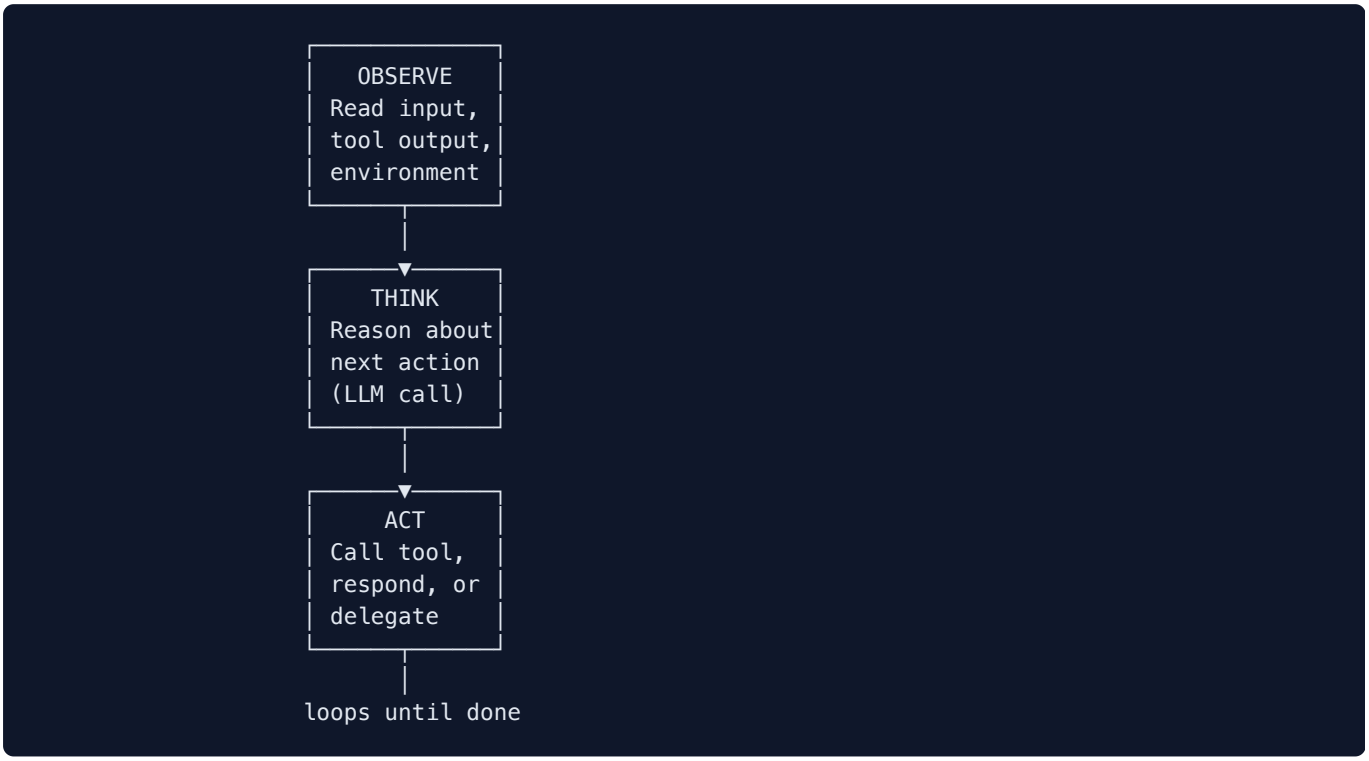
Agent Anatomy: Observe-Think-Act Loop

Every AI agent, regardless of framework, follows this core loop. Understanding it is essential before you add complexity with multi-agent patterns or custom orchestration.



Core Agent Components

An agent is more than an LLM with tools -- it needs memory, planning, and reflection to handle real-world tasks reliably. Missing any of these components leads to agents that work in demos but fail in production.

Component	Purpose	Implementation
System prompt	Identity, rules, capabilities	Static text + dynamic context
Tool definitions	Available actions	Function schemas (JSON)
Memory	Conversation context	Buffer, summary, or vector store
Planning	Task decomposition	CoT, ReAct, or explicit planner
Execution	Tool calling + result processing	Function dispatch + error handling
Reflection	Self-check, retry logic	Output validation, critic LLM

Tool Definition Patterns

Tool definitions are the contract between your agent and the outside world. Vague descriptions and sloppy schemas are the #1 cause of agents calling the wrong tool or passing bad arguments.

OpenAI-Compatible Tool Schema

```
{
  "type": "function",
  "function": {
    "name": "search_database",
    "description": "Search the product database by query. Returns top 5 results.",
    "parameters": {
      "type": "object",
      "properties": {
        "query": {
          "type": "string",
          "description": "Natural language search query"
        },
        "category": {
          "type": "string",
          "enum": ["electronics", "clothing", "books"],
          "description": "Optional category filter"
        },
        "max_results": {
          "type": "integer",
          "default": 5,
          "description": "Maximum number of results to return"
        }
      },
      "required": ["query"]
    }
  }
}
```

Tool Design Best Practices

Principle	Good	Bad
Specific name	`search_orders_by_email`	`search`
Clear description	"Finds orders by customer email. Returns order ID, date, total."	"Search stuff"
Typed parameters	`{"type": "integer", "minimum": 1}`	`{"type": "string"}` for a number
Required vs optional	Mark only truly required as required	Everything required
Return format	Documented, consistent structure	Unpredictable output
Error surface	Return error in result, don't throw	Silent failure

Memory Types

Without memory, every agent turn starts from scratch. The right memory architecture determines whether your agent can handle a 5-message chat or a 500-step workflow spanning multiple sessions.

Memory Type	How It Works	Capacity	Use Case
Buffer (sliding window)	Keep last N messages	Low-medium	Short conversations
Token buffer	Keep last N tokens of history	Medium	Token-budget aware
Summary	LLM summarizes older messages	High	Long conversations
Vector/semantic	Embed messages, retrieve relevant	Very high	Knowledge-heavy agents
Episodic	Store full episodes, retrieve by similarity	Very high	Learning from past tasks
Entity	Extract and track entity states	Medium	Customer service, CRM
Structured (KG)	Knowledge graph of facts	High	Complex domain reasoning

Memory Selection Guide

```
How long are typical conversations?
  < 10 turns    -> Buffer memory (simple, cheap)
  10–50 turns   -> Summary memory (compress old context)
  50+ turns     -> Vector memory (retrieve relevant only)

Does the agent need to learn across sessions?
  YES -> Episodic + Vector memory
  NO  -> Buffer or Summary is fine

Does the agent track many entities?
  YES -> Entity memory + structured storage
  NO  -> Standard memory is fine
```

Agent Frameworks Comparison

Choosing a framework is a build-vs-buy decision that affects your iteration speed and lock-in. Pick based on your language, complexity needs, and whether you need multi-agent support.

Framework	Language	Key Feature	Best For
LangGraph	Python/JS	Graph-based workflows	Complex stateful agents
CrewAI	Python	Role-based multi-agent	Team simulations
AutoGen	Python	Conversational agents	Research, debate patterns
Semantic Kernel	C#/Python	Enterprise integration	.NET ecosystems
Haystack	Python	Pipeline-based	RAG-heavy agents
Agents SDK (OpenAI)	Python	Handoffs, guardrails	OpenAI-centric apps
Claude Agent SDK	Python	MCP tools, model agnostic	Anthropic-centric apps
Mastra	TypeScript	Workflows, evals	TS/JS applications

ReAct Pattern

ReAct (Reason + Act) is the most widely used agent pattern because it forces the model to explain its reasoning before taking action. This makes agent behavior interpretable and debuggable.

```
Thought: I need to find the user's order status. I'll search by their email.
Action: search_orders(email="user@example.com")
Observation: Found order #1234, status: shipped, tracking: XYZ789
Thought: I have the info. I'll respond with the order status and tracking number.
Answer: Your order #1234 has been shipped. Tracking number: XYZ789.
```

ReAct Implementation

```
def react_loop(query, tools, max_steps=10):
    messages = [
        {"role": "system", "content": REACT_SYSTEM_PROMPT},
        {"role": "user", "content": query}
    ]

    for step in range(max_steps):
        response = llm.chat(messages, tools=tools)

        if response.tool_calls:
            for call in response.tool_calls:
                result = execute_tool(call.name, call.args)
                messages.append({"role": "tool", "content": result,
                                 "tool_call_id": call.id})
        else:
            return response.content  # Final answer

    return "Max steps reached without resolution."
```

Orchestration Patterns

How you wire agents together determines your system's capability ceiling and failure modes. Start with the simplest pattern that works and only add complexity when you have evidence a single agent cannot handle the task.

Pattern	Description	Use Case	Complexity
Single agent	One LLM + tools	Simple tasks	Low
Sequential pipeline	Agent A output feeds Agent B	Multi-step processing	Medium
Parallel fan-out	Same input to N agents	Multiple perspectives	Medium
Router	Classifier routes to specialist	Domain-specific handling	Medium
Supervisor-worker	Supervisor delegates, reviews	Complex task decomposition	High
Hierarchical	Multi-level supervisors	Enterprise workflows	High
Debate/consensus	Agents argue, reach agreement	High-stakes decisions	High

Supervisor-Worker Pattern

```
# Supervisor decides which worker to call
supervisor_prompt = """
You are a supervisor managing these workers:
- researcher: Finds information from documents
- calculator: Performs mathematical computations
- writer: Drafts text content

Given the user request, decide which worker(s) to call and in what order.
Respond with a plan as JSON: {"steps": [{"worker": "...", "task": "..."}]}
"""

# Workers are specialized agents with focused tool sets
workers = {
    "researcher": Agent(tools=[search, retrieve]),
    "calculator": Agent(tools=[calculate, chart]),
    "writer": Agent(tools=[draft, edit]),
}
```

Planning Strategies

Planning determines whether your agent tackles a complex task methodically or stumbles through it. The right strategy balances structure against adaptability -- too rigid and the agent can't recover from surprises, too loose and it loses track of its goal.

Strategy	How It Works	Pros	Cons
No explicit plan	LLM decides step by step	Simple, flexible	May lose track
Upfront plan	Generate full plan first, then execute	Organized	Inflexible to new info
Adaptive plan	Plan, execute, re-plan after each step	Flexible, informed	Higher LLM cost
Plan-and-solve	Decompose into sub-tasks, solve each	Good for complex tasks	Overhead

Error Handling

Agents fail in ways that traditional software does not -- they hallucinate tool names, pass invalid arguments, and get stuck in infinite loops. Robust error handling is what separates a demo agent from a production one.

Error Type	Detection	Recovery
Tool not found	Invalid tool name in response	Re-prompt with available tools
Tool execution failure	Exception from tool	Return error to LLM, let it retry
Infinite loop	Step counter exceeds max	Force response or escalate
Hallucinated tool call	Tool name not in schema	Filter, re-prompt
Wrong arguments	Schema validation failure	Return validation error to LLM

Error Type	Detection	Recovery
Context overflow	Token count exceeded	Summarize history, trim old messages

Agent Evaluation

Agent evaluation goes beyond LLM output quality -- you also need to measure tool accuracy, step efficiency, and cost. Without these metrics, you are flying blind on whether your agent is actually improving.

Metric	What It Measures	How to Measure
Task completion	Did the agent finish the task?	Binary success/failure
Tool accuracy	Correct tool called with correct args?	Compare to gold standard
Step efficiency	Number of steps to complete	Count tool calls
Cost	Total tokens consumed	Sum input + output tokens
Latency	Time to complete task	Wall clock time
Safety	No harmful actions taken	Red-team testing
User satisfaction	Did the user get what they needed?	Thumbs up/down, CSAT

Common Pitfalls

Most agent failures come from architectural over-engineering or missing safety boundaries, not from the LLM itself. Check this list before adding another agent to your system.

Pitfall	Problem	Fix
Too many tools	LLM confused, wrong tool selection	Limit to 10-15 tools, use routing
Vague tool descriptions	Wrong tool calls	Write precise descriptions with examples
No max iteration limit	Infinite loops, cost explosion	Set hard limit (5-20 steps)
Full history in context	Token overflow, high cost	Use summary or vector memory
No tool result validation	Garbage in, garbage out	Validate tool outputs before passing to LLM
Single monolithic agent	Poor at specialized tasks	Split into specialist agents
No human escalation path	Agent stuck on hard cases	Add "escalate_to_human" tool
Ignoring tool errors	Agent continues with bad data	Surface errors clearly to the LLM