

Hack Session

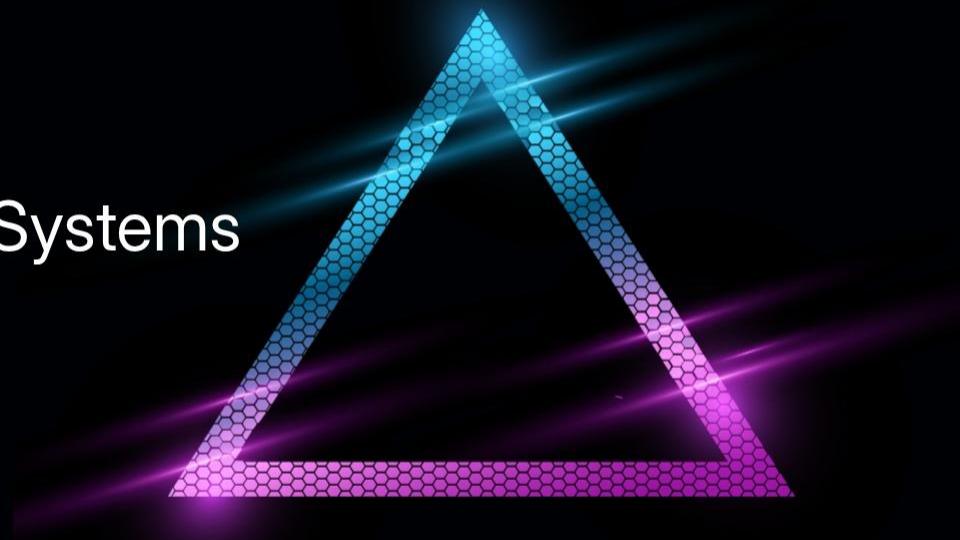
Building Effective Agentic AI Systems

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://bit.ly/agents_dhs25

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



Building Enterprise AI Agents

A comprehensive blueprint for developing scalable AI agents in regulated industries, including finance, healthcare, manufacturing, energy, and government.

cohere

SECURE AI FOR BUSINESS COHERE.COM

Cohere



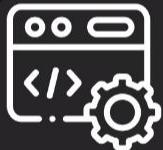
My experience building AI Agents
for the last 2 years

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



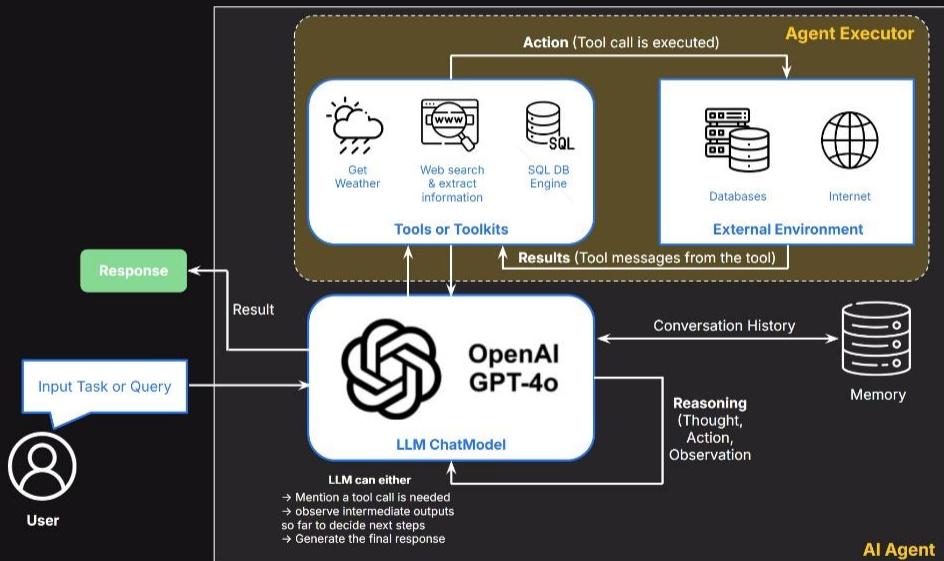
Optimizing Agentic AI Systems



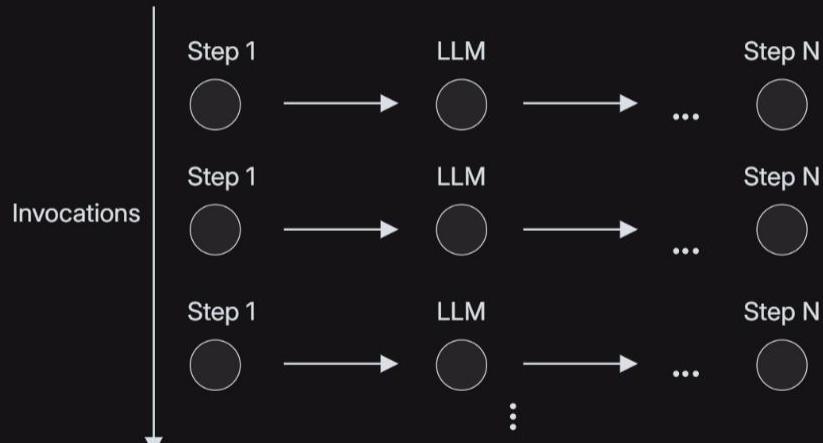
Observability for Agentic AI Systems

Key Components of an AI Agent

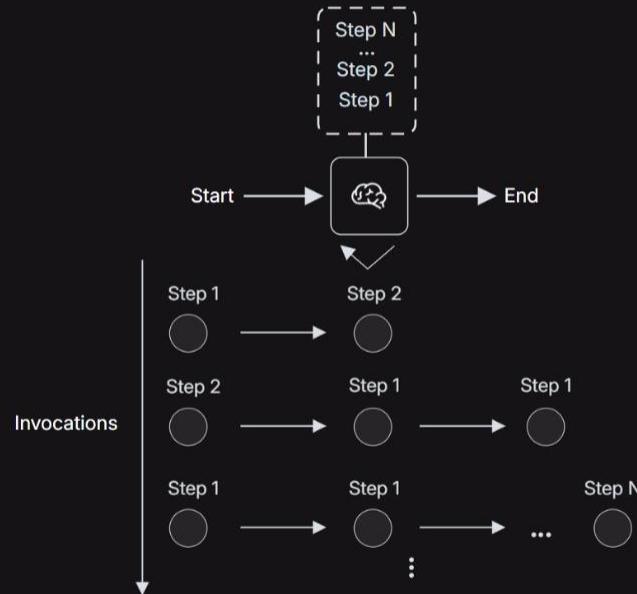
- LLM (Reasoning Engine)
- Planning Module (ReAct or Custom)
- Tools (Actions)
- External Knowledge Bases
- Memory



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



smolagents



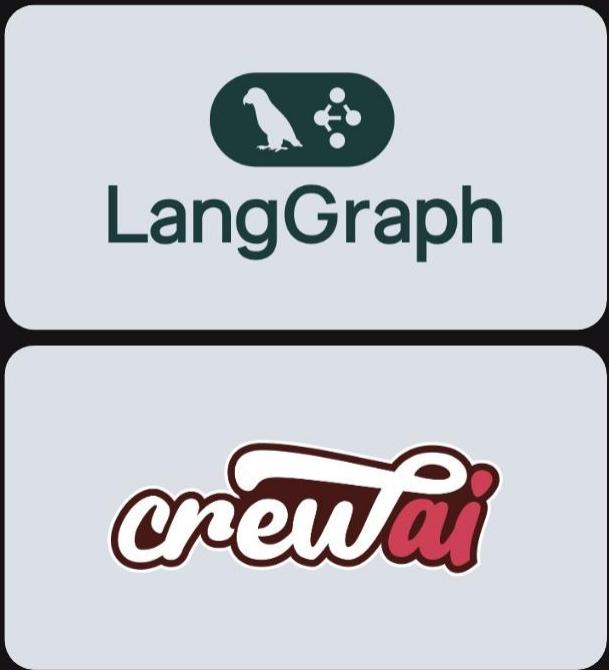
Google ADK



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)



How Does LangGraph Build AI Agents?

```
●●●  
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

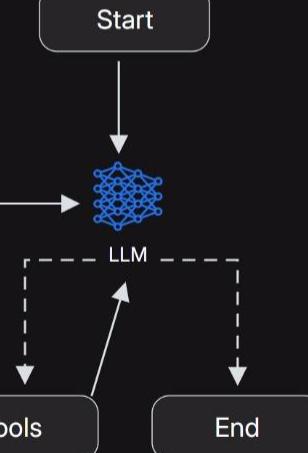
```
●●●  
# define Agent State  
class State(TypedDict):  
    messages: Annotated[list, add_messages]  
  
# bind tools to LLM  
tools = [web_search]  
llm = ChatOpenAI(model='gpt-4o')  
aug_llm = llm.bind_tools(tools)  
  
# create tool node function  
tool_node = ToolNode(tools=tools)  
  
# create 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

```
●●●  
# 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?

Initial State

```
1
state: {
  messages: [('user', 'What are latest quantum
  developments?')]
}
next: Node LLM
checkpoint_id: abc123
thread_id: t001
```

Tool Call Request

```
2
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
```

Tool Response

```
3
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
```

LLM Processing

```
4
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
```

Final State

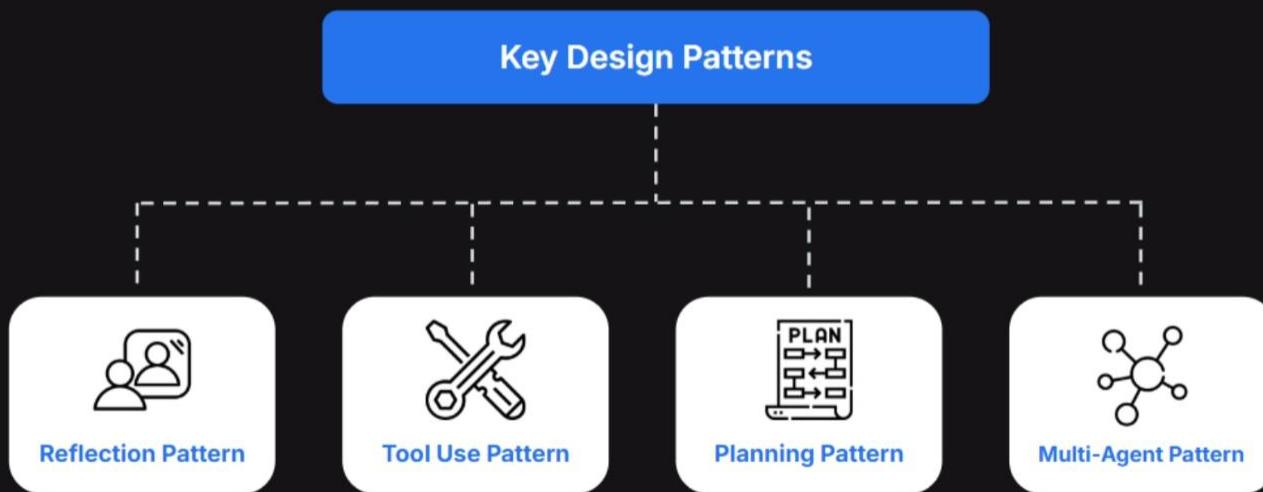
```
5
state: {
  messages: [('user', 'What are latest quantum
  developments?'),
    ('assistant', 'I'll search for quantum
  info.'),
    ('tool', 'Found 10 articles...'),
    ('assistant', 'Key quantum developments...'),
    ('user', 'Tell me about IBM?')]
}
next: Node LLM
checkpoint_id: mno345
thread_id: t001
```

Key Design Patterns for Agentic AI

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



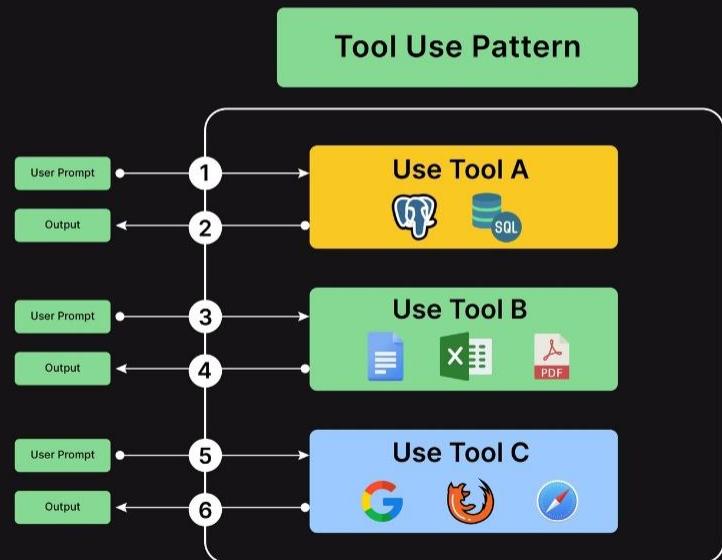
Key Design Patterns for Agentic AI



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

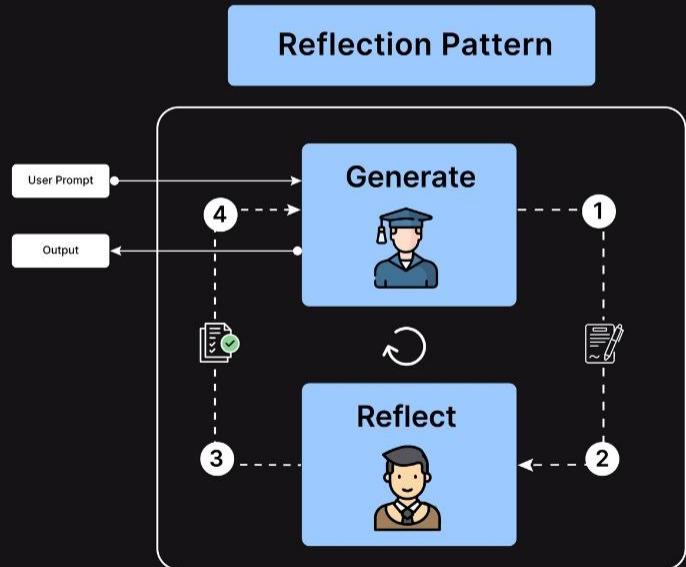


Source: <https://theneuralmaze.substack.com/>

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

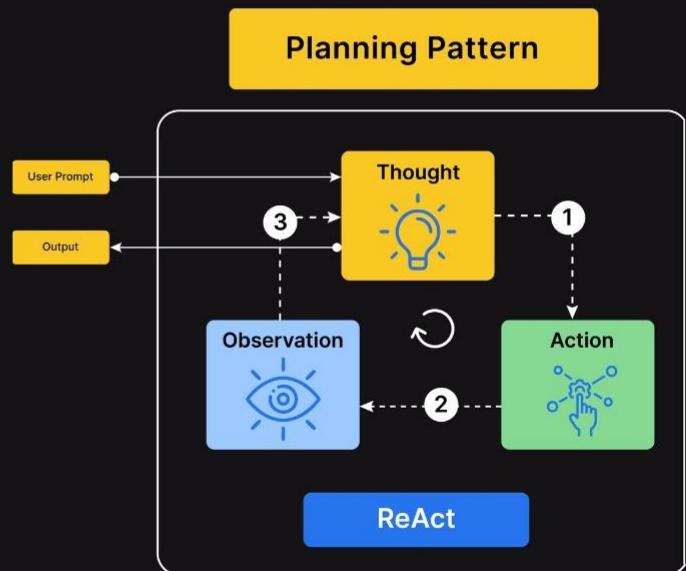


Source: <https://theneuralmaze.substack.com/>

Recommendations for using the Planning Pattern

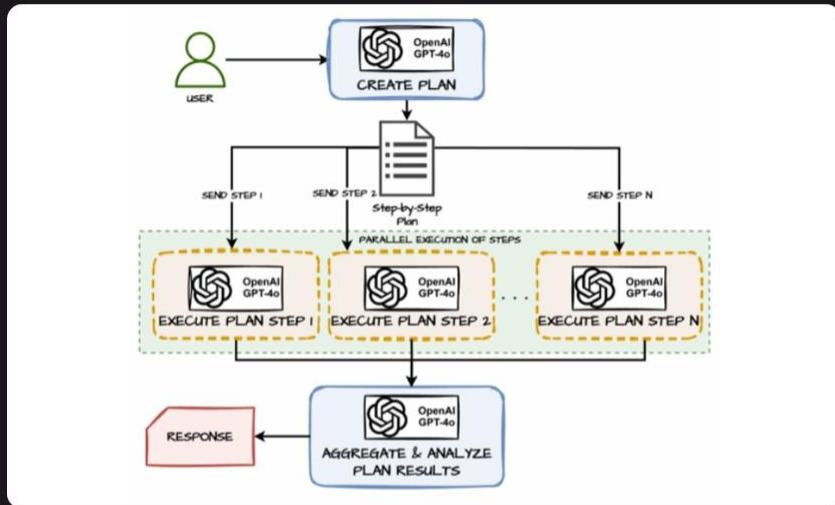
Structures and executes multi-step tasks through reasoning & planning.

- 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



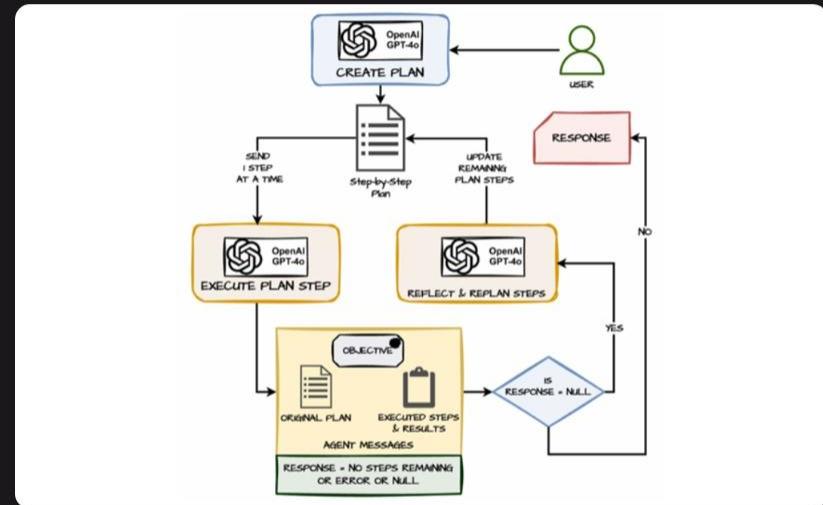
Source: <https://theneuralmaze.substack.com>

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



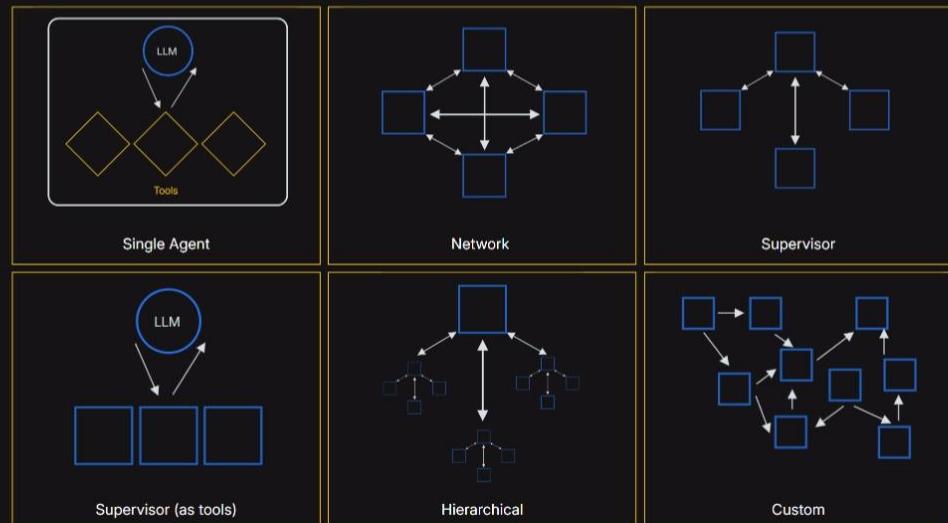
Dynamic Planners

- 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

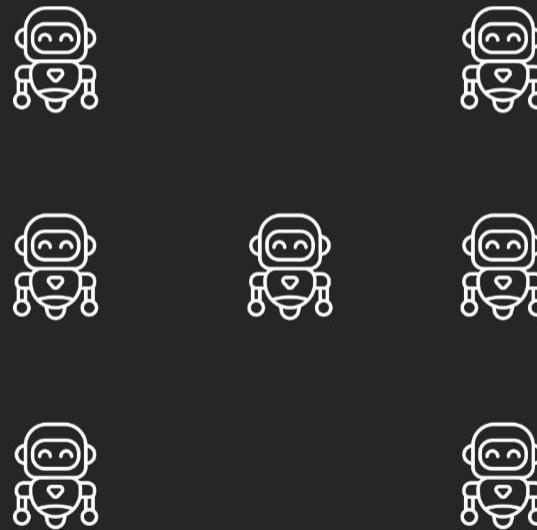
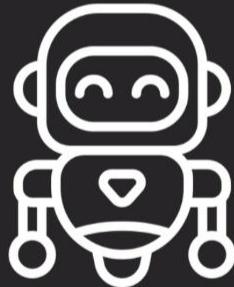
Recommendations for using the Multi-Agent Pattern

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



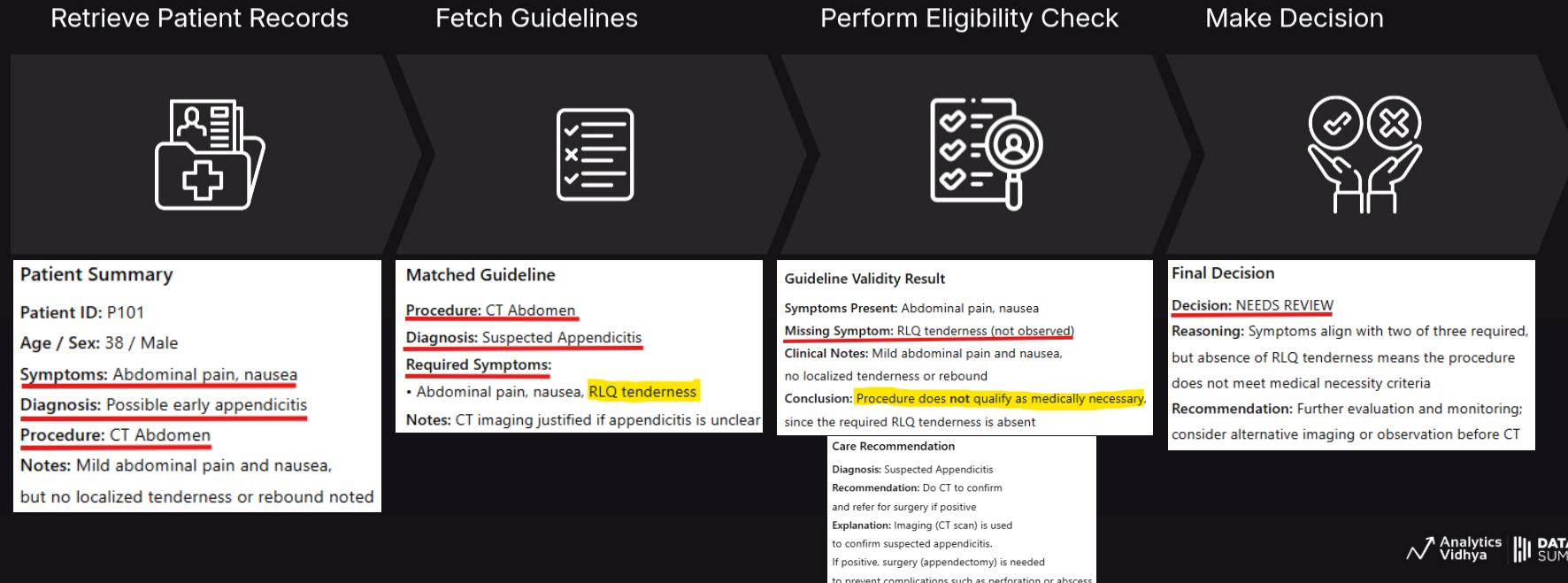
Single or Multi-Agent System?



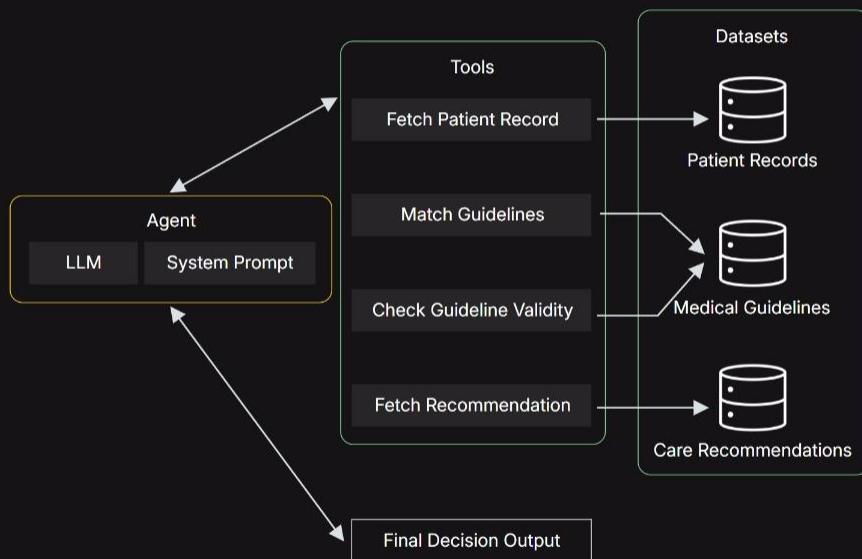
Let's look at a Real-World Use-Case

Utilization Review

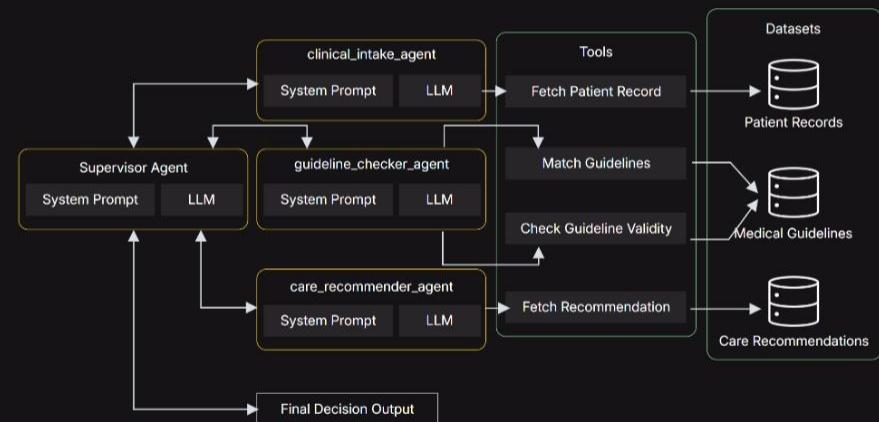
Utilization review is the process of evaluating patient medical procedures to ensure they are necessary, appropriate, and aligned with clinical guidelines and insurance coverage policies.



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

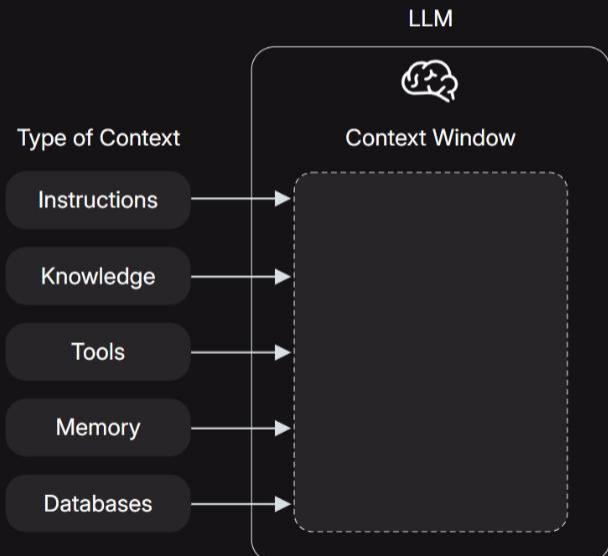
Criteria	Single-Agent	Multi-Agent
Tokens per Execution	~6K	~12K
Latency	~19s	~40s
Cost per Execution	~\$0.001	~\$0.0025
Best For	Straightforward processes	Complex processes with sub-processes
Scalability with Tools	Works well for ~10 tools	Recommended for >10 tools
Ease of Extension	Difficult to extend 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

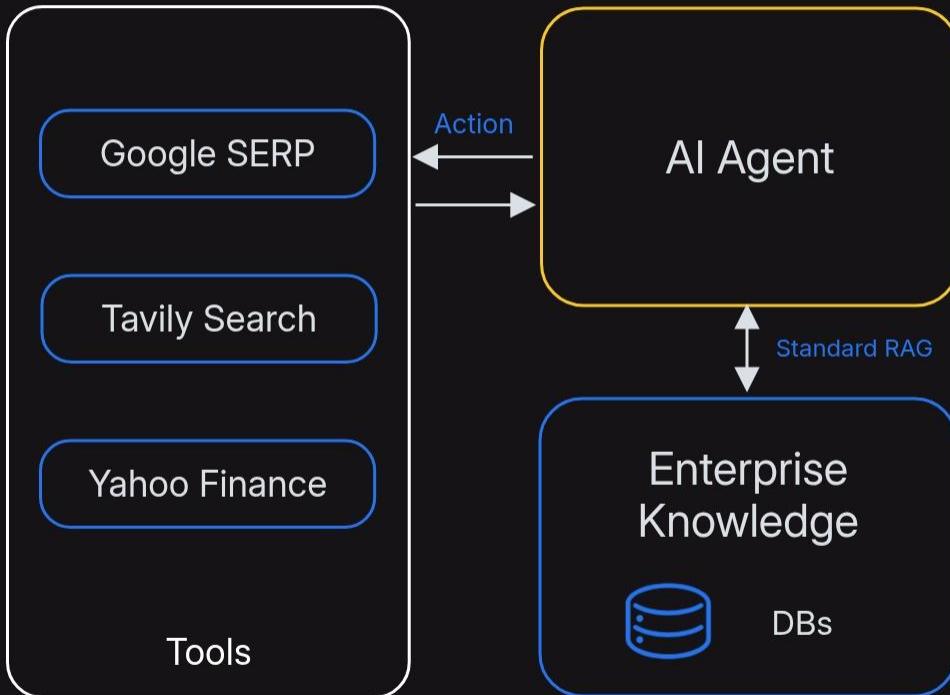
What is Context Engineering?

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



RAG + AI Agents = Agentic RAG

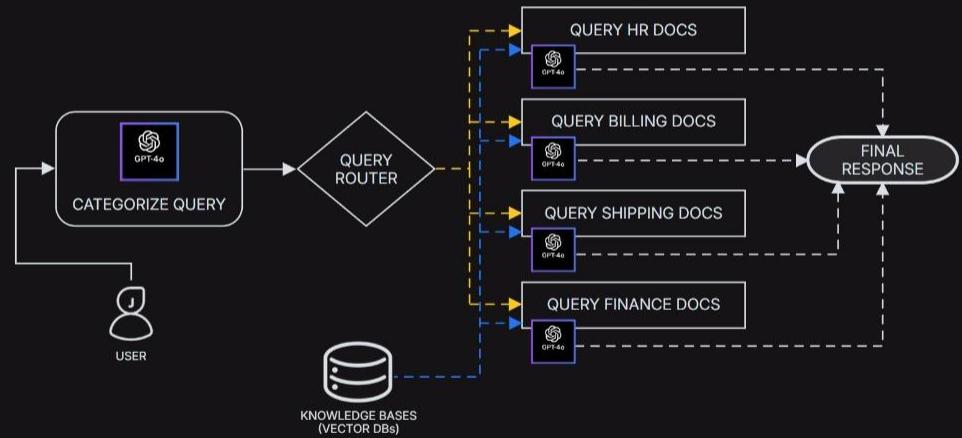


- Agentic RAG is a combination of AI Agents and RAG Systems
- AI Agents helps in reasoning, planning, calling tools
- RAG helps the Agent in getting contextual data from enterprise knowledge bases
- Examples include Router RAG, Self-Reflective RAG and more

Router Agentic RAG Systems = Routing Agents + RAG

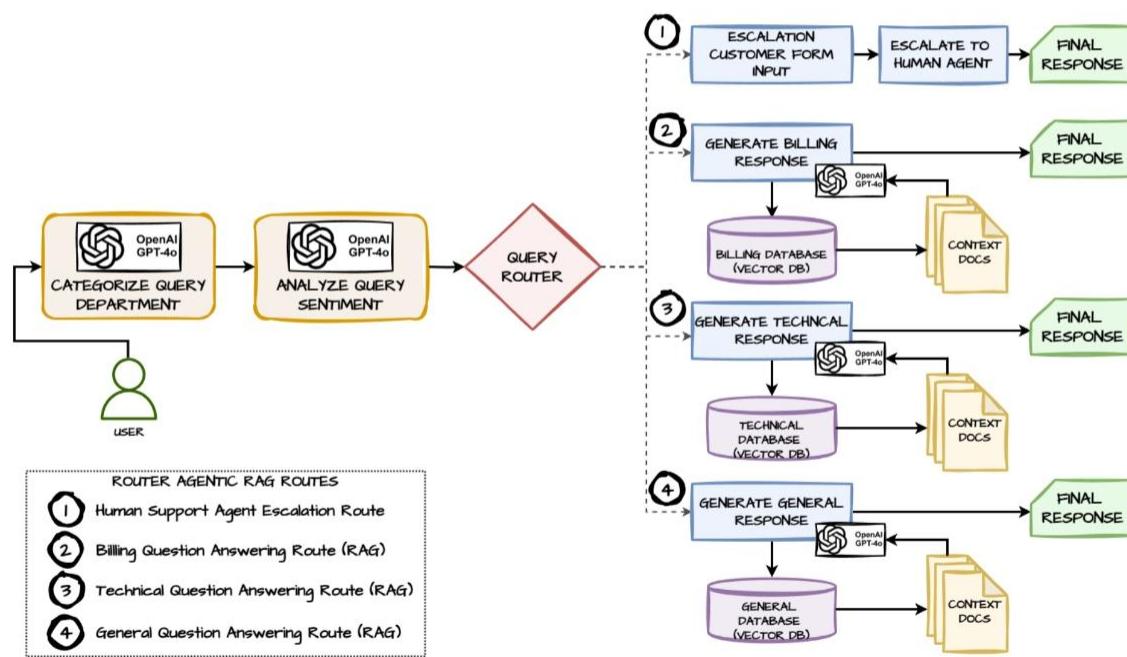
The main components of this system include:

- LLM-routing mechanisms to route user queries to specific workflows or sub-agents
- Each sub-agent or workflow handles queries of a specific type using RAG



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

Customer Support Automation



Hands-On Demo: Customer Support Automation - Router RAG Agent

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



What is MCP?

Model Context Protocol (MCP) is an open standard introduced by Anthropic to standardize how AI applications (chatbots, IDE assistants, or custom agents) connect with external tools, data, and prompts

- Think of MCP as the USB-C for AI integrations:

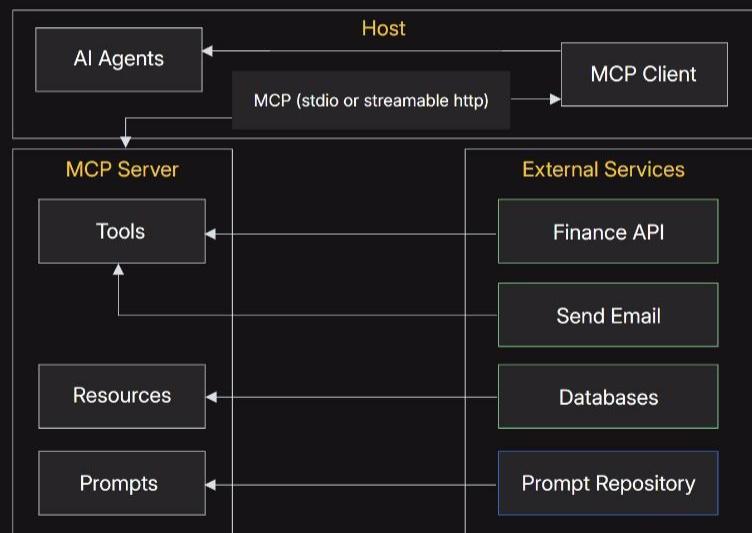
M (applications) and N (tools) do not need an MxN integration, with MCP it is now a M + N problem

- MCP defines a client-server architecture where:

- Host – Your local system or servers where apps, agents, and chatbots run
- MCP Client – Runs along with your apps and agents on your host
- MCP Server – Exposes Tools, Resources, and Prompts to the host client using the model context protocol (MCP)

- MCP goes beyond just standardizing tool-calling, use it for:

- **Access tools** to empower agent actions
- **Access data** from various sources
- **Access prompts** and instructions for specific tasks



Do we really need MCP?

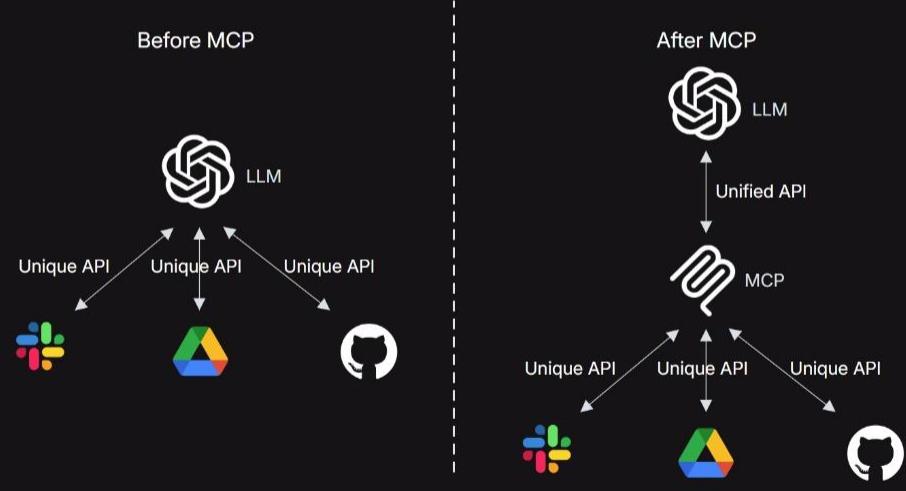
MCP has been hyped up a lot over the last few months, is it worth the hype?

- Direct Consumers

- Super useful as every major data or service provider now provides ready-made MCP servers for you to connect your apps and start using their services and tools

- Enterprise Users & Builders

- It depends on how you use it:
 - ✗ If you are building your own tools & data APIs to connect with your apps and agents, MCP is pointless
 - ✓ If you want to expose your tools & APIs to other departments and users, then it is useful
 - ✗ Use 3rd-party MCP servers with caution, considering security and regulatory implications



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

MCP Multi-Department Architecture

- Use Case: Cross-Department Auditing

- Scenario:

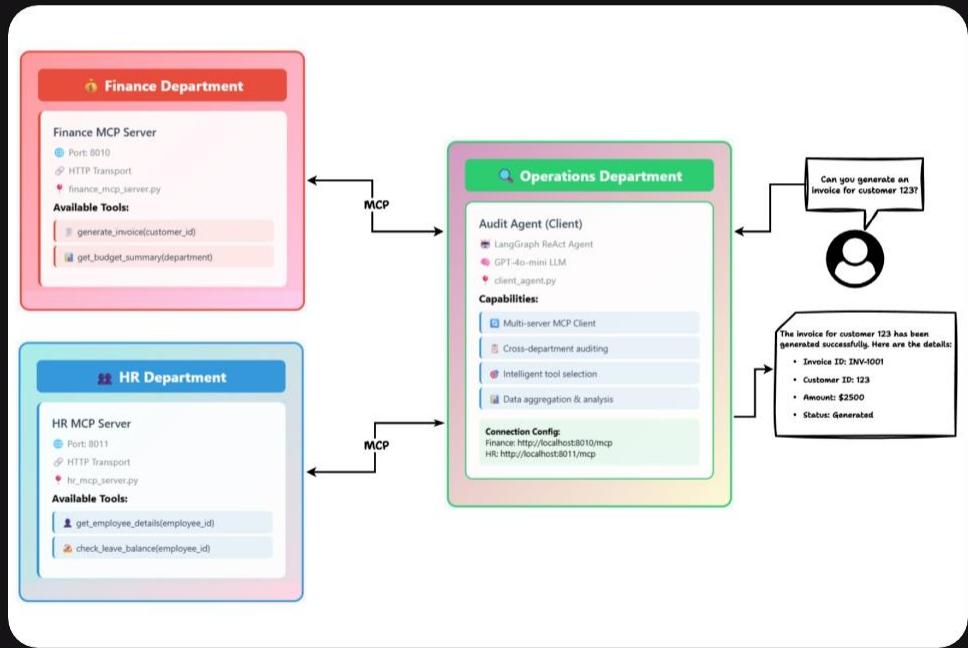
The Operations department needs to perform regular audits across multiple departments without needing to enable direct access to their internal systems.

- Solution:

Multi-Department MCP Architecture where each department enables access to their data and tools via MCP to the client agent in Operations department

- Benefits:

Centralized auditing, maintained security boundaries, standardized tool interfaces, and scalable multi-department integration.

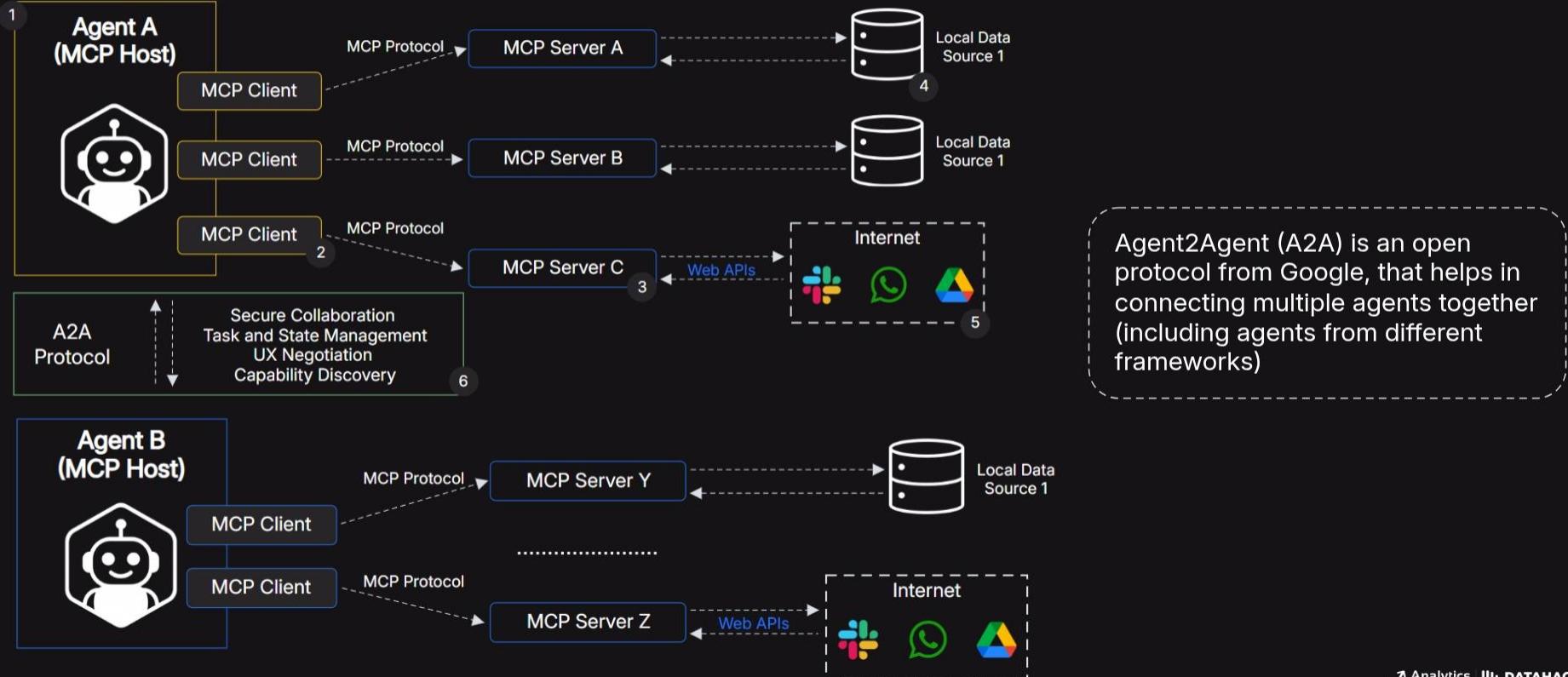


Hands-On Demo: Multi-Department MCP Architecture for AI Agents

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



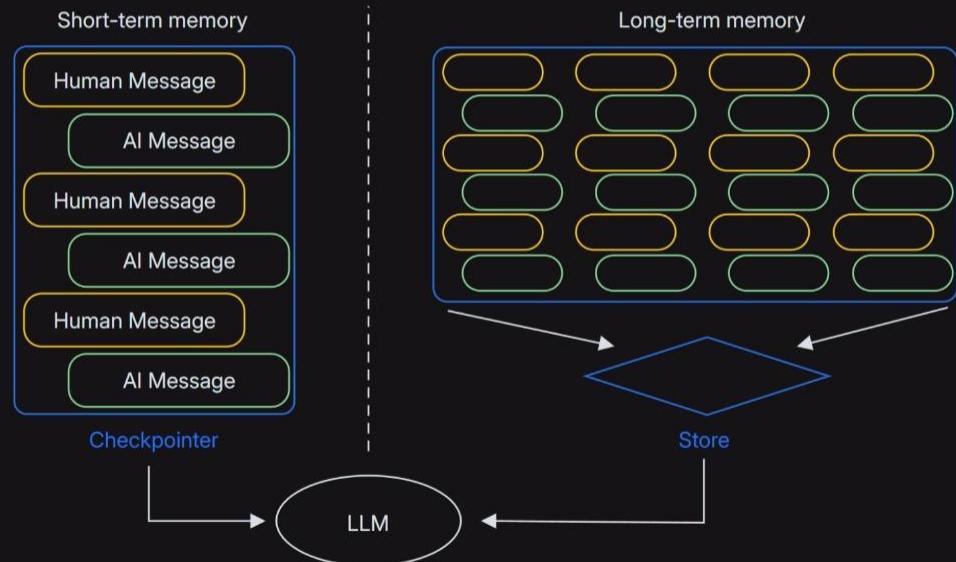
Future Scope - MCP + A2A



Memory Management for Agentic AI Systems

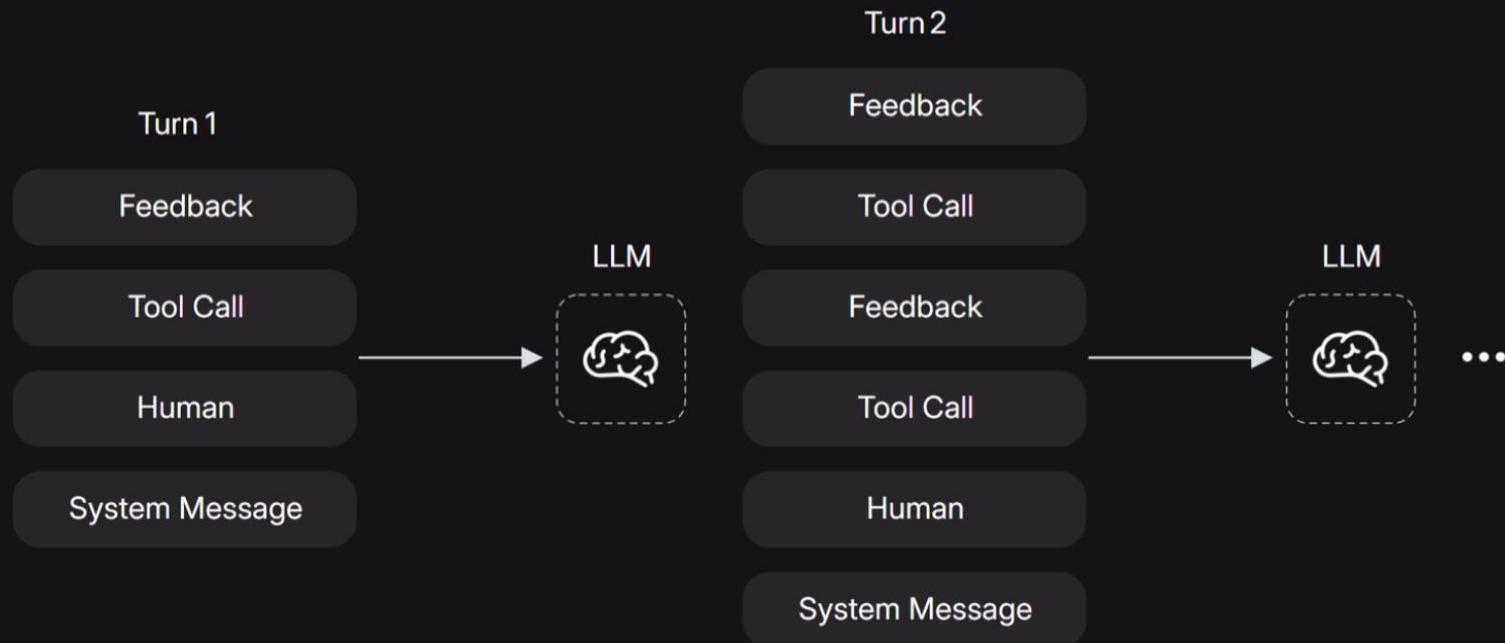
Memory enables Agentic AI Systems to have better interactions by leveraging past conversations, data, tool call results, and personalized preferences.

- Memory is a misnomer - Agents or LLMs can't really remember like humans!
- It is context engineering, to stuff past information into the context window of the Agent's LLM for future interactions
- Two major types of memory:
 - **Long-Term:** Storing all interaction information and transforming into useful artifacts like user preferences
 - **Short-Term:** Storing recent interaction information like conversations, tool calls, and more



Why care about Memory Management for Agents?

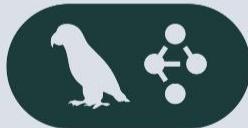
Agent context keeps increasing due to storing past interactions in memory. This can easily lead to the LLM hitting the dreaded 'Max Context Window Limit Exceeded' error!



Popular Tools & Frameworks for Building Agentic AI Systems



memØ



LangGraph

langchain-ai/
langmem



A 12
Contributors

G 258
Used by

D 12
Discussions

☆ 951
Stars

V 110
Forks

Real-World Example

Long-Term Memory Management for Personalized Agents

LangMem has tools to transform long-term agent memory interactions into well-defined user-profiles or records to enable personalized experiences

```
from langmem import create_memory_manager
from pydantic import BaseModel

# define schema to store user preferences
class UserProfile(BaseModel):
    """Save the user's preferences."""
    name: str
    preferred_name: str
    response_style_preference: str
    special_skills: list[str]
    other_preferences: list[str]

manager = create_memory_manager(
    "openai:gpt-5",
    schemas=[UserProfile],
    instructions="Extract user preferences and settings",
    enable_inserts=False,
)

# Extract user preferences from agent conversations
# e.g conversation = # get interactions from the agent
conversation = [
    {"role": "user", "content": "Hi! I'm Alex but please call me Lex. I'm a wizard at Python and love making AI systems that don't sound like boring corporate robots 😑"}, {"role": "assistant", "content": "Nice to meet you, Lex! Love the anti-corporate-robot stance. How would you like me to communicate with you?"}, {"role": "user", "content": "Keep it casual and witty - and maybe throw in some relevant emojis when it feels right ✌️ Also, besides AI, I do competitive speedcubing!"}, {"role": "assistant", "content": "I'm here to help! Let's have a fun and informative conversation."}
]

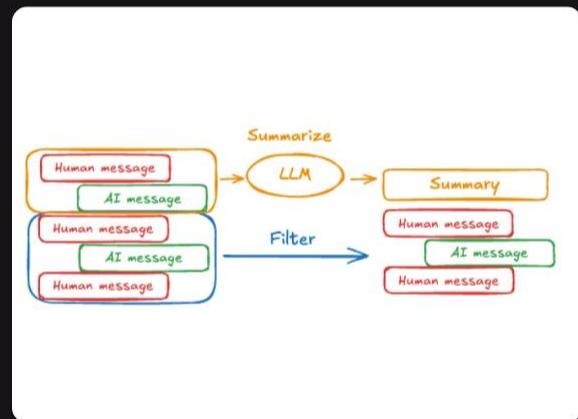
# create user profile
profile = manager.invoke({"messages": conversation})[0]
```



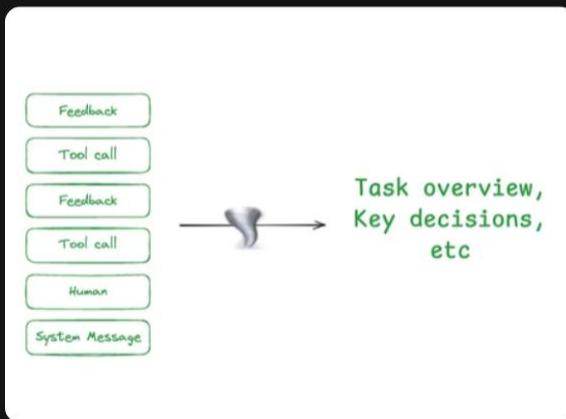
```
ExtractedMemory(
    id="6f555d97-387e-4af6-a23f-a66b4e809b0e",
    content=UserProfile(
        name="Alex",
        preferred_name="Lex",
        response_style_preference="casual and witty with appropriate emojis",
        special_skills=[
            "Python programming",
            "AI development",
            "competitive speedcubing",
        ],
        other_preferences=[
            "prefers informal communication",
            "dislikes corporate-style interactions",
        ],
    ),
)
```

Short-Term Memory Management Methodologies

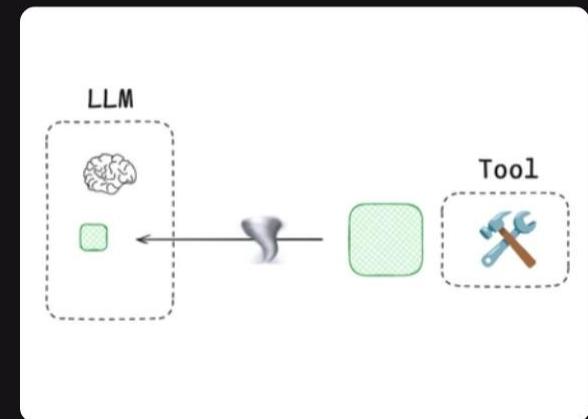
Short-term memory can be optimized by summarizing past history as well as tool responses. We can also filter out older historical messages.



Summarizing and\or Filtering out conversations



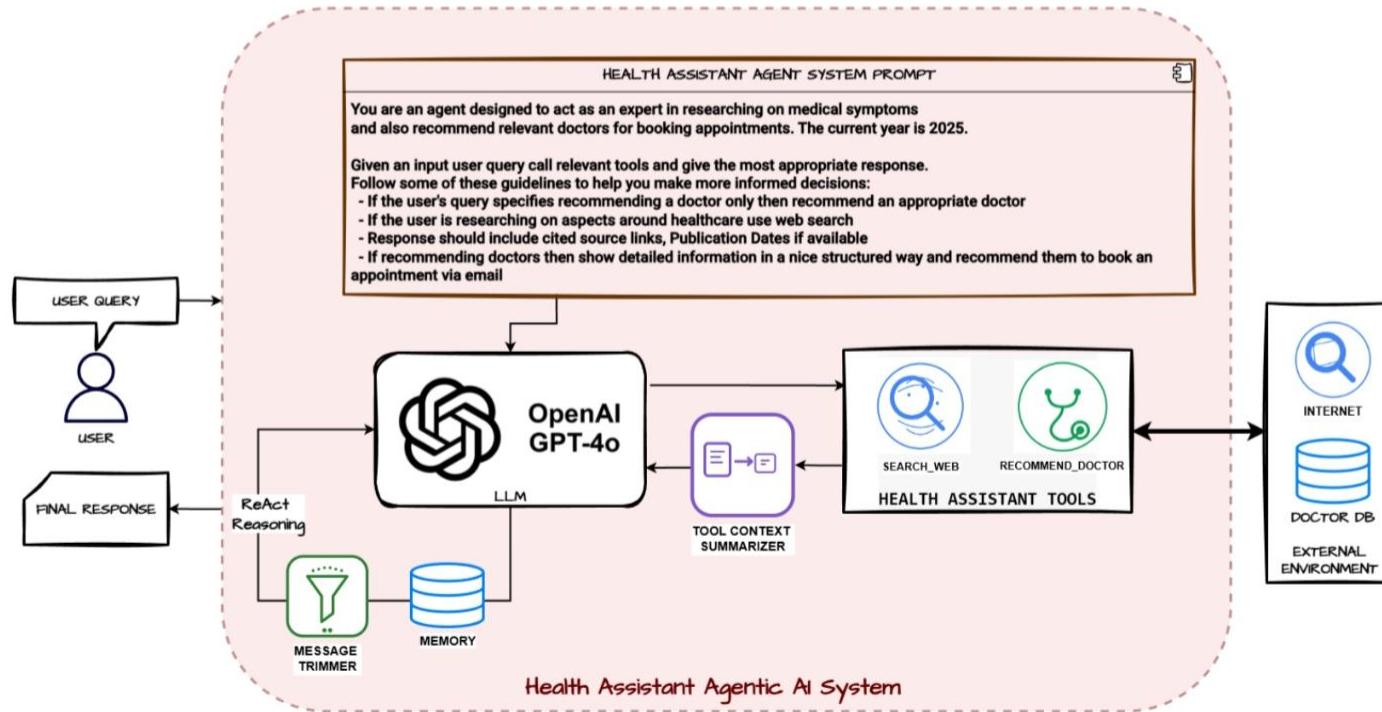
Summarizing History



Summarizing Tool Responses

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

Summarize Tool Call Responses & Filter out older history from Memory Context



Hands-On Demo: Memory Context Engineering for AI Agents

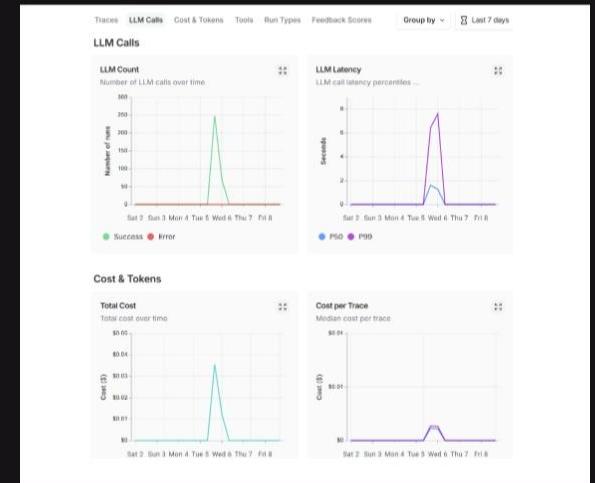
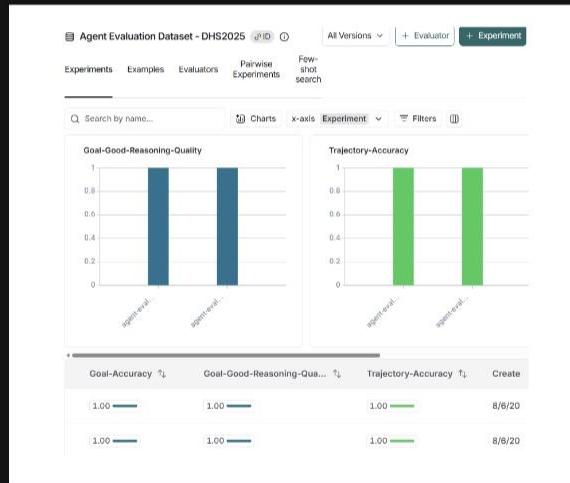
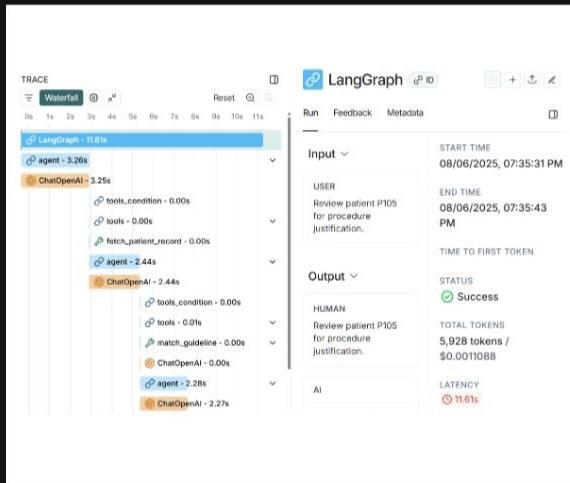
- Get the notebook from [HERE](#)
- Recommended to run on Google Colab



Observability for Agentic AI Systems

Why Care About Agent Observability?

Observability enables you to take your Agent from PoC to Production and beyond with tracing, debugging, evaluation and continuous monitoring.



Trace Agent Executions for debugging, explainability, audits and more

Evaluate your Agent's Performance with LLM-as-a-Judge & Statistical Evaluators

Monitor and Track your Agent's Usage with cost, latency, usage metrics and more

Popular Frameworks for Observability in Agentic AI Systems



LangSmith



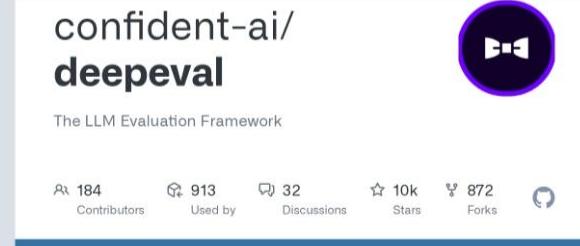
by Comet



Langfuse

confident-ai/
deepeval

The LLM Evaluation Framework



A 184 Contributors G 913 Used by D 32 Discussions ★ 10k Stars F 872 Forks

Common Metrics for Agent Monitoring

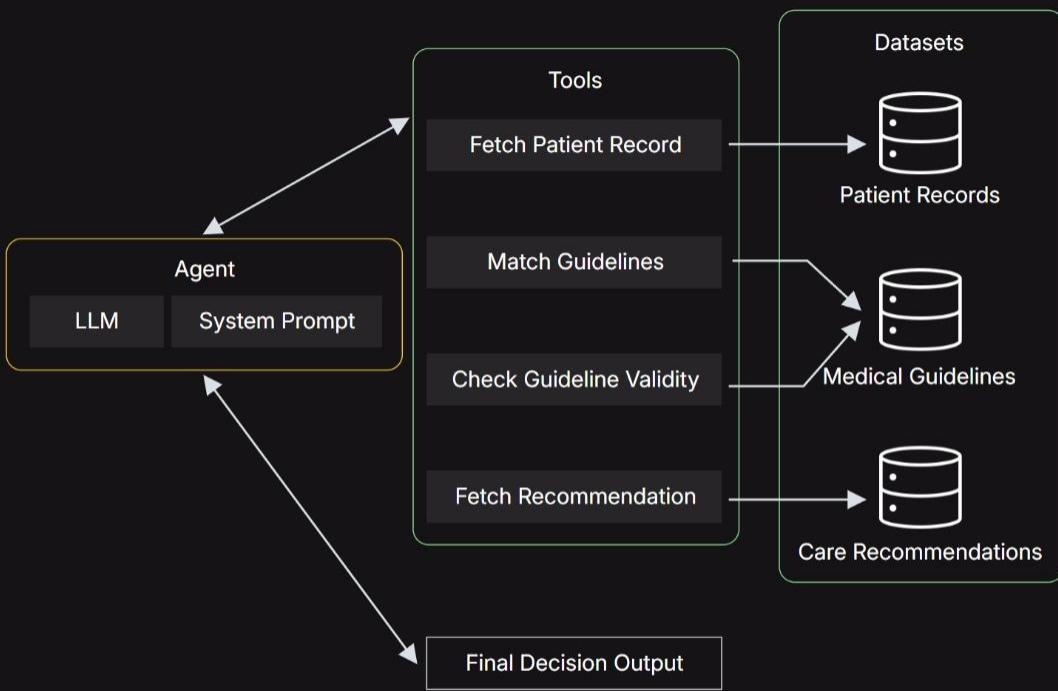
Category	Main Metric	Meaning
Traces	Trace count	Number of complete agent operation traces recorded.
	Latency (avg, p50, p99)	Time to complete a trace; percentiles show median and tail performance.
	Error rate	Percentage of traces that failed.
LLM Calls	Call count	Number of LLM invocations in traces.
	Call latency (avg, p50, p99)	Time taken per LLM call; percentiles indicate slowest cases.
Cost & Tokens	Cost	Monetary cost from token usage.
	Token breakdown	Split of prompt vs. completion tokens.
Tools	Tool run count	Frequency of tool executions.
	Tool latency (avg, p50, p99)	Time taken by tools; percentiles highlight tail latencies.
	Tool error rate	Percentage of tool runs that failed.

Common Metrics for Agent Evaluation

Recommendation: Create your own custom LLM-as-a-Judge metric prompts and avoid relying on default prompts from frameworks. Use few-shot examples and also develop an evaluation dataset; it really helps!

Metric	Description
Goal Accuracy	Measures whether the agent achieves the intended goal, using reference comparison or LLM-as-a-judge when no reference exists.
Goal Reasoning Quality	Evaluates how well the agent's reasoning supports the final decision, focusing on clarity, correctness, and alignment with the goal.
Trajectory Accuracy	Checks if the agent follows the correct sequence of steps - approaches include LLM-as-a-Judge, Strict Match, Unordered Match, and Subset/Superset Match.
Tool Call Accuracy	Assesses whether the correct tools are selected and used with appropriate inputs during execution.
Hallucination	Measures if the agent outputs unsupported or fabricated information.

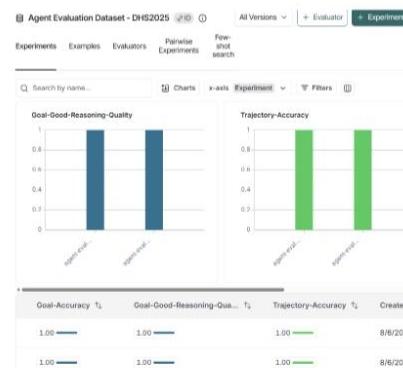
Let's now add observability to our Utilization Review Agent from earlier!



Build Agent

The screenshot shows the LangGraph interface for monitoring an AI agent's interaction. The "TRACE" section displays a waterfall of events: "tools_condition - 0.00s", "tools - 0.00s", "fetch_patient_record - 0.00s", "agent - 2.44s", "ChatOpenAI - 3.25s", and "Match Guidelines - 0.00s". The "Input" section shows a user request: "Review patient P105 for procedure justification." The "Output" section shows the AI response: "HUMAN Review patient P105 for procedure justification." The "Status" section indicates "Success" with 5,928 tokens and \$0.0011088 latency.

Monitor Agent



Evaluate Agent

Hands-On Demo: Agent Observability with LangSmith

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



Key Takeaways



Use design patterns wisely to architect robust Agentic AI systems.



Adopt MCP and A2A only when their benefits outweigh the added complexity.



Master context engineering to boost agent performance and output quality.



Implement agent observability to move beyond just building proof-of-concepts.



Apply prompt engineering effectively:

- Use strong system prompts to build better agents.
- Use well-crafted LLM-as-a-judge prompts to improve and evaluate agents.

Thank You

Slides & Code



https://bit.ly/agents_dhs25