Iterate.ai

Agentic Design Patterns for the Enterprise

A Pattern Library for Technical Leaders: From Single Agents to Production Swarms

Author: Brian Sathianathan
Cofounder/CTO, Iterate.ai

Generate (iterate.ai/platform/generate) | Interplay (iterate.ai/platform/interplay)

A comprehensive guide to 30+ agentic AI architecture patterns, organized from foundational building blocks to enterprise-scale multi-agent systems. Written for CTOs, CIOs, CDOs, VPs, Chief AI Officers, and Chief Digital Officers.

Design Patterns | Multi-Agent | RAG | Orchestration | Enterprise AI

By Iterate.ai | Generate | Interplay | 2026
For Technical Leaders and Decision Makers

Executive Summary

Agentic AI—systems where large language models autonomously plan, reason, use tools, and collaborate—is the most consequential architectural shift since microservices. Yet for most organizations, the path from a single chatbot to a production-grade multi-agent system is unclear. **Which patterns actually work? When should you use one agent versus ten? How do you maintain control, observability, and security as autonomy increases?**

This white paper answers those questions. We catalog **30+ agentic design patterns**, organized in a complexity ladder from the simplest single-agent tool caller to enterprise-scale swarm orchestration. For each pattern, we explain *what it is, when to use it, when not to use it, and the best practices that separate production-grade implementations from prototypes.* Every pattern is illustrated with an architecture diagram so that technical leaders can quickly evaluate which patterns match their use cases.

30+
Agentic design patterns cataloged and explained

6 Tiers
Foundation through enterprise multi-agent

$52.6B
Projected AI agent market value by 2030

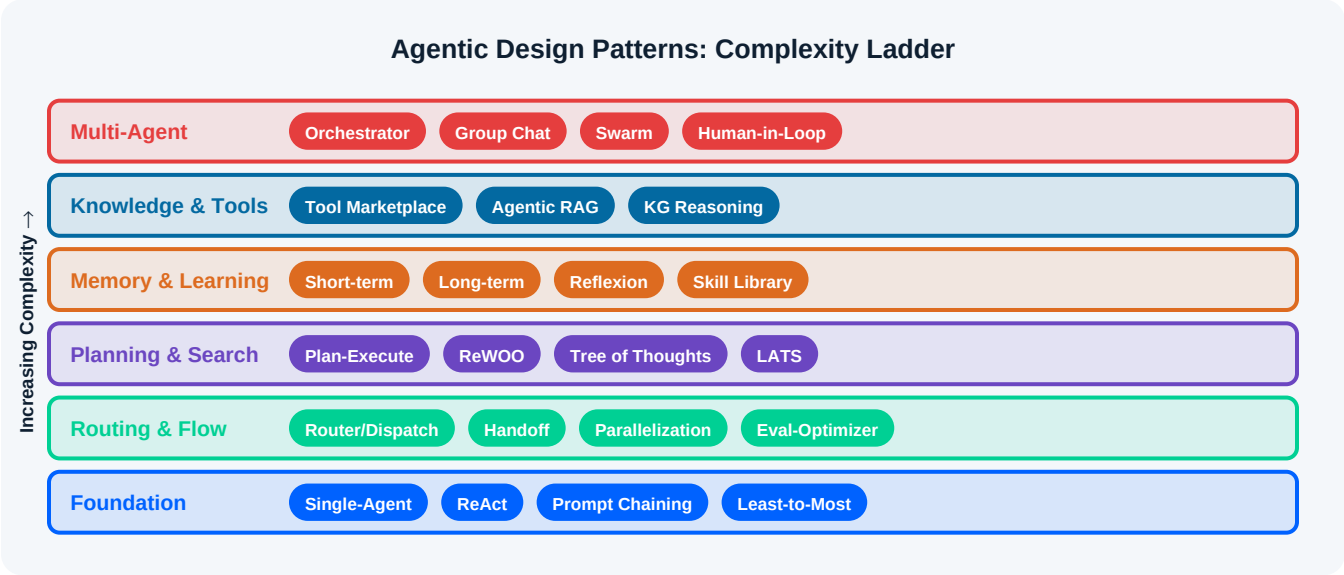
> *The difference between a demo and a production agent is not the model—it is the orchestration pattern. The right pattern turns an expensive, unreliable prototype into a system that your organization can trust, observe, and scale.*
>
> Anthropic, Building Effective AI Agents, 2025

Who This Paper Is For

This paper is written for senior technical leaders navigating agentic AI adoption: **Chief Technology Officers** evaluating architecture strategies, **Chief AI Officers** building centers of excellence, **Chief Information Officers** managing enterprise AI portfolios, **Chief Data Officers** ensuring data governance in agentic workflows, **Chief Digital Officers** driving digital transformation, and **VPs of Engineering** making build-vs-buy decisions. We assume familiarity with LLMs but not with agentic system design.

The Agentic Pattern Taxonomy

Not every problem needs a swarm. The most common mistake in agentic AI adoption is reaching for complex multi-agent patterns before exhausting what a well-designed single agent can do. Our taxonomy organizes patterns into six tiers of increasing complexity. **Start at the bottom. Move up only when the current tier cannot solve your problem.**



The six tiers of agentic design patterns, from foundation to enterprise multi-agent.

How to Read This Paper

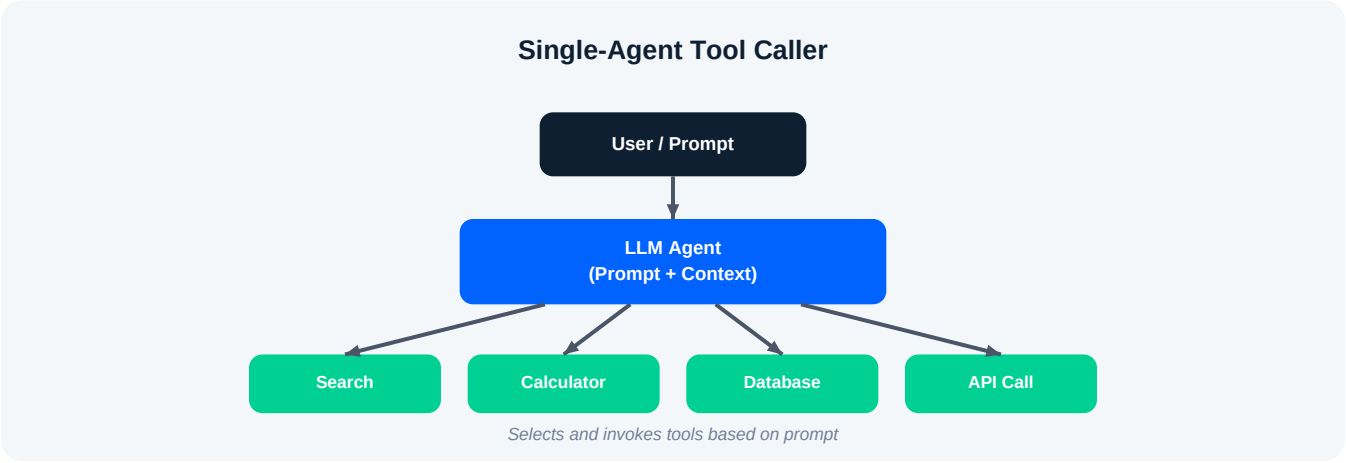
Each pattern section follows a consistent structure: a **visual diagram** showing the pattern's architecture, a **plain-language explanation** of how it works, **when to use it** (the scenarios where this pattern excels), **when not to use it** (common misapplications), and **best practices** drawn from production deployments. Patterns are grouped by tier, and each tier builds on the concepts introduced in the previous one.

The pattern descriptions are grounded in published research and official documentation from Anthropic, Google Cloud, Microsoft Azure, AWS, Hugging Face, and the academic papers that originated each pattern. References are provided at the end of each section so that your engineering teams can dive deeper.

Tier 1: Foundation Patterns

Foundation patterns are the building blocks of every agentic system. Master these before moving to more complex architectures. Most real-world tasks—including many that feel like they need multiple agents—can be solved with a well-designed foundation pattern.

Single-Agent Tool Caller

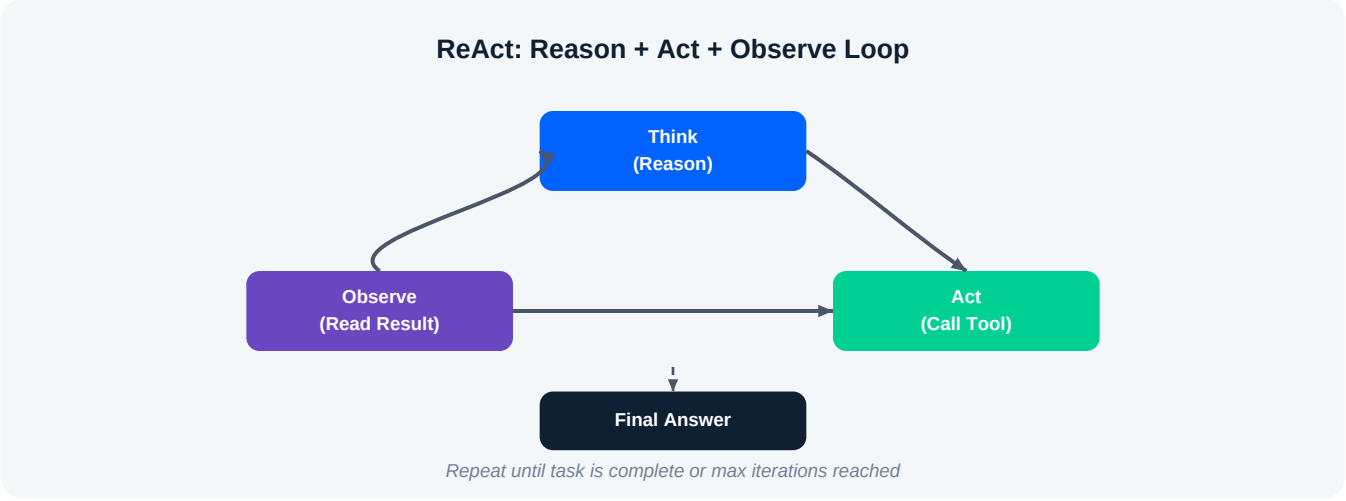


The simplest agentic pattern: one LLM, one prompt, one toolset, one shared context. The agent receives a user request, decides which tools to call (if any), executes them, and returns a response. This is the baseline—the pattern you should use until you have a concrete reason not to. Google Cloud's single-agent guidance and LangChain's core agent documentation both recommend starting here.

When to use: Customer support with tool access, code generation with file system tools, data lookups, API integrations, and any task where a single context window is sufficient. **When not to use:** Tasks that exceed the model's context window, require specialized expertise across multiple domains simultaneously, or need parallel execution for latency reasons.

Best practices: Keep tool descriptions precise and unambiguous—the agent's tool selection is only as good as the tool definitions. Limit the toolset to 10–15 tools; beyond that, tool selection accuracy degrades. Use structured output (JSON mode) for tool calls to avoid parsing failures. Implement timeout and retry logic for each tool.

ReAct (Reason + Act)

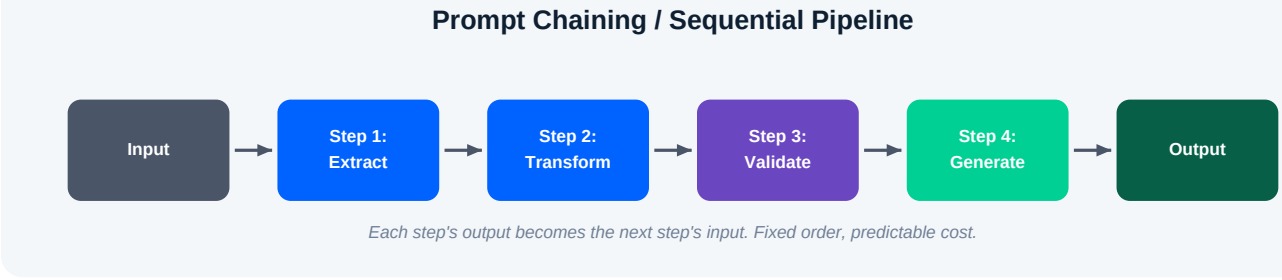


ReAct is the canonical agentic loop: **Think** → **Act** → **Observe** → **Repeat.** The agent first reasons about what to do (generating a thought trace), then takes an action (calling a tool or producing output), then observes the result, and loops until the task is complete. This is the closest thing to the "Singleton" of agentic design patterns because so many other designs extend it. The original ReAct paper by Yao et al. demonstrated that interleaving reasoning and acting significantly outperforms either reasoning-only or acting-only approaches.

When to use: Multi-step research tasks, complex question answering with tool use, debugging workflows, and any task where the agent needs to adapt its plan based on intermediate results. **When not to use:** Simple, one-shot tasks (single-agent is cheaper), tasks with known fixed workflows (prompt chaining is more predictable), or latency-sensitive applications (each loop iteration adds a full LLM call).

Best practices: Set a maximum iteration count (typically 5–10) to prevent infinite loops. Include a "I have enough information to answer" escape hatch in the prompt. Log every thought-action-observation triple for debugging. Monitor token consumption—ReAct loops can be expensive because each iteration includes the full conversation history.

Prompt Chaining / Sequential Pipeline



A fixed pipeline where each step's output becomes the next step's input. Unlike ReAct, the workflow order is determined at design time, not at runtime. Anthropic, Hugging Face, Azure, and Google all describe variants of this pattern. It is the right choice when you know the sequence of operations in advance and each step can be independently validated.

When to use: Document processing pipelines (extract → classify → summarize), data transformation workflows, content generation with review stages, and any multi-step process with a known order. **When not to use:** Tasks where the next step depends on dynamic conditions that cannot be predicted, or tasks where steps need to run in parallel.

Best practices: Add validation gates between steps—check that each step's output meets the next step's input requirements before proceeding. Keep each step focused on a single transformation. Use smaller, cheaper models for intermediate steps when possible; reserve the most capable model for the step that requires the most judgment.

Least-to-Most Decomposition

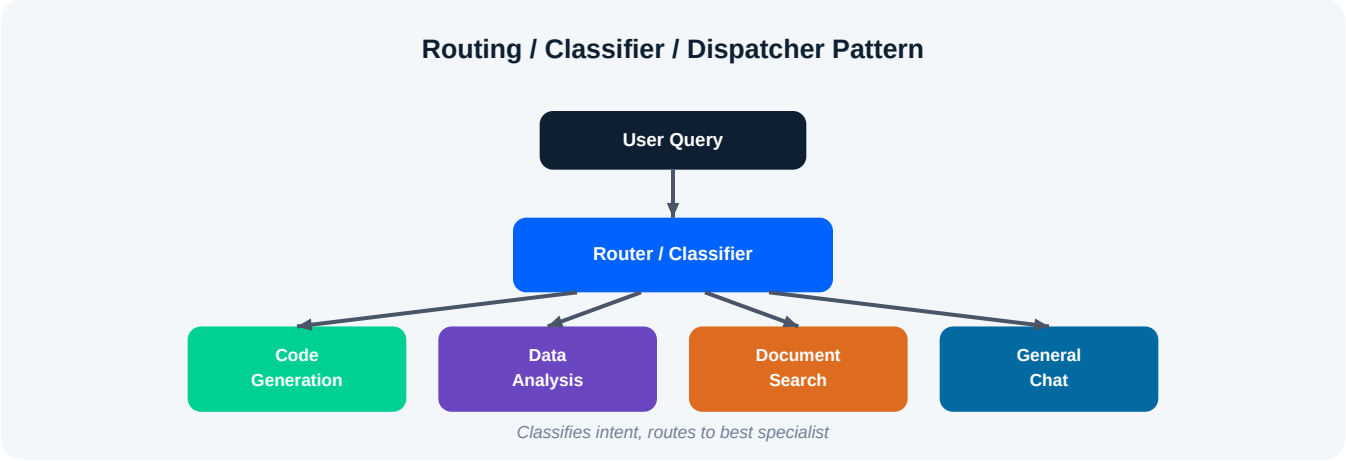
Break a hard problem into progressively simpler subproblems and solve them in sequence, using earlier solutions as context for later ones. While not always framed as an "agent" pattern, it is a critical agent-planning primitive. The key insight is that LLMs struggle with complex problems presented all at once but perform well on the decomposed subproblems.

When to use: Complex mathematical reasoning, multi-step coding challenges, tasks that require building up from simpler components. **Best practice:** Let the LLM itself generate the decomposition—it often identifies subproblems that a human designer would miss.

Tier 2: Routing and Flow Control

Once you outgrow a single agent, the first architectural decision is how to route work to the right handler. Routing patterns add a classification or dispatch layer that directs tasks to specialized agents, prompts, or tools. These patterns maintain a single point of coordination while enabling specialization.

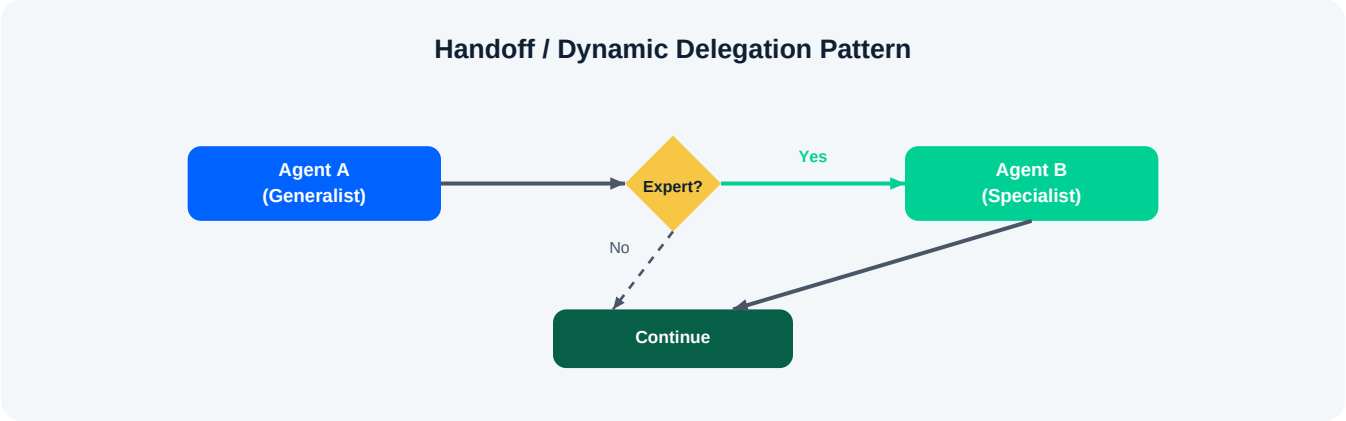
Routing / Classifier / Dispatcher



A router agent classifies the incoming request and dispatches it to the appropriate downstream handler. The router itself is lightweight—its only job is classification. AWS calls this "routing" or "dynamic dispatch"; it is one of the most deployed patterns in production because it is simple, predictable, and easy to monitor.

When to use: Customer-facing systems with diverse query types, multi-domain applications, tiered support systems, and any scenario where different inputs need fundamentally different processing. **Best practices:** Use a fast, cheap model for the router (classification does not need GPT-4-class reasoning). Define clear, non-overlapping categories. Add a "fallback" category for edge cases. Monitor classification accuracy and retrain as query distributions shift.

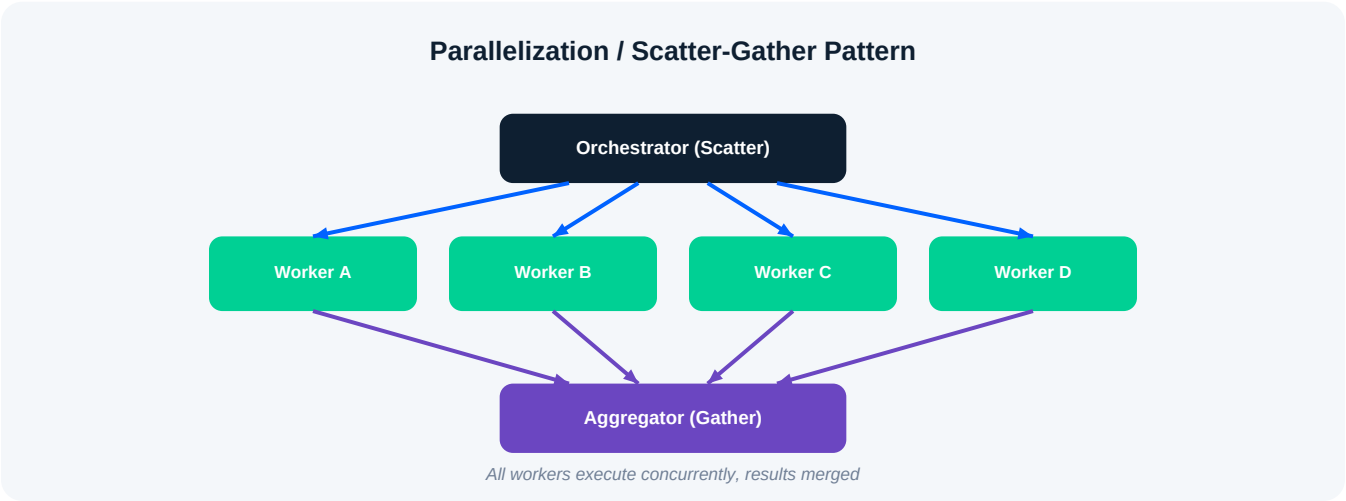
Handoff / Triage / Dynamic Delegation



Unlike static routing, the handoff pattern allows an active agent to decide *mid-flight* to transfer control to a better specialist. The generalist agent starts processing, recognizes that the task requires specialized expertise, and hands off with full context. Azure's handoff orchestration documentation provides the clearest description of this family.

When to use: Complex support workflows where initial triage cannot fully classify the task, multi-turn conversations that evolve beyond the initial agent's scope, and systems where specialists have expensive resources (so you only invoke them when needed). **Best practices:** Define clear handoff protocols—what context transfers, what does not. Avoid circular handoffs (A → B → A) by tracking handoff history.

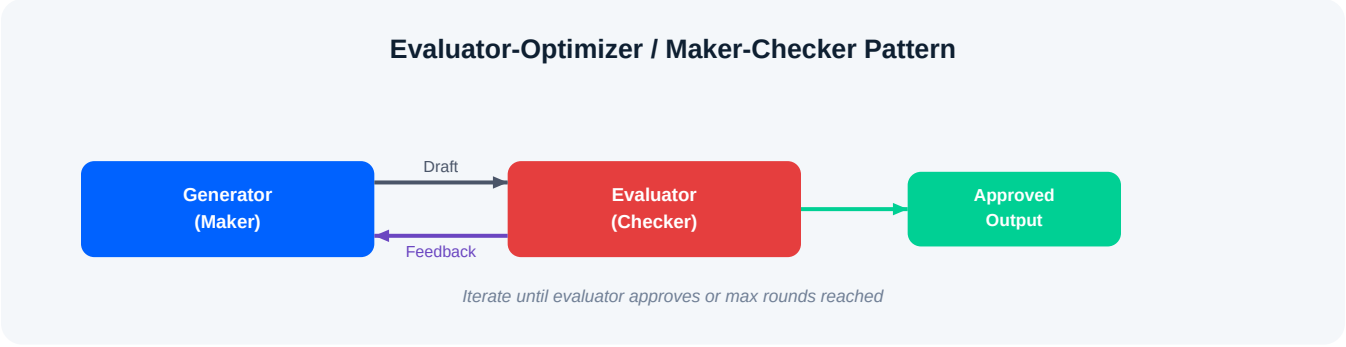
Parallelization / Scatter-Gather



Fan the task out to multiple workers simultaneously, then aggregate the results. Anthropic, AWS, Azure, Hugging Face, and Google all describe variants of this pattern. There are two main sub-patterns: **sectioning** (split the task into independent subtasks) and **voting** (run the same task multiple times and take the best or consensus result).

When to use: Tasks that decompose into independent subtasks (analyzing different sections of a document), latency-sensitive applications where parallel execution is faster than sequential, quality-critical tasks where multiple perspectives improve outcomes, and batch processing. **Best practices:** Define the aggregation strategy before building—majority vote, best-of-N, merge, or synthesize. Handle partial failures gracefully (some workers may fail). Set per-worker timeouts to prevent stragglers from blocking the entire pipeline.

Evaluator-Optimizer / Maker-Checker



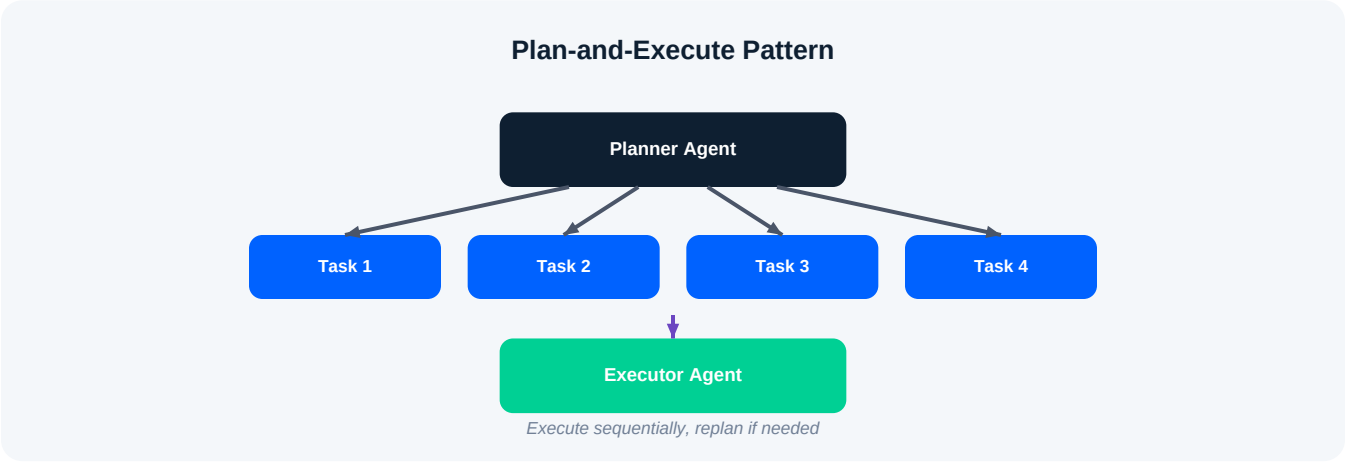
One of the most reused production patterns: one model generates, another critiques, and the first revises based on the feedback. Anthropic, Azure, AWS, Hugging Face, and Google all converge on this pattern because it adds quality control without requiring a full multi-agent system. The **Self-Refine** variant uses the same model for both generation and critique, which is simpler but sometimes less effective.

When to use: Code generation (generate then review for bugs), content creation (draft then edit for tone), translation (translate then back-translate to verify), and any task where quality improves with iteration. **Best practices:** Use a different model or temperature for the evaluator to avoid self-reinforcing errors. Define concrete evaluation criteria (not just "is this good?"). Cap iterations at 2–3 rounds—diminishing returns set in quickly.

Tier 3: Planning and Search Patterns

Planning patterns separate the "what to do" from the "how to do it." Instead of reasoning and acting in a tight loop (ReAct), these patterns first build a plan and then execute it. This separation trades some flexibility for better cost, speed, and predictability. Search patterns extend planning by exploring multiple candidate solutions simultaneously.

Plan-and-Execute



A planner agent decomposes the task into a sequence of steps, and an executor agent carries them out one by one. If a step fails or produces unexpected results, control returns to the planner for replanning. LangChain's writeup on this pattern is particularly practical. The key advantage over ReAct is that the planner can use a more capable (and expensive) model while the executor uses a cheaper one, because execution steps are typically simpler.

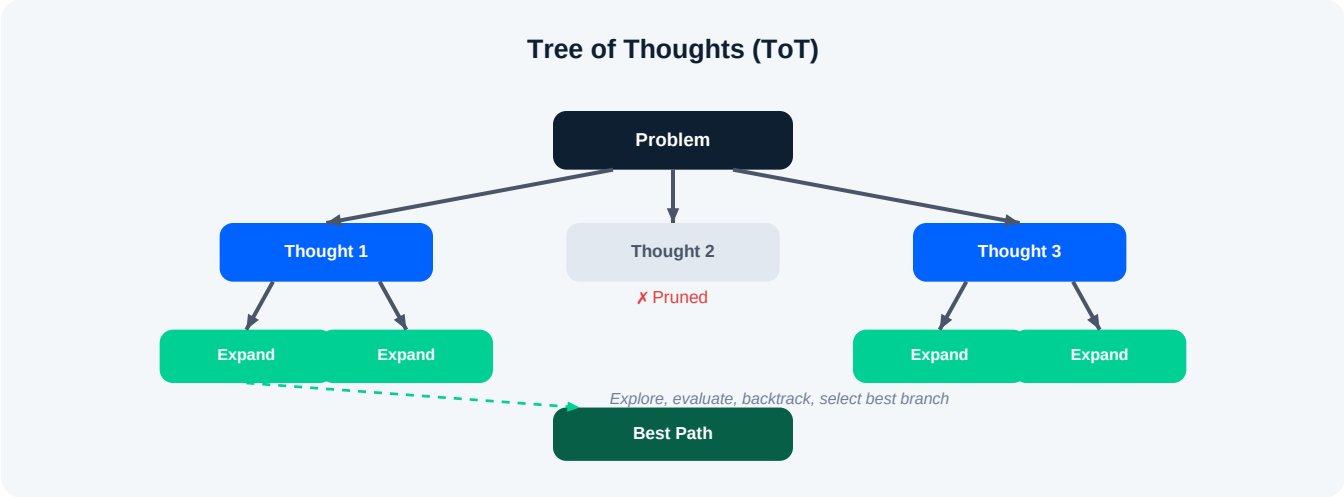
When to use: Complex multi-step tasks where you want cost control (planning once is cheaper than reasoning at every step), tasks where a plan can be reviewed by a human before execution, and workflows where predictability matters more than adaptability. **Best practices:** Include a "replan" mechanism for when execution diverges from the plan. Make plans concrete and actionable—"search for X" not "gather information." Store the plan in structured format (JSON/YAML) so it can be programmatically inspected.

ReWOO (Reasoning Without Observation)

ReWOO decouples reasoning from tool observations to reduce repeated prompting. The planner generates a complete plan with placeholder references for tool outputs (e.g., "#E1 = search('topic')"), executes all tools, then substitutes the results back into the reasoning chain. This avoids the cost of sending the full conversation history at each step (the main expense of ReAct loops). ReWOO can reduce token usage by 5–10x compared to ReAct on multi-tool tasks.

When to use: Tasks with many tool calls where token cost matters, batch processing scenarios, and tasks where tool calls are independent of each other. **When not to use:** Tasks where later tool calls depend on earlier results.

Tree of Thoughts (ToT)



Instead of following a single chain of reasoning, the agent branches over multiple candidate reasoning paths, evaluates each branch, prunes unpromising paths, and continues exploring the most promising ones. This is directly inspired by how humans solve problems that require exploration and backtracking—chess, puzzles, creative writing.

When to use: Problems with large solution spaces where the first approach may not be optimal (puzzle solving, strategic planning, creative tasks), mathematical proofs, and game playing. **When not to use:** Straightforward tasks where the first answer is usually correct—ToT adds significant latency and cost. **Best practices:** Define a clear evaluation function for branch scoring. Use breadth-first search for breadth and depth-first for depth. Prune aggressively—explore 3–5 branches per level, not dozens.

Language Agent Tree Search (LATS)

LATS extends Tree of Thoughts by combining reasoning, acting, and planning with Monte Carlo Tree Search (MCTS). It uses environment feedback to guide search, making it more effective than pure reasoning-based tree search. LATS sits above ToT in complexity and is often used as a bridge toward search-heavy agents. The key innovation is using the LLM as both the policy (deciding which action to take) and the value function (evaluating state quality).

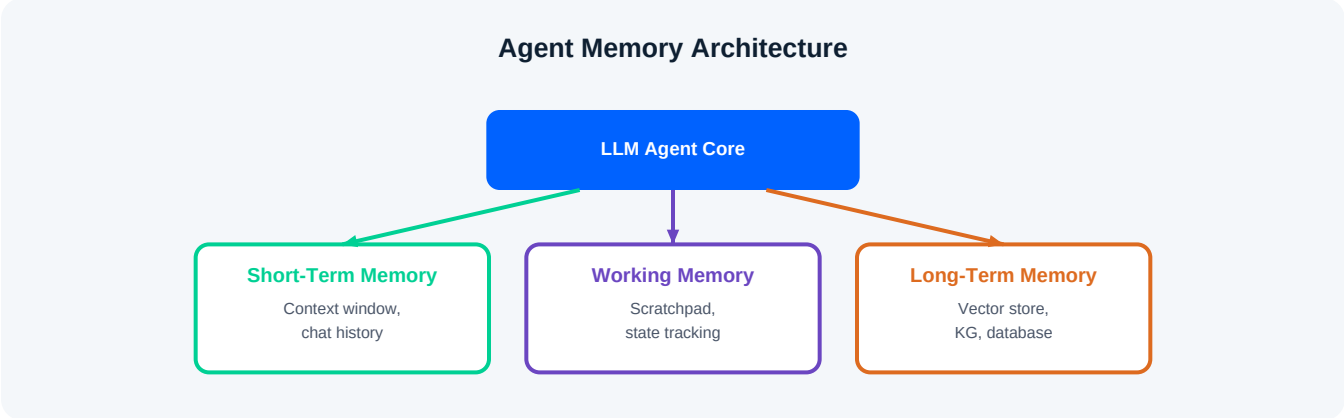
When to use: Complex decision-making with environment interaction, autonomous coding agents that need to explore multiple implementation strategies, and research tasks where systematic exploration outperforms greedy approaches.

Graph of Thoughts / Diagram of Thought

These patterns generalize chain and tree reasoning into richer graph structures where intermediate results can be combined, merged, and refined across multiple paths. They matter when the task is non-linear and multiple intermediate dependencies must be tracked—for example, synthesizing information from multiple sources where each source informs the interpretation of others. While more niche than tree-based approaches, they represent the frontier of structured reasoning.

Tier 4: Memory and Learning Patterns

An agent without memory is stateless—it cannot learn from past interactions, maintain context across sessions, or improve over time. Memory patterns give agents the ability to accumulate knowledge, reflect on past performance, and build reusable capabilities. This is the tier where agents start to feel genuinely intelligent rather than merely reactive.



Three-layer memory architecture: short-term, working, and long-term.

Short-Term Memory

Thread or session memory that keeps current state coherent within one run or conversation. This is the minimum viable memory layer for nontrivial agents. In practice, short-term memory is the conversation history plus any scratchpad or state variables the agent maintains during a task. The challenge is that context windows are finite, so agents must manage what stays in context and what gets summarized or evicted.

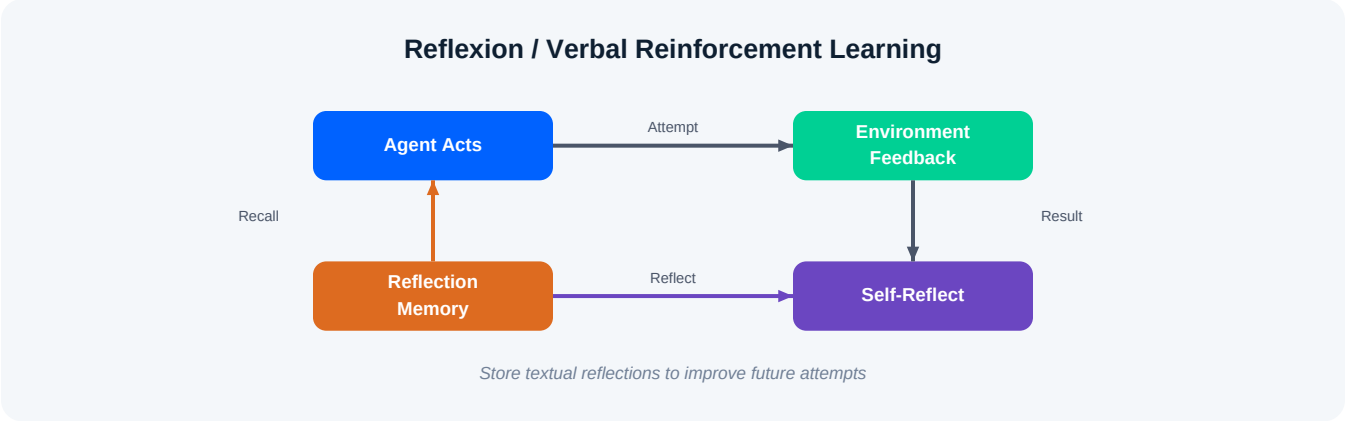
Best practices: Implement sliding-window summarization for long conversations. Use structured state objects rather than relying on raw conversation history. Separate "facts learned" from "actions taken" in the memory representation.

Long-Term Memory

Persistent cross-session memory for user facts, learned strategies, preferences, or past outcomes. This is what enables agents to improve over time and personalize their behavior. Implementations range from simple key-value stores to vector databases and knowledge graphs. LangChain's long-term memory documentation provides excellent practical guidance.

Best practices: Define a clear memory schema—what gets stored, how it is indexed, and when it expires. Use semantic search (embeddings) for retrieval rather than exact match. Implement memory consolidation—periodically summarize and merge related memories to prevent unbounded growth. Separate factual memories from procedural ones.

Reflexion / Verbal Reinforcement



The agent stores textual reflections from prior attempts and uses them to improve future actions. After each attempt, the agent generates a natural-language reflection analyzing what went wrong and what to try differently. These reflections are stored in memory and included in the prompt for subsequent attempts. This is one of the cleanest "agent learns without weight updates" patterns.

When to use: Autonomous coding agents (reflect on test failures), research agents (reflect on search strategy effectiveness), and any task where the agent needs to learn from mistakes across attempts. **Best practices:** Keep reflections specific and actionable ("the SQL query timed out because the table is not indexed on that column") rather than vague ("try a different approach"). Limit stored reflections to the 5–10 most recent to prevent context bloat.

Generative-Agent Architecture

A more cognitively inspired design where observation, memory, reflection, and planning work together in a unified architecture. Originally introduced by Park et al. for simulating believable human behavior, this pattern has practical applications in persistent assistants, social agents, and any system that needs to maintain a coherent identity and behavior over extended periods. The architecture includes a memory stream (all observations), a retrieval mechanism (what is relevant now), reflection (higher-level insights), and planning (what to do next based on goals and context).

Skill Library / Lifelong-Learning Agent

Instead of only storing text memories, the agent accumulates **reusable executable skills**—verified code snippets, procedures, or tool-use sequences that it can invoke in future tasks. Voyager (Wang et al.) is the best-known reference, demonstrating an agent that builds a growing library of skills in Minecraft and applies them to progressively harder challenges. In enterprise settings, this pattern enables agents that get measurably better at their jobs over time, building institutional knowledge in executable form.

When to use: Long-running agents that encounter recurring task patterns, domain-specific automation where procedures can be codified, and systems where accumulated expertise has high value. **Best practices:** Version skills and track their success rates. Include preconditions and postconditions with each skill. Implement skill composition—combining simple skills to solve complex tasks.

Tier 5: Knowledge and Tool Patterns

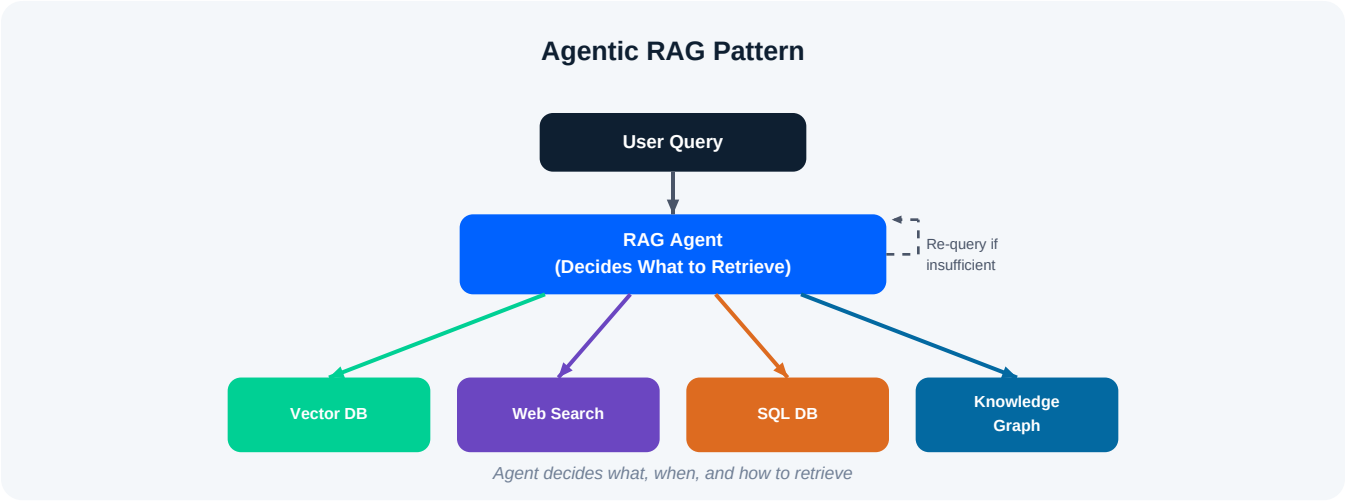
These patterns extend agents with external knowledge and capabilities. Rather than relying solely on the LLM's parametric knowledge, these agents actively retrieve, query, and reason over external data sources and tool libraries. This is where agents become genuinely useful for enterprise tasks that require access to proprietary data and systems.

Tool-Marketplace / Model-Router Agent

The LLM acts as a controller over a library of external models, tools, or APIs. HuggingGPT is the classic example: a language model receives a task, decomposes it, selects the appropriate specialist model from a registry (image generation, speech recognition, object detection), executes each subtask, and synthesizes the results. In enterprise settings, this pattern enables a single agent interface to orchestrate dozens of specialized capabilities.

When to use: Multi-modal tasks, systems that need to integrate diverse AI capabilities, and platforms where new tools or models are frequently added. **Best practices:** Maintain a structured tool registry with clear capability descriptions, input/output schemas, and cost estimates. Implement circuit breakers for unreliable tools. Cache tool results when inputs are deterministic.

Agentic RAG



RAG becomes agentic when retrieval is no longer a fixed pipeline. Instead of the pattern "retrieve → augment → generate," the agent dynamically decides **what to retrieve, when to retrieve, and whether to re-query or reflect on the results.** The agent may issue multiple retrieval queries, combine results from different sources, determine that the retrieved context is insufficient and reformulate, or decide that no retrieval is needed at all.

When to use: Complex research tasks, multi-source information synthesis, enterprise knowledge management, and any application where a single retrieval pass is insufficient. This is now a major branch of agent design—most production RAG systems are moving toward agentic architectures because static pipelines fail on nuanced queries.

Best practices: Give the agent explicit control over query formulation—do not just pass the user's question verbatim to the retriever. Implement relevance scoring so the agent can decide if retrieved context is sufficient. Support multiple retrieval backends (vector search, keyword search, SQL, knowledge graphs) and let the agent choose. Log retrieval decisions for debugging and quality improvement.

Knowledge-Graph Reasoning / Graph-Grounded Agents

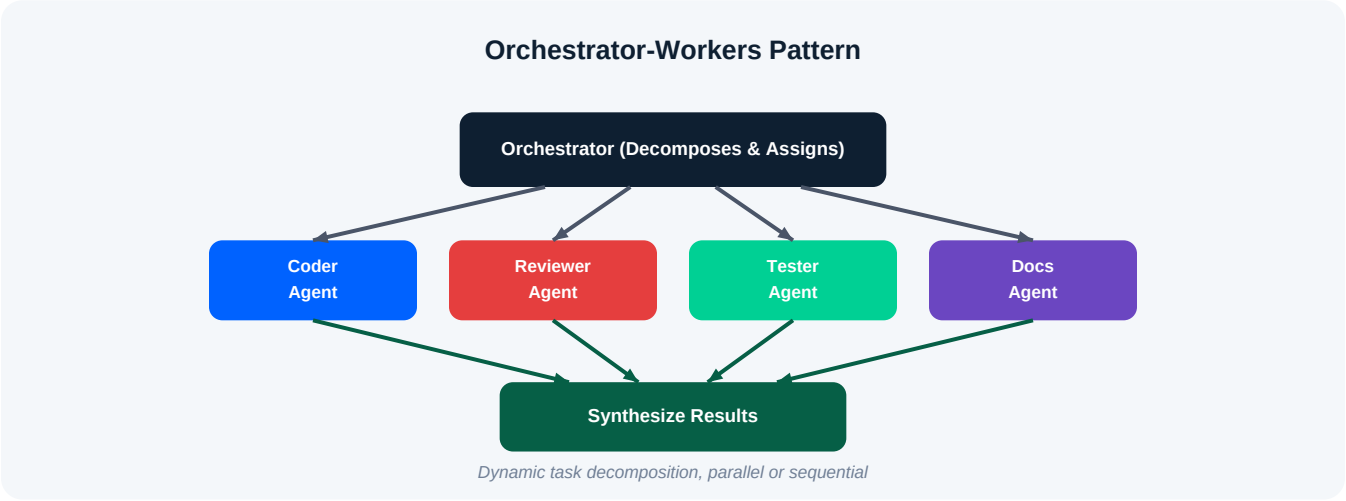
Agents that plan over a graph or use a knowledge graph as an explicit reasoning substrate. Good references include MindMap, Plan-on-Graph (PoG), and KAG. These agents navigate structured relationships between entities—following edges, querying subgraphs, and using graph topology to guide their reasoning. This is particularly powerful for enterprise applications where data has rich relational structure: supply chains, organizational hierarchies, regulatory frameworks, and technical documentation with cross-references.

When to use: Domain-specific reasoning over structured knowledge (medical diagnosis, legal analysis, technical troubleshooting), multi-hop question answering, and tasks where relationships between entities matter as much as the entities themselves. **Best practices:** Keep the graph schema clean and well-documented. Use the LLM to translate natural-language queries into graph queries rather than embedding raw graph data in prompts.

Tier 6: Multi-Agent Orchestration

Multi-agent patterns are the most powerful—and the most complex—agentic architectures. They involve multiple LLM-powered agents collaborating, debating, or competing to solve problems that no single agent can handle effectively. **Use these patterns only when you have exhausted simpler alternatives.** The coordination overhead, debugging complexity, and cost of multi-agent systems are significant. When they are the right fit, however, they unlock capabilities that are impossible with any other approach.

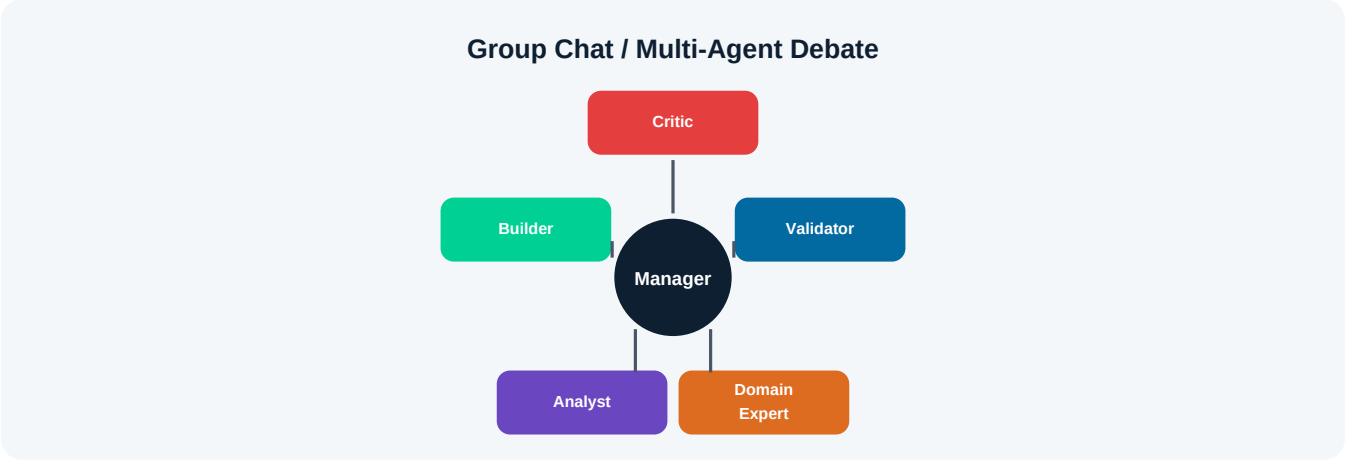
Orchestrator-Workers



A manager agent breaks down an ambiguous, open-ended task dynamically and assigns work to specialist subagents, then synthesizes the outputs into a coherent result. Unlike prompt chaining (where the pipeline is fixed), the orchestrator determines the decomposition at runtime based on the specific task. Anthropic's writeup is the clearest production-oriented description of this pattern.

When to use: Complex software engineering tasks ("refactor this module"), research synthesis ("analyze this market from five angles"), and any task that benefits from specialized experts working on different aspects. **Best practices:** Give the orchestrator a clear mandate and constraints. Define the interface between orchestrator and workers (what information flows in each direction). Implement progress tracking so the orchestrator can detect stalled workers. Use the simplest worker agents that can accomplish each subtask—not every worker needs to be a full ReAct agent.

Group Chat / Multi-Agent Debate



Multiple agents communicate in a shared thread, usually with a manager controlling turns. Each agent has a defined role (critic, builder, analyst, domain expert) and contributes its perspective. The manager decides who speaks next and when the discussion has reached a conclusion. This pattern excels at brainstorming, validation, and consensus-building.

When to use: Design reviews, complex decision-making that benefits from multiple perspectives, red-teaming (one agent attacks, another defends), and quality assurance where diverse viewpoints catch more issues. **Best practices:** Keep the group small (3–5 agents)—larger groups generate more tokens than insight. Give each agent a distinct, non-overlapping role. Implement a turn-taking protocol to prevent agents from talking past each other. Set a maximum number of rounds.

Hierarchical Task Decomposition

A coordinator decomposes large objectives into nested subtasks and delegates them down a tree of agents. Google Cloud and Microsoft both document this family clearly. Unlike flat orchestrator-workers, hierarchical decomposition supports arbitrary depth—a coordinator can delegate to sub-coordinators, which delegate to workers. This mirrors how large engineering organizations actually operate: VPs delegate to directors, who delegate to managers, who delegate to individual contributors.

When to use: Very large, complex projects that require multiple levels of decomposition, enterprise workflows that mirror organizational hierarchy, and tasks where subtasks themselves are complex enough to require their own orchestration.

Magentic / Manager-Led Open-Ended Orchestration

Microsoft's Magentic-One framework introduces a particularly useful pattern where a manager builds and continuously updates a task ledger—a living plan that tracks progress, identifies blockers, and reassigns work—while specialist agents act on tools and real systems. The manager does not just decompose and dispatch; it actively monitors, adapts, and steers the overall effort. This is closest to how a skilled project manager operates.

Role-Playing / Society of Agents

Agents are assigned explicit roles and collaborate according to those roles. CAMEL (Li et al.) is foundational: it demonstrated that giving two agents complementary roles (e.g., "Python programmer" and "stock trader") and letting them communicate produces better task completion than a single agent. The key insight is that **role assignment constrains agent behavior in productive ways**—an agent told it is a "security reviewer" catches more vulnerabilities than a general-purpose agent asked to review code.

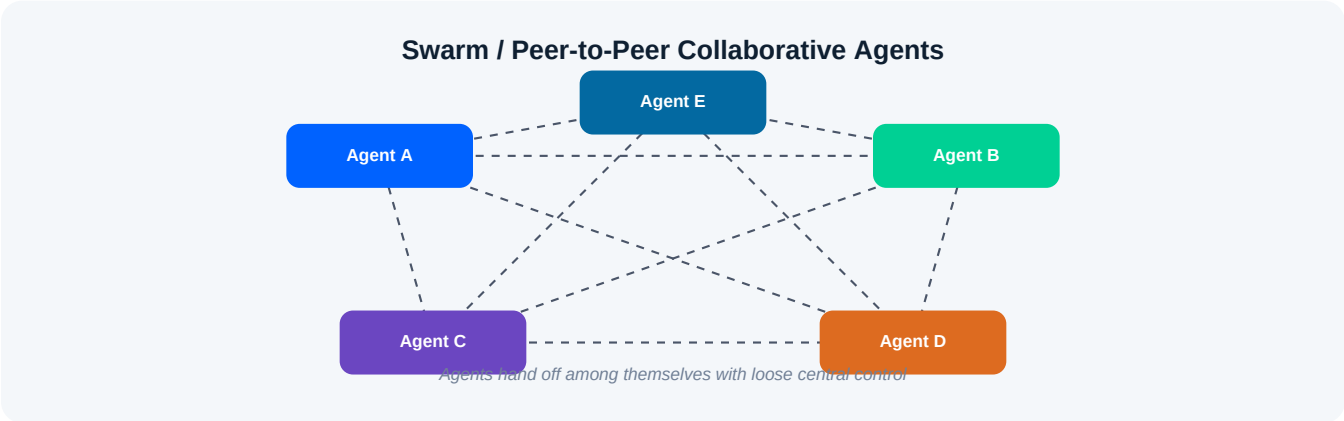
When to use: Simulations, training data generation, creative collaboration, and tasks where specialized perspectives add value. **Best practices:** Define roles with concrete expertise and responsibilities, not vague descriptions. Include constraints in role definitions ("you can only approve changes to the API layer").

SOP / Assembly-Line Multi-Agent

A role-based system that encodes Standard Operating Procedures so specialist agents collaborate like a structured team. MetaGPT (Hong et al.) is the main reference: it assigns agents roles like Product Manager, Architect, Engineer, and QA, then has them follow defined workflows (requirements → design → implement → test). The SOPs prevent the chaos that can occur in unstructured multi-agent systems by enforcing process discipline.

When to use: Software development workflows, content production pipelines, and any process that benefits from defined handoffs and quality gates between stages.

Swarm / Peer-to-Peer Collaborative Agents



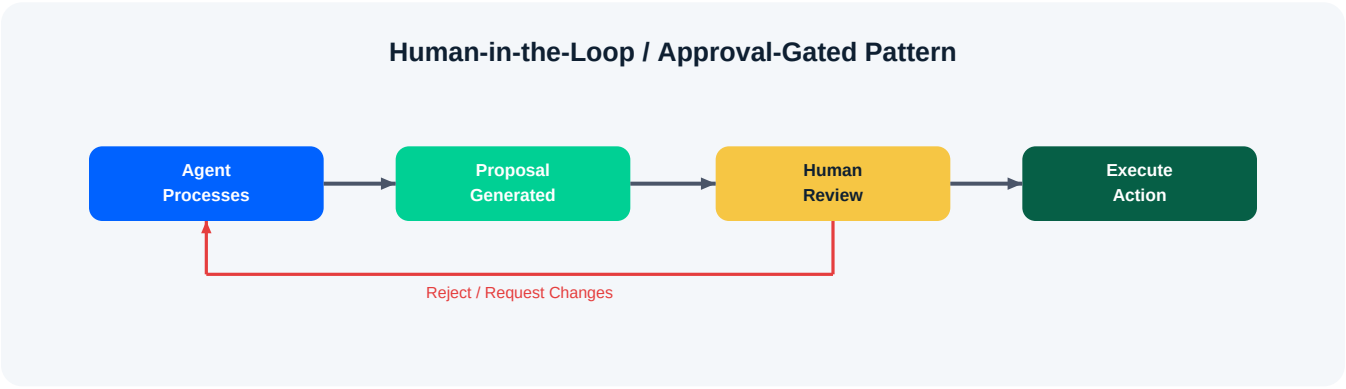
Agents can hand off among themselves with loose or no central control. Each agent has its own tools and specialization, and the system relies on peer-to-peer communication and emergent coordination rather than top-down orchestration. Google and LangGraph both expose swarm-style patterns. Swarms are powerful and creative—but also the most expensive and hardest to debug of all multi-agent patterns.

When to use: Open-ended exploration tasks, complex problem-solving where the optimal strategy is not known in advance, and research tasks that benefit from diverse, self-organizing behavior. **When not to use:** Tasks with predictable workflows (use orchestrator-workers instead), latency-sensitive applications, and cost-sensitive deployments. **Best practices:** Implement comprehensive logging—without it, debugging swarm behavior is nearly impossible. Set budget limits (tokens, time, cost). Start with 2–3 agents and scale up only as needed.

Enterprise Integration Patterns

Production agentic systems require patterns that address enterprise-specific concerns: human oversight, custom workflow composition, and interaction with real-world environments. These patterns are rarely used in isolation—they are layered on top of the patterns described in earlier tiers.

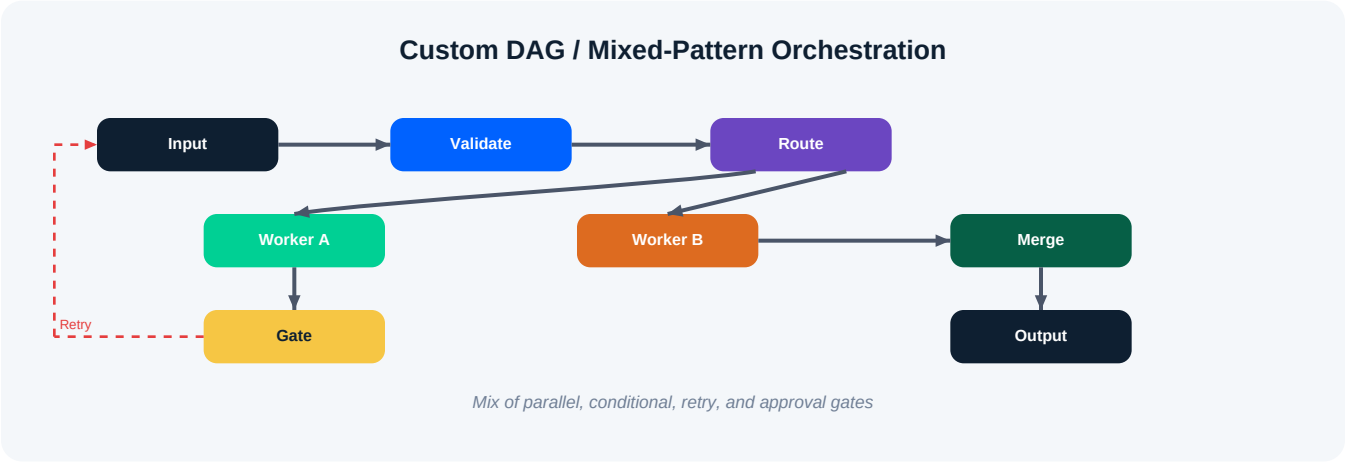
Human-in-the-Loop / Approval-Gated Agents



Critical actions pause for human validation before execution. This is essential in enterprise workflows, compliance-heavy settings, and high-risk operations where agent errors have real consequences—financial transactions, customer communications, infrastructure changes, and anything that cannot be easily reversed.

When to use: Any production system where agent actions have real-world consequences. This is not a "nice to have"—it is a requirement for responsible deployment. **Best practices:** Define clear approval thresholds (what requires human review vs. what can proceed automatically). Make the approval interface show the agent's reasoning, not just its proposed action. Implement timeouts—if no human responds within N minutes, default to safe behavior (reject or escalate). Log all approval decisions for audit.

Custom Graph / DAG / Mixed-Pattern Orchestration



A hand-built mix of parallel checks, conditional branches, retries, approval gates, and agent calls. This is where serious production systems end up once the simple templates no longer fit. Frameworks like LangGraph, CrewAI, and AutoGen provide the building blocks, but the architecture is custom to your use case.

When to use: Production systems that have outgrown single-pattern architectures, workflows with complex conditional logic, and systems that need to compose multiple patterns. **Best practices:** Document the DAG

visually—complex orchestrations become unmaintainable without diagrams. Implement comprehensive error handling at each node. Use idempotent operations wherever possible so that retries are safe. Monitor end-to-end latency and cost, not just individual node performance.

Browser / Computer-Use Agents

Agents that operate in web or GUI environments through browsing, clicking, scrolling, typing, and observation. WebGPT is an early precursor; newer implementations like Anthropic's computer use and OpenAI's Operator demonstrate agents that can navigate arbitrary web interfaces. This is a distinct pattern family because it requires fundamentally different tooling—screen capture, DOM parsing, mouse/keyboard control—and introduces unique challenges around state management and error recovery.

When to use: Automating tasks in systems that lack APIs, quality assurance testing, web scraping with complex interaction patterns, and legacy system integration. **Best practices:** Implement robust error recovery (pages load slowly, elements move, pop-ups appear). Take screenshots at each step for debugging. Set strict URL allowlists for security. Use structured selectors when possible rather than visual matching.

Choosing the Right Pattern: A Decision Framework

With 30+ patterns to choose from, selection can be overwhelming. Use the following decision criteria to narrow your options systematically. Start simple and add complexity only when you hit a concrete limitation.

1. **Start with Single-Agent**

   Can one agent with tools solve the task in a single context window? If yes, stop here. Most tasks can be handled by a well-prompted single agent with good tool definitions.

2. **Add ReAct for Multi-Step Reasoning**

   Does the agent need to adapt its approach based on intermediate results? Use ReAct. If the workflow is fixed, use prompt chaining instead.

3. **Add Routing for Diverse Inputs**

   Do you receive fundamentally different query types? Add a router to dispatch to specialists. This is cheaper and more reliable than one agent that tries to handle everything.

4. **Add Parallelization for Latency or Quality**

   Can the task be split into independent subtasks? Scatter-gather reduces latency. Running the same task multiple times (voting) improves quality on critical outputs.

5. **Add Evaluation Loops for Quality Control**

   Is output quality critical? Add an evaluator-optimizer loop. Start with self-refine (same model critiques its own output) and upgrade to separate evaluator models if needed.

6. **Add Memory for Continuity**

   Does the agent need context across sessions or must it learn from past interactions? Add short-term memory first, then long-term memory, then reflexion.

7. **Add Multi-Agent for Complex Collaboration**

   Is the task genuinely too complex for one agent? Use orchestrator-workers for known decompositions, group chat for creative/analytical tasks, and swarms only when you need emergent behavior.

8

Add Human-in-the-Loop for High-Stakes Actions

Can agent errors cause real harm? Add approval gates. This is a requirement, not an option, for production systems in regulated industries.

Production Best Practices for Agentic Systems

Regardless of which patterns you choose, the following principles apply to every production agentic system. These are the lessons learned from organizations that have moved agentic AI from prototype to production.

Observability Is Non-Negotiable

Every agent action, tool call, memory read/write, and inter-agent message must be logged with structured metadata. Without comprehensive observability, debugging multi-step agent behavior is effectively impossible. Implement trace IDs that follow a request through the entire agent pipeline. Build dashboards that show token consumption, latency, error rates, and cost by agent, pattern, and task type. Tools like **AgentWatch** by Iterate.ai (iterate.ai/applications/agentwatch) provide purpose-built AI observability and privacy policy enforcement for agentic systems.

Set Explicit Boundaries

Every agent needs a budget (tokens, time, cost, iterations). Without hard limits, agents can enter infinite loops, consume thousands of dollars in API calls, or run for hours on a task that should take minutes. Define maximum iterations for loops, maximum depth for recursive decomposition, and maximum total cost per task. Fail gracefully when limits are reached—return the best partial result, not an error.

Test Agent Behavior, Not Just Outputs

Traditional unit tests verify function outputs. Agent tests must also verify *behavior*: Did the agent use the right tools? Did it follow the expected reasoning pattern? Did it stay within its boundaries? Build evaluation suites that test agent trajectories (the sequence of actions taken), not just final answers. Include adversarial test cases that attempt to trigger prompt injection, tool misuse, and boundary violations.

Design for Graceful Degradation

External services fail. Models hallucinate. Tools return unexpected results. Design every pattern to handle failures at each stage. A scatter-gather pattern should produce useful output even if one worker fails. An agentic RAG system should answer from its existing context if retrieval fails. A human-in-the-loop system should default to safe behavior if no human responds.

Start Simple, Measure, Then Evolve

The most common mistake is over-engineering. Start with the simplest pattern that could work (usually single-agent or prompt chaining). Measure its performance against your specific requirements. Identify the concrete limitations (not theoretical ones). Then evolve to a more complex pattern to address those specific limitations. This iterative approach is faster, cheaper, and produces better architectures than trying to design the "perfect" multi-agent system upfront.

References and Further Reading

Official Documentation

- Anthropic — Building Effective AI Agents
- Google Cloud — Agent Design Patterns for Agentic AI Systems
- Microsoft Azure — AI Agent Orchestration Patterns
- AWS Prescriptive Guidance — Agentic AI Patterns and Workflows on AWS
- Hugging Face — Design Patterns for Building Agentic Workflows

Foundational Research

- Yao et al. — ReAct: Synergizing Reasoning and Acting in Language Models
- Shinn et al. — Reflexion: Language Agents with Verbal Reinforcement Learning
- Yao et al. — Tree of Thoughts: Deliberate Problem Solving with LLMs
- Kim et al. — LLMCompiler: An LLM Compiler for Parallel Function Calling
- Xu et al. — ReWOO: Decoupling Reasoning from Observations for Efficient Augmented LMs
- Li et al. — CAMEL: Communicative Agents for Mind Exploration of LLMs
- Hong et al. — MetaGPT: Meta Programming for Multi-Agent Collaboration
- Park et al. — Generative Agents: Interactive Simulacra of Human Behavior
- Wang et al. — Voyager: An Open-Ended Embodied Agent with LLMs
- Zhou et al. — Language Agent Tree Search Unifies Reasoning, Acting, and Planning
- Gao et al. — Retrieval-Augmented Generation for LLMs: A Survey

Surveys and Recent Work

- Wang et al. — A Survey on Large Language Model Based Autonomous Agents
- Guo et al. — A Survey on LLM-Based Multi-Agent Systems
- Masterman et al. — The Landscape of Emerging AI Agent Architectures: A Survey
- Xi et al. — The Rise and Potential of LLM-Based Agents: A Survey
- Durante et al. — Agent AI: Surveying the Horizons of Multimodal Interaction
- Anthropic — Claude Agent SDK and Multi-Agent Patterns
- OpenAI — Agent SDK: Building Production Agents

Securing Agentic AI: From Design to Production

Security in agentic systems is not an afterthought—it must be designed in from the first architectural decision. Agents that autonomously plan, use tools, access data, and take actions introduce attack surfaces that do not exist in traditional software. The more autonomous the agent, the more critical the security architecture.

Prompt Injection and Input Validation

Every agent that accepts user input or processes external data is vulnerable to prompt injection. Agentic systems are especially at risk because injected instructions can cause the agent to misuse tools, exfiltrate data, or bypass safety controls. **Best practices:** Validate and sanitize all inputs before they reach the LLM. Use separate system prompts that the user cannot override. Implement input/output guardrails that flag suspicious patterns. Never trust content retrieved from external sources—treat RAG results as untrusted input.

Tool and API Security

Agents with tool access can read files, execute code, make API calls, and modify databases. Each tool is a potential attack vector. **Best practices:** Apply the principle of least privilege—give each agent only the tools it needs. Implement authentication and authorization for every tool call. Use allowlists for file paths, URLs, and API endpoints. Sandbox code execution environments. Log every tool invocation with full parameters for audit.

Data Privacy and Access Control

Agentic RAG systems and memory-enabled agents handle sensitive enterprise data. Without proper controls, agents can leak confidential information across conversations, users, or tenants. **Best practices:** Implement row-level and document-level access controls in retrieval systems. Ensure agents cannot access data beyond the requesting user's permissions. Encrypt data at rest and in transit. Implement data retention policies for agent memory stores. Audit what data the agent accesses, stores, and returns.

Multi-Agent Trust Boundaries

In multi-agent systems, each agent is a potential point of compromise. A compromised or manipulated agent can propagate bad instructions to others. **Best practices:** Define explicit trust boundaries between agents. Validate messages passed between agents—do not assume that inter-agent communication is safe. Implement output validation at each agent boundary. Use separate credentials and permissions per agent role. Monitor for anomalous agent behavior patterns (unusual tool calls, excessive data access, unexpected inter-agent messages).

Human Oversight as a Security Control

Human-in-the-loop is not just a design pattern—it is a critical security control. For high-stakes actions (financial transactions, data deletion, external communications), human approval gates prevent both agent errors and security breaches from causing irreversible harm. **Best practices:** Classify agent actions by risk level and require human approval above a threshold. Show the agent's full reasoning chain in the approval interface, not just the proposed action. Implement dead-man switches—if the human oversight system fails, the agent should default to safe behavior, not proceed autonomously.

Observability and Incident Response

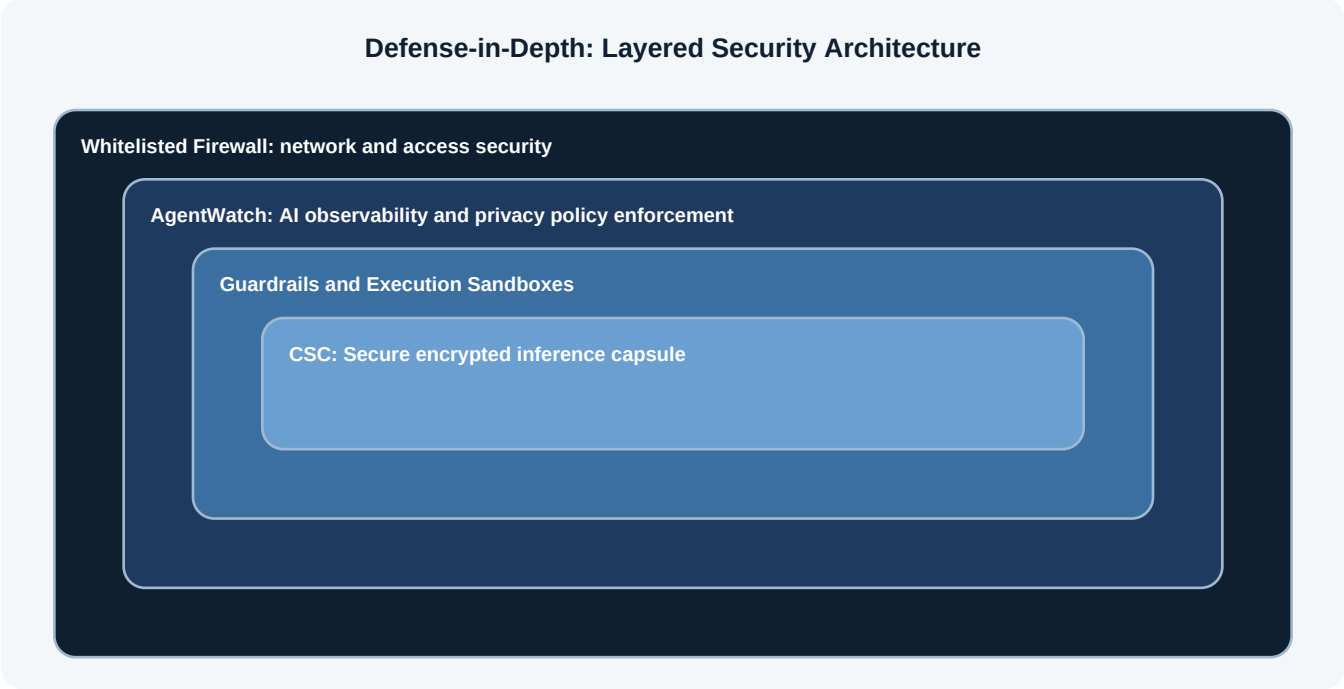
You cannot secure what you cannot see. Comprehensive observability is the foundation of agentic security. **Best practices:** Log every agent action, tool call, and inter-agent message with structured metadata and trace IDs. Build alerting for anomalous patterns: unexpected tool usage, unusual data access volumes, prompt injection attempts, and boundary violations. Maintain audit trails that can reconstruct the full chain of agent decisions for any incident. Test your incident response playbook specifically for agent-related scenarios.

Open-Source Agent Frameworks: Security Risks

Open-source agent frameworks like OpenClaw and similar tools lower the barrier to building agentic systems—but they also introduce significant security risks. Many open-source agent frameworks ship with permissive defaults: unrestricted tool access, no sandboxing, no input validation, and no output filtering. Deploying these frameworks without hardening them is equivalent to running an unsecured server on the public internet. **Best practices:** Audit every open-source agent framework before deployment. Disable default tools you do not need. Add input/output guardrails. Run agent execution in sandboxed environments with restricted network access. Never grant agents access to production databases or systems without explicit authorization layers.

Defense-in-Depth: Layered Security Architecture

The most robust approach to securing agentic AI is **defense-in-depth**—multiple concentric security layers, each providing independent protection. From the outside in: a **whitelisted firewall** controls network and access security, allowing only approved endpoints. **AgentWatch** (iterate.ai/applications/agentwatch) provides AI observability and privacy policy enforcement, monitoring all agent activity. **Guardrails and execution sandboxes** constrain what agents can do at runtime—restricting file access, network calls, and system commands. At the core, a **Confidential Security Capsule (CSC)** runs inference inside a secure encrypted session—even when one session is breached, the blast radius is contained and other sessions remain protected. CSC is a software-based protection layer, not hardware-dependent, making it deployable across any cloud or on-premise environment. This layered approach ensures that no single point of failure can compromise the entire agentic system.



Defense-in-depth: concentric security layers for agentic AI systems.

Agentic Patterns in Practice: Real-World Systems

The patterns described in this paper are not theoretical—they power the most capable AI systems shipping today. Understanding how production systems combine these patterns helps technical leaders make informed architecture decisions.

Claude Code (Anthropic)

Anthropic's Claude Code is a coding agent built on the **ReAct pattern** with extensive tool calling. It operates as a single agent with a large toolset (file read/write, shell execution, search, browser) and uses **plan-and-execute** for complex multi-file tasks. Claude Code also demonstrates **memory patterns**—it maintains project context across sessions via CLAUDE.md files and a persistent memory system. For large tasks, it spawns **orchestrator-worker** subagents that operate in parallel on isolated worktrees.

Manus (Manus AI)

Manus uses a **multi-agent orchestrator** pattern with **browser/computer-use agents**. A planning agent decomposes user goals into tasks, then delegates to specialist agents that can browse the web, write code, manipulate files, and interact with APIs. Manus combines **plan-and-execute** with **tool marketplace** patterns—the orchestrator selects which specialist to invoke based on the current subtask. Its architecture demonstrates how production systems layer multiple patterns together.

OpenAI Operator and Agent SDK

OpenAI's Operator is a **browser/computer-use agent** that navigates web interfaces autonomously. The OpenAI Agent SDK enables developers to build agents using **handoff patterns**, **routing**, and **tool calling** with built-in guardrails. The SDK's design explicitly supports multi-agent handoff—agents can transfer control to specialists mid-conversation—and includes tracing for observability, reflecting the production best practices in this paper.

Devin and SWE-Agent (Cognition / Princeton)

Devin and SWE-Agent are autonomous coding agents that combine **ReAct loops** with **long-term memory** and **skill library** patterns. They operate in sandboxed environments, using shell tools and file manipulation to solve software engineering tasks. Both systems demonstrate the importance of **reflexion**—learning from failed attempts to improve subsequent approaches—and **evaluation loops** where test results feed back into the agent's reasoning.

Microsoft Magentic-One and AutoGen

Magentic-One implements the **manager-led orchestration** pattern with a task ledger that tracks progress across multiple specialist agents (web surfer, coder, file handler). AutoGen provides a framework for **group chat / multi-agent debate** patterns, enabling multiple agents to collaborate in shared conversation threads with configurable turn-taking and termination conditions.

From Patterns to Production with Generate and Interplay

Understanding agentic design patterns is essential—but translating that understanding into production systems is where the real work begins. Building agentic architectures from scratch means solving orchestration, state management, tool integration, observability, and runtime efficiency for every pattern you deploy. Most engineering teams find that this infrastructure work consumes more effort than the agent logic itself.

Generate and **Interplay**, by Iterate.ai, are platforms designed to make building and operating agentic systems straightforward. **Generate** (iterate.ai/platform/generate) enables rapid prototyping and deployment of AI-powered applications, providing the foundation for experimenting with different agentic patterns quickly. **Interplay** (iterate.ai/platform/interplay) provides a visual composition environment where the patterns described in this paper—from single-agent tool callers to multi-agent orchestrations—can be assembled, tested, and deployed without building the underlying infrastructure from scratch.

Together, Generate and Interplay cover the full lifecycle: Generate for rapid experimentation and Interplay for production-grade orchestration. Interplay's drag-and-connect interface maps directly to the architectural diagrams in this paper: agents, tools, routers, evaluators, memory stores, and human approval gates are all first-class building blocks. The runtime engine handles automatic scaling, token budget enforcement, comprehensive tracing and observability, multi-model routing, failover between providers, and cost tracking at the agent and task level.

> *"The goal of this paper is to help technical leaders build a mental model of agentic design patterns so they can make informed architecture decisions. Generate and Interplay exist to make those decisions easy to implement and operate at enterprise scale.*
> Iterate.ai | iterate.ai/platform/generate | iterate.ai/platform/interplay

Iterate.ai is trusted by leading Fortune 500 companies for its enterprise AI solutions. Technology partners include Intel, NVIDIA, Qualcomm, Dell, IBM, NetApp, and others. Learn more at iterate.ai/platform/generate and iterate.ai/platform/interplay.