Agents: Landscape and Challenges

Manish Sharma
Gen AI Design & Adoption

Agenda

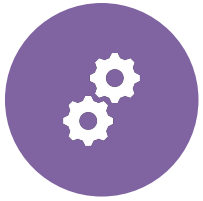
AN OPEN DISCUSSION

(IDEA TO LIVE)

What are Agents & Use Cases

Agents are autonomous systems that perform tasks using reasoning and planning

Identify use cases by Common domains: IT ticketing, finance reconciliation, document processing

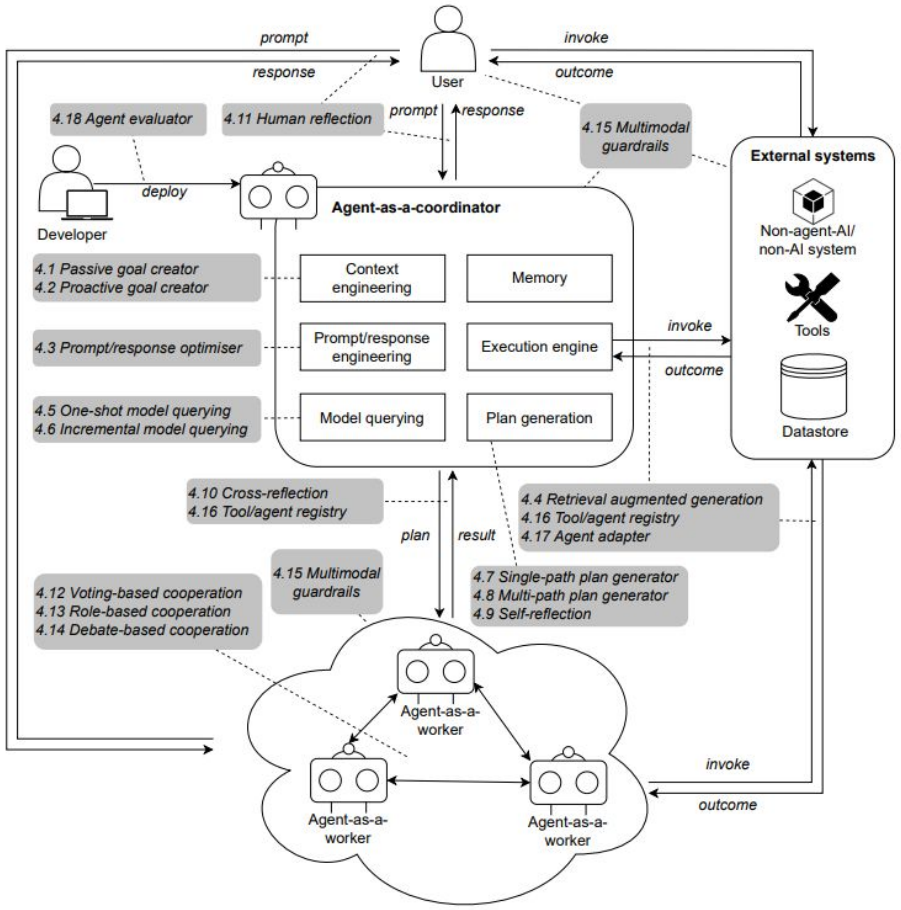


High volume, repetitive workflows

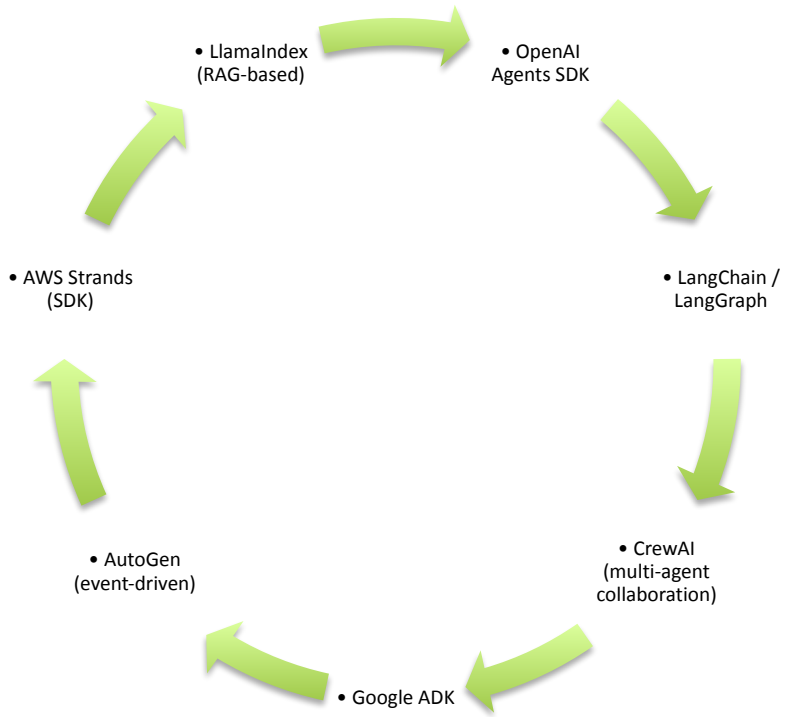
Tasks involving multiple systems

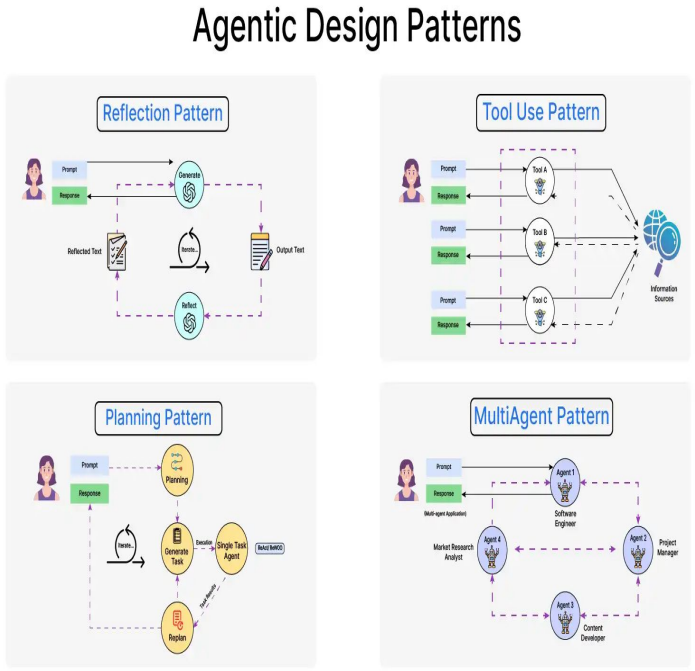
Manual interventions

Agents Ecosystem

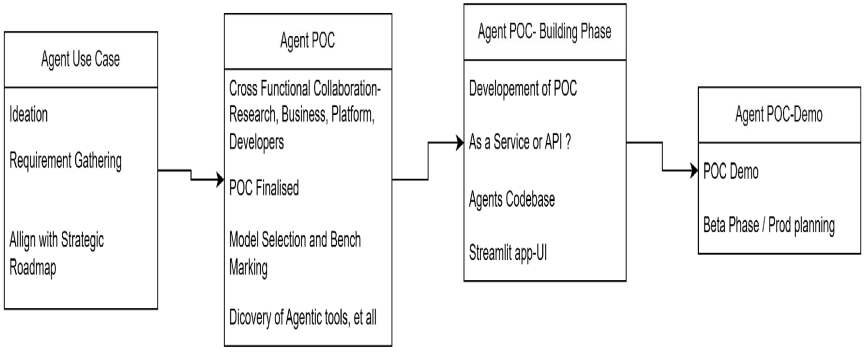


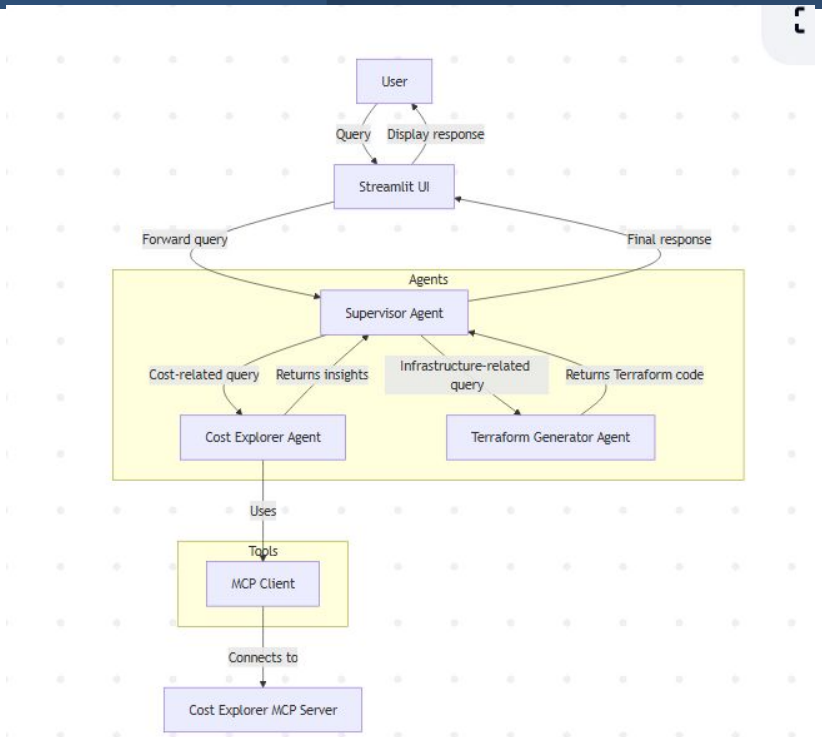
Popular Agent Frameworks





Idea to POC


Agent Use Case
Ideation
Requirement Gathering
Allign with Strategic Roadmap
Agent POC
Cross Functional Collaboration- Research, Business, Platform, Developers
POC Finalised
Model Selection and Bench Marking
Dicovery of Agentic tools, et all
Agent POC- Building Phase
Developement of POC
As a Service or API ?
Agents Codebase
Streamlit app-UI
Agent POC-Demo
POC Demo
Beta Phase / Prod planning

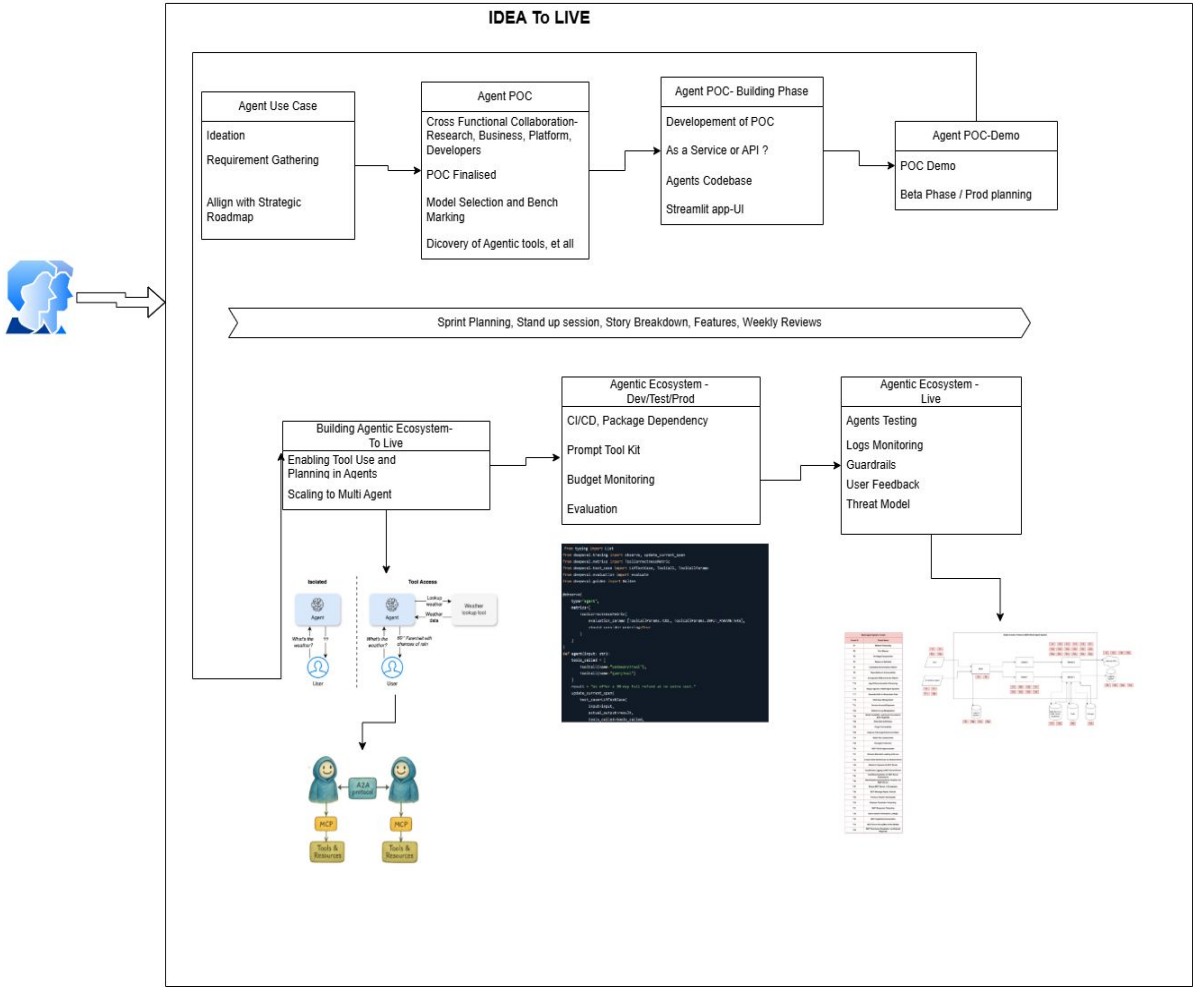

User
Query
Display response
Streamlit UI
Forward query
Final response
Agents
Supervisor Agent
Cost-related query
Returns insights
Infrastructure-related query
Returns Terraform code
Cost Explorer Agent
Terraform Generator Agent
Uses
Tools
MCP Client
Connects to
Cost Explorer MCP Server

Idea to POC

```
# Supervisor / Orchestrator Agent
# ------------------------------------------------------------------------
SUPERVISOR_PROMPT = (
    "You are the AI system supervisor. Route user requests to the appropriate specialist.\n"
    "If the user's question is about AWS costs, billing, or spending, use the `cost_agent` tool. "
    "If it's about generating or understanding AWS infrastructure (e.g. creating resources, Terraform code, architecture), use `terraform_agent`. "
    "Respond with the chosen tool's output."
)

supervisor_agent = Agent(
    system_prompt=SUPERVISOR_PROMPT,
    tools=[cost_agent, terraform_agent],
    callback_handler=None  # suppress internal logs for clarity
)

## Streamlit Chatbot UI

import asyncio
import streamlit as st
from strands import Agent
# (Assume the above agent code is in the same scope or imported)

st.set_page_config(page_title="Multi-Agent AWS Chatbot")
st.header("AWS Multi-Agent Assistant")

# Initialize the supervisor agent (ideally done outside the request loop)
# (Reuse the supervisor_agent defined above)

# Chat history
if "history" not in st.session_state:
    st.session_state.history = []

# User input
user_input = st.text_input("You:", key="input")

if st.button("Send"):
    prompt = user_input.strip()
    if prompt:
        # Append user message to chat history
        st.session_state.history.append({"role": "user", "text": prompt})

        # Call the supervisor agent (streaming)
        response_text = ""
        placeholder = st.empty()

        # Use Strands streaming interface (asynchronous)
        async def chat_stream():
            try:
                async for event in supervisor_agent.stream_async(prompt):
                    if "data" in event:
                        # Append new chunk to response
                        nonlocal response_text
                        response_text += event["data"]
                        placeholder.markdown(response_text)  # update display
            except Exception as e:
                placeholder.text(f"Error: {e}")

        # Run the async streaming generator
asyncio.run(chat_stream())

        # Save assistant response
        st.session_state.history.append({"role": "assistant", "text": response_text})
```

```
'''
Multi-agent AI system using Strands Agents to handle cloud cost analysis and infrastructure-as-code generation.
The system has three specialized agents a **Cost Explorer Agent**, a **Terraform Generator Agent**, and a **Supervisor (orchestrator) Agent**
A **Streamlit chatbot UI**.
'''

from strands import Agent, tool
from strands.tools.mcp import MCPClient
from mcp.client.sse import sse_client

# (A) Cost Explorer Agent - uses MCP to fetch AWS cost data.
# ------------------------------------------------------------------------
# Create an MCP client connected to the local Cost Explorer MCP server.
cost_mcp_client = MCPClient(lambda: sse_client("http://localhost:8000/sse"))
# List the tools provided by the MCP server (this fetches cost-related APIs)
cost_tools = []
with cost_mcp_client:
    cost_tools = cost_mcp_client.list_tools_sync()  # list of Tool objects

# Define a specialized cost-analysis agent as a tool.
COST_AGENT_PROMPT = (
    "You are a specialized AWS Cost Explorer assistant. "
    "Given a user's request about AWS spending or cost breakdown, use the available cost tools "
    "to query the AWS Cost Explorer data and return insights (service-wise spend, usage patterns, breakdowns, etc.) "
    "in clear language."
)

@tool("cost_agent")
def cost_agent(query: str) -> str:
    try:
        # Create a new agent with the cost tools and a guiding prompt
        agent = Agent(
            system_prompt=COST_AGENT_PROMPT,
            tools=cost_tools
            model = <Bedrock-Model>
        )
        response = agent(query)
        # The agent returns a textual answer including cost insights.
        return str(response)
    except Exception as e:
        return f"Error in Cost Agent: {str(e)}"

# (B) Terraform Generator Agent - generates Terraform code.
# ------------------------------------------------------------------------
TERRAFORM_AGENT_PROMPT = (
    "You are an AWS Terraform expert. The user will describe an AWS architecture "
    "or resource requirements in natural language. Generate a complete, valid Terraform "
    "configuration (HCL code) that implements the described infrastructure. "
    "Return only the Terraform code, with appropriate resources and modules."
)

@tool("terraform_agent")
def terraform_agent(query: str) -> str:
    try:
        agent = Agent(
            system_prompt=TERRAFORM_AGENT_PROMPT,
            tools=[]  # no external tools needed, just rely on the LLM for code generation
            model = <Bedrock-Model>

        )
        response = agent(query)
        return str(response)
    except Exception as e:
        return f"Error in Terraform Agent: {str(e)}"
```

Idea to Live

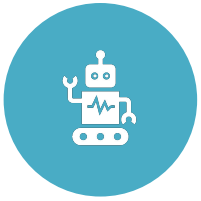

IDEA To LIVE
Agent Use Case
Ideation
Requirement Gathering
Allign with Strategic Roadmap
Agent POC
Cross Functional Collaboration- Research, Business, Platform, Developers
POC Finalised
Model Selection and Bench Marking
Dicovery of Agentic tools, et all
Agent POC- Building Phase
Developement of POC
As a Service or API ?
Agents Codebase
Streamlit app-UI
Agent POC-Demo
POC Demo
Beta Phase / Prod planning
Sprint Planning, Stand up session, Story Breakdown, Features, Weekly Reviews
Building Agentic Ecosystem- To Live
Enabling Tool Use and Planning in Agents
Scaling to Multi Agent
Agentic Ecosystem - Dev/Test/Prod
CI/CD, Package Dependency
Prompt Tool Kit
Budget Monitoring
Evaluation
Agentic Ecosystem - Live
Agents Testing
Logs Monitoring
Guardrails
User Feedback
Threat Model

Architecture Options

- Microservices for decoupling logic

- GenAI platforms (Bedrock, Azure AI Studio)

- Event-driven flows with queues (Pub/Sub, SQS)



- Stateless LLM ‘brain’ + API-based tools

- Recommended: Modular, cloud-native, agent-as-a-service

Key KPIs

- Task Efficiency – Completion rate, latency

- Reliability – Errors, uptime

- Engagement – Usage frequency, sessions

- Satisfaction – CSAT, feedback

- Business Impact – ROI, hours saved, cost/request

References

- Agent Design Pattern Catalogue: A Collection of Architectural Patterns for Foundation Model based Agents
- Agentic or Tool use – Ragas
- Multi-agent Systems
- Introduction - Model Context Protocol
- ADK Tutorials! - Agent Development Kit