创始人梁文锋之前也说过，公司差的从来都不是钱，而是高端芯片被禁运。

所以中国的大模型公司，像是字节和阿里这样的大厂，卡能管够，把年收入的1/10拿出来卷AI，问题不大，但初创公司没这么多弹药，保持不下牌桌的唯一方法就是玩命创新。

李开复今年也一直在表达一个观点，中国做AI的优势从来不是在不设预算上限的情况下去做突破性研究，而是在好、快、便宜和可靠性之间找出最优解。

零一和DeepSeek用的都是MoE（混合专家）模式，相当于是在事先准备的高质量数据集上去做特定训练，不能说在跑分上完全没有水分，但市场并不关心原理，只要质价比够看，就一定会有竞争力。

当然DeepSeek不太一样的是，它不太缺卡，2021年就囤了1万张英伟达A100，那会儿ChatGPT还没影呢，和Meta为了元宇宙囤卡却阴差阳错的赶上AI浪潮很像，DeepSeek买那么多卡，是为了做量化交易⋯⋯

我最早对梁文锋有印象，是「西蒙斯传」里有他写的序，西蒙斯是文艺复兴科技公司的创始人，用算法模型去做自动化投资的开创者，梁文锋当时管着600亿人民币的量化私募，写序属于顺理成章的给行业祖师爷致敬。

交待这个背景，是想说，梁文锋的几家公司，从量化交易做到大模型开发，并不是一个金融转为科技的过程，而是数学技能在两个应用场景之间的切换，投资的目的是预测市场，大模型的原理也是预测Token。

后来看过几次梁文锋的采访，对他的印象很好，非常清醒和聪明的一个人，我贴几段你们感受一下：

「暗涌」：大部分中国公司都选择既要模型又要应用，为什么DeepSeek目前选择只做研究探索？

梁文锋：因为我们觉得现在最重要的是参与到全球创新的浪潮里去。过去很多年，中国公司习惯了别人做技术创新，我们拿过来做应用变现，但这并非是一种理所当然。这一波浪潮里，我们的出发点，就不是趁机赚一笔，而是走到技术的前沿，去推动整个生态发展。

「暗涌」：互联网和移动互联网时代留给大部分人的惯性认知是，美国擅长搞技术创新，中国更擅长做应用。

梁文锋：我们认为随着经济发展，中国也要逐步成为贡献者，而不是一直搭便车。过去三十多年IT浪潮里，我们基本没有参与到真正的技术创新里。我们已经习惯摩尔定律从天而降，躺在家里18个月就会出来更好的硬件和软件。Scaling Law也在被如此对待。但其实，这是西方主导的技术社区一代代孜孜不倦创造出来的，只因为之前我们没有参与这个过程，以至于忽视了它的存在。

「暗涌」：但这种选择放在中国语境里，也过于奢侈。大模型是一个重投入游戏，不是所有公司都有资本只去研究创新，而不是先考虑商业化。

梁文锋：创新的成本肯定不低，过去那种拿来主义的惯性也和过去的国情有关。但现在，你看无论中国的经济体量，还是字节、腾讯这些大厂的利润，放在全球都不低。我们创新缺的肯定不是资本，而是缺乏信心以及不知道怎么组织高密度的人才实现有效的创新。

「暗涌」：但做大模型，单纯的技术领先也很难形成绝对优势，你们赌的那个更大的东西是什么？

梁文锋：我们看到的是中国AI不可能永远处在跟随的位置。我们经常说中国AI和美国有一两年差距，但真实的gap是原创和模仿之差。如果这个不改变，中国永远只能是追随者，所以有些探索也是逃不掉的。英伟达的领先，不只是一个公司的努力，而是整个西方技术社区和产业共同努力的结果。他们能看到下一代的技术趋势，手里有路线图。中国AI的发展，同样需要这样的生态。很多国产芯片发展不起来，也是因为缺乏配套的技术社区，只有第二手消息，所以中国必然需要有人站到技术的前沿。

「暗涌」：很多大模型公司都执着地去海外挖人，很多人觉得这个领域前50名的顶尖人才可能都不在中国的公司，你们的人都来自哪里？

梁文锋：V2模型没有海外回来的人，都是本土的。前50名顶尖人才可能不在中国，但也许我们能自己打造这样的人。

「暗涌」：所以你对这件事也是乐观的？

梁文锋：我是八十年代在广东一个五线城市长大的。我的父亲是小学老师，九十年代，广东赚钱机会很多，当时有不少家长到我家里来，基本就是家长觉得读书没用。但现在回去看，观念都变了。因为钱不好赚了，连开出租车的机会可能都没了。一代人的时间就变了。以后硬核创新会越来越多。现在可能还不容易被理解，是因为整个社会群体需要被事实教育。当这个社会让硬核创新的人功成名就，群体性想法就会改变。我们只是还需要一堆事实和一个过程。

⋯⋯

是不是很牛逼？反正我是被圈粉了，做最难的事情，还要站着把钱赚了，一切信念都基于对真正价值的尊重和判断，这样的80后、90后越来越多的站上了主流舞台，让人非常宽慰，你可以说他们在过去是所谓的「小镇做题家」，但做题怎么了，参与世界未来的塑造，就是最有挑战性的题，喜欢解这样的题，才有乐趣啊。