Multi-Agent AI Patterns for Developers: Pick the Right Pattern for the Right Problem

Suman Das Follow 16 min read · Apr 1, 2026

 137  1   


Multi-Agent AI Patterns
RESEARCH
DEPLOY
Planner
Coder
Critic
Executor
Collaborate & Solve Complex Problems
Task Orchestration
Data Integration
API
Tool Utilization
Which Pattern Will You Choose?

Multi-Agent AI Patterns

In this **agentic world** we're living in, it's tempting to believe that one powerful AI agent can do it all — research, write, code, review, and deploy. And honestly, for simple tasks, it can.

But before we go further, let's get on the same page about what we actually mean by an **"agent."**

An agent isn't just an LLM responding to a prompt. It's an LLM with **autonomy** — it can **reason** about a goal, **decide** what steps to take, **use tools** (APIs, databases, search engines, code interpreters), **observe** the results, and **adjust** its approach. Think of it as the difference between asking someone a question and hiring someone to get a job done. A chatbot answers. An agent **acts**.

Now here's where it gets interesting.

The moment we ask a **single agent** to juggle ten things at once — pull data from five sources, cross-check facts, draft a report, get it reviewed, and fix what's wrong — **the cracks show**. It loses context. It hallucinates under pressure. It forgets what it said three steps ago.

Sound familiar? It should. We solved this exact problem in software engineering decades ago. We stopped building **monoliths**. We broke systems

into **microservices** — small, focused, independently deployable. Each service did one thing well and talked to others through clean interfaces.

Multi-agent AI is that same evolution, and we're right in the middle of it.

Instead of asking one LLM to be the architect, builder, tester, and project manager all at once, we give each role to a **specialized agent.** One plans. One

The real question for us isn't **whether** to use multiple agents — it's **which pattern** fits our problem and **when.**

That's exactly what we'll cover in this article. We'll walk through the most widely adopted **multi-agent design patterns** — each with a visual diagram, a real-world use case, clear guidance on when to use it, and which frameworks support it out of the box.

Whether we're building with **LangGraph, CrewAI, AutoGen, Google ADK, or the Anthropic Agent SDK** — by the end, we'll know exactly which pattern to reach for and why.

Let's dive in.

What We'll Cover

Core Patterns

1. **Sequential / Pipeline** — Agents in a straight line, one after another
2. **Orchestrator-Worker** — A manager delegates to specialists
3. **Parallel / Fan-Out → Fan-In** — Independent tasks run concurrently
4. **Reflection / Self-Critique** — Generate, review, improve, repeat
5. **Router / Dispatch** — Classify and route to the right agent
6. **Planning + Execution** — Separate thinking from doing
7. **Handoff** — Transfer control when out of scope
8. **Evaluator-Optimizer Loop** — Generate, score, improve

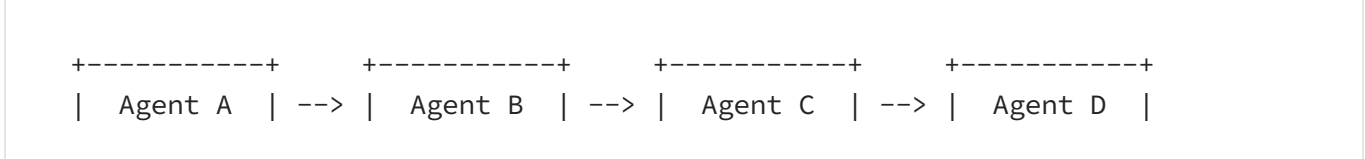
Bonus

- **Emerging Patterns** — Debate, Group Chat, Mixture of Agents

Putting It All Together

- **Quick Reference:** Pick Your Pattern
- **Combining Patterns**
- **Final Thoughts**

1. Sequential / Pipeline



```
| (Extract) |     |(Transform)|     | (Validate)|     |(Summarize)|
+-----------+     +-----------+     +-----------+     +-----------+
```

The simplest pattern we can start with. We line up our agents like an assembly line — each one does its job and passes the result to the next.

When to use:

- Our task has clear, ordered stages
- Each stage has a distinct responsibility
- We need auditability and traceability at every step

When NOT to use:

- Stages are independent and don't need sequential ordering
- We need speed — this pattern is only as fast as the slowest agent

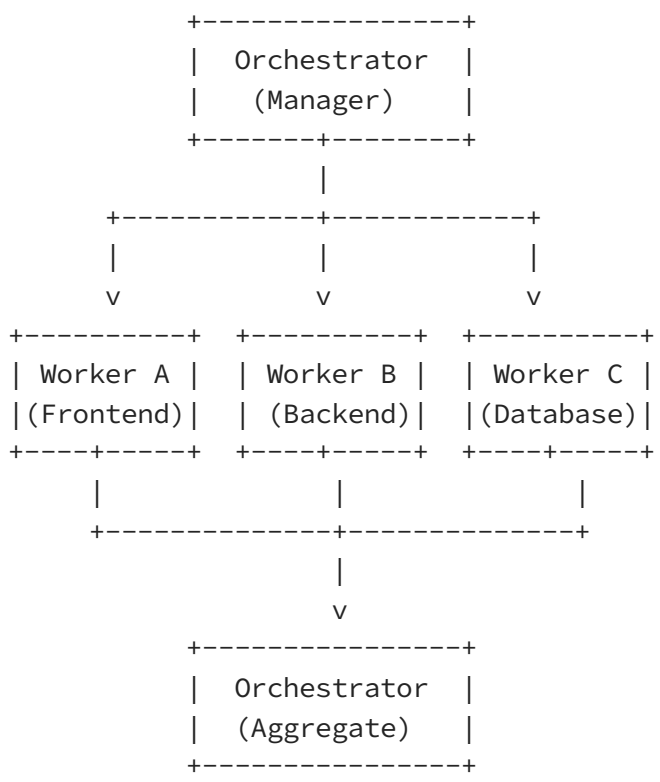
Sample Use Case: A content publishing pipeline — Agent A extracts key points from raw research, Agent B transforms them into a blog draft, Agent C validates facts and grammar, Agent D generates the final summary and SEO metadata.

Framework Support:

- **LangGraph** — Native linear graph edges
- **CrewAI** — `Process.sequential`
- **AutoGen** — `initiate_chats` with sequential carryover

- **Google ADK** — `SequentialAgent` (native workflow agent)
- **Anthropic Agent SDK** — No dedicated sequential API; Claude naturally chains steps through its reasoning loop

2. Orchestrator-Worker (Hierarchical)



This is probably the most widely used pattern in production. We have a **smart manager agent** that understands the big picture, **dynamically decides** what sub-tasks to create, delegates to specialists, monitors progress, and stitches the results together. The key word here is **dynamic** — the orchestrator reasons about the task and may change its plan mid-execution based on what workers return.

How is this different from Parallel Fan-Out? In Fan-Out, we know all the sub-tasks upfront and just run them simultaneously. Here, the orchestrator **figures out** the sub-tasks at runtime, may run some in sequence and others in parallel, and can reassign or retry if something fails. Think of it as a project manager vs a mail room — the project manager thinks and adapts, the mail room just distributes.

When to use:

- Sub-tasks aren't known upfront and need **dynamic decomposition**
- Workers may have **dependencies** on each other's output
- We need **centralized coordination**, error handling, and adaptive replanning

When NOT to use:

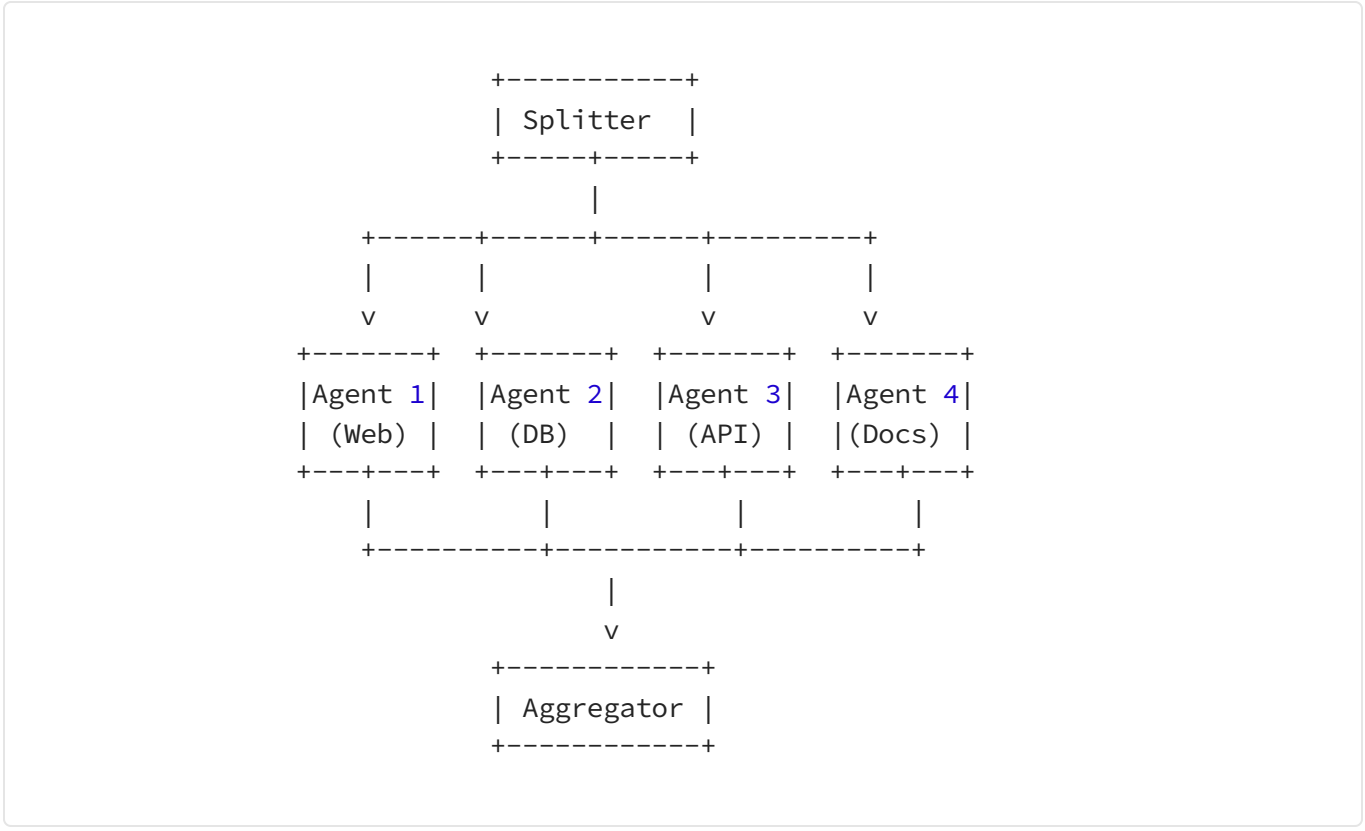
- The task is simple enough for a single agent
- All sub-tasks are independent and known upfront — just use Parallel Fan-Out instead

Sample Use Case: An e-commerce order processing system — a customer places an order, and the orchestrator agent takes over. It delegates to an Inventory agent (check stock availability), a Payment agent (process the payment), and a Shipping agent (calculate delivery options and schedule dispatch). But here's the key: if the Inventory agent reports "out of stock," the orchestrator **adapts** — it asks a Recommendation agent to suggest alternatives, updates the customer, and only proceeds to Payment once the customer confirms. A simple fan-out can't handle this kind of dynamic decision-making.

Framework Support:

- **LangGraph** — Supervisor pattern with sub-graphs
- **CrewAI** — `Process.hierarchical` with `manager_llm`
- **AutoGen** — `GroupChat` with `GroupChatManager`
- **Google ADK** — `LlmAgent` with `sub_agents` and `transfer_to_agent`
- **Anthropic Agent SDK** — Parent agent spawns subagents via Agent tool (core pattern)

3. Parallel / Fan-Out → Fan-In



This is the **speed pattern.** When we have independent sub-tasks that don't depend on each other, why run them one by one? We fan out to multiple agents running concurrently, then fan in to merge the results. Unlike the

Orchestrator-Worker, there's **no smart manager here** — we know all the sub-tasks upfront, fire them all at once, and a simple aggregator combines the results.

How is this different from Orchestrator-Worker? The splitter doesn't think — it just distributes pre-defined tasks. There's no dynamic decision-making, no replanning, no dependency management between workers. It's pure parallelism for speed. If any worker's output should change what other workers do, we need the Orchestrator pattern instead.

When to use:

- All sub-tasks are **known upfront** and **independent** of each other
- **Latency is critical** — we want results as fast as the slowest agent
- We're gathering data from multiple sources that don't affect each other

When NOT to use:

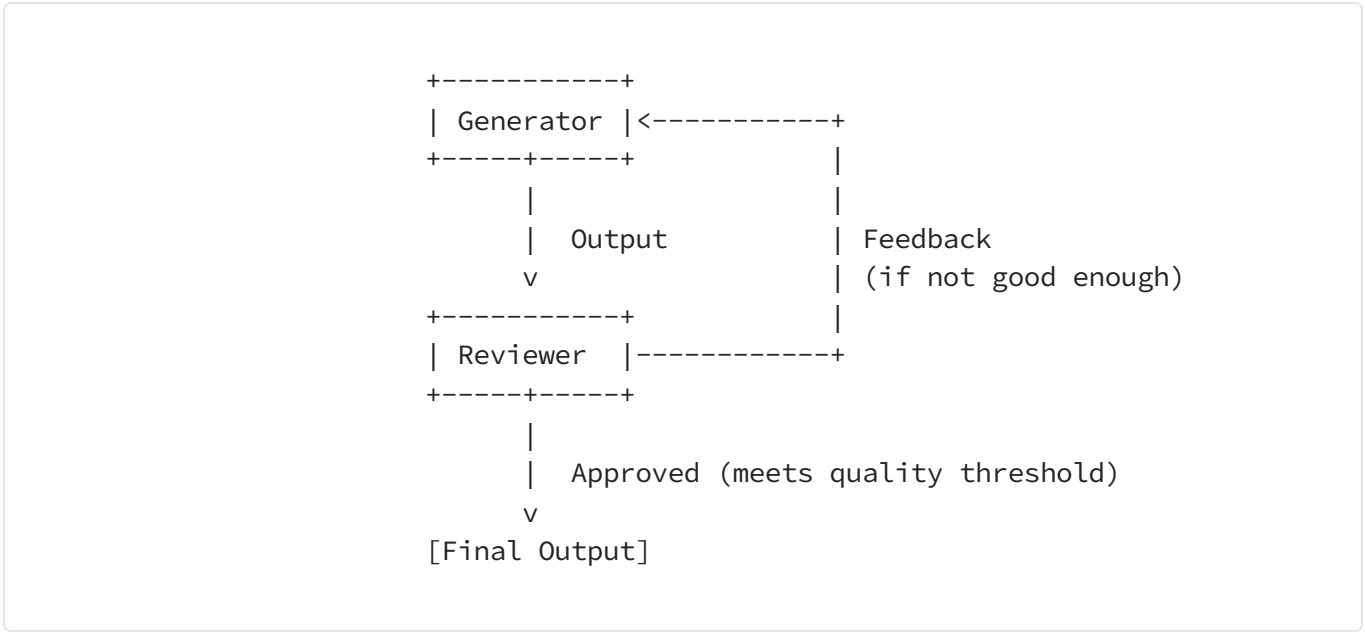
- Tasks have dependencies on each other — use Orchestrator-Worker instead
- We need dynamic task creation based on intermediate results

Sample Use Case: A competitive analysis tool — we need data from four independent sources. Agent 1 scrapes competitor websites, Agent 2 pulls financial data from APIs, Agent 3 gathers social media sentiment, Agent 4 searches news articles. None of them need each other's output. They all run at the same time, and the aggregator combines everything into a single report. Total time = slowest agent, not the sum of all four.

Framework Support:

- **LangGraph** — Parallel branches with fan-in node, `Send` API for map-reduce
- **CrewAI** — `async_execution=True` on tasks
- **AutoGen** — `a_initiate_chats` with concurrent execution
- **Google ADK** — `ParallelAgent` (native workflow agent, runs sub-agents concurrently)
- **Anthropic Agent SDK** — Multiple Agent tool calls in a single message execute concurrently

4. Reflection / Self-Critique



We all write better with a good editor. This pattern gives our AI a built-in editor — one agent generates, another reviews, and we iterate until the output meets our quality bar.

When to use:

- Code generation (write → test → fix cycles)
- Content creation where first drafts aren't good enough
- Any task where we can define clear quality criteria

When NOT to use:

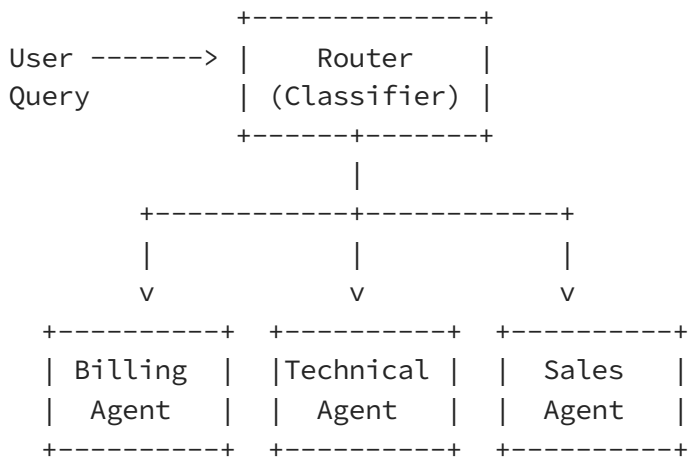
- We need real-time responses — iteration adds latency
- The task is simple enough that the first pass is usually correct
- We don't have clear criteria for "good enough"

Sample Use Case: An automated code review system — the Generator agent writes a function, the Reviewer agent runs tests, checks for edge cases, security issues, and style. If anything fails, it sends feedback. The Generator rewrites. We repeat until all tests pass and the review score exceeds our threshold.

Framework Support:

- **LangGraph** — Cycles with conditional stop (well-documented pattern)
- **CrewAI** — `guardrail` for LLM-based validation, `max_iter` for retries
- **AutoGen** — `register_nested_chats` for inner critic conversations
- **Google ADK** — `LoopAgent` with `max_iterations` and `escalate=True` to exit
- **Anthropic Agent SDK** — No dedicated reflection API; Claude naturally self-evaluates and retries through its reasoning loop

5. Router / Dispatch



Not every query needs a specialist. But every query needs the *right* specialist. We use a lightweight router agent to **classify the input once** and send it to the best-fit agent. That's it — the router's job is done.

How is this different from Orchestrator-Worker? The router is a **traffic cop**, not a project manager. It makes **one decision** (which agent should handle this?) and steps aside. It doesn't break the task into sub-tasks, doesn't wait for results, doesn't coordinate between agents, and doesn't aggregate anything. The Orchestrator does all of that. Use a Router when the task goes to **one** specialist. Use an Orchestrator when the task needs to be **split across multiple** specialists.

When to use:

- We receive diverse input types that each need a **single** specialist
- We want to optimize cost (route simple queries to cheaper/smaller models)

- Customer support, helpdesks, and multi-domain assistants

When NOT to use:

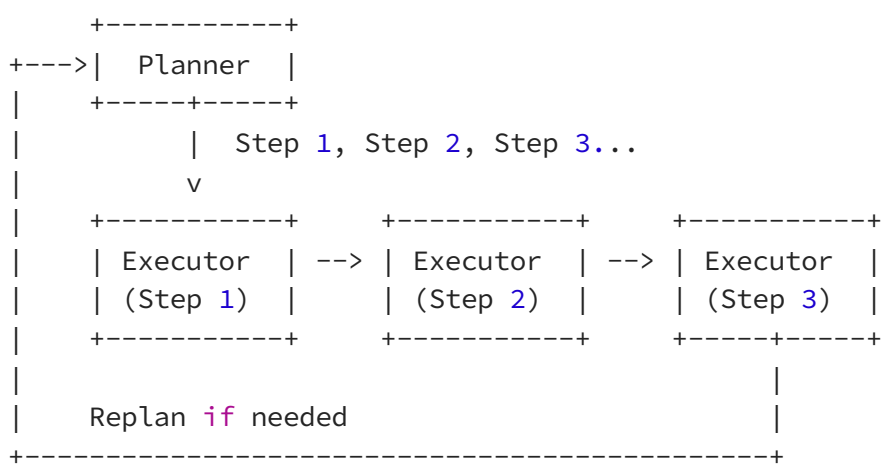
- The task needs to be split across multiple agents — use Orchestrator-Worker instead
- All queries need the same type of processing
- We only have one specialist — just call it directly

Sample Use Case: A SaaS customer support system — the Router classifies incoming tickets as billing, technical, account, or feature-request. A billing query goes **entirely** to the Billing agent. A technical issue goes **entirely** to the Technical agent. The router doesn't coordinate between them — each specialist handles the full request on its own.

Framework Support:

- **LangGraph** — Conditional edges with routing functions (native)
- **CrewAI** — `@router` decorator in Flows, `ConditionalTask`
- **AutoGen** — Custom `speaker_selection_method` in GroupChat
- **Google ADK** — `LlmAgent` with `sub_agents` and routing instructions (no dedicated RouterAgent)
- **Anthropic Agent SDK** — LLM-driven dispatch via Agent tool descriptions

6. Planning + Execution



We separate the **thinking** from the **doing**. A planner agent creates the full roadmap upfront, then executor agents carry out each step. The planner steps back and only re-engages if something fails and we need to replan.

How is this different from Orchestrator-Worker? The Orchestrator is a **micromanager** — it stays involved at every step, makes decisions between tasks, and coordinates in real-time. Plan + Execute is more like an **architect and builders** — the architect draws the full blueprint upfront, hands it to builders, and only comes back if the foundation cracks. The planning and execution are **clearly separated phases**, not interleaved.

When to use:

- Complex, multi-step goals where we benefit from **thinking before acting**
- We need the ability to **replan** when intermediate results change the approach
- Research tasks, coding projects, data analysis workflows

When NOT to use:

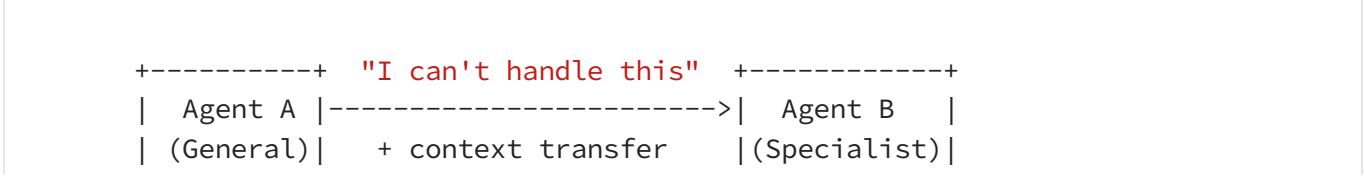
- The workflow is fixed and well-known — just use a Pipeline
- We need real-time decision-making at every step — use Orchestrator-Worker instead
- The task is too simple to justify a separate planning step

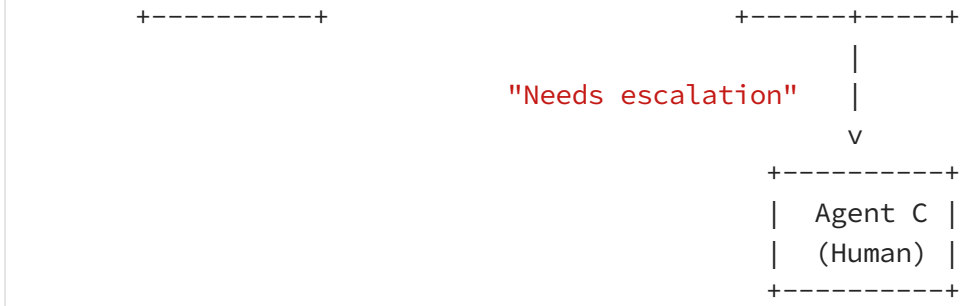
Sample Use Case: An automated research assistant — the Planner agent breaks down "write a market analysis report on EV batteries" into: (1) identify top 5 companies, (2) gather financial data, (3) analyze patents, (4) compare market share, (5) draft report. The planner hands this off and steps back. Executor agents handle each step sequentially. If Step 2 reveals a major company we missed, the Planner re-engages, revises the plan, and executors continue with the updated steps.

Framework Support:

- **LangGraph** — Plan-and-execute pattern (well-documented with tutorials)
- **CrewAI** — Task `context` dependencies + Flows for planning
- **AutoGen** — Planner + executor via two-agent or GroupChat (manual setup)
- **Google ADK** — Compose with `SequentialAgent` + `LoopAgent` for replanning
- **Anthropic Agent SDK** — No dedicated planning API; Claude naturally plans and executes through its reasoning loop

7. Handoff





Sometimes an agent knows it's out of its depth. Instead of hallucinating an answer, it hands off the conversation — with full context — to a more capable agent or a human.

When to use:

- Customer service with escalation tiers
- Multi-domain assistants where scope changes mid-conversation
- Human-in-the-loop workflows where some cases need manual review

When NOT to use:

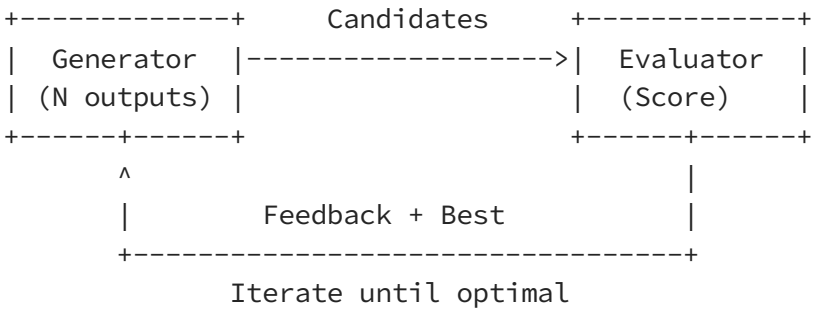
- We can handle everything with a single agent
- We prefer routing upfront rather than mid-conversation handoffs

Sample Use Case: A healthcare triage bot — Agent A handles general wellness questions. When it detects symptoms that need medical advice, it hands off to Agent B (a medically-trained agent with access to clinical guidelines). If Agent B determines the case is urgent, it escalates to Agent C — a human doctor — with the full conversation history attached.

Framework Support:

- **LangGraph** — Conditional edges with full state transfer between nodes
- **CrewAI** — `allow_delegation=True` for autonomous agent delegation
- **AutoGen** — `register_hand_off` in Swarm pattern (v0.4+)
- **Google ADK** — `transfer_to_agent` with session state context
- **Anthropic Agent SDK** — Subagents return to parent (no direct peer handoff)

8. Evaluator-Optimizer Loop



We generate multiple candidates, score them, and use the feedback to generate better ones. It's evolution in action — each iteration gets us closer to optimal.

When to use:

- We can define a clear scoring function or evaluation criteria
- Prompt optimization, query refinement, search ranking

- Code optimization where we can measure performance

When NOT to use:

- We don't have clear evaluation criteria
- We need a single quick answer, not an optimized one
- The generation cost is too high to produce multiple candidates

Sample Use Case: An ad copy optimizer — the Generator creates 10 variations of ad copy. The Evaluator agent scores each on readability, brand alignment, emotional appeal, and CTA strength. The top 3 go back to the Generator with feedback. We repeat for 3 rounds and pick the winner.

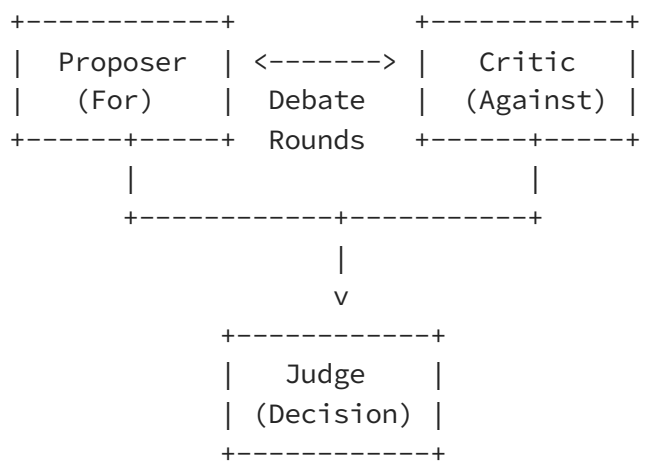
Framework Support:

- **LangGraph** — Cycles with scoring nodes and conditional exit
- **CrewAI** — Task output validation functions + `guardrail` + `max_iter`
- **AutoGen** — `register_nested_chats` for evaluation before reply
- **Google ADK** — `LoopAgent` with evaluator `LlmAgent` (custom composition)
- **Anthropic Agent SDK** — No dedicated scoring API; implement scoring as a custom tool and Claude iterates through its reasoning loop

Bonus: Emerging Patterns

These patterns are powerful in specific scenarios but less commonly used in production today. They're worth knowing as our multi-agent systems mature.

Debate / Adversarial:



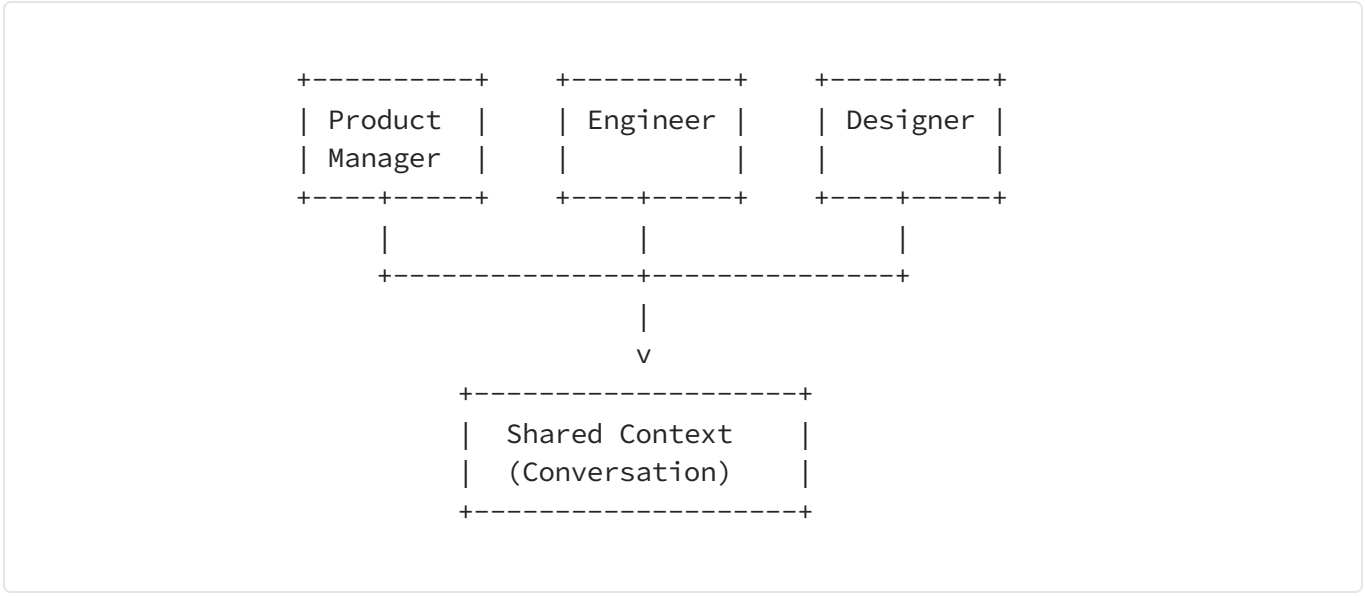
Sometimes the best way to find the truth is to argue about it. We set up agents on opposing sides and let them challenge each other. A judge agent listens to both and makes the final call. Best for **high-stakes decisions**, fact verification, and red-teaming.

Sample Use Case: An investment analysis system — a Bull agent argues for, a Bear agent argues against. A Judge agent weighs both sides and generates a balanced recommendation.

Framework Support:

- **AutoGen** — Two-agent `initiate_chat` (built-in, natural fit)
- **LangGraph** — Cyclic graphs with conditional edges
- **Google ADK** — `LoopAgent` with two `LlmAgent` sub-agents

Multi-Agent Group Chat:



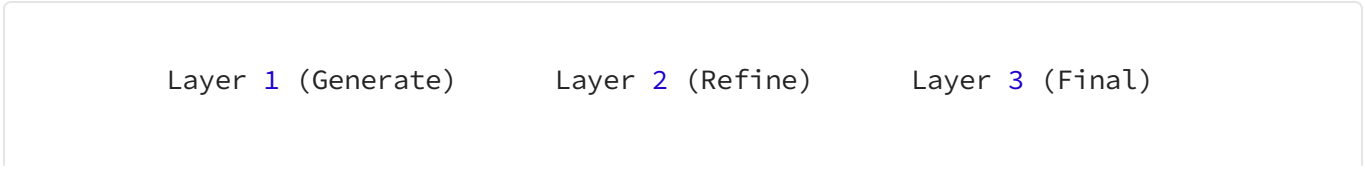
Multiple agents sit in a shared conversation, each contributing from their expertise. Best for **brainstorming** and simulating cross-functional team discussions. Expensive — every agent reads every message.

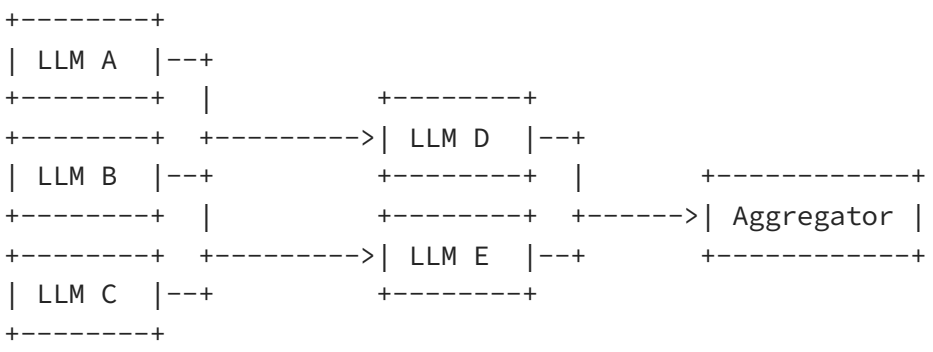
Sample Use Case: Designing a new API — PM, Engineer, Security, and QA agents iterate in shared conversation until we have a complete spec.

Framework Support:

- **AutoGen** — `GroupChat` + `GroupChatManager` (native, best support)
- **CrewAI** — Crew shares task outputs between agents (not true group chat)
- **Anthropic Agent SDK** — Not supported (subagents run in isolated contexts)

Mixture of Agents (MoA):





We use multiple models to generate diverse responses, then refine and aggregate. Different models have different strengths — together, they're better than any one alone. Best for **maximum accuracy** when cost and latency aren't constraints.

Sample Use Case: A legal contract review — Claude analyzes clause risks, GPT-4 checks regulatory compliance, Gemini cross-references case law. An aggregator synthesizes findings into a unified risk report.

Framework Support:

- **LangGraph** — Parallel nodes with different model configs per node
- **CrewAI** — Each agent configurable with different `llm` parameter
- **AutoGen** — Each agent in GroupChat can use different `llm_config`
- **Google ADK** — Model-agnostic; each `LlmAgent` can use Gemini, Claude, GPT via LiteLLM
- **Anthropic Agent SDK** — Claude-only (no multi-provider support)

Quick Reference: Pick Your Pattern

Core Patterns:

core pattern comparison

Emerging Patterns:

Emerging patterns comparison

Combining Patterns: How Production Systems Actually Work

In the real world, we rarely use a single pattern in isolation. Most production systems combine two or three patterns together. Here are some common combinations we see:

Router + Reflection We route incoming queries to specialists, and each specialist uses a reflection loop to ensure quality before responding.

Orchestrator + Parallel + Reflection The orchestrator decomposes a task, fans out to workers in parallel, each worker reflects on its output, and the orchestrator aggregates the results.

Plan + Execute + Debate The planner creates a strategy, executors carry it out, and at critical decision points, a debate between agents validates the

approach before we proceed.

Router + Handoff + Human-in-the-Loop The router classifies the query, specialist agents handle it, and if confidence is low, we hand off to a human — with full context preserved.

Final Thoughts

Multi-agent design patterns aren't just academic concepts — they're practical tools we can reach for when a single agent isn't enough. The key is to **start simple** and **add complexity only when we need it.**

Here's a simple decision framework we can follow:

1. **Can one agent handle it?** → Don't use multi-agent. Keep it simple.
2. **Do we need specialization?** → Orchestrator-Worker or Router.
3. **Do we need speed?** → Parallel Fan-Out.
4. **Do we need accuracy?** → Reflection, Debate, or Mixture of Agents.
5. **Do we need adaptability?** → Plan + Execute.
6. **Do we need graceful escalation?** → Handoff.

The agentic era is just getting started, and these patterns will evolve as we build more sophisticated systems. But the fundamentals — decomposition, specialization, coordination, and feedback — these are timeless engineering principles wearing a new hat.

Now let's go build something.

References

Framework Documentation

- LangGraph — https://langchain-ai.github.io/langgraph/
- CrewAI — https://docs.crewai.com/
- AutoGen — https://microsoft.github.io/autogen/
- Google ADK — https://google.github.io/adk-docs/
- Anthropic Agent SDK — https://platform.claude.com/docs/en/agent-sdk/overview

Pattern Deep Dives

- LangGraph Multi-Agent Patterns — https://langchain-ai.github.io/langgraph/concepts/multi_agent/
- Google ADK Multi-Agent Systems — https://google.github.io/adk-docs/agents/multi-agents/
- Google Developers Blog: Multi-Agent Patterns in ADK — https://developers.googleblog.com/en/developers-guide-to-multi-agent-patterns-in-adk/
- AutoGen Conversation Patterns — https://microsoft.github.io/autogen/0.2/docs/tutorial/conversation-patterns/
- CrewAI Processes (Sequential & Hierarchical) — https://docs.crewai.com/concepts/processes
- Anthropic: Building Effective Agents — https://www.anthropic.com/research/building-effective-agents

Research Papers & Articles

- “Mixture-of-Agents Enhances Large Language Model Capabilities” (2024) — https://arxiv.org/abs/2406.04692
- “Plan-and-Solve Prompting” — https://arxiv.org/abs/2305.04091
- “Reflexion: Language Agents with Verbal Reinforcement Learning” — https://arxiv.org/abs/2303.11366
- “CAMEL: Communicative Agents for Mind Exploration” — https://arxiv.org/abs/2303.17760

Workflow Agent Types (Google ADK)

- SequentialAgent — https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/
- ParallelAgent — https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/
- LoopAgent — https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/

AI Agent | Agentic Ai | Agents | Genai | LLM

Written by Suman Das

1.2K followers · 141 following

An AI Enthusiast & Principal AI Engineer 🌟

Follow

Responses (1)

Sean Lewis

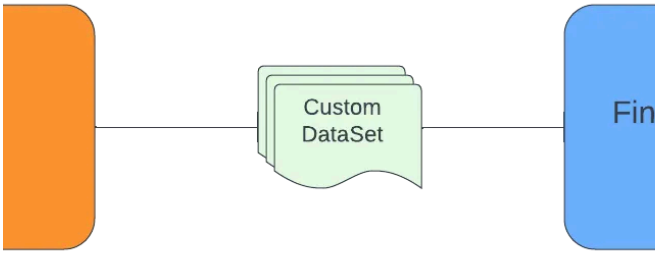
What are your thoughts?

William Wang
1 day ago

Great pattern taxonomy. The one that's underrepresented in most discussions is the "reviewer as separate agent" pattern — having a different model (or at least a different prompt/persona) review the output of the implementation agent. It catches... more

1 reply Reply

More from Suman Das



Suman Das

Fine Tune Large Language Model (LLM) on a Custom Dataset with...

The field of natural language processing has been revolutionized by large language...

Jan 24, 2024 2.4K 23

In Crux Intelligence by Suman Das

Auto Scaling Microservices with Kubernetes Event-Driven...

Autoscaling (sometimes spelled as auto scaling or auto-scaling) is the process of...

Feb 25, 2023 85 2

Suman Das

Building REST APIs using FastAPI, SQLAlchemy & Uvicorn

FastAPI

Oct 1, 2021 191 2

In Crux Intelligence by Suman Das

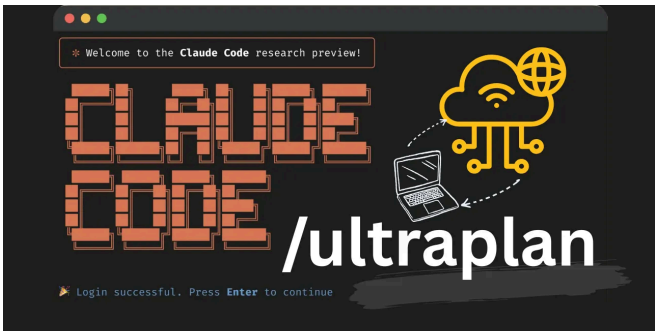
Async Architecture with FastAPI, Celery, and RabbitMQ

In one of my earlier tutorials, we have seen how we can optimize the performance of a...

May 10, 2022 348 4

See all from Suman Das

Recommended from Medium



In Level Up Coding by Fareed Khan

Building Claude Code with Harness Engineering

Multi-agents, MCP, skills system, context pipelines and more

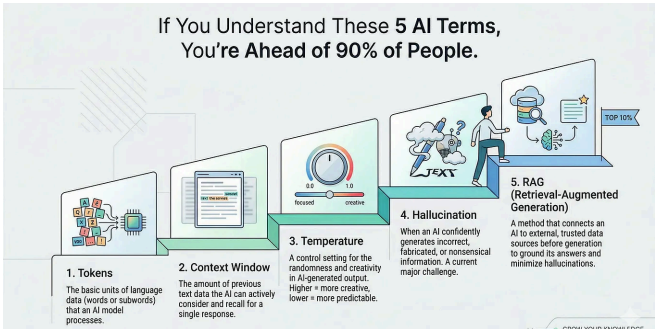
6d ago 1K 8

Joe Njenga

Claude Code Ultraplan Launched: I Just Tested It (And It's Better Tha...

Anthropic has added Claude Code ultraplan, and I was quick to test it. You might like it or...

Apr 4 744 23



In Towards AI by Shreyas Naphad

If You Understand These 5 AI Terms, You're Ahead of 90% of...

Master the core ideas behind AI without getting lost

Mar 29 9.4K 190

Bibek Poudel

The SKILL.md Pattern: How to Write AI Agent Skills That Actuall...

If your skill does not trigger, it is almost never the instructions. It is the description.

Feb 25 127

TechEon

The Complete Agentic AI System Design Interview Guide 2026

A senior engineer's handbook for navigating the toughest agentic AI interviews

Jan 28 107 2

In Data Science Colle... by Han HELOIR YAN, Ph....

Everyone Analyzed Claude Code's Features. Nobody Analyzed Its...

Five hundred thousand lines of leaked source code reveal that the moat in AI coding tools i...

Mar 31 4.9K 12

See more recommendations