Introduction to Agents, Agentic Supervisors and Agentic RAG

Compiled by D.Gueorguiev, 9/24/2025

# Workflows and Agents

Distinction between workflows and agents

From [1]:

*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.*

When and when not to use agents

From [1]:

*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.*

A diagram of a workflow

AI-generated content may be incorrect. Figure 1: Visualizing the difference between Workflows and 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. The current LLMs actively use these capabilities , generating their own search queries, selecting appropriate tools, and determining what information to retain.

A diagram of a process

AI-generated content may be incorrect.

Figure 2: Building an augmented LLM

## Workflow: Prompt chaining

Prompt chaining decomposes a task into a sequence of steps, where each LLM call processes the output of the previous one. One can add programmatic checks (aka gates) on the intermediate steps to ensure the process is progressing as expected.

A diagram of a flowchart

AI-generated content may be incorrect.

Figure 3: The prompt chaining workflow

When to use such workflow:

This workflow is used when the task can be easily decomposed into fixed list of subtasks. The main goal is to trade latency for higher accuracy, by making each LLM call an easier task.

Examples where prompt chaining is useful are:

* 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 follow-up 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.

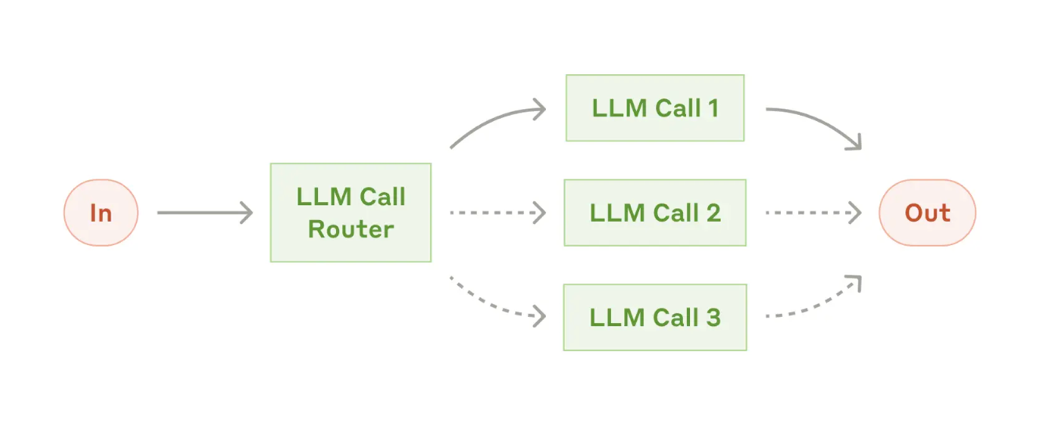


Figure 4: the routing workflow

When to use this workflow:

When there are distinct categories that are better handled separately, and where classification can be handled accurately, either by an LLM or a traditional ML classification model/algorithm.

Examples where routing is useful are:

* 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 LM and harder/unusual questions to larger/specialized LMs

## Workflow: Parallelization

LLMs can work simultaneously on a task and have their outputs aggregated programmatically. This workflow is known as parallelization and has two main variations

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

A diagram of a call

AI-generated content may be incorrect.

Figure 5: the parallelization workflow

When to use this workflow:

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 evaluation 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.

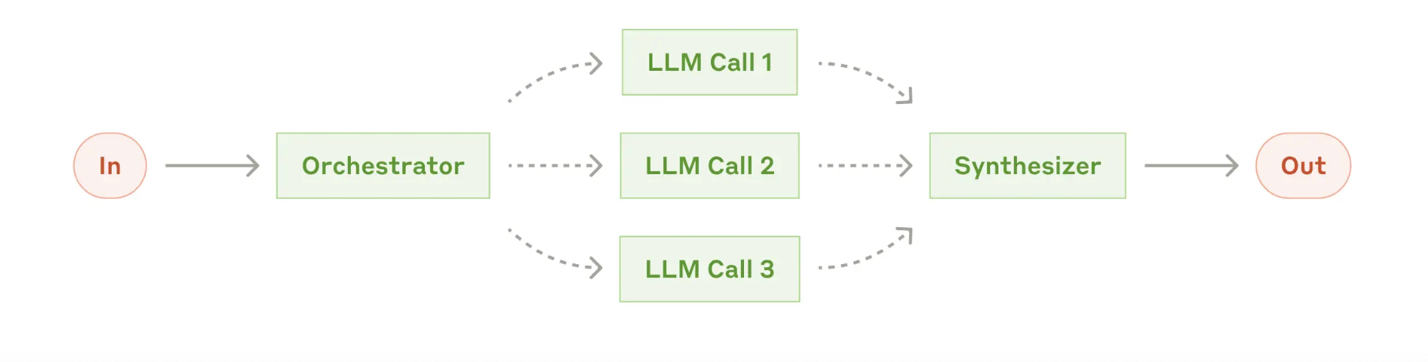


Figure 6: The orchestrator-workers workflow

When to use this workflow:

This workflow is 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 is similar to parallelization topologically the key difference is the level of flexibility – subtasks are not predefined but predetermined by the orchestrator based on the specific input.

Examples where orchestrator-workers are 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.

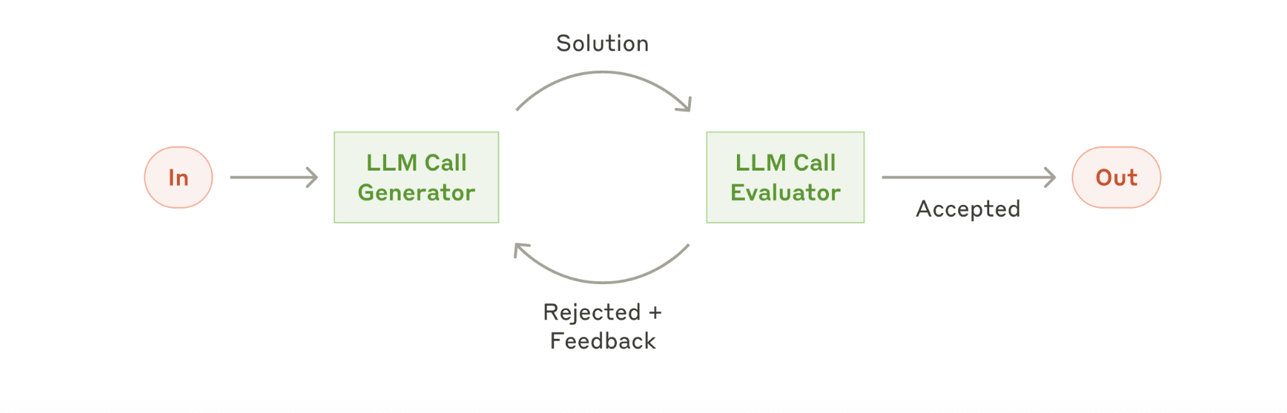


Figure 7: The evaluator-optimizer workflow

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 form, 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 is crucial for the agents to gain the *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 is also common to include stopping conditions (such as a maximum number of iterations) to maintain control.

Agents can handle sophisticated tasks with straightforward implementation. They are typically just LLMs using tools based on environmental feedback in a loop.

When to use agents:

Agents can be used for open-ended problems where it is difficult or impossible to predict the required number of steps, and where you can’t hardcode a fixed path. The LLM wil 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 potential for compounding errors. Therefore, extensive testing in sandboxed environments with imposed guardrails is recommended.

Example where agents are useful is a coding Agent to resolve [SWE-bench tasks](https://www.anthropic.com/engineering/swe-bench-sonnet), which involve edits to many files based on a task description.

A diagram with text and arrows

AI-generated content may be incorrect. Figure 8: high-level flow of a coding agent

A chart of a company's company

AI-generated content may be incorrect.

### Reactive Agents

Reactive agents represent the simplest form of AI architecture, operating on a stimulus-response paradigm. These agents lack internal representation of their environment or worldview, relying instead on a set of predefined rules to map inputs directly to actions.

Key characteristics of reactive agents include:

rapid response times due to minimal processing between input and action;

high efficiency in well-defined, stable environments;

limited ability to learn and adapt in new situations.

Their simplicity means low latencies making these types of agents suitable for real-time applications where low latency is essential.

Limitation of these types of agents is the inability to improve performance through experience, lack of long-term planning or strategic thinking capabilities, sub-optimal behavior in complex and novel situations.

### Deliberative Agents

In contrast with reactive agents, deliberative agents possess internal world models that allow them to reason about their environment and plan future actions. These agents typically employ symbolic AI techniques to represent knowledge and use logical inference to make decisions.

Key characteristics of deliberative agents include

* plans for achieving long-term goals
* the capacity for complex reasoning and problem solving
* the ability to handle uncertainty and incomplete information through probabilistic reasoning

Deliberative agents excel in complex, strategic tasks that require foresight and planning. Their ability to model and reason about their environment makes them particularly suited for scenarios where longer-term optimization is important. **These types of agents are still subject of active research in a rapidly evolving field.**

### Hybrid Agents

Recognizing the complementary strengths of reactive and deliberative architectures, hybrid agents aim to combine the best of both worlds. These agents typically feature a layered architecture, with lower layers handling reactive behaviors and upper layers managing deliberative planning.

# Building a Research Agent

# Multi-agent supervisor

# References

[1] <https://www.anthropic.com/engineering/building-effective-agents>

[2] <https://langchain-ai.github.io/langgraph/tutorials/workflows/#agent>

[3] <https://langchain-ai.github.io/langgraph/tutorials/multi_agent/agent_supervisor/>

[4] <https://langchain-ai.github.io/langgraph/tutorials/rag/langgraph_agentic_rag/>

[5] <https://academy.langchain.com/courses/take/deep-research-with-langgraph/>

[6] <https://github.com/langchain-ai/deep_research_from_scratch>