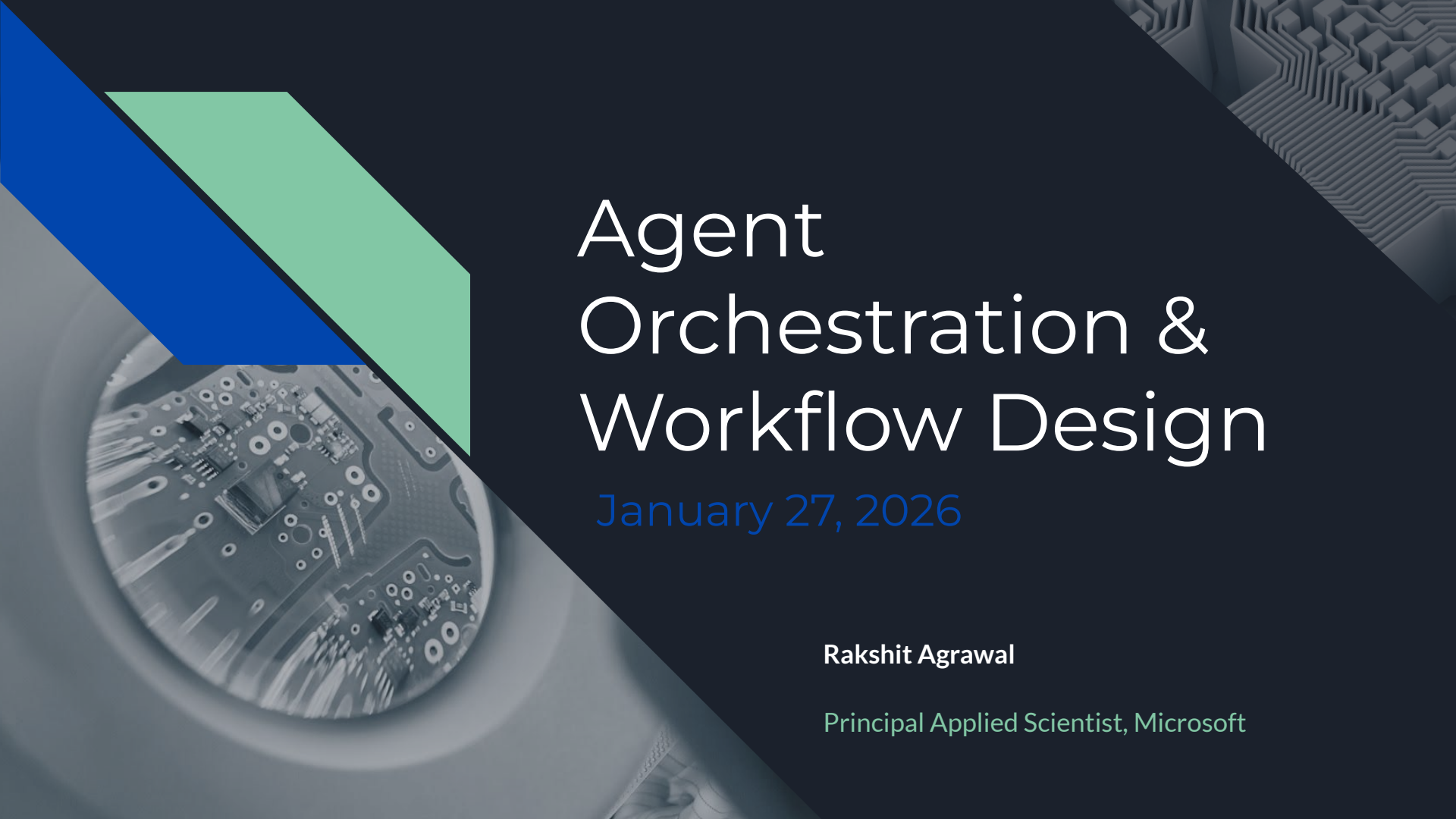


Agent Orchestration & Workflow Design

January 27, 2026

Rakshit Agrawal

Principal Applied Scientist, Microsoft

About me

- Principal Applied Scientist at Microsoft
- PhD in Computer Science (Machine Learning and Graphs)
- Previous startups in AI (healthcare, graphs, computer vision)
- Working in AI for >10 years now
- Advisor to The Nature Conservancy on AI and Computer Vision

Agenda

1. Foundations: What is an Agent?
2. Agent Architectures: Single vs. Multi-Agent
3. Orchestration Frameworks: LangGraph vs. CrewAI
4. Agentic Patterns (Workflows)
5. Tool Use & Function Calling
6. Agent Communication Protocols (MCP & A2A)
7. State Management
8. Real-World Applications
9. Safety & Containment

FOUNDATIONS

What is an "Agent"?

Definition (Anthropic):

"Systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks."

Key Distinction:

- Workflow: Predefined code paths (deterministic)
- Agent: Dynamic, model-driven decision-making

The Four Pillars of Agency

by psychologist Albert Bandura

1. Intentionality (Planning): The agent sets its own goals
2. Forethought: The agent anticipates future states
3. Self-Reactiveness: The agent monitors its own performance
4. Self-Reflectiveness: The agent adjusts its strategy based on outcomes

When to Use Agents (Anthropic)

"Start with simple prompts, optimize them with comprehensive evaluation, and add multi-step agentic systems only when simpler solutions fall short."

Trade-offs:

- Pros: Flexibility, can handle novel tasks
- Cons: Higher latency, higher cost

When Workflows Suffice:

- Well-defined tasks with known steps
- Predictability and consistency are paramount

Types of AI Agents (Russell & Norvig)

A progression from simple to complex:

1. Simple Reflex Agent: Condition-Action rules (if dirty, vacuum)
2. Model-Based Reflex Agent: Maintains internal state (memory)
3. Goal-Based Agent: Plans actions to achieve a specific goal
4. Utility-Based Agent: Optimizes for a "utility" function
5. Learning Agent: Improves performance over time

Most modern "AI Agents" are Goal-Based or higher.

AGENT ARCHITECTURES

Single-Agent Architecture

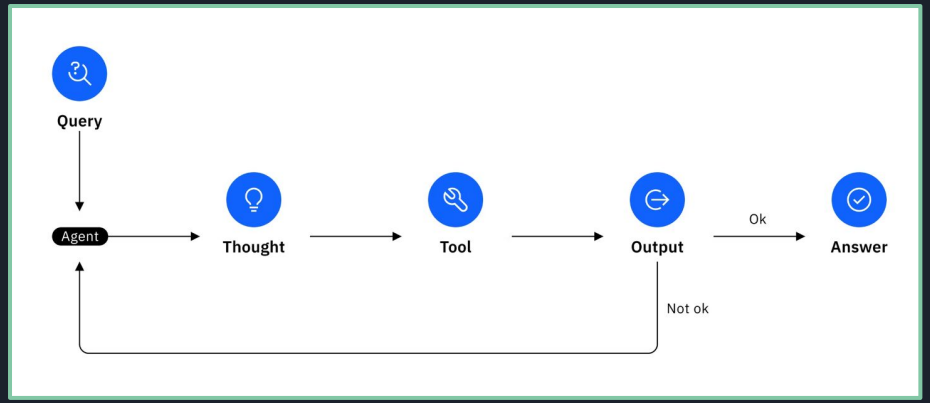
- **Definition**: A single autonomous entity making centralized decisions.
- **Structure**: One LLM, one set of tools, one context window
- **Strengths**:
  - Simplicity, lower latency, easier to debug
  - Fewer integration challenges
- **Weaknesses**:
  - Context window fills up fast
  - Struggles with complex, multi-domain tasks

Single-Agent: The ReAct Pattern

ReAct = Reasoning + Acting

Loop:

1. Thought: "I need to find the population of Paris."
2. Action: search("population of Paris")
3. Observation: "The population of Paris is 2.1 million..."
4. Thought: "I now have the answer."
5. Final Answer: "The population is approximately 2.1 million."



Source: https://www.ibm.com/think/topics/react-agent#1793360183

Multi-Agent Systems (MAS)

Definition: Specialized agents collaborating to solve problems.

Key Insight: Partition the problem space.

- Each agent has a focused role (Coder, Reviewer, Researcher)
- Each agent has its own system prompt and tools

Single vs. Multi-Agent: Trade-offs

Feature	Single Agent	Multi-Agent System (MAS)
Context	Single, shared window	Partitioned by role
Prompting	One "monolithic" prompt	Modular, specialized instructions
Control	Heuristic & fluid	Strict graph-based workflows
Performance	Lower latency	Higher latency (inter-agent calls)
Maintenance	Easy to debug	Complex orchestration

Guidance: Start Single. Upgrade to MAS when you need distinct "personas".

ORCHESTRATION FRAMEWORKS

LangGraph: The Low-Level Choice

Model: State Machine (Nodes & Edges)

Core Concepts:

- State: A shared TypedDict that persists across steps
- Nodes: Python functions that modify the state
- Edges: Define transitions (can be conditional)

LangGraph: Key Features

- Persistence: Save checkpoints to resume later
- Human-in-the-Loop: Pause execution, allow human edit, resume
- Subgraphs: Nest graphs for modularity
- Streaming: Stream tokens as they are generated

Best For: Complex, custom workflows requiring fine-grained control.

CrewAI: The High-Level Choice

Model: Role-Based Organization (like a company)

Core Concepts:

- Agent: Defined by Role, Goal, Backstory
- Task: A unit of work assigned to an Agent
- Crew: A collection of Agents working on Tasks
- Process: sequential or hierarchical

Best For: Rapid prototyping, standard "team" topologies.

Framework Comparison

Feature	LangGraph	CrewAI
Logic Layer	Low: Explicit State Machines	High: Role & Task based
Flexibility	Infinite: Granular control	Opinionated: Pre-defined patterns
Setup Effort	High (Steeper learning curve)	Low (Gentler learning curve)
Human Oversight	Native: Built-in "interrupts"	Limited / Basic
Ideal For	Custom, high-precision pipelines	Rapidly scaling "agent teams"

AGENTIC PATTERNS (WORKFLOWS)

Pattern Overview

The building blocks of agentic systems:

1. Prompt Chaining: Fixed sequence of steps
2. Routing: Classify and dispatch to specialists
3. Parallelization: Simultaneous independent work
4. Orchestrator-Workers: Dynamic delegation
5. Evaluator-Optimizer: Generate, test, iterate
6. ReAct: The foundational agent loop (Reasoning + Acting)
7. ReWOO: Plan upfront, execute in parallel

Supporting Colab Notebook: *Link*

Prompt Chaining

Concept: Decompose a task into a fixed sequence of steps.

Flow: Step A → Step B → Step C

Example:

1. Generate marketing copy
2. Translate to Spanish
3. Check for compliance

Benefit: Higher accuracy by simplifying each step.

Routing

Concept: Classify an input and direct it to a specialized handler.

Types:

- LLM Router: Ask the LLM to classify (Flexible, Slower)
- Semantic Router: Embedding similarity (Fast, Deterministic)
- Keyword Router: Regex/Keyword match (Fastest, Low flexibility)

Use Case: Customer Service Triage (Refunds vs. Tech Support vs. General Q&A)

Benefit: Separation of concerns; allows specialized prompts for each category.

Parallelization

Concept: LLMs work simultaneously on independent subtasks.

Variations:

- Sectioning: Breaking a large task into independent chunks
  - Example: Processing 5 different security guardrails simultaneously
- Voting: Running the same task multiple times for diversity
  - Example: Generating 3 code fixes, having a "Judge" pick the best

Benefit: Latency = slowest step (not sum of all steps).

Orchestrator-Workers

Concept: A central "Orchestrator" LLM dynamically breaks down a plan and delegates to "Workers".

Difference from Parallelization: Subtasks are NOT pre-defined; the Orchestrator determines them at runtime.

Example (Coding Agent):

1. User: "Add a login page."
2. Orchestrator: "I need to modify auth.py, create login.html, update routes.py."
3. Workers execute each file change.

Evaluator-Optimizer

Concept: Generate → Evaluate → Feedback Loop

Flow:

1. Generator: Produces a candidate solution
2. Evaluator: Scores or tests the solution
3. Feedback: If fail, provide error to Generator for retry

Use Case: Coding Agents

- Generator: Writes code
- Evaluator: Runs unit tests
- Feedback: "Test failed on line 10." → Generator fixes it

ReAct (The Foundation)

ReAct = Reasoning + Acting

The core loop that powers most single-agent systems:

1. Thought: The LLM reasons about what to do next
2. Action: The LLM outputs a tool call
3. Observation: The runtime executes the tool and returns the result
4. Repeat: Until the LLM decides to output a final answer

Limitation: Sequential execution. Each tool call waits for the previous one.

ReAct: Example Trace

User: What is the weather in Paris and London?

- Thought: I need to get weather for two cities. Let me start with Paris.
- Action: get_weather(city="Paris")
- Observation: Paris: 15°C, partly cloudy.
- Thought: Now I need to get weather for London.
- Action: get_weather(city="London")
- Observation: London: 12°C, rainy.
- Thought: I have both. Let me summarize.
- Final Answer: Paris is 15°C and partly cloudy. London is 12°C and rainy.

ReWOO (Reasoning Without Observation)

Problem with ReAct: Sequential "Stop-and-Wait" for each tool call.

ReWOO Solution: Separate Planning from Execution.

Flow:

1. Planner: Generates the full plan upfront (Search X, Search Y, Calculate Z)
2. Workers: Execute all search steps in PARALLEL
3. Solver: Synthesizes the final answer

Benefit: 2x-5x latency reduction for multi-tool tasks.

TOOL USE & FUNCTION CALLING

What is Function Calling?

The mechanism by which an LLM signals that it needs to use an external tool.

The Loop:

1. User sends message
2. LLM generates a tool call (JSON)
3. Runtime executes the tool
4. Runtime sends observation back to LLM
5. LLM generates final response

Function Calling: The "Stop Reason"

LLMs don't magically execute code. They output a structured request.

```
{
  "stop_reason": "tool_use",
  "tool_calls": [
    {
      "name": "search",
      "arguments": { "query": "population of Paris" }
    }
  ]
}
```

The runtime (your code) intercepts this and calls the actual function.

Defining Tools: JSON Schema

Tools are defined for the LLM via JSON Schema.

```
{
   "name": "search",
   "description": "Search the web for information.",
   "parameters": {
      "type": "object",
      "properties": {
         "query": {
            "type": "string",
            "description": "The search query."
         }
      },
      "required": ["query"]
   }
}
```

Tool Definition Best Practices (Anthropic)

1. Clear Names: get_stock_price > gsp
2. Detailed Descriptions: The docstring IS the prompt. Be descriptive.
3. Simple Interfaces: Use primitives (strings, ints). Avoid complex nested objects.
4. Fault Tolerance: Handle errors gracefully. Return informative error messages.
5. Idempotency: Where possible, ensure calling a tool twice has the same effect as once.

AGENT COMMUNICATION PROTOCOLS

The Interoperability Problem

As agents proliferate, we face new challenges:

- Tool Fragmentation: Every framework defines tools differently
- Agent Isolation: Agents from different vendors can't communicate
- Integration Burden: Developers re-implement the same integrations

Solution: Standardized protocols for agent-tool and agent-agent communication.

Two Key Protocols

- MCP (Anthropic):
  - Connect agents to tools & data sources.
  - Supports Human-to-Agent and Agent-to-Tool interactions.
- A2A (Google):
  - Enable agents to communicate with each other.
  - Supports Agent-to-Agent collaboration.
- Relationship: MCP and A2A are complementary, not competing.

MCP: Model Context Protocol

- Goal: A universal standard for connecting AI agents to external systems.
- Analogy: MCP is to agents what USB is to peripherals.
- Core Idea: Define tools once, use everywhere.

MCP: Key Concepts

- Tools: Functions the agent can call (e.g., search, send_email)
- Resources: Read-only data sources (e.g., files, DB records)
- Prompts: Pre-built prompt templates for common tasks
- Sampling: Let the server request LLM completions from the host

MCP: Benefits

- Write Once, Use Everywhere: Your MCP server works with any MCP-compatible agent.
- Security: The protocol defines permission scopes.
- Discovery: Agents can discover available tools at runtime.
- Ecosystem: Growing library of pre-built MCP servers (Slack, GitHub, DBs).

A2A: Agent-to-Agent Protocol

- Goal: A standard for agents from different vendors to collaborate.
- Problem: How does a "Travel Agent" (Vendor A) ask a "Booking Agent" (Vendor B) to reserve a hotel?
- Solution: A2A defines the message format and discovery mechanism.

A2A: Core Concepts

- Agent Card: A JSON document describing an agent's capabilities
  - Example: { "name": "Booking Agent", "skills": ["hotel", "flight"], "endpoint": "..." }
- Tasks: Units of work delegated between agents
- Messages: Structured communication (request, response, status)
- Artifacts: Files or data passed between agents

STATE MANAGEMENT

The "Memory" Problem

- LLMs are stateless.
- They don't remember past conversations.
- Solution: Manage state externally and inject it into the prompt.

Short-Term Memory (Thread Context)

- What: The list of messages in the current conversation
- Where: In-memory, passed with each LLM call
- Challenge: Context window limits (e.g., 128k tokens)
- Strategy: Summarization, sliding window, truncation

Long-Term Memory (Persistence)

- Key-Value Store: User preferences, session metadata
  - Example - { "user_id": "123", "preferred_language": "Spanish" }
- Vector Database (RAG): Semantic search for past knowledge
  - Store embeddings of documents/conversations
  - Retrieve relevant chunks based on query similarity

Checkpointing (LangGraph)

Concept: Save a "snapshot" of the graph's state after every node.

Capabilities:

- Resumption: Pick up where you left off after a disconnect
- Human-in-the-Loop: Pause, let a human approve, then resume
- Time Travel: Rewind to a previous state and fork execution

REAL-WORLD APPLICATIONS

Case Study A: Customer Support Agent

- Goal: Automate Tier-1 support with human escalation.
- Architecture: Router → Specialized Sub-Agents → Human Handoff

Customer Support: Flow

1. Guardrails: Filter PII, toxicity
2. Router: Classify intent (Billing, Tech, General)
3. Sub-Agent (Billing):
    a. Retrieve account info (API call)
    b. Check refund eligibility
    c. If eligible, call issue_refund tool
4. Human Handoff: If confidence < threshold or high-value action

Case Study B: Coding Agent

- Goal: Autonomously solve coding tasks (e.g., GitHub issues).
- Architecture: Orchestrator-Workers + Evaluator-Optimizer

Coding Agent: Flow

1. Planner: Read issue description, break down into file changes
2. Coder (Worker): Modify each file
3. Executor: Save files to disk
4. Tester: Run pytest or linter
5. Evaluator:
    a. If tests pass → Commit
    b. If tests fail → Send stderr back to Coder for retry
6. Loop: Max 3 retries, then escalate to human

SAFETY & CONTAINMENT

Why Safety Matters

Agents have real-world side effects:

- Send emails
- Execute code
- Make API calls (purchases, deletions)

An unconstrained agent is a liability.

Principle 1: Start Simple

"Build the RIGHT system, not the MOST SOPHISTICATED system."

- Optimize single LLM calls first
- Add agentic loops only when needed
- Reduce abstraction layers in production

Principle 2: Transparency

Show the user what the agent is thinking.

- Streaming: Display "Thought" steps in real-time
- Reasoning Trace: Log all tool calls and intermediate outputs
- Explainability: "I issued a refund because the policy allows it within 30 days."

Principle 3: Guardrails

Input Guardrails:

- Filter PII (credit cards, SSNs)
- Block toxic or adversarial prompts
- Detect jailbreak attempts

Output Guardrails:

- Verify response doesn't violate policy
- Check for hallucinations against source documents

Principle 4: Sandboxing

Code Execution Tools are Dangerous.

- Run in isolated containers (Docker, etc.)
- Limit network access
- Set resource quotas (CPU, memory, time)
- Disable sensitive syscalls

Principle 5: Human-in-the-Loop

Critical Actions Require Approval:

- Sending emails
- Committing code
- Making financial transactions

Implementation: Use checkpointing to pause, present action to human, resume on approval.

Principle 6: Evals & Monitoring

- Before Deployment:
  - Create a golden dataset (queries + expected outputs)
  - Use "LLM-as-a-Judge" to score accuracy, tool usage, tone
- After Deployment:
  - Monitor latency, error rates, tool call frequency
  - Log all interactions for audit and debugging

SUMMARY

Key Takeaways

- Agents vs. Workflows: When to use each
- Architectures: Single-Agent (ReAct) vs. Multi-Agent
- Frameworks: LangGraph (control) vs. CrewAI (speed)
- Patterns: ReAct, Chaining, Routing, Parallelization, Orchestrator-Workers, Evaluator-Optimizer, ReWOO
- Tools: JSON Schema, clear docstrings, fault tolerance
- Protocols: MCP (Agent-to-Tool), A2A (Agent-to-Agent)
- State: Short-term (context), Long-term (RAG/KV), Checkpointing
- Safety: Guardrails, Sandboxing, Human-in-the-Loop, Evals

Next Steps

- Run the Notebook: agent_orchestration_lecture.ipynb
  - See ReAct and ReWOO in action with live web search
- Experiment: Modify tools, prompts, and graph structure
- Build Your Own Agent: Pick a use case and apply these patterns

References

- Anthropic: Building Effective Agents
  https://www.anthropic.com/engineering/building-effective-agents
- IBM: What is Agentic Architecture?
  https://www.ibm.com/think/topics/agentic-architecture
- LangGraph Documentation
  https://langchain-ai.github.io/langgraph/
- CrewAI Documentation
  https://docs.crewai.com/
- MCP (Model Context Protocol)
  https://modelcontextprotocol.io/
- A2A (Agent-to-Agent Protocol)
  https://a2a-protocol.org/latest/

Thank you!

Rakshit Agrawal

Principal Applied Scientist, Microsoft

LinkedIn:
linkedin.com/in/rakshit-agrawal/