Analytics Vidhya | DATAHACK SUMMIT

Hack Session

Building Effective Agentic AI Systems

Lessons From the Field

Speaker

Dipanjan Sarkar

Head of Artificial Intelligence & Community, Analytics Vidhya

Google Developer Expert - ML & Cloud Champion Innovator

Published Author

Get Slides & Code Notebooks Here...

https://github.com/dipanjanS/building-effective-agentic-ai-systems-dhs2025

This session is inspired by...


Engineering at Anthropic

Building effective agents

Published Dec 19, 2024

We've worked with dozens of teams building LLM agents across industries. Consistently, the most successful implementations use simple, composable patterns rather than complex frameworks.

Over the past year, we've worked with dozens of teams building large language model (LLM) agents across industries. Consistently, the most successful implementations weren't using complex frameworks or specialized libraries. Instead, they were building with simple, composable patterns.

In this post, we share what we've learned from working with our customers and building agents ourselves, and give practical advice for developers on building effective agents.

Anthropic Research

Cohere

My experience building AI Agents for the last 2 years

Common Challenges in Building Agentic AI Systems

- What frameworks should I use to build AI Agents?
- How should I design and architect Agentic AI Systems?
- Single-agent vs Multi-agent?
- How do I integrate RAG with Agents (Agentic RAG)?
- How can I optimize my Agent's context (Context Engineering)
- To MCP or not to MCP?
- How do I monitor and evaluate AI Agents (Observability)?

What we will cover today...

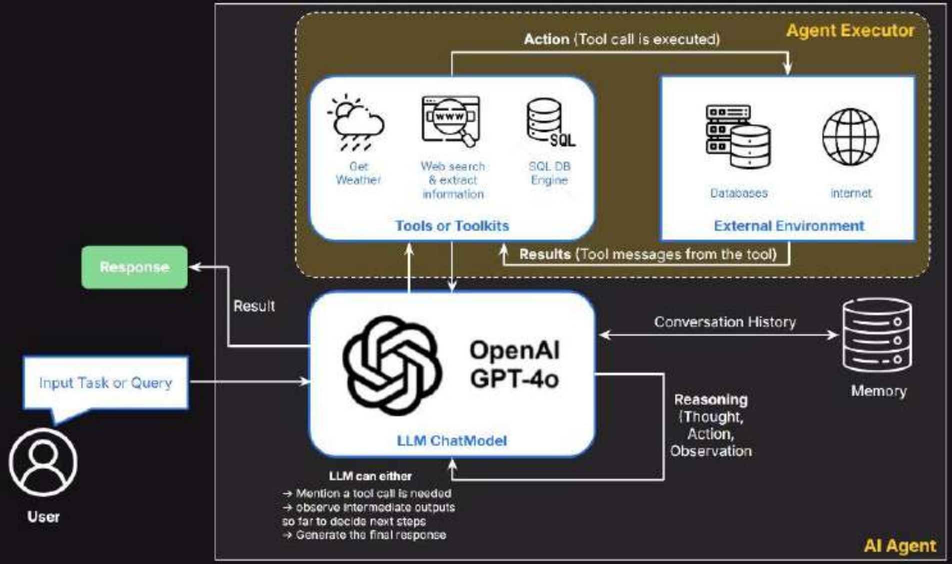
Architecting Agentic AI Systems

Optimizing Agentic AI Systems

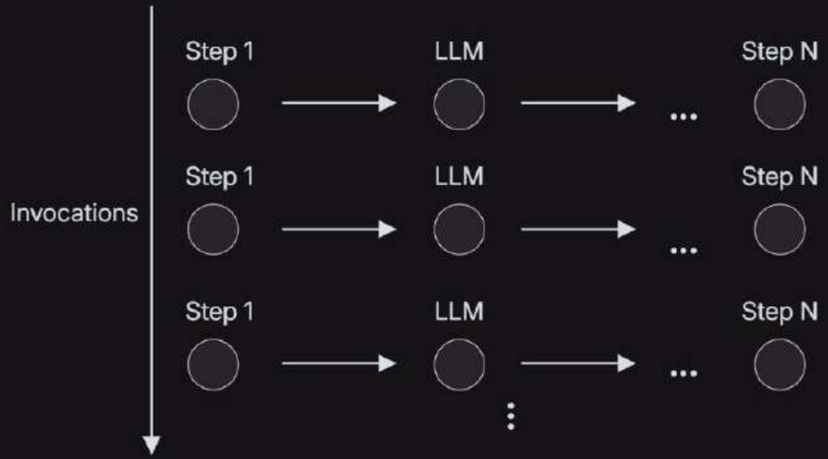
Observability for Agentic AI Systems

Key Components of an AI Agent

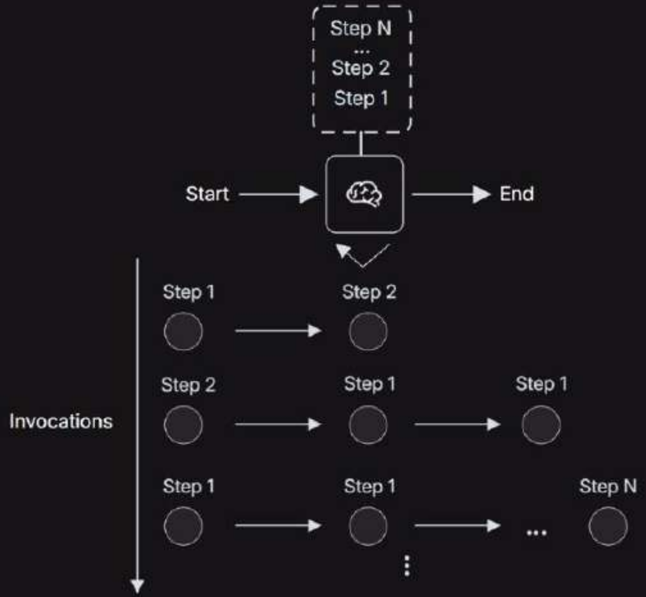
- LLM (Reasoning Engine)
- Planning Module (ReAct or Custom)
- Tools (Actions)
- External Knowledge Bases
- Memory



AI Workflows vs. AI Agents



AI workflows always execute the same flow



Agentic AI systems rely on the LLM to control the flow

Architecting Effective Agentic AI Systems

Popular Tools & Frameworks for Building Agentic AI Systems

LangGraph

crewai

smolagents

Google ADK

AG2

Semantic Kernel

My Personal Choice?

- LangGraph has strong Graph-based API and a functional API to build simple and complex agents with low-level control
- CrewAI makes building multi-agent systems really easy (now AG2 isn't far behind)
- Both of these frameworks have the highest adoption across various industry verticals (so far)

LangGraph

crewai

How Does LangGraph Build AI Agents?

```python
tavily_search = TavilySearchAPIWrapper()

@tool
def search_web(query: str) -> list:
    """Search the web for a query."""
    # Perform web search on the internet
    results = tavily_search.raw_results(
        query=query,
        max_results=5,
        include_raw_content=True
    )
    docs = results['results']
    return docs
```

Tools

```python
# define Agent State
class State(TypedDict):
    messages: Annotated[list, add_messages]

# bind tools to LLM
tools = [web_search]
llm = ChatOpenAI(model='gpt-4o')
aug_llm = llm.bind_tools(tools)

# create tool node function
tool_node = ToolNode(tools=tools)

# create LLM node function
def llm_node(state: State) -> State:
    SYS_PROMPT = '.....' # system instructions
    current_state = state["messages"]
    state_with_prompt = SYS_PROMPT + current_state
    response = [aug_llm.invoke(current_state)]
    # update agent state
    return {"messages": response}
```

Node Functions

Agent Graph

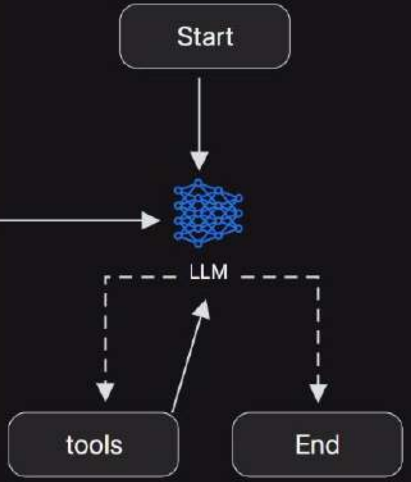
```python
# Build the graph
graph = StateGraph(State)

# Add nodes
graph.add_node("llm", llm_node)
graph.add_node("tools", tool_node)

# Add edges
graph.add_edge(START, "llm")
# Conditional tool call or generate response
graph.add_conditional_edges(
    "tool_calling_llm",
    tools_condition,
    ["tools", END]
)
graph.add_edge("tools", "tool_calling_llm")

# Compile Agent Graph
agent = graph.compile()
```

ReAct Agent



How Does LangGraph Execute AI Agents?

1

Initial State

```
state: {
  messages: [('user', 'What are latest quantum
  developments?')]
}
next: Node LLM
checkpoint_id: abc123
thread_id: t001
```

2

Tool Call Request

```
state: {
  messages: [('user', 'What are latest quantum
  developments?'),
  ('assistant', 'I'll search for quantum
  info.')]
}
Tool Request: web_search
Params: {"query": "quantum 2025"}
next: Tool Execution
checkpoint_id: def456
thread_id: t001
```

3

Tool Response

```
state: {
  messages: [('user', 'What are latest quantum
  developments?'),
  ('assistant', 'I'll search for quantum
  info.'),
  ('tool', 'Found 10 articles...')]
}
Tool Response: search_results
Content: IBM quantum, Google advances...
next: Node LLM
checkpoint_id: ghi789
thread_id: t001
```

4

LLM Processing

```
state: {
  messages: [('user', 'What are latest quantum
  developments?'),
  ('assistant', 'I'll search for quantum
  info.'),
  ('tool', 'Found 10 articles...'),
  ('assistant', 'Key quantum developments...')]
}
next: Human Review
checkpoint_id: jkl012
thread_id: t001
```

5

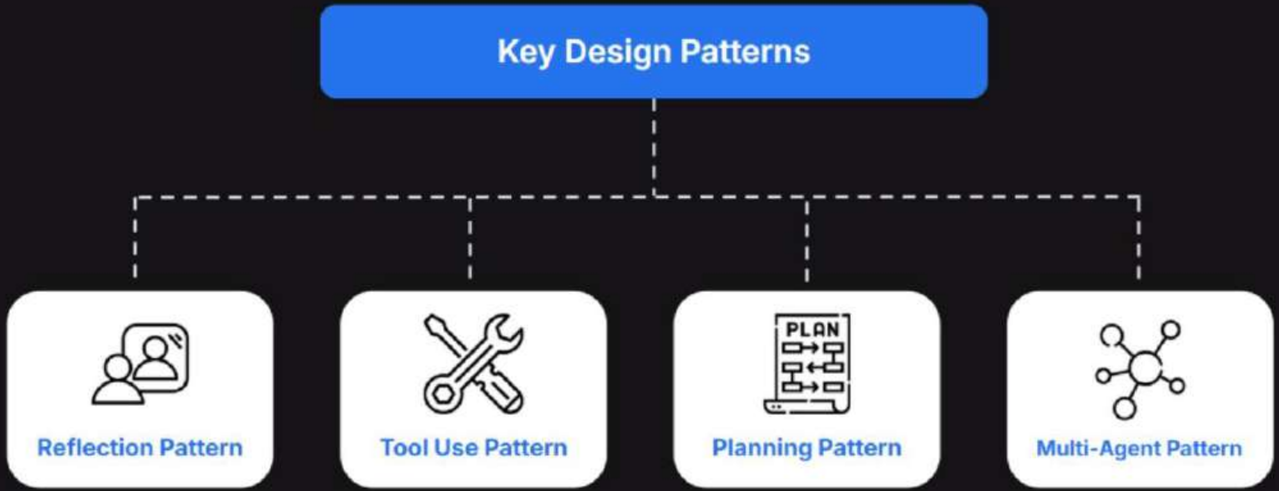
Final State

```
state: {
  messages: [('user', 'What are latest quantum
  developments?'),
  ('assistant', 'I'll search for quantum
  info.'),
  ('tool', 'Found 10 articles...'),
  ('assistant', 'Key quantum developments...'),
  ('user', 'Tell me about IBM?')]
}
next: Node LLM
checkpoint_id: mno345
thread_id: t001
```

Key Design Patterns for Agentic AI

Almost a year ago, Andrew Ng defined four design patterns recognizable in Agentic AI Systems

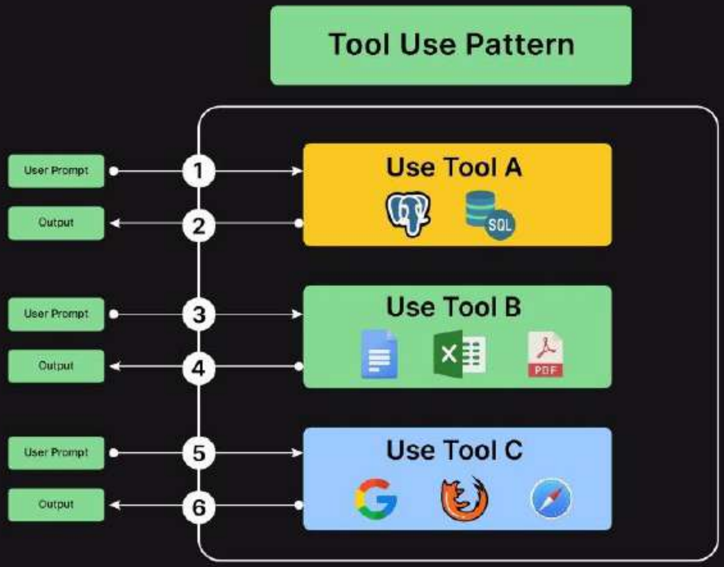
Key Design Patterns for Agentic AI



Recommendations for using the Tool Use Pattern

Enables AI Agents to interact with external tools, APIs, and resources for improved functionality and context to support their reasoning.

- These systems can easily handle ~10 tools.
- Can also handle multi-step and multi-tool call executions (ReAct is built-in)
- Best Practices:
  - Well-defined tool schemas for accurate function calling
  - Well-structured System Prompt with detailed instructions
  - Powerful LLMs already trained for function (tool) calling

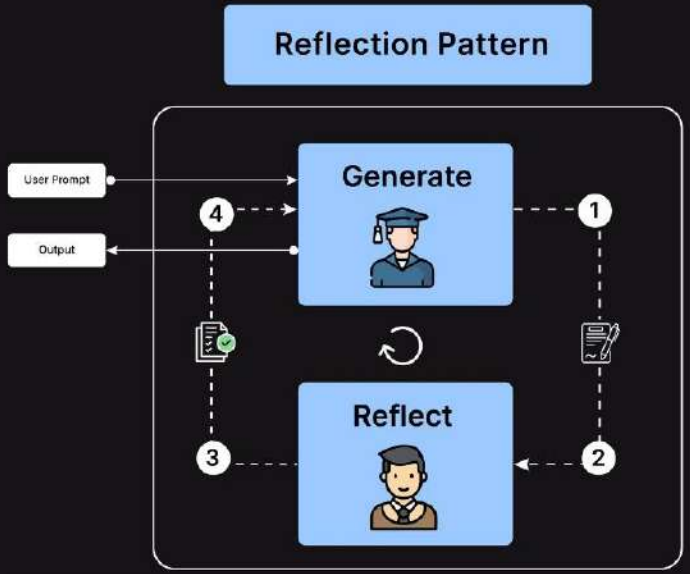


Source: https://theneuralmaze.substack.com/

Recommendations for using the Reflection Pattern

Enables AI Agents to alternate between generating and critiquing for iterative improvement of the generated response.

- Define **clear evaluation criteria** and use only when **iterative refinement** provides **measurable value**.
- Examples:
  - Judging and grading the quality of an LLM response
  - Validating specific guidelines e.g, claims processing
- Best Practices:
  - Use a powerful LLM as a Judge (avoid SLMs)
  - Create well-defined prompts for judging (prefer categories to ranges)
  - Have a max iterations cutoff to prevent infinite loops, besides clear stopping criteria

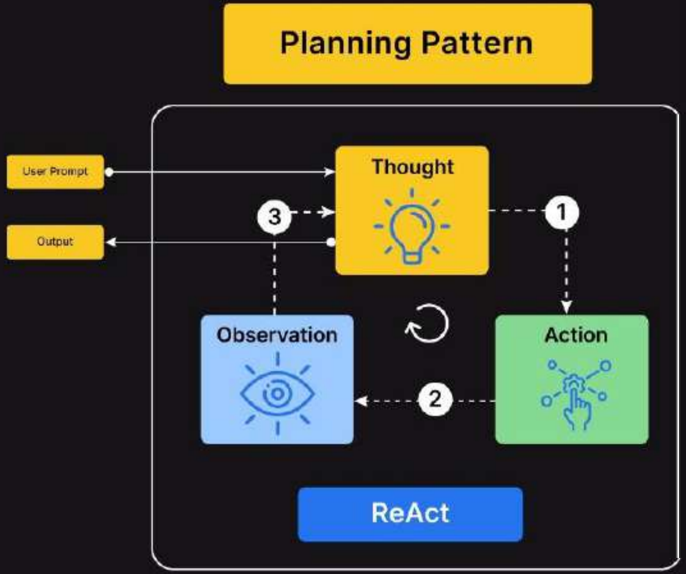


Source: https://theneuralmaze.substack.com/

Recommendations for using the Planning Pattern

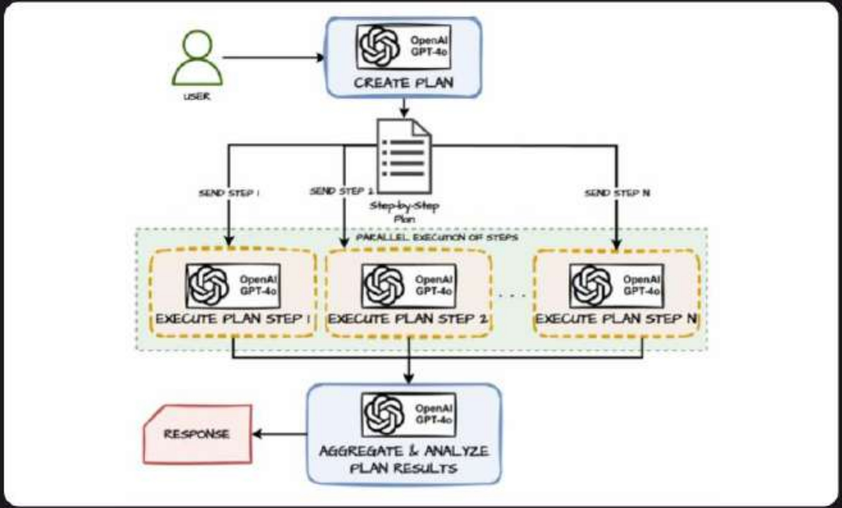
Structures and executes multi-step tasks through reasoning & planning.

- Most ReAct Agents already have planning built-in so first start with simple ReAct Agents
- Best Practices:
  - For more complex tasks, consider adding additional custom planning modules
  - Planning modules or patterns are typically:
    - Static Planners with Parallel Task Execution & Synthesis
    - Dynamic Planners with Task Execution, Reflection & Replanning



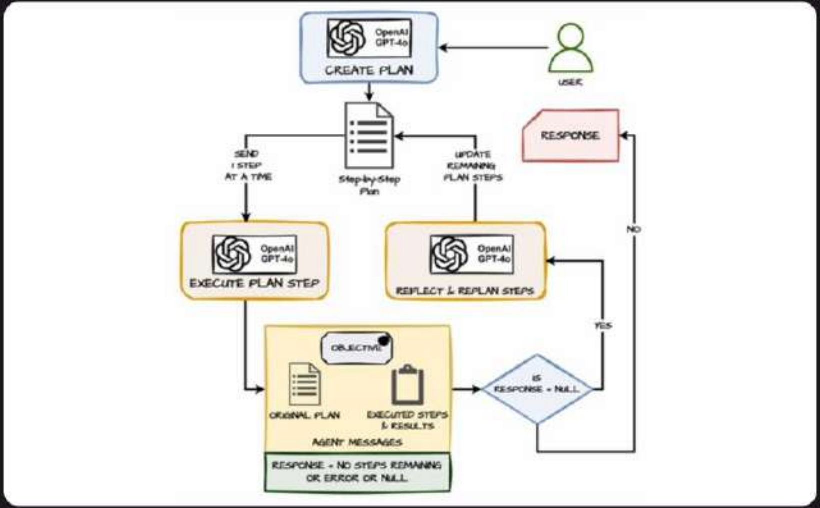
Source: https://theneuralmaze.substack.com

Custom Planning Patterns



Static Planners

- Break down a task into multiple steps
- Execute all steps in parallel
- Synthesize results from all steps and generate final response (map-reduce)
- Useful when steps do not have dependencies



Dynamic Planners

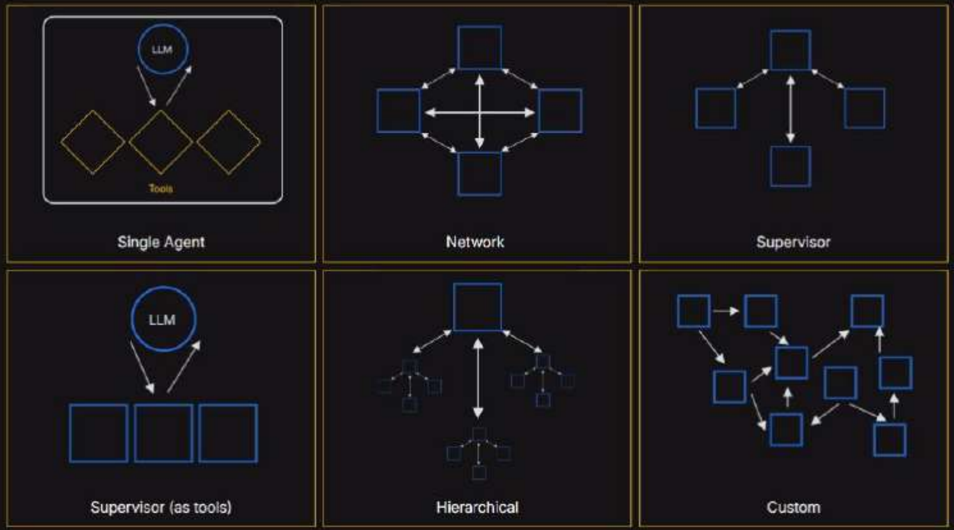
- Break down a task into multiple steps
- Executes one step at a time
- Reflect and replan remaining steps if needed
- Synthesize results from all steps and generate final response
- Useful when steps have dependencies

Recommendations for using the Multi-Agent Pattern

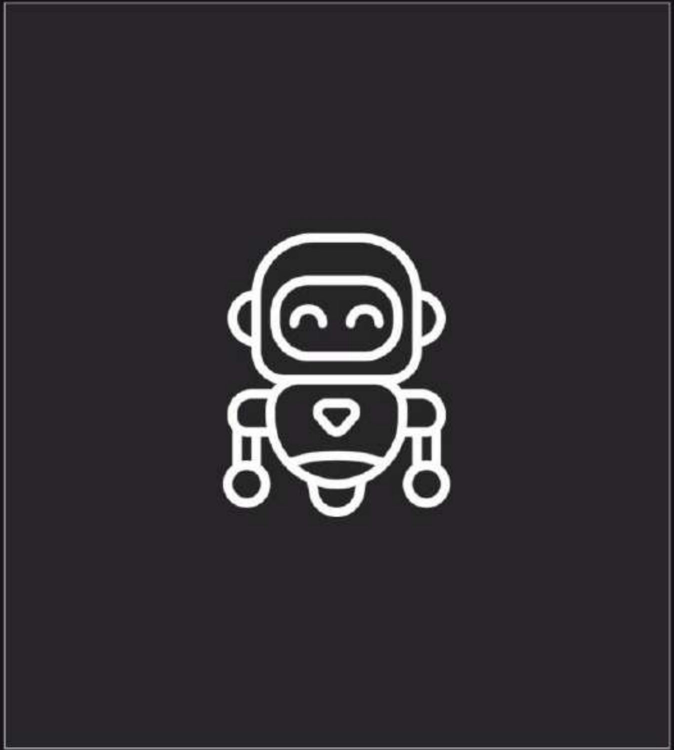
Enables multiple AI Agents to solve complex problems through communication and coordination.

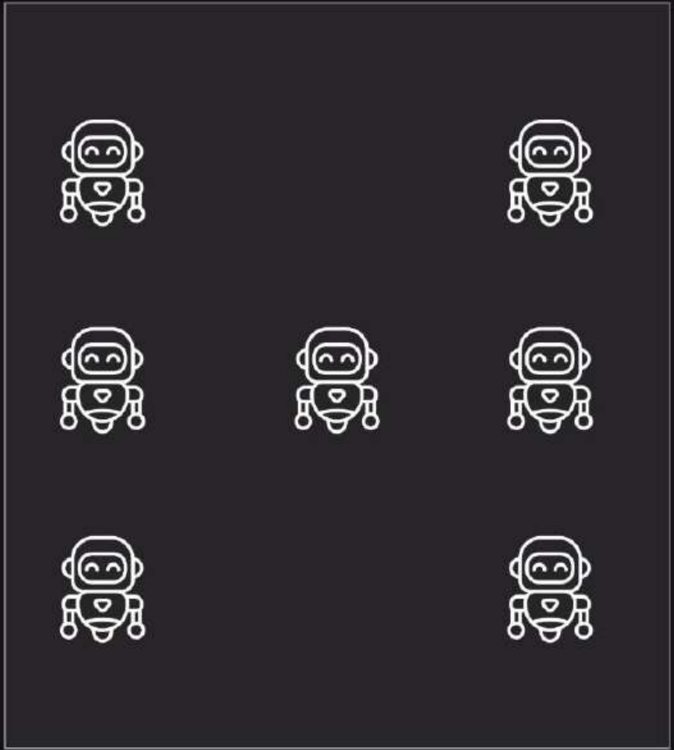
- Common architecture patterns:
  - **Network**: Each agent can communicate with every other agent.
  - **Supervisor**: Each sub-agent communicates with a single supervisor agent, which makes decisions.
  - **Hierarchical**: Multi-agent system with a supervisor of supervisors.

- Best Practices:
  - Always start with a simple supervisor or network architecture, and then expand
  - Create separate agents based on specific processes, tasks, tools, and flows



Single or Multi-Agent System?





Let's look at a Real-World Use-Case

Utilization Review

Utilization review is the process of evaluating patient medical procedures to ensure they are necessary, appropriate, and aligned with clinical guidelines and insurance coverage policies.

Retrieve Patient Records	Fetch Guidelines	Perform Eligibility Check	Make Decision

Patient Summary

Patient ID: P101

Age / Sex: 38 / Male

Symptoms: Abdominal pain, nausea

Diagnosis: Possible early appendicitis

Procedure: CT Abdomen

Notes: Mild abdominal pain and nausea, but no localized tenderness or rebound noted

Matched Guideline

Procedure: CT Abdomen

Diagnosis: Suspected Appendicitis

Required Symptoms:

- Abdominal pain, nausea, RLQ tenderness

Notes: CT imaging justified if appendicitis is unclear

Guideline Validity Result

Symptoms Present: Abdominal pain, nausea

Missing Symptom: RLQ tenderness (not observed)

Clinical Notes: Mild abdominal pain and nausea, no localized tenderness or rebound

Conclusion: Procedure does **not** qualify as medically necessary, since the required RLQ tenderness is absent

Care Recommendation

Diagnosis: Suspected Appendicitis

Recommendation: Do CT to confirm and refer for surgery if positive

Explanation: Imaging (CT scan) is used to confirm suspected appendicitis. If positive, surgery (appendectomy) is needed to prevent complications such as perforation or abscess.

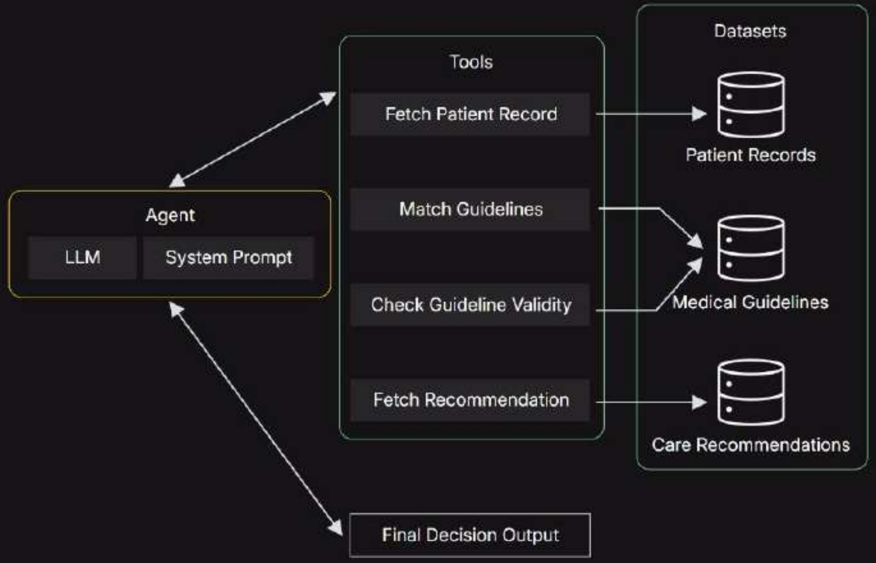
Final Decision

Decision: NEEDS REVIEW

Reasoning: Symptoms align with two of three required, but absence of RLQ tenderness means the procedure does not meet medical necessity criteria

Recommendation: Further evaluation and monitoring; consider alternative imaging or observation before CT

Single-Agent vs. Multi-Agent Architecture



Single-Agent Architecture

Multi-Agent Architecture

Hands-On Demo: Single-Agent vs. Multi-Agent Architecture

- Get the notebook from [HERE]
- Recommended to run on Google Colab

Single-Agent vs. Multi-Agent

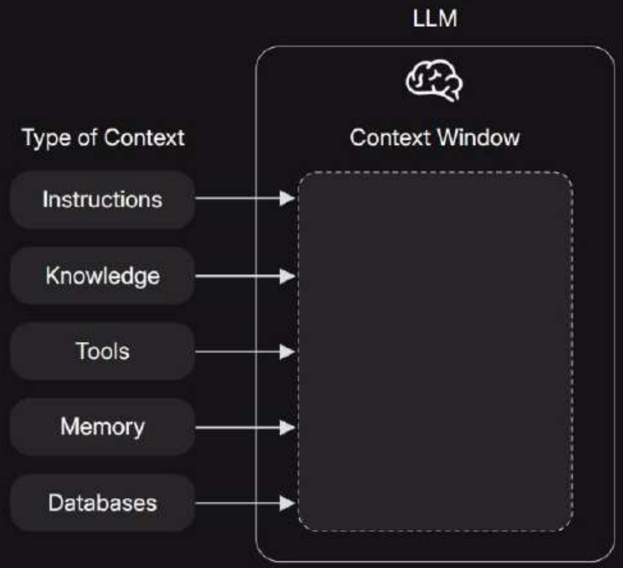
Criteria	Single-Agent	Multi-Agent
Tokens per Execution	~6K	~12K
Latency	~19s	~40s
Cost per Execution	~$0.001	~$0.0025
Best For	Straightforward processes	Complex processes with sub-processes
Scalability with Tools	Works well for ~10 tools	Recommended for >10 tools
Ease of Extension	Difficult to extend to new tasks	Easy to extend with new agents
Observability	Easy to trace and debug	Harder to trace and debug
Modularity	Low	High
Reusability	Intermediate	High across similar task agents

Optimizing Agentic AI Systems

What is Context Engineering?

Context Engineering is the delicate art and science of filling the context window with just the right information for the next step - Andrej Karpathy

- For AI Agents, context includes:
  - Instructions – prompts, few-shot examples, tool descriptions
  - Knowledge – facts, memories
  - Tools – feedback from tool calls
  - Memory - agent state, conversations
  - Databases - enterprise knowledge
- We will look at the following patterns for Context Engineering:
  - Agentic RAG as a way to infuse enterprise knowledge into Agents
  - MCP as a standardized tool-calling protocol for improving agent context
  - Memory management techniques for longer and better context retention



Complete Deck is in the Repo here:

https://github.com/dipanjanS/building-effective-agentic-ai-systems-dhs2025