# AI Agent Architecture Reference



Reference Card · Updated April 2026

**ReAct (thought-action-observation loop)** The agent loops through three steps: it thinks about the current state and goal, picks an action or tool to call, then observes the result before thinking again. This interleaves reasoning and acting in a dynamic feedback loop. It grounds decisions in real tool outputs instead of hallucinated steps. [1]

**Plan-and-Execute (upfront planning then execution)** The agent first creates a complete multi-step plan, then executes it step by step with minimal deviation. A separate executor or controller follows the plan while the planner stays out of the loop. This adds structure compared to pure reactive approaches but struggles when the environment changes mid-execution.[1]

**Tool Calling / Function Calling (structured tool use)** The model outputs structured JSON or schema-compliant calls instead of free-form text. The framework parses the call, runs the tool or API, and feeds the result back. This gives reliable, auditable tool use with type safety and validation gates.[1]

**Code Execution Agents (sandboxed code running)** The agent writes and runs code in a restricted interpreter or container. It generates, executes, inspects output, and iterates on the code until the task succeeds. This excels at data analysis and computation but requires strong sandboxing to limit damage from bad code.[2]

**Multi-Agent. Supervisor pattern** A central supervisor agent routes tasks to specialized worker agents, collects outputs, and produces the final result. The supervisor maintains the overall plan and context. It works well for clear task decomposition but creates a single point of failure and coordination bottleneck.[3]

**Multi-Agent. Debate pattern** Multiple agents generate competing solutions or critiques and discuss them in rounds. They challenge assumptions and refine ideas through structured back-and-forth. This improves reasoning on ambiguous or subjective tasks at the cost of higher token use and latency.

**Multi-Agent. Pipeline pattern** Agents sit in a fixed linear sequence. Each takes the previous agent's output, performs its specialized transformation, and passes the result forward. It enforces quality gates in content or data workflows. The whole pipeline slows to the speed of its weakest stage.[3]

**Memory. Context window (short-term)** The agent keeps recent messages, tool outputs, and thoughts inside the model's active context window. This provides immediate working memory for the current session. It's simple but hits token limits quickly and forgets everything when the window rolls.[1]

**Memory. Vector store (long-term semantic)** Embeddings of past documents, observations, or conversations go into a vector database. The agent queries for semantically similar items when needed. This scales to large histories but depends on embedding quality and retrieval precision.[4]

**Memory. Conversation history (session tracking)** The system stores and summarizes full conversation turns across sessions. It injects relevant past context or summaries into new prompts. This maintains continuity without bloating every context window but requires good summarization to avoid noise.

| PATTERN | BEST FOR | WEAKNESS | EXAMPLE FRAMEWORK |
|---|---|---|---|
| ReAct | Dynamic tool use, research | High latency, error compounding | LangGraph, LangChain |
| Plan-and-Execute | Structured multi-step tasks | Inflexible to changes | LangGraph |
| Tool Calling | Reliable API/function use | Schema maintenance | OpenAI SDK, Anthropic |
| Code Execution Agents | Data analysis, computation | Sandbox security risks | LangChain code interpreter |
| Multi-Agent. Supervisor | Task routing, mixed expertise | Supervisor bottleneck | LangGraph, CrewAI |
| Multi-Agent. Debate | Complex reasoning, evaluation | High token cost, slow | AutoGen, custom |
| Multi-Agent. Pipeline | Linear content/data workflows | No parallelism, brittle | LangGraph, CrewAI |
| Memory. Context window | Short sessions | Token limits | All base models |
| Memory. Vector store | Large knowledge bases | Retrieval quality variance | LlamaIndex, LangChain |
| Memory. Conversation history | Long user sessions | Summary drift over time | LangGraph checkpointers |

These patterns rarely appear in isolation. Most production agents combine two or three (ReAct plus vector memory plus supervisor routing). Pick the simplest pattern that solves your actual failure modes.