Agentic AI and Multi Agent Systems

Dr. Shailesh Sivan

December 12, 2025

Outline

Agents and AI

Agent Patterns

Applications of Agentic AI

Multi Agent Systems

Multi Agent System Architectures

Agentic AI Implementation Frameworks

LangGraph

Agentic RAG

Agents and AI

What is Agent and Agentic AI?

Agent

An **agent** is an entity that autonomously perceives its environment, makes decisions, and acts to achieve specific goals.

Agentic AI

Agentic AI refers to artificial intelligence systems that act autonomously toward goals, exhibiting behaviors similar to human-like agents: planning, decision-making, and learning from the environment.

Example

An AI personal assistant that schedules meetings, books travel, or completes tasks based on user instructions without being micromanaged.

Traditional Agents vs Agentic AI Agents

Traditional Agents

- Rule-based / model-based intelligence
- Finite-state machines, logic rules
- Symbolic planning (STRIPS, PDDL)
- Fixed policies and workflows
- Communication protocols (ACL, FIPA)
- Limited adaptability
- Strong formal guarantees

Agentic AI Agents

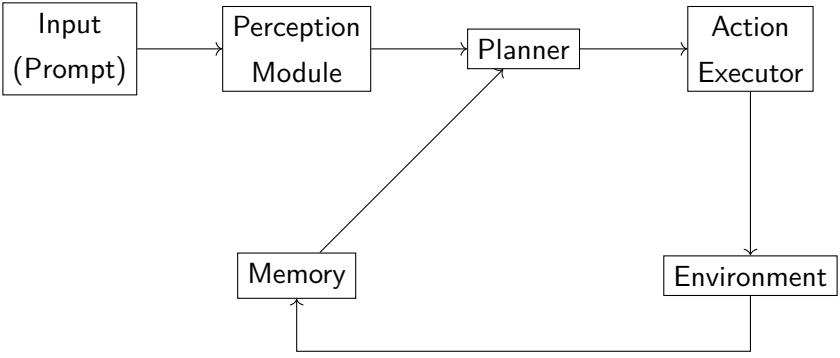
- LLM-driven reasoning and decision-making
- Natural language planning
- Think–Plan–Act–Reflect loops
- Dynamic tool selection and composition
- Language-based coordination
- Self-correction via critique loops
- High autonomy and generalization

Traditional agents emphasize *control and correctness* where as Agentic AI agents emphasize *reasoning, adaptability, and autonomy*.

Key Characteristics of Agents and Agentic AI

- **Goal-Directed Behavior**
- **Planning and Decision Making**
- **Autonomy and Reusability**
- **Memory and Adaptability**
- **Environment Interaction**

Simple Agent Architecture



Types of Agents in AI

- **Reflex Agents**
  - Respond only to the current percept (stateless).
  - **Example:** A thermostat that turns heating ON/OFF purely based on the current temperature reading.
- **Model-Based Agents**
  - Maintain an internal state that represents unobserved parts of the world.
  - **Example:** A robot vacuum cleaner that builds a map of the environment and navigates even in dark areas.
- **Goal-Based Agents**
  - Select actions based on the desired goal state.
  - **Example:** A navigation system (e.g., Google Maps) that plans a route to reach the destination.

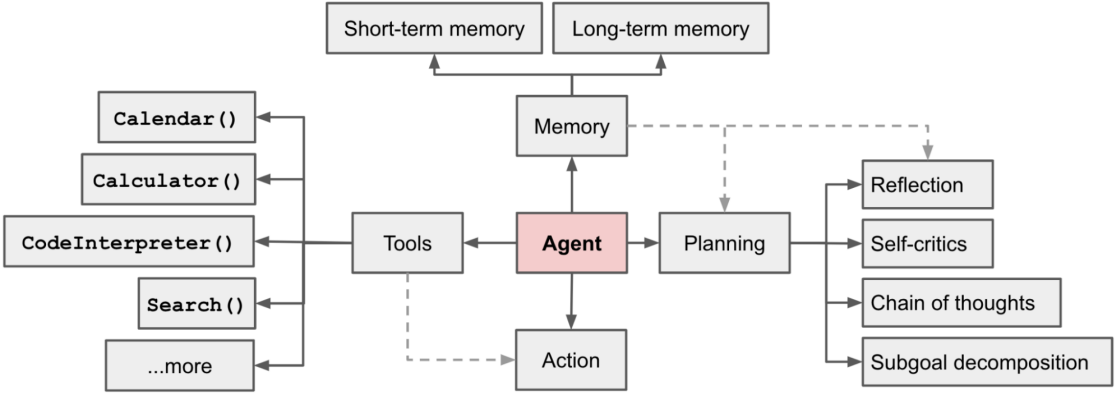
Types of Agents in AI

- **Utility-Based Agents**
  - Choose actions that maximize expected utility considering preferences and trade-offs.
  - **Example:** A self-driving car choosing between routes by balancing time, safety, and comfort.
- **Learning Agents**
  - Improve performance over time based on experience.
  - **Example:** AlphaGo/AlphaZero learning optimal strategies through repeated self-play.

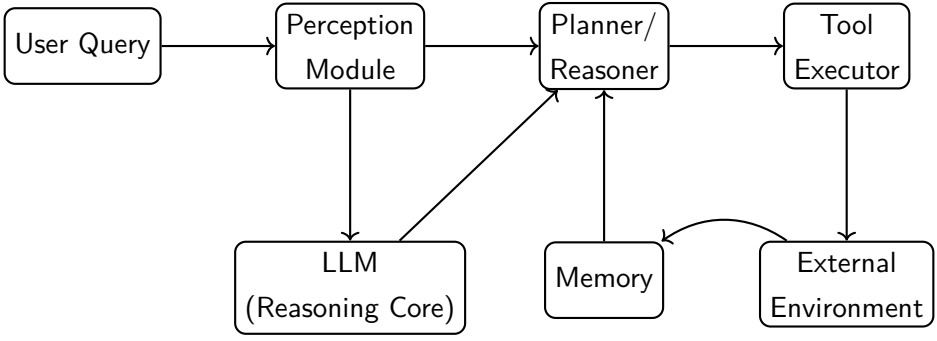
Note

Agentic AI blends goal-based, utility-based, and learning capabilities to create autonomous and adaptive systems.

LLM-powered autonomous agent system.



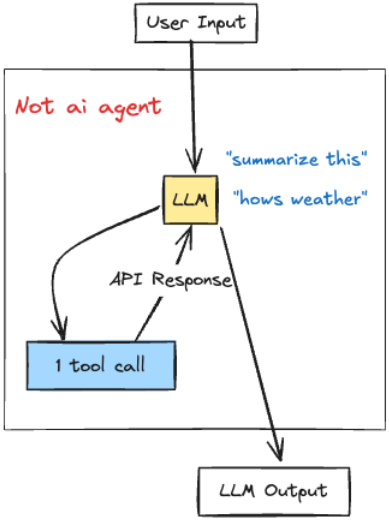
LLM Agent Workflow

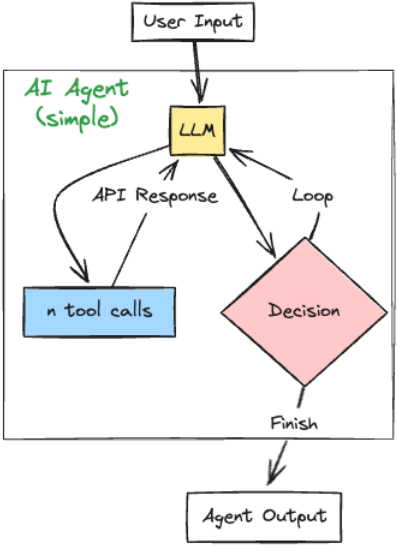


Agent vs. Tool: Comparison

Aspect	Tool AI	AI Agent
Initiative	Reactive	Proactive
Memory	Stateless	Stateful
Autonomy	Low	High
Learning	Minimal	Continuous
Goal Orientation	Task-Specific	Goal-Driven

Agent vs. Tool: Comparison

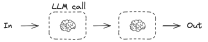




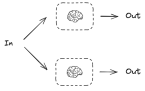
Agents and Workflow

Workflows

Prompt Chaining



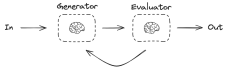
Parallelization



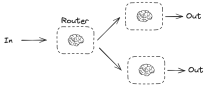
LLM is embedded in predefined code paths

Orchestrator-Worker

Evaluator-optimizer

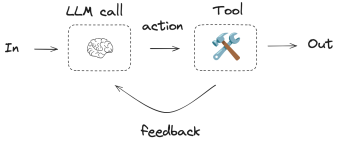


Routing



LLM directs control flow through predefined code paths

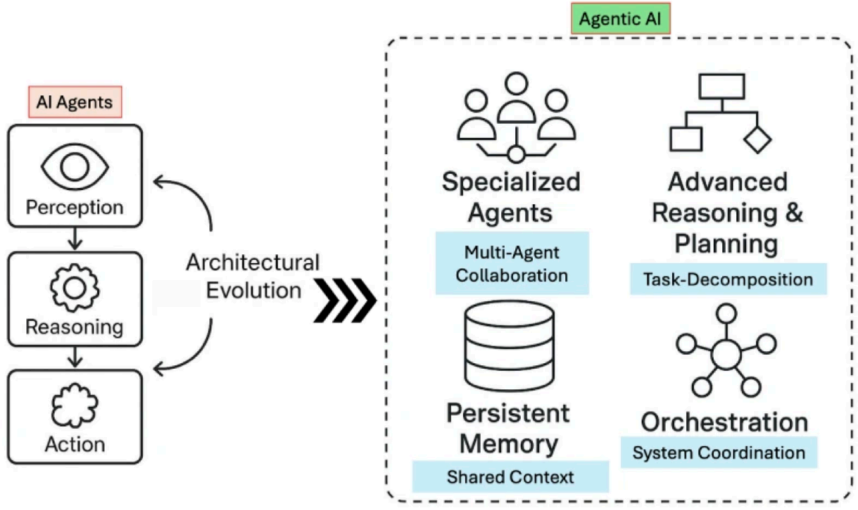
Agent



LLM directs its own actions based on environmental feedback

Workflows can also be implemented as Agent pattern

AI Agents and Agentic AI

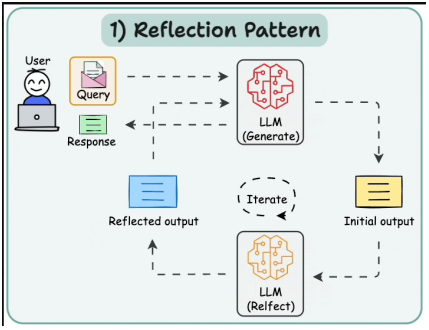


Agent Patterns

Reflection Pattern

The AI reviews its work to spot mistakes and iterate until it produces the final response.

- Self-evaluation and iterative refinement
- Useful for hallucination reduction and answer polishing
- Implementations: chain-of-thought + critique loops

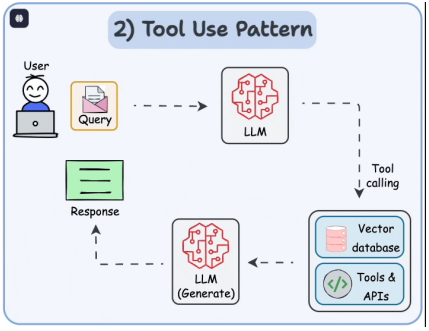


Tool Use Pattern

Tools allow LLMs to gather more information by:

- Querying a vector/database
- Executing Python scripts
- Invoking external APIs

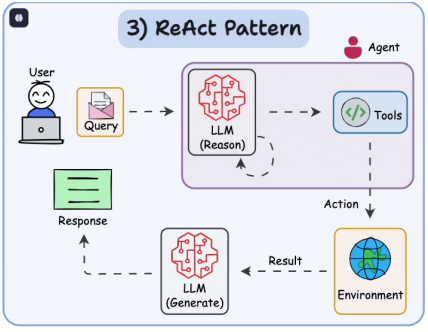
This reduces reliance on internal knowledge and improves factuality.



ReAct (Reason + Act) Pattern

ReAct combines reflection and tool use:

- Agents can reflect on generated outputs
- Agents can call tools and observe results
- Effective for multi-step, interactive tasks

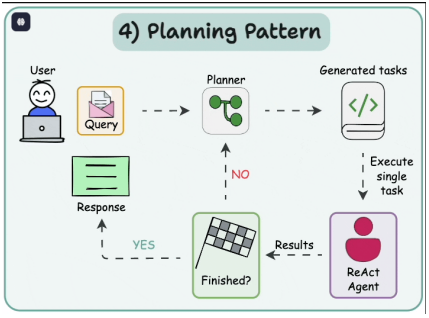


Planning Pattern

Instead of solving a request in one go, the AI creates a roadmap by:

- Subdividing tasks
- Outlining objectives and steps
- Iteratively executing and revising the plan

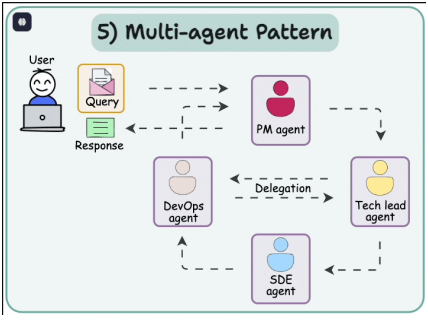
Strategic thinking helps solve complex tasks more effectively.



Multi-Agent Pattern

Multiple agents collaborate, each with dedicated roles and access to tools:

- Role-based decomposition (planner, worker, critic, router)
- Parallelization and delegation of sub-tasks
- Agents coordinate via messages or shared memory



Pattern Comparison

Each pattern offers unique strengths for building intelligent systems. Depending on task complexity, environment, and goals, one or more patterns can be combined for optimal performance.

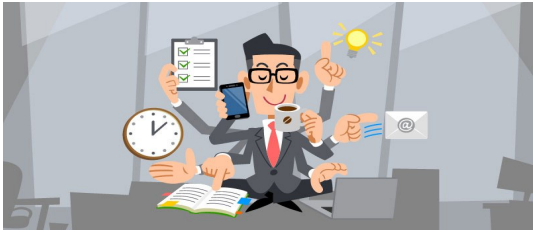
Pattern	**Strengths**	**Best For**
Reflection	Improves self-awareness and adaptability	Debugging, strategy adjustment
Tool Use	Extends capabilities via external tools	Math, retrieval, real-world tasks
Planning	Provides structure and foresight	Multi-step reasoning, code generation
Multi-Agent	Enables collaboration and modularity	Complex, distributed task environments

Applications of Agentic AI

Customer Service & Productivity

Customer Service

- Automated chatbots handling initial customer inquiries
- Virtual assistants providing 24/7 support
- AI agents resolving common issues without human intervention



Personal Productivity

- Digital assistants managing calendars and scheduling
- Email sorting and prioritization agents
- Task management systems with intelligent recommendations

Healthcare and Financial Services

Healthcare

- Diagnostic support agents analyzing patient symptoms
- Medication management assistants for patients
- Administrative agents handling appointment scheduling and records

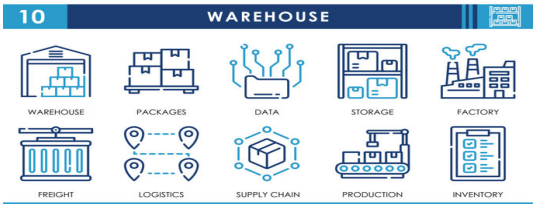
Financial Services

- Robo-advisors managing investment portfolios
- Fraud detection agents monitoring transactions
- Insurance claim processing automation

Sales and Supply Chain

Sales and Marketing

- Lead qualification and scoring agents
- Personalized product recommendation engines
- Marketing automation tools optimizing campaigns



Manufacturing and Supply Chain

- Predictive maintenance agents monitoring equipment
- Inventory optimization systems
- Quality control agents detecting defects

Home Automation and Education

Home Automation

- Smart home assistants controlling devices and systems
- Security monitoring agents detecting unusual activity
- Energy optimization agents managing consumption

Education

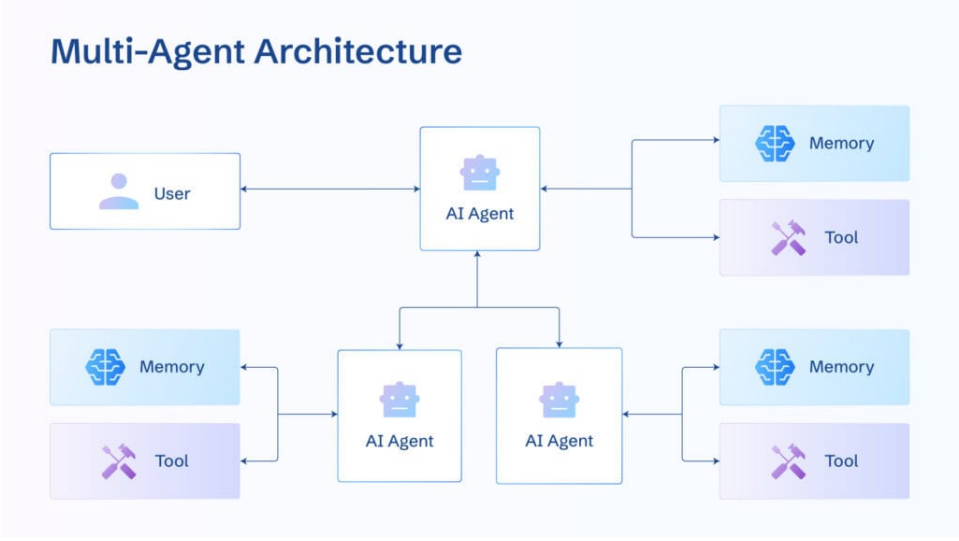
- Tutoring agents providing personalized learning
- Grading assistants evaluating assignments
- Course recommendation agents suggesting appropriate classes

Multi Agent Systems

What is a Multi-Agent System?

- A **multi-agent system** consists of multiple autonomous agents that collaborate or coordinate to solve a task.
- Each agent has:
  - its own goal,
  - its own reasoning logic,
  - and its own tools or knowledge.
- Agents communicate through messages and operate in a shared environment.
- Useful for:
  - decomposition of complex tasks,
  - parallel reasoning,
  - fact-checking and evaluation,
  - retrieval + reasoning + validation loops.

Multi-Agent Systems



Single-agent vs Multi-agent

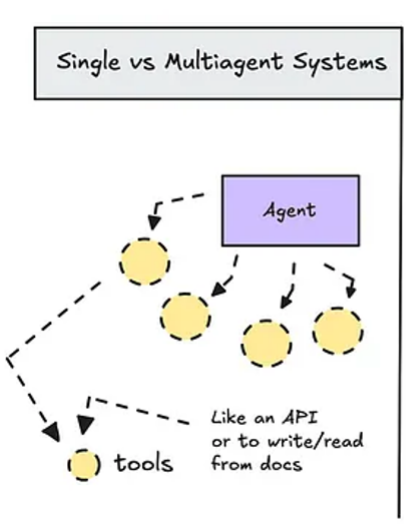
Single-Agent Systems

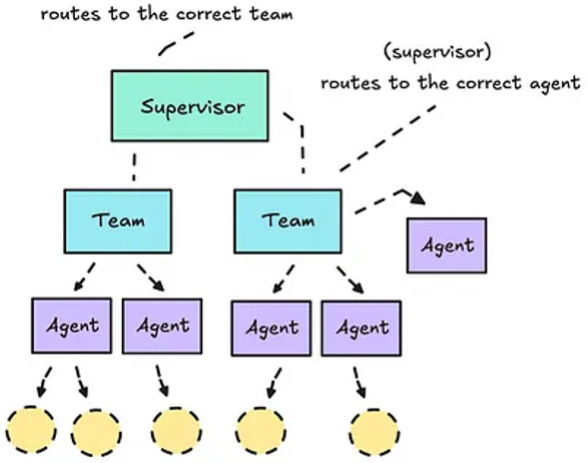
- One autonomous agent interacts with environment.
- Plans, calls tools (APIs, DBs, other agents-as-tools), and responds.
- If another agent is invoked as a tool, that agent is treated as part of the environment no further modeling or coordination.

Multi-Agent Systems

- Multiple autonomous agents that model each other's goals, memory and plans.
- Agents communicate (directly or via shared environment), cooperate and coordinate.
- Supports distributed problem solving and multi-agent reinforcement learning.

Single-agent vs Multi-agent





Architectures of Multi-Agent Systems

Centralized Networks

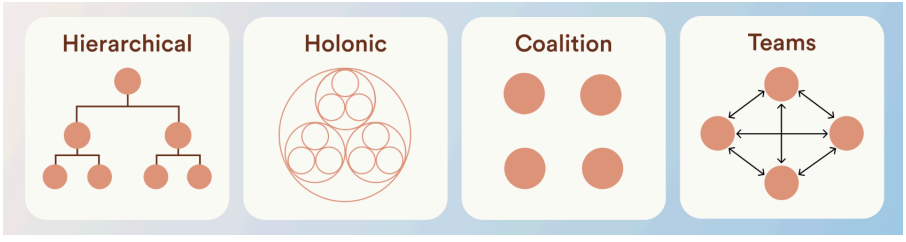
- A central unit stores global knowledge and orchestrates agents.
- Pros: easy communication, uniform knowledge.
- Cons: single point of failure.

Decentralized Networks

- Agents share information with neighbours; no global store.
- Pros: robustness, modularity.
- Cons: harder coordination and global optimization.

Structures of Multi-Agent Systems

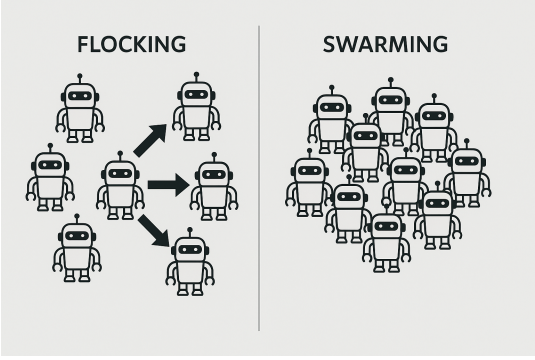
- **Hierarchical:** tree-like with varying authority levels. Can be centralized or distributed.
- **Holonic:** agents grouped into holarchies; a holon acts as a composite entity composed of subagents.
- **Coalitions:** temporary unions to boost utility; dissolve after goals met.
- **Teams:** persistent cooperative groups with interdependent roles (more dependence than coalitions).



Agent Behaviours: Flocking & Swarming

Flocking heuristics

- Separation: avoid collisions with nearby agents.
- Alignment: match velocity/direction of neighbours.
- Cohesion: remain close to the group.



Swarming

- Emergent self-organization with decentralized control.
- Useful when one operator manages an entire swarm (reduced operator-per-agent cost).

Advantages of Multi-Agent Systems

- **Flexibility:** add/remove/adapt agents to changing environments.
- **Scalability:** leverage pooled information and compute.
- **Domain specialization:** different agents hold different expertise.
- **Higher performance:** collective learning, shared experiences and parallelism.

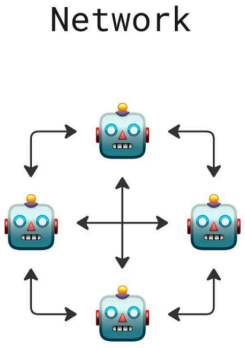
Multi Agent System Architectures

1. Network

Definition: Agents can communicate directly with any other agent in the system. Each agent decides which agent to interact with next based on local state and policies.

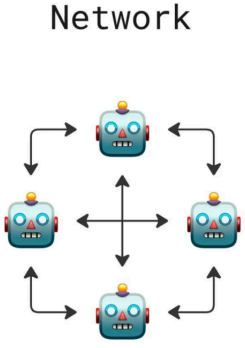
Key Behaviour

- Peer-to-peer communication (no central coordinator).
- Dynamic routing: agents pick interaction partners at runtime.
- Emergent cooperation via local decisions and messages.



Practical Use Cases

- **Ad-hoc sensor networks**: distributed sensors negotiating which node aggregates or forwards data.
- **Collaborative chat assistants**: specialized assistants (code, math, facts) exchange messages to resolve a query.
- **Decentralized marketplaces**: buyer and seller agents discover and negotiate prices without central registry.

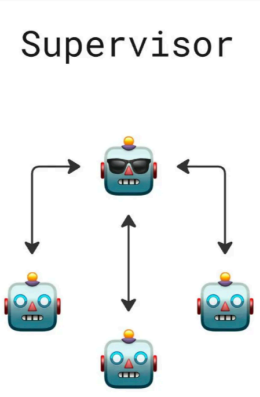


2. Supervisor

Definition: A central supervisor agent coordinates communication and allocates tasks among other agents. The supervisor decides which agent to call next for a given task.

Key Behaviour

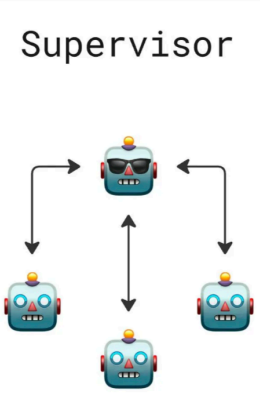
- Centralized decision-making for task routing.
- Supervisor holds or has access to global state / task queue.
- Sub-agents specialized for execution; supervisor orchestrates flow.



2. Supervisor

Practical Use Cases

- **Customer support pipeline**: supervisor assigns intents sub-agents (billing, technical, refunds).
- **Data processing pipeline**: coordinator schedules cleaning, enrichment, analysis agents.
- **Multi-step automation**: supervisor ensures ordered execution (validate compute publish).

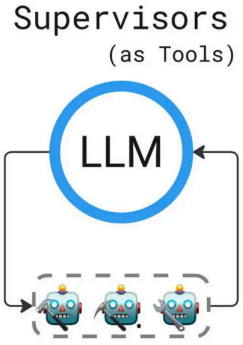


3. Supervisor with Tool Calling

Definition: The supervisor treats individual agents (or capabilities) as tools/APIs. Instead of delegating to a live agent, the supervisor invokes tool endpoints (or agent-as-service) as needed.

Key Behaviour

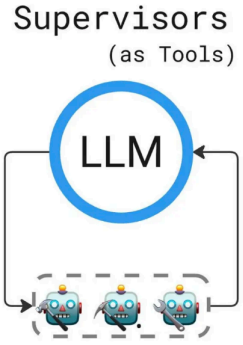
- Supervisor orchestrates calls to tool-interfaces (search API, calculator, domain models).
- Tools are stateless or minimally stateful; logic lives in supervisor or tool.
- Easier integration with existing services and safer sandboxing.



3. Supervisor with Tool Calling

Practical Use Cases

- **RAG-enabled QA**: supervisor calls vector-search, web-scraper, or DB connector tool to answer a query.
- **Enterprise automation**: supervisor invokes HR payroll API, CRM updates, reporting tools.
- **Robotics control**: supervisor calls motion-planning service, vision API, and safety-check tool sequentially.

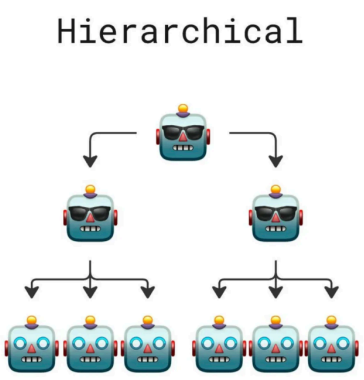


4. Hierarchical

Definition: A multi-tiered system where supervisors manage groups of agents or other supervisorsenables control of complex workflows and better scalability.

Key Behaviour

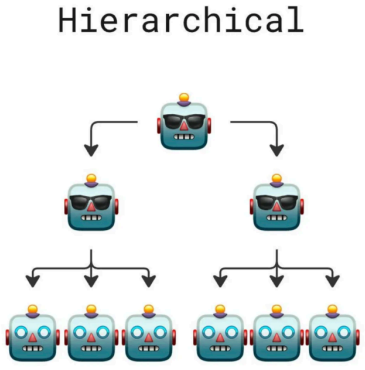
- Layered supervision: top-level planners mid-level coordinators worker agents.
- Task decomposition and delegation across layers.
- Error isolation and logging scoped to layers.



4. Hierarchical

Practical Use Cases

- **Large-scale enterprise orchestration**: global supervisor assigns region supervisors which manage local agents (billing, logistics).
- **Smart city management**: city-level controller district controllers sensors/actuators.
- **Complex ML pipelines**: top-level experiment planner dataset stewards model-training workers.

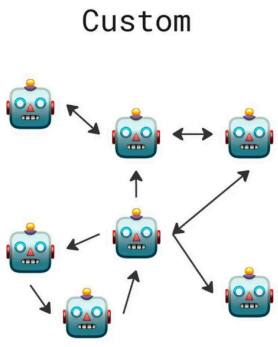


5. Custom Workflow

Definition: Agents communicate only with a subset of other agents according to predefined rules; workflows combine deterministic routing and agent decision-making.

Key Behaviour

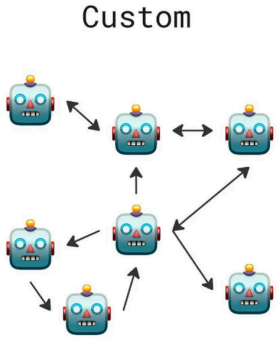
- Hybrid routing: deterministic rules for critical steps, agent autonomy for exploratory steps.
- Role-based visibility: agents only see subset of global state.
- Rule engine and agent policies co-exist.



5. Custom Workflow

Practical Use Cases

- **Regulated workflows**: banking KYC deterministic verification then agent-based risk assessment.
- **Healthcare triage**: fixed triage rules route critical cases, non-critical go to specialized agents for extended assessment.
- **Hybrid manufacturing lines**: fixed safety checks plus agentic workers optimizing assembly substeps.



Decision Guide & Trade-offs

Choosing the right pattern

- **Network** choose when decentralization and robustness are required; expect harder coordination.
- **Supervisor** choose when deterministic task ordering or centralized policy is needed.
- **Supervisor + Tools** choose to integrate existing services and reduce inter-agent complexity.
- **Hierarchical** choose for scale and error isolation in large deployments.
- **Custom Workflow** choose for regulated domains where partial determinism is required.

Common concerns: latency, single points of failure, security of inter-agent channels, observability and debugging.

Challenges of Multi-Agent Systems

- Agent malfunctions and shared failure modes.
- Coordination complexity and negotiation overhead.
- Unpredictable emergent behaviour in decentralized systems.
- Security, governance, and data-quality concerns.

Agentic AI Implementation Frameworks

Popular Frameworks

- LangChain + LangGraph
- AutoGPT / BabyAGI
- ReAct (Reasoning + Acting)
- CrewAI / AgentVerse

Languages and Libraries for Agentic AI

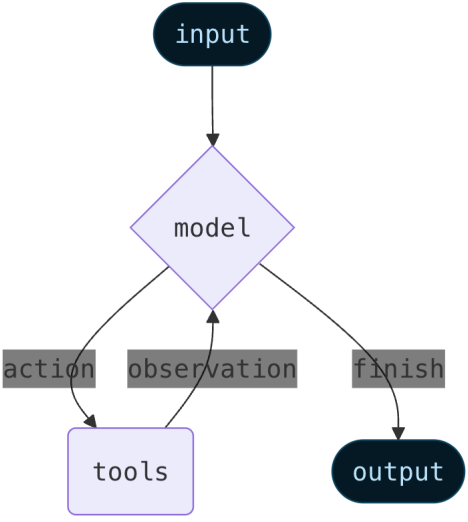
Popular Languages

- Python (dominant in LLM tools)
- JavaScript (for front-end agents)
- TypeScript (used in LangChain.js)

Core Libraries

- `LangChain, LangGraph`
- `AutoGPT, BabyAGI`
- `CrewAI, OpenAgents`
- `Transformers ( )`
- `ReAct, Semantic Kernel`

LangChain Agent Architecture



Simple Agent (search + calc) example

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import Tool, initialize_agent, AgentType

def calc_tool(expr: str) -> str:
    return str(eval(expr))

def search_tool(q: str) -> str:
    return "SEARCH_RESULT: product X released in 2010"

t1=Tool(name="search", func=search_tool, description="Search for product
    info and return results.")

t2=Tool(name="calc", func=calc_tool, description="Evaluate arithmetic
    expressions safely.")
```

Simple Agent (search + calc) example

```
llm = ChatGoogleGenerativeAI(
       model="gemini-2.0-flash",temperature=0,
       google_api_key="YOUR_GOOGLE_API_KEY"
   )

agent = initialize_agent(tools=[t1,t2],
       llm=llm,
       agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
       verbose=True
   )

print(agent.run('Find the year product X released (search), then compute
    years since release (calc).'))
```

Simple Python Agentic AI Example

A basic autonomous agent that answers and searches

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import initialize_agent, load_tools

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0,
    google_api_key="YOUR_GOOGLE_API_KEY")

tools = load_tools(["serpapi", "llm-math"],
llm=llm, serpapi_api_key="YOUR_API_KEY")

agent = initialize_agent(tools=tools,llm=llm, agent="zero-shot-react-
    description", verbose=True)

result = agent.run("What's the weather in Paris and square root of 123?")
print(result)
```

Agent Demo: Weather-Based Match Scheduling

Goal: Build an agent that checks tomorrows weather and decides: **PLAY** or **POSTPONE** a match.

Agent Workflow:

- Input: location, date (tomorrow)
- Node 1: WeatherTool fetch forecast
- Node 2: LLM Reasoning evaluate conditions
- Node 3: DecisionTool return PLAY/POSTPONE
- Optional: Loops for self-correction

Why this example?

- Demonstrates agent nodes
- Uses tools in LangChain/LangGraph
- Shows prompt design for decision-making
- Easy real-world use case to explain agents

Components:

- WeatherTool (API call)
- DecisionTool (rule reasoning)
- LLM node (analysis)
- Agent graph

Prompt Used by the Match-Decision Agent

Agent Role + Inputs

```
You are MATCH-DECIDER, an agent that
decides whether to schedule an
    outdoor
match tomorrow at {location}.
Inputs:
- forecast: {weather_json}
- constraints: thresholds for rain,
  temperature, wind, player
      preferences.
```

Decision Rules + Output

```
Task:
1) Decide: "PLAY" or "POSTPONE".
2) Give 2-3 bullet reasons.
3) If POSTPONE -> suggest alternate slot.
Rules:
- Rain chance >= threshold => POSTPONE.
- Temp < min_temp => POSTPONE.
- Wind > wind_limit => POSTPONE.

Output JSON:
{ "decision": "...", "reason": "..." }
```

Weather Tool (Tomorrow's Forecast)

Purpose:

- Fetch tomorrows weather for a location
- Return simplified JSON used by LLM

Inputs:

- location name
- date (implicitly tomorrow)

Code

```
from langchain.tools import tool
import requests, json, datetime
@tool
def weather_tool(location: str):
    lat, lon = 12.97, 77.59 # Example coords
    url = f"https://api.openweathermap.org/.."
    r = requests.get(url)
    data = r.json()
    tomorrow = (datetime.date.today()
    + datetime.timedelta(days=1)).isoformat()
    return json.dumps({"date": tomorrow, "
        forecast": data.get("daily", []),})
```

Decision Tool + Assembling the Agent

Decision Tool

```
@tool
def scheduler_decision(forecast_json: str,constraints: str):
    fc = json.loads(forecast_json)
    cons = json.loads(constraints)
    daily = fc["forecast"][0]
    temp = daily["temp"]["min"]
    rain = daily["pop"]
    wind = daily["wind_speed"]
    if rain >= cons["precip_threshold"]
       or temp < cons["min_temp"]
       or wind > cons["wind_limit"]:
        return {"action":"POSTPONE"}
    return {"action":"PLAY"}
```

Decision Tool + Assembling the Agent

Building the Agent

```
from langchain import ChatOpenAI
from langchain.agents import create_agent

model = ChatOpenAI(model="gpt-4o-mini")
tools = [weather_tool, scheduler_decision]
agent = create_agent(model=model,tools=tools)
result = agent.run({
   "input": "Should we play tomorrow?",
   "location": "Bengaluru",
   "constraints": {
     "min_temp": 15,
     "precip_threshold": 0.3,
     "wind_limit": 8 } })
print(result)
```

What Makes Agent Prompts Different?

- Agents act autonomously.
- They need structured instructions.
- Prompts must define role, goals, tools, constraints.
- Clear action loops improve reliability.

Agent Identity (Role Prompt)

Clearly define what the agent is and its level of expertise.

Template:

```
You are <AGENT_NAME>, a highly reliable <ROLE>.
Your purpose is to <PRIMARY_OBJECTIVE>.
```

Example:

```
You are DATA-CURATOR, an expert in data cleaning.
Your purpose is to clean and validate datasets for ML workflows.
```

Goals and Success Criteria

Agents work better with measurable objectives.

Template:

```
Your goals:
1. <Goal 1>
2. <Goal 2>
3. <Goal 3>

Success means:
- <Success Condition 1>
- <Success Condition 2>
```

Example:

```
Your goals:
1. Identify errors in tabular data.
2. Repair missing values.
3. Produce a validated dataset.

Success means:
- No missing values.
- Transformations are logged.
```

Tools the Agent Can Use

Define available tools and when to use them.

Template:

```
You can use the following tools:
- <Tool A>: <What it does>
- <Tool B>: <What it does>

Use tools ONLY when needed and return valid tool input.
```

Example:

```
Tools:
- PythonTool: For running Python code.
- WebSearch: For searching information online.
```

Action Loop / Reasoning Style

Define how the agent should Think Plan Act Reflect.

Template:

```
Follow this loop:
1. THINK: Analyze the state.
2. PLAN: Break the goal into steps.
3. ACT: Use a tool or output text.
4. REFLECT: Evaluate and improve.
Repeat until goal is achieved.
```

Example:

```
If the tool result is wrong, reflect and retry.
```

Constraints and Output Format

Prevent hallucinations with strict rules.

Template:

```
Constraints:
- Do not fabricate facts.
- Do not assume missing data.
Output Format:
{
  "step": "...",
  "action": "...",
  "result": "..."
}
```

Example:

```
Never generate synthetic data unless allowed.
Return output as structured JSON.
```

Complete Sample Agent Prompt

A ready-to-use agent prompt.

Role, Goals, Success

```
You are RESEARCH-ASSISTANT,
an academic research agent.
GOALS:
1. Summarize literature.
2. Identify research gaps.
3. Propose methodology.
SUCCESS:
- All info cited.
- Output structured.
TOOLS:
- WebSearch
- PythonTool
```

Tools, Loop, Constraints, Output

```
ACTION LOOP:
1. THINK
2. PLAN
3. ACT
4. REFLECT
CONSTRAINTS:
- No hallucinated citations.
OUTPUT FORMAT:
{ "summary": "...",
  "gaps": [...],
  "methodology": "..." }
```

Best Practices for Agent Prompt Design

- Be explicit, not implicit.
- Use constraints to avoid drift.
- Use checklists and structured output.
- Add self-verification.
- Define memory usage if needed.

LangGraph

Why LangGraph?

- Traditional LangChain agents are powerful but:
  - hard to debug,
  - unpredictable (LLM decides planning),
  - stateless across steps,
  - not suited for multi-agent orchestration.
- **LangGraph** solves these by:
  - Representing agent workflows as **graphs** (nodes = steps),
  - Adding **stateful memory** across runs,
  - Supporting **multi-agent collaboration**,
  - Allowing **interrupt/resume**, retries, and human-in-loop.

What is LangGraph?

LangGraph is a framework for building:

- Reliable, stateful LLM workflows
- Multi-step, multi-tool agent systems
- Agents with recoverability (checkpoints, retries)
- Deterministic logic instead of LLM-driven planning

Key Idea: Represent agent logic as a **graph**: each node is a step (tool, retriever, LLM), edges define execution flow.

LangGraph: Basic Structure

- **State** Shared memory during execution.
- **Nodes** Computation units (LLM calls, tools, retrievers).
- **Edges** Execution transitions.
- **Graph** Connect nodes to define an agent workflow.

Example workflow:

1. User input
2. Retrieval node (FAISS)
3. Reasoning node (Gemini)
4. Output node

Sample LangGraph Implementation

```
import os
from pydantic import BaseModel, Field
from langgraph.graph import StateGraph, END
from langchain_google_genai import ChatGoogleGenerativeAI

# -------- State --------
class State(BaseModel):
    input: str = Field(...)
    output: str | None = None
# -------- Gemini LLM --------
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0,
    google_api_key=YOUR_GEMINI_API_KEY
)
```

Sample LangGraph Implementation

```
# -------- Node --------
def respond(state: State) -> State:
    raw = llm.invoke(state.input)
    text = raw.content if hasattr(raw, "content") else str(raw)
    return State(input=state.input, output=text)

# -------- Build Graph --------
graph = StateGraph(State)
graph.add_node("respond", respond)
graph.set_entry_point("respond")
graph.add_edge("respond", END)
app = graph.compile()

result = app.invoke({"input": "Explain LangGraph in one line"})
print(result['output'])
```

Agentic RAG

Agentic RAG Flow in LangGraph

- Node 1: **Plan Step** Decide: retrieve or compute?
- Node 2: **Retriever Step** Retrieve passages via FAISS
- Node 3: **Reason Step (Gemini)** Synthesize and answer
- Node 4: **Verification Step** Critic agent checks correctness

Each node is explicit and predictable unlike classical agents.

Multi-Agent Architecture (LangGraph + Gemini)

- **Planner Agent** Determines if retrieval is required and dispatches actions.
- **Retriever Agent** Uses FAISS to fetch evidence from local documents.
- **Reasoner Agent (Gemini)** Synthesizes an answer using LLM reasoning.
- **Critic Agent** Evaluates quality, correctness, and completeness.
- LangGraph coordinates these agents through a **stateful graph**.

Multi-Agent Workflow using LangGraph + Gemini

```
class MASState(BaseModel):
    query: str
    docs: list | None = None
    draft: str | None = None
    final: str | None = None
    action: str | None = None

# --- Planner Agent ---
def planner(state):
    if "year" in state.query or "released" in state.query:
        state.action = "retrieve"
    else:
        state.action = "reason_direct"
    return state
```

Multi-Agent Workflow using LangGraph + Gemini

```
# --- Retriever Agent ---
def retriever_agent(state):
    state.docs = db.similarity_search(state.query)
    return state

# --- Reasoner Agent (Gemini) ---
def reasoner_agent(state):
    ctx = "\n".join(d.page_content for d in (state.docs or []))
    prompt = f"Context: {ctx}\nQuestion: {state.query}\nAnswer:"
    raw = llm.invoke(prompt)
    state.draft = str(getattr(raw, "content", raw))
    return state
```

Multi-Agent Workflow using LangGraph + Gemini

```
# --- Critic Agent ---
def critic_agent(state):
    critique = llm.invoke(
        f"Evaluate this answer:\n{state.draft}\nProvide corrections only."
    )
    state.final = str(getattr(critique, "content", critique))
    return state

# --- Build Graph ---
graph = StateGraph(MASState)
graph.add_node("planner", planner)
graph.add_node("retriever", retriever_agent)
graph.add_node("reasoner", reasoner_agent)
graph.add_node("critic", critic_agent)
```

Multi-Agent Workflow using LangGraph + Gemini

```
graph.set_entry_point("planner")
graph.add_conditional_edges("planner",
    lambda s: s.action,
    {"retrieve": "retriever", "reason_direct": "reasoner"}
)
graph.add_edge("retriever", "reasoner")
graph.add_edge("reasoner", "critic")
graph.add_edge("critic", END)

app = graph.compile()
```

Running the Multi-Agent System

```
result = app.invoke({"query": "When was Product X released?"})

print("Draft answer:", result.draft)
print("Final corrected answer:", result.final)
```

Pipeline Summary:

- Planner decides retrieval required.
- Retriever fetches documents from FAISS.
- Reasoner Gemini generates answer.
- Critic Gemini evaluates and improves.

Why Multi-Agent Systems in LangGraph?

- LangGraph allows you to model agents as **graph nodes**.
- Each node can:
  - run different LLMs (e.g., Gemini),
  - use different tools or vector stores,
  - maintain separate state.
- Edges define:
  - communication paths,
  - collaboration rules,
  - control flow.
- Perfect for:
  - Planner $\rightarrow$ Retriever $\rightarrow$ Solver,
  - Solver $\rightarrow$ Critic $\rightarrow$ Refiner loops,
  - multi-turn workflows with checkpoints.

Benefits of Multi-Agent Systems in LangGraph

- Decompose complex tasks into specialized agents.
- Achieve higher-quality answers through:
  - Retrieval grounding,
  - Multi-step reasoning,
  - Critique and refinement.
- Agents operate independently but share state.
- Easy to debug each node's behavior is visible.
- Works seamlessly with Gemini models.

Challenges

- Alignment and Safety
- Hallucinations and Goal Drift
- Evaluation and Benchmarking
- Multi-agent conflicts

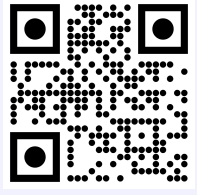
Conclusion

- **LangGraph** provides a reliable and structured framework for building advanced agentic systems.
- **Multi-agent workflows** (Planner, Retriever, Reasoner, Critic) improve accuracy, robustness, and interpretability.
- **Gemini LLMs** combine high-quality reasoning with efficient tool use, making them ideal for multi-agent and RAG-based pipelines.
- **FAISS integration** enables fast, scalable retrieval to ground answers in factual context.
- Overall: LangGraph + Gemini = a powerful foundation for **autonomous, verifiable, and production-ready Agentic AI**.

Thank You!

For your attention and participation !!!

Dr. Shailesh Sivan



Email: shaileshsivan@gmail.com

Phone: +91-8907230664

Website: www.shaileshsivan.info