Iterate.ai × VeroMesh.ai

WHITE PAPER · 2026

A PATTERN LIBRARY FOR TECHNICAL LEADERS

Agentic Design Patterns for the Enterprise.

From single agents to production swarms — a guide to 30+ agentic AI architecture patterns, organized from foundational building blocks to enterprise-scale multi-agent systems.

DESIGN PATTERNS · MULTI-AGENT · AGENTIC RAG · ORCHESTRATION · ENTERPRISE AI

AUTHORS

Brian Sathianathan · Sibito Morley

Cofounder/CTO, Iterate.ai ·
Cofounder/President, VeroMesh.ai

SCOPE

30+ Patterns

6 tiers · foundation to swarm

FOR

Technical Leaders

CTO · CAIO · CIO · CDO · VP Eng

ABOUT THIS PAPER

The difference between a demo and a production agent is not the model. It is the orchestration pattern.

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.

This paper catalogs 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.

WHO THIS PAPER IS 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, Chief Digital Officers driving transformation, and VPs of Engineering making build-vs-buy decisions. We assume familiarity with LLMs — but not with agentic system design.

GROUNDED IN

Published research and official documentation from Anthropic, Google Cloud, Microsoft Azure, AWS, and Hugging Face — plus the academic papers that originated each pattern.

PUBLISHED BY

Iterate.ai & **VeroMesh.ai** — Generate · Interplay.

EXECUTIVE SUMMARY

Which patterns actually work — and when to use one agent versus ten.

This white paper answers the questions every technical leader is asking: which patterns actually work, when you should use one agent versus a swarm, and how you maintain control, observability, and security as autonomy increases. Every pattern is illustrated with an architecture diagram so 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 right pattern turns an expensive, unreliable prototype into a system that your organization can trust, observe, and scale.**
>
> *Anthropic, Building Effective AI Agents, 2025.*

HOW TO READ THIS PAPER

Each pattern section follows a consistent structure: a diagram of the architecture, a plain-language explanation of how it works, when to use it, when not to use it, and best practices drawn from production deployments. Patterns are grouped by tier, and each tier builds on the one before it.

A NOTE ON ICONOGRAPHY

Architecture diagrams in this paper are reconstructions of the source figures, redrawn in the Iterate visual system. References are provided at the end so engineering teams can dive deeper into each pattern's originating work.

CONTENTS

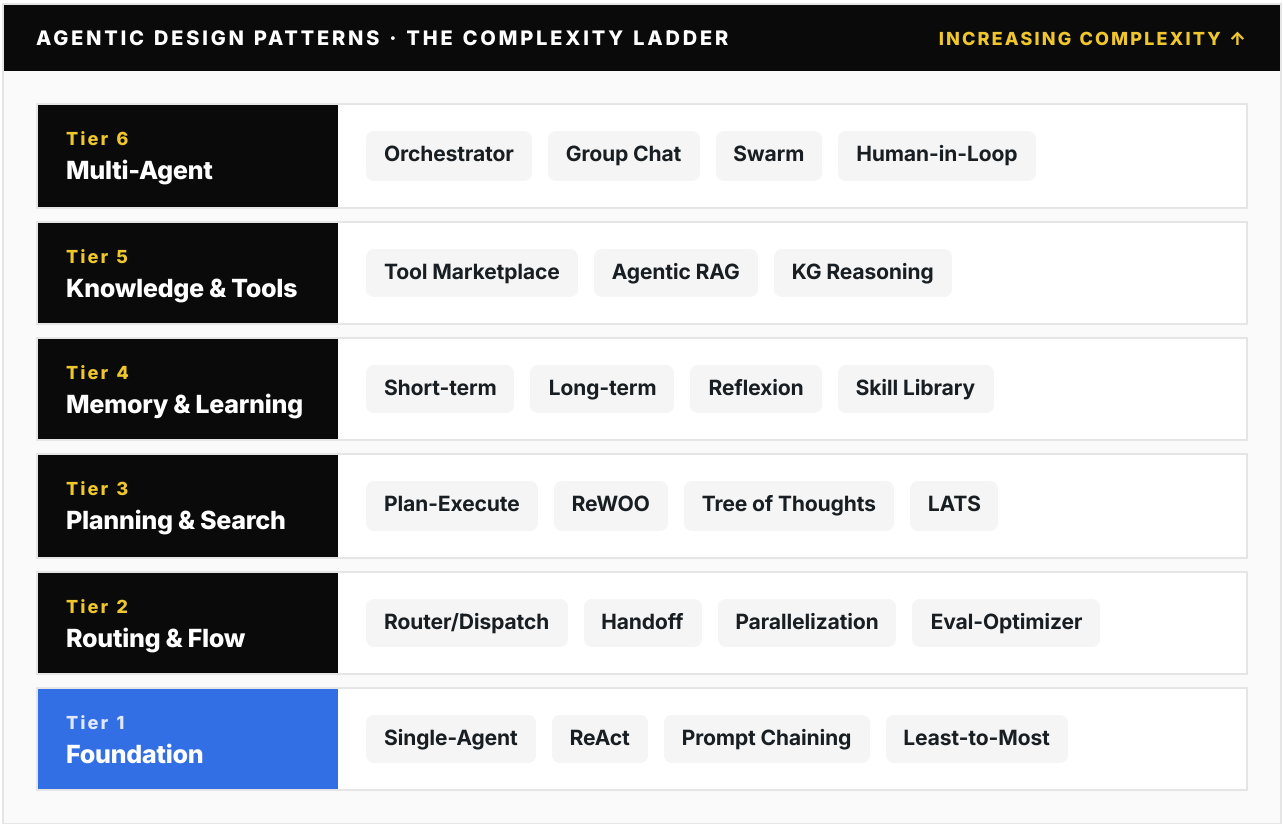
The complexity ladder.

Start at the bottom. Move up only when the current tier cannot solve your problem. The most common mistake in agentic adoption is reaching for a swarm before exhausting what a well-designed single agent can do.

00 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.



The six tiers of agentic design patterns, from foundation to enterprise multi-agent. Start at the bottom; move up only when the current tier cannot solve your problem.

HOW TO READ THIS PAPER

Each pattern is presented with a diagram, a plain-language explanation, when to use it, when not to, and best practices from production deployments. Patterns are grouped by tier; each tier builds on the concepts in the one below.

BEFORE THE FIRST TIER

There is a prerequisite layer the tiers depend on. We call it **Tier 0: Governance Infrastructure** — the part most enterprise AI programs get wrong before they even begin. It is addressed on the facing page.

T0

TIER 0 · THE GOVERNANCE PREREQUISITE

Pattern selection is the second decision, not the first.

Architectures fail not because the pattern was wrong, but because the data it reasoned over was ungoverned, the outputs were unobservable, and the deployment environment could not satisfy the legal requirements. The first decision is whether you have the governance infrastructure the pattern depends upon.

FIVE QUESTIONS TO ANSWER "YES" BEFORE SELECTING A PATTERN

1 · Data Governance. Are the data sources your agents reason over modeled, semantically grounded, and reliable?

2 · Governed Data Fabric. Can agents access the right data at inference time without moving it across prohibited boundaries?

3 · Observability. Can you trace what your agents did, what data they accessed, what decisions they made, and why?

4 · AI Gateway / Control Plane. Is there a single layer through which all model interactions flow, enforcing cost, DLP, and audit structurally?

5 · Deployment Sovereignty. Does your target environment permit the architecture you are selecting? Not all patterns are permissible in all environments.

TIER 0 CAPABILITIES · WHY EACH MATTERS FOR AGENTIC PATTERNS

TIER 0 CAPABILITY	WHY IT MATTERS ACROSS THE TIERS
Governed Data (ontologies, canonical models)	Reduces hallucination across all tiers; enables high-confidence RAG.
Federated Data Access	Enables patterns that require cross-system retrieval without data movement.
Observability Infrastructure	Makes every tier's behavior auditable and debuggable.
AI Gateway / Control Plane	Enforces cost, compliance, and DLP structurally across all model calls.
Sovereign Deployment Surface	Determines which patterns are permissible in the target environment.

The compounding argument. Each Tier 0 layer makes the pattern tiers stronger. When these capabilities are in place, the patterns compound powerfully. When they are not, even the simplest patterns — a single ReAct loop with two tools — become governance liabilities at scale. *Read this paper twice: once to understand the landscape, once to audit your Tier 0 against each pattern you want to adopt.*

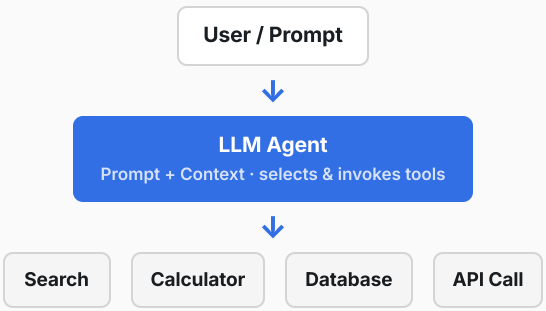
01 TIER 1 · FOUNDATION PATTERNS

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 BASELINE

SINGLE-AGENT TOOL CALLER



One LLM, one prompt, one toolset, one shared context. The agent decides which tools to call, executes them, and returns a response.

The simplest agentic pattern — the one you should use until you have a concrete reason not to. Google Cloud's single-agent guidance and LangChain's core agent docs both recommend starting here.

WHEN TO USE

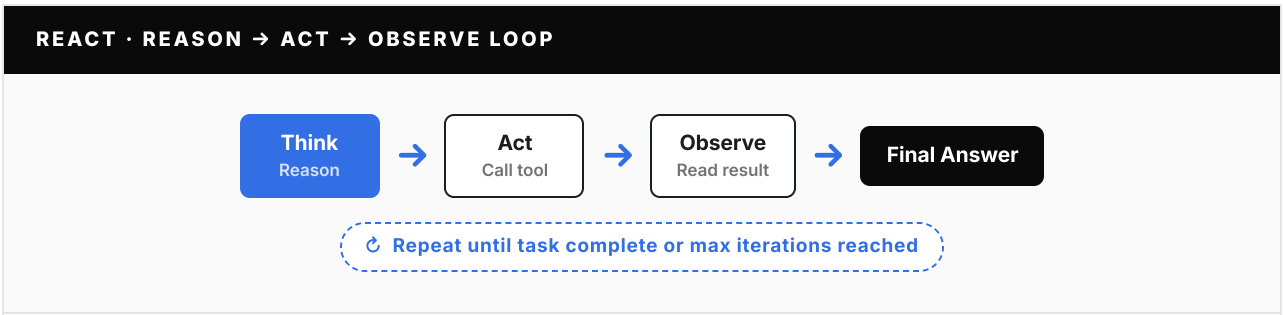
Customer support with tool access, code generation, data lookups, API integrations — any task where a single context window is sufficient.

WHEN NOT TO USE

Tasks that exceed the context window, require specialized expertise across many domains at once, or need parallel execution for latency.

Best practices: keep tool descriptions precise — selection is only as good as the definitions. Limit the toolset to 10–15 tools. Use structured output (JSON mode) for tool calls. Implement timeout and retry logic per tool.

ReAct — Reason + Act THE CANONICAL LOOP



The agent reasons, acts, observes, and loops — the closest thing to a "singleton" of agentic design, because so many other patterns extend it (Yao et al.).

WHEN TO USE

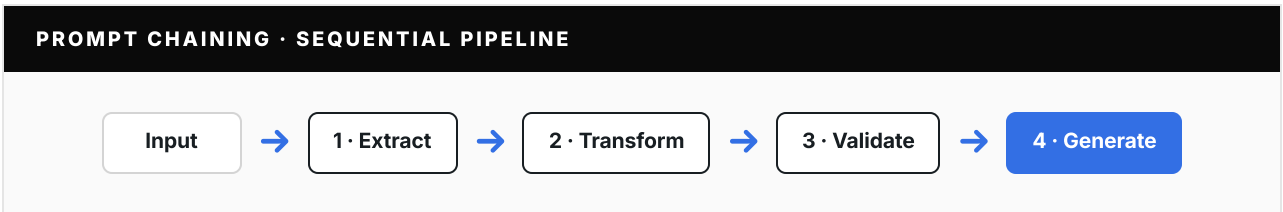
Multi-step research, complex QA with tool use, debugging workflows — any task where the agent must adapt its plan to intermediate results.

WHEN NOT TO USE

Simple one-shot tasks, fixed known workflows (use prompt chaining), or latency-sensitive apps — each loop adds a full LLM call.

Best practices: cap iterations (5–10) to prevent infinite loops. Include an "I have enough information" escape hatch. Log every thought-action-observation triple. Monitor token consumption — each iteration includes the full history.

Prompt Chaining / Sequential Pipeline FIXED ORDER



Each step's output becomes the next step's input. The order is fixed at design time, not runtime — predictable cost, independently validatable steps.

When to use: document pipelines (extract → classify → summarize), data transformation, content generation with review stages — any multi-step process with a known order. **Best practices:** add validation gates between steps; keep each step to a single transformation; use cheaper models for intermediate steps and reserve the most capable model for the step that needs the most judgment.

Least-to-Most Decomposition PLANNING PRIMITIVE

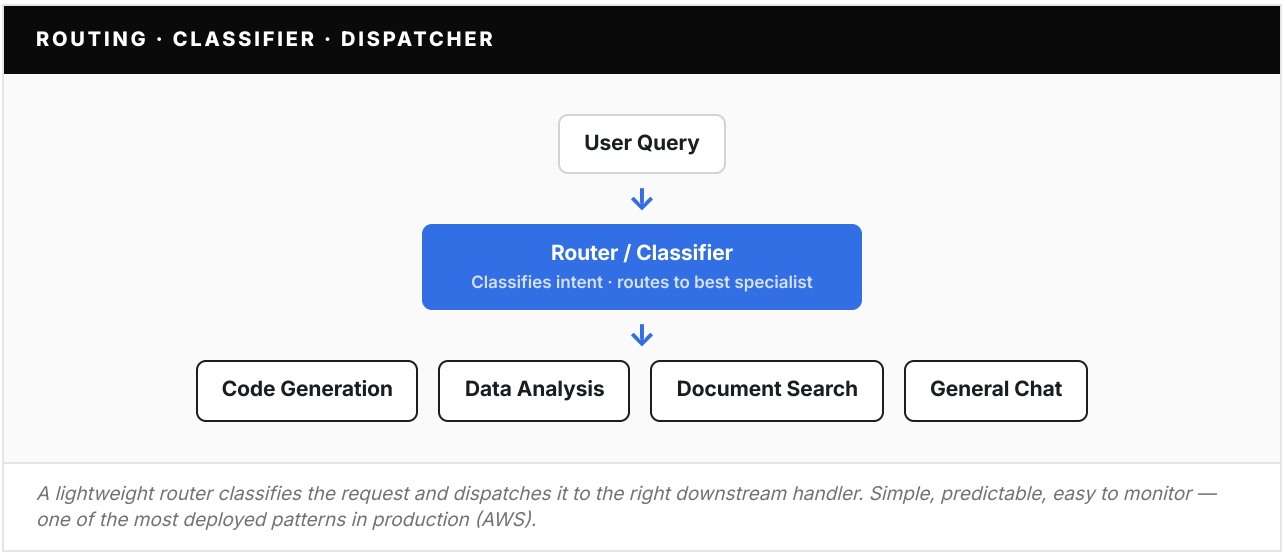
Break a hard problem into progressively simpler subproblems and solve them in sequence, using earlier solutions as context for later ones. LLMs struggle with complex problems presented all at once but perform well on the decomposed subproblems. **When to use:** complex mathematical reasoning, multi-step coding, building up from simpler components. **Best practice:** let the LLM itself generate the decomposition — it often identifies subproblems a human designer would miss.

02 TIER 2 · ROUTING & FLOW CONTROL

Once you outgrow a single agent, the first decision is how to route work.

Routing patterns add a classification or dispatch layer that directs tasks to specialized agents, prompts, or tools — maintaining a single point of coordination while enabling specialization.

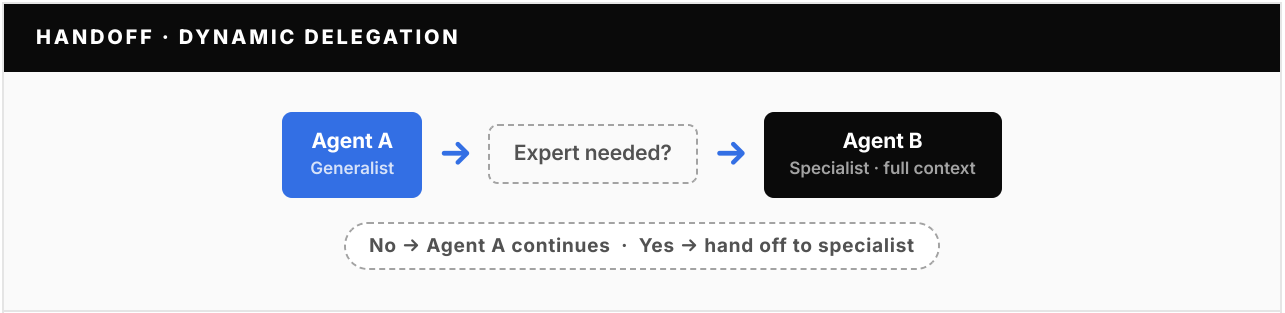
Routing / Classifier / Dispatcher MOST-DEPLOYED



A lightweight router classifies the request and dispatches it to the right downstream handler. Simple, predictable, easy to monitor — one of the most deployed patterns in production (AWS).

When to use: customer-facing systems with diverse query types, multi-domain apps, tiered support. **Best practices:** use a fast, cheap model for the router; define clear, non-overlapping categories; add a "fallback" category; monitor classification accuracy and retrain as the query distribution shifts.

Handoff / Triage / Dynamic Delegation MID-FLIGHT TRANSFER

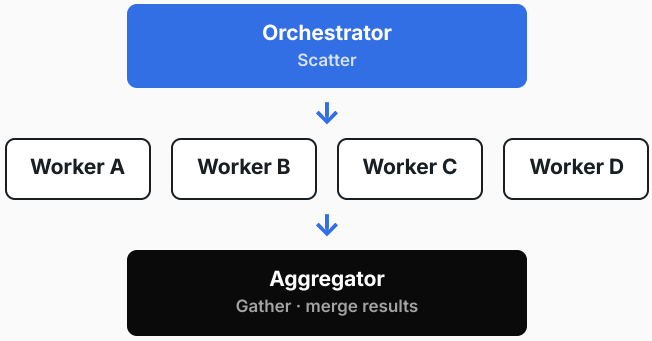


Unlike static routing, the active agent decides mid-flight to transfer control to a better specialist, passing full context (Azure handoff orchestration).

When to use: complex support workflows where initial triage cannot fully classify the task; multi-turn conversations that evolve beyond the first agent's scope; systems where specialists are expensive. **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 CONCURRENT WORKERS

PARALLELIZATION · SCATTER-GATHER



Fan the task out to multiple workers simultaneously, then aggregate. Two sub-patterns: sectioning (split into independent subtasks) and voting (run the same task N times, take the best/consensus).

WHEN TO USE

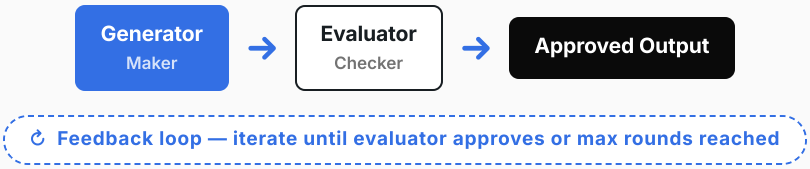
Independent subtasks (different document sections), latency-sensitive apps, quality-critical tasks where multiple perspectives help, batch processing.

BEST PRACTICES

Decide the aggregation strategy first (majority vote, best-of-N, merge, synthesize). Handle partial failures. Set per-worker timeouts to stop stragglers.

Evaluator-Optimizer / Maker-Checker QUALITY CONTROL

EVALUATOR-OPTIMIZER · MAKER-CHECKER



One model generates, another critiques, the first revises. Adds quality control without a full multi-agent system. The Self-Refine variant uses one model for both — simpler, sometimes less effective.

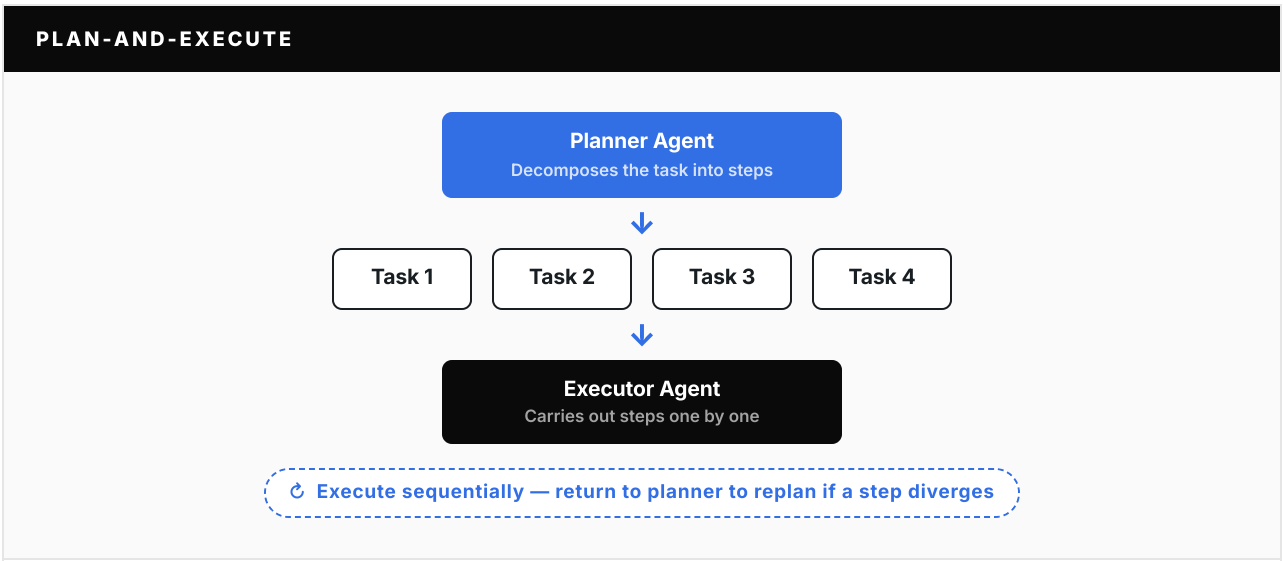
When to use: code generation (generate then review), content creation (draft then edit), translation (translate then back-translate) — any task that improves with iteration. **Best practices:** use a different model or temperature for the evaluator to avoid self-reinforcing errors; define concrete evaluation criteria, not "is this good?"; cap iterations at 2–3 rounds — diminishing returns set in quickly.

03 TIER 3 · PLANNING & SEARCH

Separate the “what to do” from the “how to do it.”

Instead of reasoning and acting in a tight loop, planning patterns first build a plan and then execute it — trading some flexibility for better cost, speed, and predictability. Search patterns extend planning by exploring multiple candidate solutions at once.

Plan-and-Execute PLAN ONCE, RUN CHEAP



A planner decomposes the task; an executor carries the steps out. The key advantage over ReAct: the planner can use a capable (expensive) model while the executor uses a cheaper one.

WHEN TO USE

Complex multi-step tasks where cost control matters, plans that can be reviewed by a human before execution, and workflows where predictability beats adaptability.

BEST PRACTICES

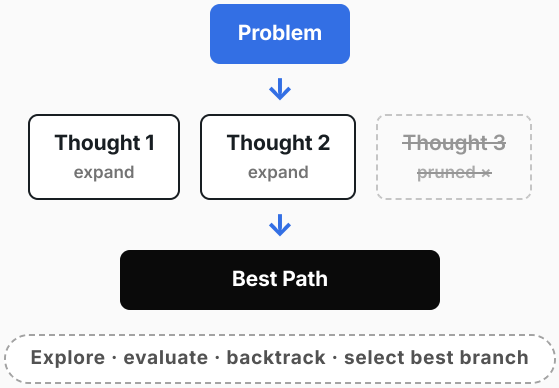
Include a replan mechanism. Make plans concrete and actionable (“search for X” not “gather info”). Store the plan as structured JSON/YAML so it can be inspected.

ReWOO — Reasoning Without Observation 5–10× FEWER TOKENS

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 — avoiding the cost of sending full history at each step. **This can reduce token usage by 5–10×** versus ReAct on multi-tool tasks. **When to use:** tasks with many tool calls where token cost matters, batch processing, and tasks where tool calls are independent. **When not to use:** tasks where later tool calls depend on earlier results.

Tree of Thoughts (ToT) EXPLORE & BACKTRACK

TREE OF THOUGHTS



Rather than a single chain, the agent branches over multiple candidate reasoning paths, evaluates each, prunes the unpromising ones, and continues on the most promising — like a human solving a puzzle.

WHEN TO USE

Large solution spaces where the first approach may not be optimal — puzzle solving, strategic planning, creative tasks, mathematical proofs, game playing.

WHEN NOT TO USE

Straightforward tasks where the first answer is usually right — ToT adds significant latency and cost. Prune aggressively: 3–5 branches per level, not dozens.

Language Agent Tree Search (LATS) MCTS

LATS extends Tree of Thoughts by combining reasoning, acting, and planning with Monte Carlo Tree Search. It uses environment feedback to guide search — using the LLM as both the policy (which action to take) and the value function (evaluating state quality). **When to use:** complex decision-making with environment interaction, autonomous coding agents exploring multiple implementation strategies, and research tasks where systematic exploration beats greedy approaches.

Graph of Thoughts / Diagram of Thought NON-LINEAR

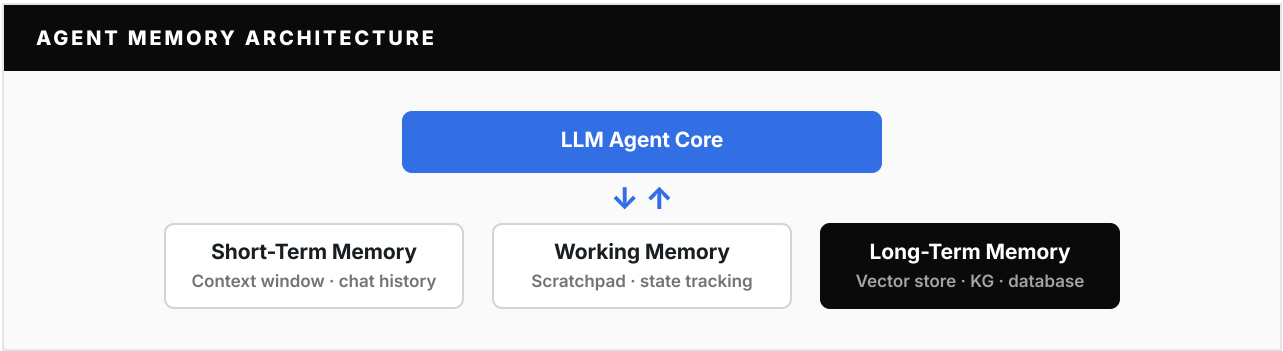
These generalize chain and tree reasoning into richer graph structures where intermediate results can be combined, merged, and refined across multiple paths. They matter when the task is non-linear and multiple intermediate dependencies must be tracked — for example, synthesizing information from multiple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

04 TIER 4 · MEMORY & LEARNING

An agent without memory is stateless.

It cannot learn from past interactions, maintain context across sessions, or improve over time. Memory patterns let agents accumulate knowledge, reflect on performance, and build reusable capabilities — the tier where agents start to feel genuinely intelligent rather than merely reactive.

Agent Memory Architecture THREE LAYERS



Three-layer memory architecture: short-term keeps current state coherent, working memory tracks the active task, and long-term persists facts and strategies across sessions.

Short-Term Memory

Thread or session memory that keeps current state coherent within one run — the minimum viable layer for nontrivial agents. In practice it is the conversation history plus scratchpad state. The challenge: context windows are finite, so agents must manage what stays and what gets summarized.

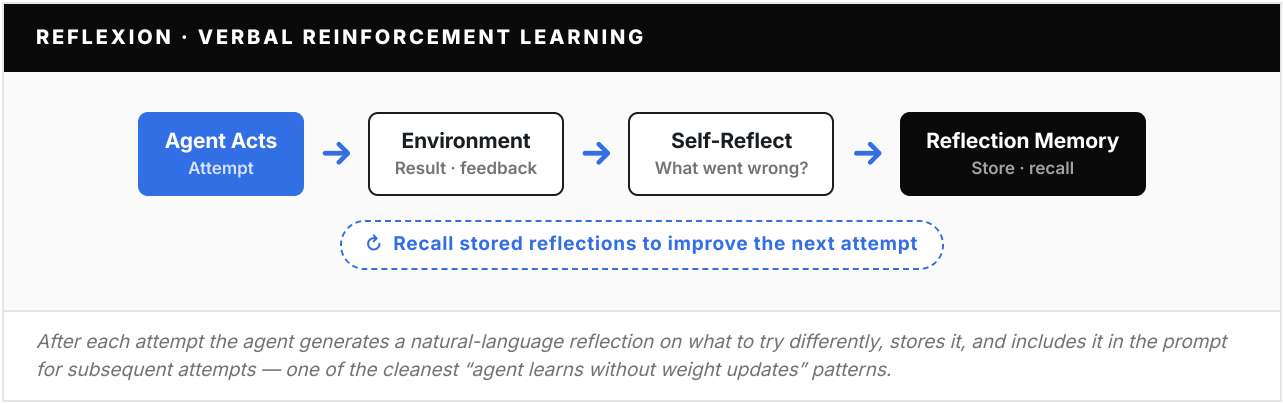
Best practices: implement sliding-window summarization for long conversations. Use structured state objects, not raw history. Separate "facts learned" from "actions taken."

Long-Term Memory

Persistent cross-session memory for user facts, learned strategies, preferences, or past outcomes — what enables agents to improve and personalize. Implementations range from key-value stores to vector databases and knowledge graphs.

Best practices: define a clear memory schema (what is stored, indexed, when it expires). Use semantic search for retrieval. Implement memory consolidation — periodically summarize and merge to prevent unbounded growth.

Reflexion / Verbal Reinforcement LEARNS WITHOUT WEIGHT UPDATES



After each attempt the agent generates a natural-language reflection on what to try differently, stores it, and includes it in the prompt for subsequent attempts — 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), any task where the agent must 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"), not vague. Limit stored reflections to the 5–10 most recent to prevent context bloat.

Skill Library / Lifelong-Learning Agent EXECUTABLE SKILLS

Instead of only storing text memories, the agent accumulates reusable executable skills — verified code snippets, procedures, or tool-use sequences it can invoke later. Voyager (Wang et al.) is the best-known reference. In enterprise settings this enables agents that get measurably better over time, building institutional knowledge in executable form. **When to use:** long-running agents with recurring task patterns, domain-specific automation. **Best practices:** version skills and track success rates; include preconditions and postconditions; implement skill composition.

Generative-Agent Architecture COGNITIVELY INSPIRED

A cognitively inspired design where observation, memory, reflection, and planning work together. Introduced by Park et al. for simulating believable human behavior, it applies to persistent assistants and any system that must maintain a coherent identity over extended periods. The architecture combines a memory stream (all observations), a retrieval mechanism (what is relevant now), reflection (higher-level insights), and planning (what to do next).

05 TIER 5 · KNOWLEDGE & TOOLS

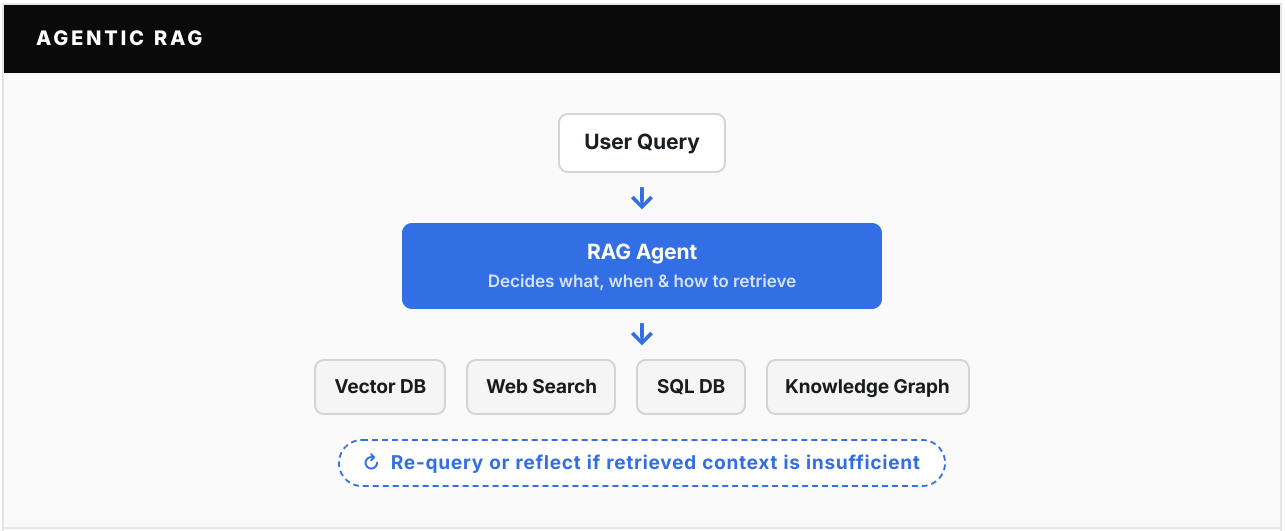
Where agents become genuinely useful for enterprise tasks.

Rather than relying solely on the LLM's parametric knowledge, these agents actively retrieve, query, and reason over external data sources and tool libraries — including proprietary data and systems.

Tool-Marketplace / Model-Router Agent HUGGINGGPT

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. **Best practices:** maintain a structured tool registry with capability descriptions, I/O schemas, and cost estimates; implement circuit breakers for unreliable tools; cache results when inputs are deterministic.

Agentic RAG RETRIEVAL BECOMES DYNAMIC



RAG becomes agentic when retrieval is no longer a fixed pipeline. The agent may issue multiple queries, combine sources, decide context is insufficient and reformulate, or decide no retrieval is needed at all.

When to use: complex research, multi-source synthesis, enterprise knowledge management — any application where a single retrieval pass is insufficient. 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 (don't just pass the user's question verbatim); implement relevance scoring; support multiple backends and let the agent choose; log retrieval decisions.

Knowledge-Graph Reasoning / Graph-Grounded Agents RELATIONAL SUBSTRATE

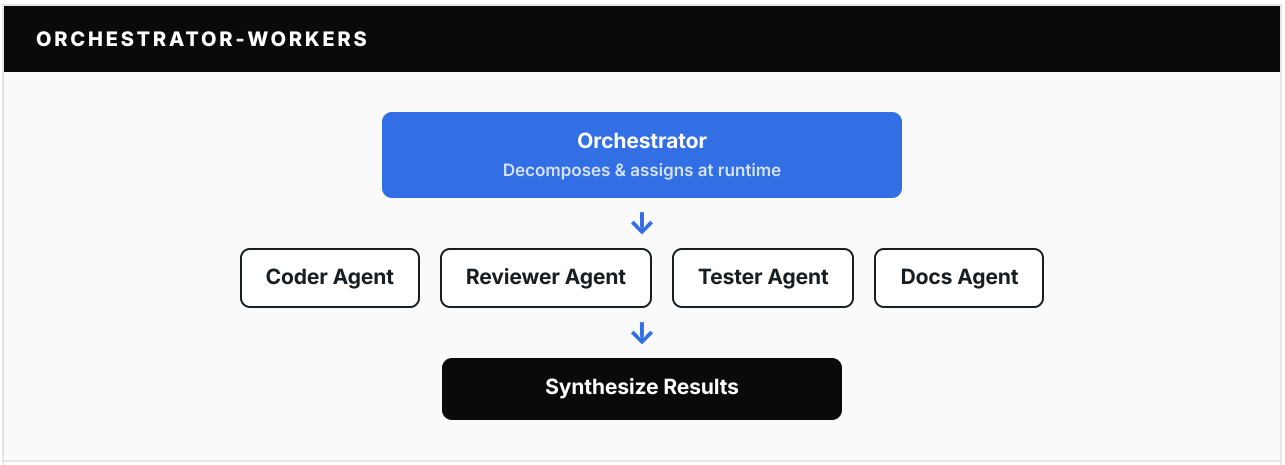
Agents that plan over a graph or use a knowledge graph as an explicit reasoning substrate (MindMap, Plan-on-Graph, KAG). They navigate structured relationships between entities — following edges, querying subgraphs, using topology to guide reasoning. Powerful for enterprise data with rich relational structure: supply chains, org hierarchies, regulatory frameworks. **Best practices:** keep the graph schema clean and documented; use the LLM to translate natural-language queries into graph queries rather than embedding raw graph data in prompts.

06 TIER 6 · MULTI-AGENT ORCHESTRATION

The most powerful — and most complex — architectures.

Multiple LLM-powered agents collaborate, debate, or compete to solve problems no single agent can handle. Use these only when you have exhausted simpler alternatives — the coordination overhead, debugging complexity, and cost are significant.

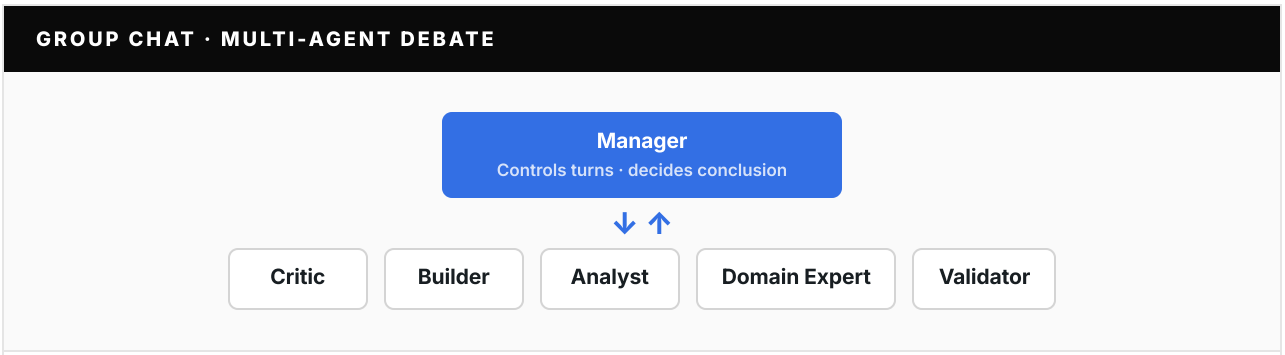
Orchestrator-Workers DYNAMIC DECOMPOSITION



A manager breaks down an open-ended task dynamically, assigns work to specialist subagents, then synthesizes outputs. Unlike prompt chaining, the decomposition is determined at runtime (Anthropic).

When to use: complex software tasks ("refactor this module"), research synthesis ("analyze this market from five angles"). **Best practices:** give the orchestrator a clear mandate and constraints; define the orchestrator↔worker interface; implement progress tracking; use the simplest worker that can do each subtask — not every worker needs to be a full ReAct agent.

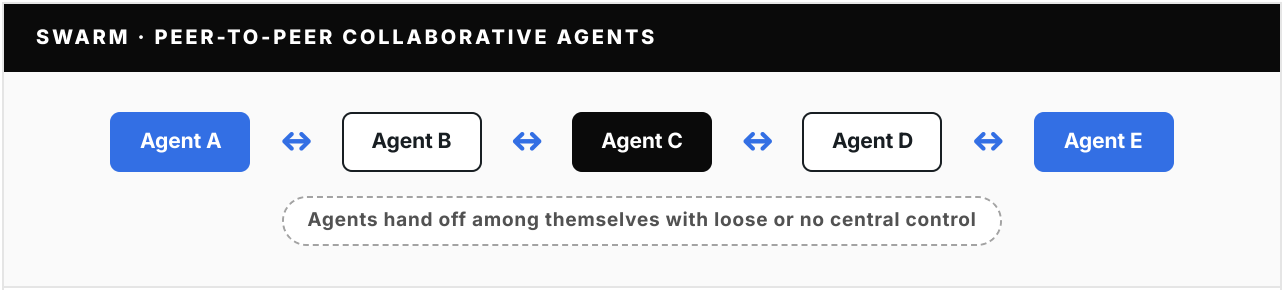
Group Chat / Multi-Agent Debate SHARED THREAD



Multiple agents communicate in a shared thread, each with a defined role. The manager decides who speaks next and when the discussion concludes — excels at brainstorming, validation, and red-teaming.

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 and set a maximum number of rounds.

Swarm / Peer-to-Peer Collaborative Agents EMERGENT COORDINATION



Each agent has its own tools and specialization; the system relies on peer-to-peer communication and emergent coordination rather than top-down orchestration. Powerful and creative — but the most expensive and hardest to debug.

WHEN TO USE

Open-ended exploration, complex problem-solving where the optimal strategy is unknown in advance, research that benefits from diverse self-organizing behavior.

WHEN NOT TO USE

Predictable workflows (use orchestrator-workers), latency-sensitive or cost-sensitive deployments. Start with 2–3 agents; log comprehensively; set budget limits.

Hierarchical Task Decomposition

A coordinator decomposes large objectives into nested subtasks and delegates down a tree of agents. Unlike flat orchestrator-workers, it supports arbitrary depth — coordinators delegate to sub-coordinators, which delegate to workers. Mirrors how large engineering orgs operate (Google Cloud, Microsoft).

Magentic / Manager-Led Orchestration

Microsoft's Magentic-One: 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 real systems. The manager actively monitors, adapts, and steers.

Role-Playing / Society of Agents

CAMEL (Li et al.) is foundational: complementary roles (e.g., "Python programmer" + "stock trader") produce better completion than a single agent. Role assignment constrains behavior productively — an agent told it is a "security reviewer" catches more vulnerabilities.

SOP / Assembly-Line Multi-Agent

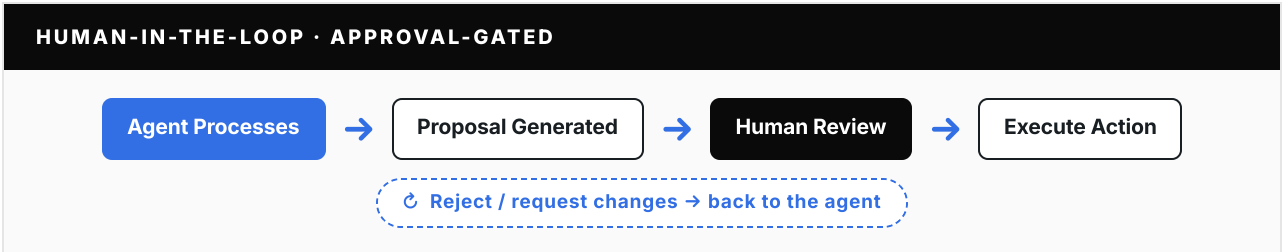
MetaGPT (Hong et al.) encodes Standard Operating Procedures: agents take roles like Product Manager, Architect, Engineer, QA and follow defined workflows (requirements → design → implement → test). The SOPs prevent the chaos of unstructured multi-agent systems.

07 ENTERPRISE INTEGRATION PATTERNS

Layered on top of everything else.

Production systems require patterns that address enterprise-specific concerns — human oversight, custom workflow composition, and interaction with real-world environments. These are rarely used in isolation.

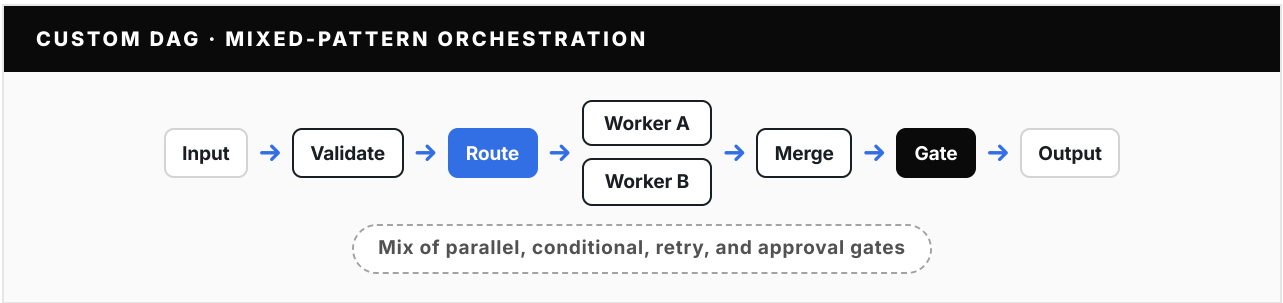
Human-in-the-Loop / Approval-Gated Agents A REQUIREMENT, NOT A NICETY



Critical actions pause for human validation before execution — essential for financial transactions, customer communications, infrastructure changes, anything that cannot be easily reversed.

Best practices: define clear approval thresholds (what needs review vs. what can proceed). 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 WHERE SERIOUS SYSTEMS END UP



A hand-built mix of parallel checks, conditional branches, retries, approval gates, and agent calls — where serious production systems end up once simple templates no longer fit (LangGraph, CrewAI, AutoGen).

Best practices: document the DAG visually — complex orchestrations become unmaintainable without diagrams. Implement comprehensive error handling at each node. Use idempotent operations so retries are safe. Monitor end-to-end latency and cost, not just per-node performance.

Browser / Computer-Use Agents GUI ENVIRONMENTS

Agents that operate in web or GUI environments through browsing, clicking, typing, and observation (WebGPT; Anthropic computer use; OpenAI Operator). A distinct family because it requires different tooling — screen capture, DOM parsing, mouse/keyboard control. **When to use:** automating systems that lack APIs, QA testing, legacy integration. **Best practices:** robust error recovery (pages load slowly, elements move); screenshots at each step; strict URL allowlists; structured selectors over visual matching.

A DECISION FRAMEWORK

Choosing the right pattern.

With 30+ patterns to choose from, selection can be overwhelming. 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 based on intermediate results? Use ReAct. If the workflow is fixed, use prompt chaining instead.

3 **Add routing for diverse inputs.**
Receiving fundamentally different query types? Add a router to dispatch to specialists — cheaper and more reliable than one agent that handles everything.

4 **Add parallelization for latency or quality.**
Can the task 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 and upgrade to separate evaluator models if needed.

6 **Add memory for continuity.**
Need context across sessions, or must the agent learn? Add short-term memory first, then long-term, then reflexion.

7 **Add multi-agent for complex collaboration.**
Genuinely too complex for one agent? Orchestrator-workers for known decompositions, group chat for creative/analytical tasks, swarms only when you need emergent behavior.

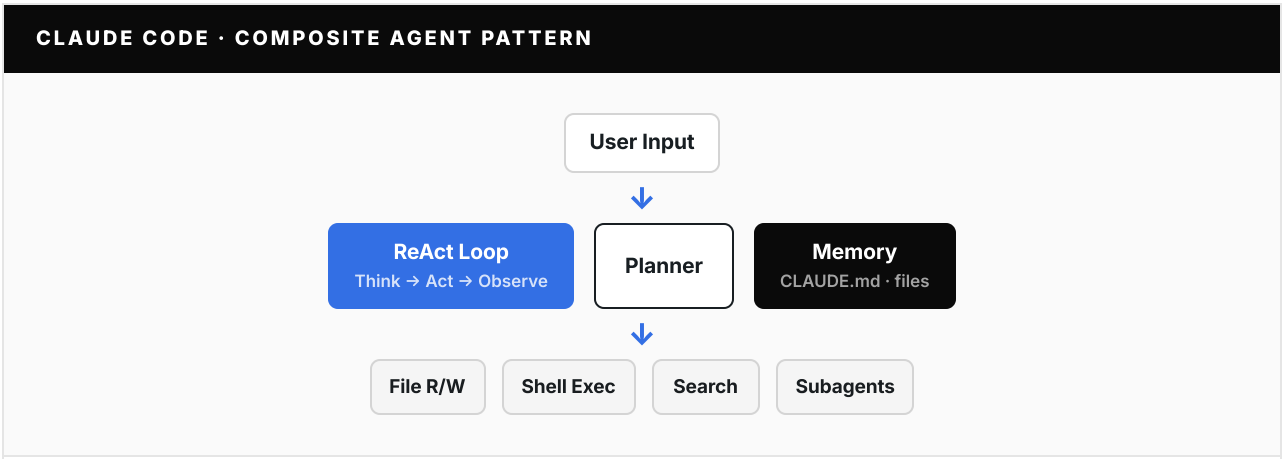
8 **Add human-in-the-loop for high-stakes actions.**
Can agent errors cause real harm? Add approval gates. A requirement, not an option, for production systems in regulated industries.

08 COMPOSITE PATTERNS

Real systems compose multiple patterns.

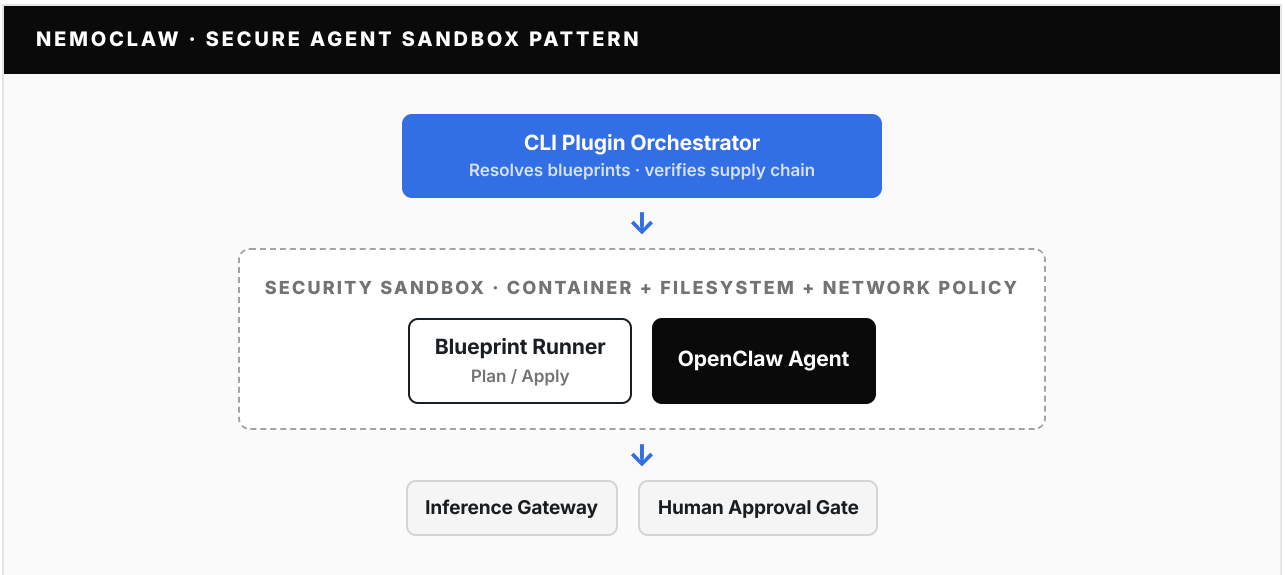
The patterns in this paper are building blocks. Production systems rarely use a single pattern in isolation — they compose several into integrated architectures.

Claude Code REACT + PLANNING + MEMORY + ORCHESTRATION



A single-agent architecture incorporating multiple patterns. Core is a ReAct loop; a planner decomposes complex multi-file tasks; memory persists across sessions; large tasks spawn orchestrator-worker subagents on isolated git worktrees.

NemoClaw PLAN-EXECUTE + SANDBOXING + HUMAN-IN-LOOP

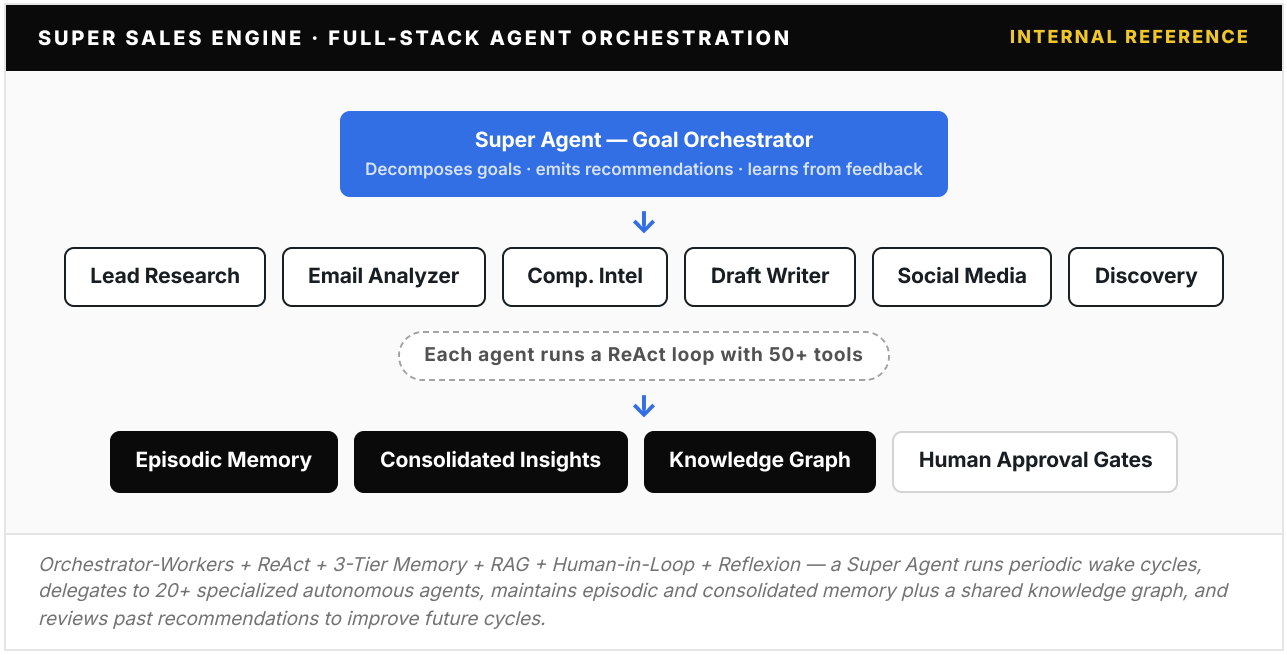


The OpenClaw coding agent wrapped in a multi-layer security sandbox driven by a deterministic plan-and-execute pipeline. Filesystem policies (Landlock LSM) restrict paths; network defaults deny; unlisted hosts surface a human approval gate; all inference routes through a gateway.

ITERATE.AI SUPER SALES ENGINE · INTERNAL REFERENCE ARCHITECTURE

Full-stack agent orchestration.

An internal Iterate.ai solution demonstrating the most comprehensive pattern composition: every tier of the taxonomy working together in a single production application.



Orchestrator-Workers + ReAct + 3-Tier Memory + RAG + Human-in-Loop + Reflexion — a Super Agent runs periodic wake cycles, delegates to 20+ specialized autonomous agents, maintains episodic and consolidated memory plus a shared knowledge graph, and reviews past recommendations to improve future cycles.

Human approval gates operate at multiple levels — agent-level, tool-level, and action-level risk assessment. The Super Agent also implements reflexion: it reviews past recommendations and user feedback to improve future wake cycles. While internal, the system serves as a reference architecture for how enterprises can compose the full pattern taxonomy into a production application.

09 PRODUCTION BEST PRACTICES

Principles that apply to every production agentic system.

Regardless of which patterns you choose, these are the lessons learned from organizations that have moved agentic AI from prototype to production.

1 **Observability is non-negotiable.**

Every agent action, tool call, memory read/write, and inter-agent message must be logged with structured metadata. Implement trace IDs that follow a request through the entire pipeline; build dashboards for token consumption, latency, error rates, and cost by agent, pattern, and task type. Tools like **AgentWatch by Iterate.ai** provide purpose-built AI observability and privacy policy enforcement for agentic systems.

2 **Set explicit boundaries.**

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

3 **Test agent behavior, not just outputs.**

Agent tests must verify behavior: did the agent use the right tools, follow the expected reasoning pattern, stay within its boundaries? Build evaluation suites that test trajectories — the sequence of actions — not just final answers. Include adversarial cases for prompt injection, tool misuse, and boundary violations.

4 **Design for graceful degradation.**

External services fail, models hallucinate, tools return surprises. A scatter-gather pattern should produce useful output even if one worker fails. An agentic RAG system should answer from existing context if retrieval fails. A human-in-the-loop system should default to safe behavior if no human responds.

5 **Start simple, measure, then evolve.**

The most common mistake is over-engineering. Start with the simplest pattern that could work, measure its performance against your requirements, identify the concrete (not theoretical) limitations, then evolve to address those specific limitations. Faster, cheaper, and better than designing the “perfect” multi-agent system upfront.

FROM DESIGN TO PRODUCTION

Security 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 & Input Validation

Every agent that accepts user input or external data is vulnerable. Validate and sanitize all inputs; use system prompts the user cannot override; flag suspicious patterns. **Never trust content retrieved from external sources — treat RAG results as untrusted input.**

Data Privacy & Access Control

Implement row- and document-level access controls in retrieval. Ensure agents cannot access data beyond the requesting user's permissions. Encrypt at rest and in transit. Define retention policies for memory stores; audit what data the agent accesses, stores, and returns.

Human Oversight as a Control

Human-in-the-loop is a critical security control, not just a design pattern. Classify actions by risk; require approval above a threshold; show the full reasoning chain; implement dead-man switches — if oversight fails, default to safe, do not proceed.

Tool & API Security

Each tool is a potential attack vector. Apply least privilege; authenticate and authorize every tool call; use allowlists for paths, URLs, and endpoints; sandbox code execution; log every invocation with full parameters.

Multi-Agent Trust Boundaries

A compromised agent can propagate bad instructions to others. Define explicit trust boundaries; validate inter-agent messages; implement output validation at each boundary; use separate credentials per agent role; monitor for anomalous behavior.

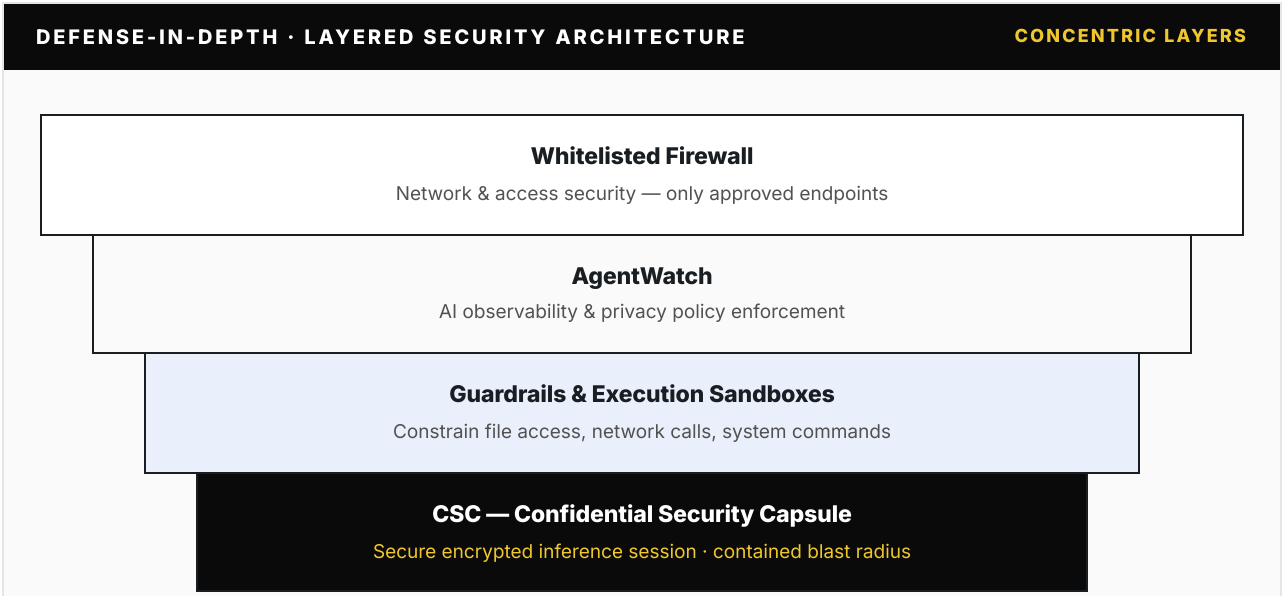
Observability & Incident Response

You cannot secure what you cannot see. Log every action with trace IDs; alert on anomalous patterns (unexpected tool usage, unusual data access, injection attempts); maintain audit trails that reconstruct decisions; test an incident-response playbook for agent scenarios.

OPEN-SOURCE FRAMEWORKS & LAYERED DEFENSE

Permissive defaults are an unsecured server on the public internet.

Open-source agent frameworks lower the barrier to building agentic systems — but many ship with unrestricted tool access, no sandboxing, no input validation, and no output filtering. Audit every framework before deployment; disable default tools you do not need; add guardrails; run execution in sandboxes with restricted network access.



Multiple concentric security layers, each providing independent protection. At the core, the Confidential Security Capsule runs inference inside a secure encrypted session — even when one session is breached, the blast radius is contained. CSC is software-based, deployable across any cloud or on-premise environment.

This layered approach ensures that no single point of failure can compromise the entire agentic system. From the outside in: a whitelisted firewall controls network and access security, AgentWatch monitors all agent activity, guardrails and execution sandboxes constrain runtime behavior, and the CSC protects the inference itself.

10 SOVEREIGN AI · THE BINDING CONSTRAINT

For a growing segment of deployments, the environment is the constraint — not the pattern.

Defense, intelligence, government, healthcare handling sensitive patient data, financial institutions under data-residency rules, telecommunications under sector regulations — for these, the deployment envelope binds before any pattern does.

THE REGULATORY LANDSCAPE IS TIGHTENING

GDPR & Schrems II

An agent calling a US-hosted model API to reason over EU personal data may be structurally non-compliant, regardless of vendor terms of service.

The EU AI Act

High-risk systems face transparency, human-oversight, and auditability requirements that directly constrain available agentic patterns — employment, credit scoring, law enforcement, critical infrastructure.

Sector-specific frameworks

ITAR/EAR, HIPAA, CJIS, FedRAMP, FISMA, CMMC — each constrains not just what the agent does, but *where it runs* and *what data it can access*.

Proliferating data residency

Beyond the EU, residency requirements have expanded across the Middle East, Southeast Asia, and South Asia — complex, overlapping rules on where data may be stored and processed.

THE SOVEREIGN AI CONSTRAINT

The data cannot leave. The model must come to the data.

This eliminates a significant portion of the pattern space as typically implemented — patterns that assume real-time calls to cloud-hosted model APIs. When the infrastructure stack is redesigned so the model, data, inference compute, and application logic all operate within the sovereign boundary, the full pattern library becomes available again — including multi-agent orchestration, tool-calling, and RAG — because agents reason over governed local data without external egress.

PATTERN GUIDANCE FOR CONSTRAINED ENVIRONMENTS

Start with the deployment envelope, not the pattern.

Map the environment's constraints — connectivity, accreditation, data-handling rules, permissible hardware — before selecting an architecture. These constraints will eliminate some patterns and constrain others.

- **Prefer stateless, compact patterns in disconnected environments.** Tactical edge deployments may operate without reliable connectivity. Simple ReAct loops against locally-cached knowledge bases are often more appropriate than sophisticated multi-agent orchestrations.
- **Sovereign RAG requires local vector stores.** RAG is one of the highest-value patterns for regulated environments. In sovereign deployments, local vector infrastructure — embedded in the runtime or served by a sovereign data platform — is the architectural prerequisite.
- **Audit logging is a compliance requirement, not optional observability.** Every action, tool call, data access, and inference must be logged with enough detail for forensic reconstruction — and the logging infrastructure must be local and must not egress.
- **Use small, purpose-trained models where frontier models cannot deploy.** SLMs (1B–14B parameters) fine-tuned on domain corpora can deliver high-accuracy inference at sub-50ms latency on commodity hardware. Frequently the superior architecture, not a compromise.
- **Design for the airgapped case even when not required.** Disconnected-first design imposes discipline that benefits all deployments: data locality, minimal external dependencies, observable inference, and sovereign control of model weights and data.

FOR VEROMESH ENTERPRISE DEPLOYMENTS

VeroMesh's Modern Data and AI Architecture Framework is built for this class of deployment. Its three-layer reference architecture operates correctly from hyperscale cloud to single-node air-gapped facilities. The organizing principle: **AI comes to the data, not the other way around.**

Governed Data Foundation
The data fabric governs and enriches data at the source.

Governed Model Control Plane
The AI gateway enforces compliance and cost on every interaction.

Autonomous Agents & Private Inference
The agent platform executes against sovereign data without egress.

REAL-WORLD SYSTEMS

These patterns power the most capable AI systems shipping today.

The patterns in this paper are not theoretical. Understanding how production systems combine them helps technical leaders make informed architecture decisions.

Claude Code Anthropic	A coding agent on the ReAct pattern with extensive tool calling. Single agent, large toolset (file R/W, shell, search, browser), plan-and-execute for complex multi-file tasks, memory across sessions via CLAUDE.md, and orchestrator-worker subagents on isolated worktrees for large tasks.
Manus Manus AI	A multi-agent orchestrator with browser/computer-use agents. A planning agent decomposes goals into tasks, then delegates to specialists that browse, write code, manipulate files, and call APIs — combining plan-and-execute with tool-marketplace patterns.
Operator & Agent SDK OpenAI	Operator is a browser/computer-use agent that navigates web interfaces autonomously. The Agent SDK supports multi-agent handoff — agents transfer control to specialists mid-conversation — plus routing, tool calling with built-in guardrails, and tracing for observability.
Devin & SWE-Agent Cognition / Princeton	Autonomous coding agents combining ReAct loops with long-term memory and skill-library patterns, operating in sandboxed environments. Both demonstrate reflexion — learning from failed attempts — and evaluation loops where test results feed back into reasoning.
Magentic-One & AutoGen Microsoft	Magentic-One implements manager-led orchestration with a task ledger tracking progress across specialists (web surfer, coder, file handler). AutoGen provides a framework for group-chat / multi-agent debate with configurable turn-taking and termination.

FROM PATTERNS TO PRODUCTION

Understanding the patterns is essential. Operating them 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 teams find this infrastructure work consumes more effort than the agent logic itself.

ITERATE.AI

Generate

Rapid prototyping and deployment of AI-powered applications — the foundation for experimenting with different agentic patterns quickly.

ITERATE.AI

Interplay

A visual composition environment where the patterns in this paper — from single-agent tool callers to multi-agent orchestrations — are assembled, tested, and deployed without building the infrastructure from scratch.

Interplay's drag-and-connect interface maps directly to the architecture 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 is trusted by leading Fortune 500 companies for its enterprise AI solutions. Technology partners include Intel, NVIDIA, Qualcomm, Dell, IBM, NetApp, and others.

ABOUT THIS PAPER

A collaboration between Iterate.ai and VeroMesh.ai.

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 of Engineering, Chief AI Officers, and Chief Digital Officers. Pattern descriptions are grounded in published research and official documentation; the Tier 0 governance framework and sovereign-deployment guidance reflect VeroMesh's enterprise deployment experience across regulated industries.

AUTHORS

ITERATE.AI

Brian Sathianathan

Cofounder & CTO, Iterate.ai.

VEROMESH.AI

Sibito Morley

Cofounder & President, VeroMesh.ai.

OFFICIAL DOCUMENTATION

01 Anthropic — *Building Effective AI Agents*.

02 Google Cloud — *Agent Design Patterns for Agentic AI Systems*.

03 Microsoft Azure — *AI Agent Orchestration Patterns*.

04 AWS Prescriptive Guidance — *Agentic AI Patterns and Workflows on AWS*.

05 Hugging Face — *Design Patterns for Building Agentic Workflows*.

SURVEYS & RECENT WORK

17 Wang et al. — *A Survey on LLM-Based Autonomous Agents*.

18 Guo et al. — *A Survey on LLM-Based Multi-Agent Systems*.

19 Masterman et al. — *The Landscape of Emerging AI Agent Architectures*.

FOUNDATIONAL RESEARCH

06 Yao et al. — *ReAct: Synergizing Reasoning and Acting in Language Models*.

07 Shinn et al. — *Reflexion: Language Agents with Verbal Reinforcement Learning*.

08 Yao et al. — *Tree of Thoughts*.

09 Xu et al. — *ReWOO: Decoupling Reasoning from Observations*.

10 Li et al. — *CAMEL: Communicative Agents*.

11 Hong et al. — *MetaGPT: Multi-Agent Collaboration*.

12 Park et al. — *Generative Agents*.

13 Wang et al. — *Voyager: An Open-Ended Embodied Agent*.

14 Zhou et al. — *Language Agent Tree Search*.

15 Gao et al. — *Retrieval-Augmented Generation for LLMs: A Survey*.

PLATFORMS

Generate · Interplay

by Iterate.ai

ITERATE.AI

iterate.ai/platform

/generate · /interplay

VEROMESH.AI

veromesh.ai

Sovereign enterprise AI

EDITION

2026