Day 3: Agentic Systems

Agentic AI Intensive Training

5-Day Program

Today's Agenda

Morning: Agent Foundations

- What are agents?
- Core architectures
- ReAct pattern

Afternoon: Tool Use & Multi-Agent

- Function calling
- Tool integration
- Agent collaboration

Evening: Building Agents

- Hands-on: Build research agent

What is an Agent?

Definition

- Autonomous system that perceives and acts
- Uses LLM for reasoning/decision-making
- Interacts with tools/environment
- Pursues goals over multiple steps

Key Characteristics

- Goal-directed
- Adaptive
- Multi-step reasoning
- Tool usage

Agent vs Standard LLM

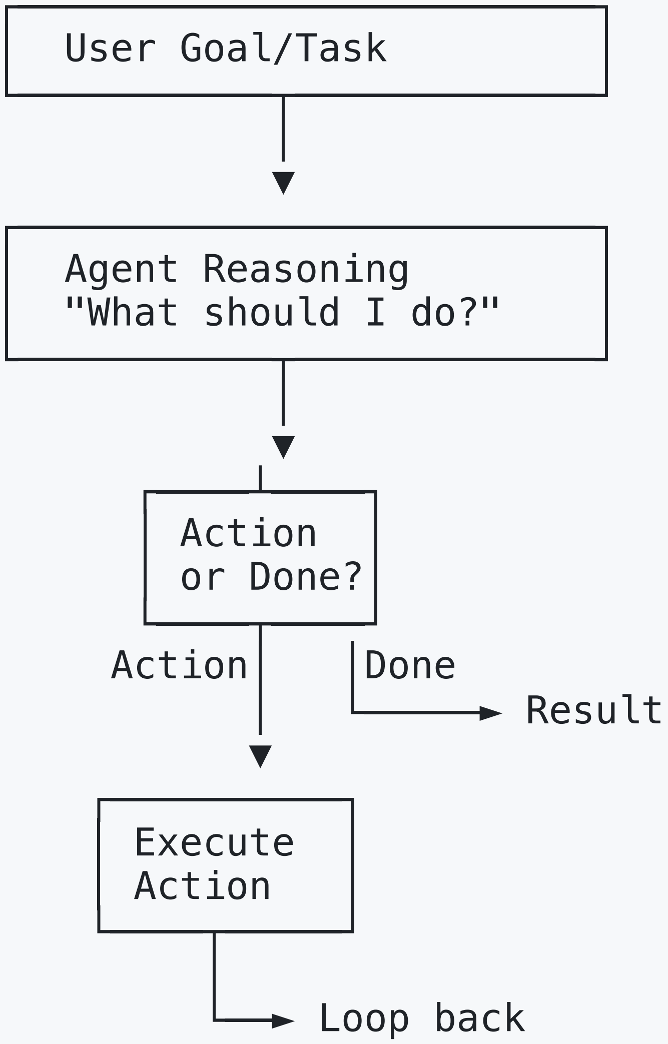
Standard LLM

- Single request/response
- No environment interaction
- Stateless (without implementation)
- Direct output

Agent

- Multi-turn reasoning loop
- Tool/API calls
- Maintains state
- Iterative problem-solving

Simple Agent Loop



Core Agent Architectures

1. **ReAct** (Reason + Act)
    - Interleave reasoning and actions
    - Explicit thought traces
2. **Plan-and-Execute**
    - Create plan first
    - Execute steps sequentially
3. **Reflexion**
    - Self-reflection and refinement
    - Learn from mistakes

ReAct Pattern

Reason → Act → Observe → Repeat

```
Thought: I need to find population of Tokyo
Action: search("Tokyo population 2024")
Observation: Tokyo has 14 million people

Thought: Now I need Paris population
Action: search("Paris population 2024")
Observation: Paris has 2.1 million people

Thought: I can now compare
Answer: Tokyo is ~6.7x larger than Paris
```

Python: Simple ReAct Agent

```python
def react_agent(question, tools, max_steps=5):
    context = f"Question: {question}\n"

    for step in range(max_steps):
        # Reasoning
        thought = llm(f"{context}\nThought:")
        context += f"\nThought: {thought}"

        # Decide action or finish
        if "Answer:" in thought:
            return extract_answer(thought)

        # Execute action
        action = parse_action(thought)
        result = execute_tool(action, tools)
        context += f"\nObservation: {result}"

    return "Max steps reached"
```

TypeScript: Simple ReAct Agent

```typescript
async function reactAgent(
  question: string,
  tools: Map<string, Function>,
  maxSteps = 5
): Promise<string> {
  let context = `Question: ${question}\n`;

  for (let step = 0; step < maxSteps; step++) {
    const thought = await llm(`${context}\nThought:`);
    context += `\nThought: ${thought}`;

    if (thought.includes("Answer:")) {
      return extractAnswer(thought);
    }

    const action = parseAction(thought);
    const result = await executeTools(action, tools);
    context += `\nObservation: ${result}`;
  }
  return "Max steps reached";
}
```

Tools and Functions

What are Tools?

- Functions agents can call
- Access external data/systems
- Perform actions (API calls, calculations)

Common Tool Categories

- Search (web, docs, databases)
- Computation (calculator, code exec)
- Data retrieval (APIs, databases)
- Actions (send email, file ops)

Tool Definition Schema

```
tools = [
    {
        "name": "web_search",
        "description": "Search the web for information",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "Search query"
                }
            },
            "required": ["query"]
        }
    }
]
```

OpenAI Function Calling

```python
from openai import OpenAI

client = OpenAI()

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current weather",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {"type": "string"},
                "unit": {"type": "string", "enum": ["C", "F"]}
            },
            "required": ["location"]
        }
    }
}]
```

Python: Function Calling Flow

```python
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "Weather in NYC?"}],
    tools=tools
)

# Check if function called
tool_call = response.choices[0].message.tool_calls[0]
if tool_call:
    function_name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)

    # Execute function
    result = get_weather(**args)

    # Send result back
    messages.append(response.choices[0].message)
    messages.append({
        "role": "tool",
        "tool_call_id": tool_call.id,
        "content": result
    })
```

TypeScript: Function Calling

```typescript
const response = await client.chat.completions.create({
  model: "gpt-4o",
  messages: [{ role: "user", content: "Weather in NYC?" }],
  tools: tools
});

const toolCall = response.choices[0].message.tool_calls?.[0];
if (toolCall) {
  const functionName = toolCall.function.name;
  const args = JSON.parse(toolCall.function.arguments);

  // Execute function
  const result = await getWeather(args);

  // Send result back
  messages.push(response.choices[0].message);
  messages.push({
    role: "tool",
    tool_call_id: toolCall.id,
    content: result
  });
}
```

Building a Calculator Tool

```python
def calculator_tool(operation: str, x: float, y: float) -> float:
    """Perform mathematical operations"""
    ops = {
        "add": lambda a, b: a + b,
        "subtract": lambda a, b: a - b,
        "multiply": lambda a, b: a * b,
        "divide": lambda a, b: a / b if b != 0 else None
    }
    return ops.get(operation, lambda a, b: None)(x, y)

# Tool schema
calculator_schema = {
    "name": "calculator",
    "description": "Perform math operations",
    "parameters": { ... }
}
```

Building a Search Tool

```python
import requests

def web_search(query: str, num_results: int = 3) -> str:
    """Search web and return results"""
    # Using a search API (e.g., Serper, Tavily)
    response = requests.post(
        "https://api.search.com/search",
        json={"q": query, "num": num_results},
        headers={"Authorization": f"Bearer {API_KEY}"}
    )

    results = response.json()["results"]
    return "\n\n".join([
        f"{r['title']}\n{r['snippet']}"
        for r in results
    ])
```

Plan-and-Execute Pattern

```
1. Planning Phase
   ┌──────────────────┐
   │ Create detailed  │
   │ step-by-step     │
   │ plan             │
   └────────┬─────────┘
            │
2. Execution Phase
   ┌────────▼─────────┐
   │ Execute step 1   │
   ├──────────────────┤
   │ Execute step 2   │
   ├──────────────────┤
   │ Execute step 3   │
   └────────┬─────────┘
            │
3. Synthesis
   ┌────────▼─────────┐
   │ Combine results  │
   └──────────────────┘
```

Python: Plan-and-Execute

```python
def plan_and_execute(task: str, tools: dict):
    # Step 1: Create plan
    plan_prompt = f"Create step-by-step plan for: {task}"
    plan = llm(plan_prompt)
    steps = parse_steps(plan)

    # Step 2: Execute each step
    results = []
    for step in steps:
        result = execute_step(step, tools)
        results.append(result)

    # Step 3: Synthesize
    synthesis = llm(f"""
    Task: {task}
    Results: {results}
    Provide final answer.
    """)
    return synthesis
```

Memory in Agents

Types of Memory

1. **Short-term** (Working memory)
    - Current conversation context
    - Recent observations
2. **Long-term** (Persistent)
    - Past conversations
    - Learned facts/patterns
    - User preferences
3. **Procedural**
    - How to use tools

Python: Agent Memory

```python
class AgentMemory:
    def __init__(self):
        self.short_term = []  # Recent messages
        self.long_term = {}   # Vector DB
        self.facts = []       # Extracted facts

    def add_to_short_term(self, item, max_size=10):
        self.short_term.append(item)
        if len(self.short_term) > max_size:
            # Archive to long-term
            self.archive(self.short_term.pop(0))

    def archive(self, item):
        # Store in vector DB
        embedding = get_embedding(item)
        self.long_term[hash(item)] = embedding

    def retrieve(self, query, k=3):
        # Semantic search
        return search_similar(query, self.long_term, k)
```

Reflexion Pattern

Self-reflection for improvement

```
Attempt 1
├─ Execute task
├─ Evaluate result
└─ Identify failures

Reflect
├─ Analyze what went wrong
├─ Generate improvement plan
└─ Update strategy

Attempt 2
├─ Apply lessons learned
├─ Execute with improvements
└─ Better result
```

Python: Reflexion Agent

```python
def reflexion_agent(task, max_attempts=3):
    reflections = []

    for attempt in range(max_attempts):
        # Execute task
        result = execute_task(task, reflections)

        # Evaluate
        success = evaluate_result(result, task)
        if success:
            return result

        # Reflect on failure
        reflection = llm(f"""
        Task: {task}
        Attempt: {result}
        What went wrong? How to improve?
        """)
        reflections.append(reflection)

    return "Failed after max attempts"
```

Multi-Agent Systems

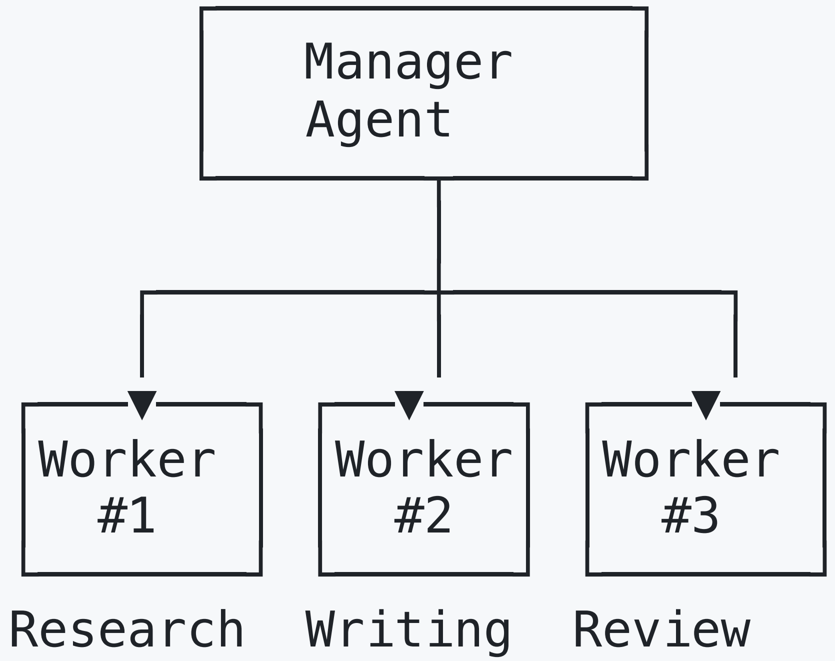
Why Multiple Agents?

- Specialization (each agent = expert)
- Parallelization (concurrent tasks)
- Modularity (easier to debug/improve)
- Collaboration (agents work together)

Patterns

- Hierarchical (manager + workers)
- Peer-to-peer (collaborate as equals)
- Sequential (pipeline/workflow)

Hierarchical Multi-Agent



Manager: Delegates tasks

Workers: Specialized execution

Python: Multi-Agent System

```python
class Agent:
    def __init__(self, name, role, tools):
        self.name = name
        self.role = role
        self.tools = tools

    def execute(self, task):
        prompt = f"You are {self.role}. Task: {task}"
        return llm_with_tools(prompt, self.tools)

class ManagerAgent:
    def __init__(self, workers):
        self.workers = workers

    def delegate(self, task):
        # Decompose task
        subtasks = self.plan(task)

        # Assign to workers
        results = []
        for subtask in subtasks:
            worker = self.select_worker(subtask)
            result = worker.execute(subtask)
            results.append(result)

        return self.synthesize(results)
```

Agent Communication

Message Passing

```
class Message:
    def __init__(self, sender, receiver, content):
        self.sender = sender
        self.receiver = receiver
        self.content = content
        self.timestamp = time.time()

class MessageBus:
    def __init__(self):
        self.messages = []

    def send(self, msg):
        self.messages.append(msg)
        self.deliver(msg)

    def deliver(self, msg):
        msg.receiver.receive(msg)
```

Agent Collaboration Example

Research Paper Writer

```
┌────────────┐
│ Researcher │   Finds papers, data
└─────┬──────┘
      │
      ▼
┌────────────┐
│ Analyzer   │   Extracts insights
└─────┬──────┘
      │
      ▼
┌────────────┐
│ Writer     │   Drafts sections
└─────┬──────┘
      │
      ▼
┌────────────┐
│ Reviewer   │   Checks quality
```

Tool Safety & Validation

Security Concerns

- Malicious tool calls
- Data leakage
- Unintended actions
- Cost overruns

Mitigation

```
def safe_tool_execution(tool_name, args):
    # Validate tool exists
    if tool_name not in ALLOWED_TOOLS:
        return "Tool not allowed"

    # Validate arguments
    if not validate_args(args):
        return "Invalid arguments"
```

Error Handling in Agents

```python
def robust_agent_step(agent, task):
    max_retries = 3

    for attempt in range(max_retries):
        try:
            result = agent.execute(task)
            if validate_result(result):
                return result
            else:
                # Invalid result, retry with feedback
                task = f"{task}\nPrevious attempt invalid: {result}"

        except ToolError as e:
            logger.error(f"Tool error: {e}")
            # Try alternative tool

        except TimeoutError:
            logger.error("Timeout")
            # Simplify task

    return "Failed after retries"
```

Agent Evaluation Metrics

Performance

- Task success rate
- Steps to completion
- Token usage
- Latency

Quality

- Answer accuracy
- Tool usage appropriateness
- Reasoning coherence

Cost

Python: Agent Metrics

```python
class AgentMetrics:
    def __init__(self):
        self.steps = 0
        self.tool_calls = 0
        self.tokens = 0
        self.start_time = time.time()

    def record_step(self):
        self.steps += 1

    def record_tool_call(self, tool_name):
        self.tool_calls += 1

    def record_tokens(self, count):
        self.tokens += count

    def get_summary(self):
        return {
            "steps": self.steps,
            "tool_calls": self.tool_calls,
            "tokens": self.tokens,
            "duration": time.time() - self.start_time
        }
```

LangChain Agents

```python
from langchain.agents import AgentExecutor, create_openai_functions_agent
from langchain_openai import ChatOpenAI
from langchain.tools import Tool

# Define tools
tools = [
    Tool(
        name="Calculator",
        func=calculator,
        description="Useful for math"
    ),
    Tool(
        name="Search",
        func=web_search,
        description="Search the web"
    )
]

# Create agent
llm = ChatOpenAI(model="gpt-4o")
agent = create_openai_functions_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)
```

LangGraph for Complex Agents

```python
from langgraph.graph import Graph

# Define agent workflow
workflow = Graph()

# Add nodes (agent steps)
workflow.add_node("research", research_node)
workflow.add_node("analyze", analyze_node)
workflow.add_node("write", write_node)

# Add edges (flow)
workflow.add_edge("research", "analyze")
workflow.add_edge("analyze", "write")

# Set entry point
workflow.set_entry_point("research")

# Compile
app = workflow.compile()
result = app.invoke({"task": "Write report on AI"})
```

Crew AI Framework

```python
from crewai import Agent, Task, Crew

# Define agents
researcher = Agent(
    role="Researcher",
    goal="Find relevant information",
    tools=[search_tool],
    verbose=True
)

writer = Agent(
    role="Writer",
    goal="Create engaging content",
    tools=[],
    verbose=True
)

# Define tasks
task1 = Task(description="Research AI trends", agent=researcher)
task2 = Task(description="Write blog post", agent=writer)

# Create crew
crew = Crew(agents=[researcher, writer], tasks=[task1, task2])
result = crew.kickoff()
```

AutoGPT-style Agent

Continuous autonomous operation

```python
class AutoAgent:
    def __init__(self, goal):
        self.goal = goal
        self.memory = []
        self.tools = load_tools()

    def run(self):
        while not self.is_goal_achieved():
            # Think
            thoughts = self.think()

            # Plan
            plan = self.plan(thoughts)

            # Act
            action = self.select_action(plan)
            result = self.execute(action)

            # Remember
            self.memory.append((action, result))
```

Agent Limitations

Current Challenges

- Unreliable long-term reasoning
- Expensive (many LLM calls)
- Hard to debug
- Can go off-track
- Safety concerns

Mitigation Strategies

- Max steps limit
- Regular checkpoints
- Human-in-the-loop
- Sandboxed execution

Human-in-the-Loop

```python
class HumanApprovalAgent:
    def __init__(self):
        self.actions_requiring_approval = [
            "send_email",
            "delete_file",
            "make_payment"
        ]

    def execute_action(self, action, args):
        if action in self.actions_requiring_approval:
            print(f"Agent wants to: {action}({args})")
            approval = input("Approve? (y/n): ")
            if approval.lower() != 'y':
                return "Action rejected by user"

        return self.run_action(action, args)
```

Workshop: Research Agent

Goal: Build agent that researches topics

Requirements

1. Search web for information
2. Extract key facts
3. Synthesize findings
4. Cite sources

Tools Needed

- Web search
- Content extraction
- Note-taking

Research Agent Architecture

```
User Query
    │
    ▼
┌─────────────────┐
│  Query          │
│  Understanding  │
└─────────────────┘
        │
        ▼
┌─────────────────┐
│  Search & Gather│
│  (multiple      │
│   sources)      │
└─────────────────┘
        │
        ▼
┌─────────────────┐
│  Extract & Store│
│  (key facts)    │
└─────────────────┘
        │
        ▼
┌─────────────────┐
│  Synthesize     │
│  Report         │
└─────────────────┘
```

Research Agent Starter

```python
class ResearchAgent:
    def __init__(self):
        self.tools = {
            "search": web_search_tool,
            "extract": extract_content_tool
        }
        self.findings = []

    def research(self, topic: str):
        # 1. Generate search queries
        queries = self.generate_queries(topic)

        # 2. Search and collect
        for query in queries:
            results = self.tools["search"](query)
            self.findings.extend(results)

        # 3. Synthesize
        return self.synthesize_report(topic, self.findings)
```

Exercise 1: Basic Agent

Build: Simple ReAct agent with calculator

```python
def exercise1():
    tools = {"calculator": calculator_tool}

    question = """
    If I buy 3 items at $12.50 each and 2 items
    at $8.75 each, what's my total?
    """

    agent = ReactAgent(tools)
    answer = agent.solve(question)
    print(answer)
```

Expected: Agent reasons through steps, uses calculator

Exercise 2: Multi-Tool Agent

Build: Agent with search + calculator

```
def exercise2():
    tools = {
        "search": web_search_tool,
        "calculator": calculator_tool
    }

    question = """
    What's the GDP of the US and China combined?
    """

    agent = ReactAgent(tools)
    answer = agent.solve(question)
    print(answer)
```

Expected: Search GDPs, then calculate sum

Exercise 3: Research Agent

Build: Full research agent

```
def exercise3():
    agent = ResearchAgent(
        tools=["search", "extract", "summarize"]
    )

    report = agent.research(
        "What are the main applications of "
        "transformers in NLP?"
    )

    print(report)
```

Expected: Multi-source synthesis with citations

Best Practices

Agent Design

- Start simple, add complexity gradually
- Clear stopping conditions
- Limit max steps
- Comprehensive logging
- Error recovery

Tool Design

- Clear descriptions
- Input validation
- Timeout handling
- Idempotency when possible

Debugging Agents

Common Issues

1. Infinite loops → Add max steps
2. Wrong tool selection → Improve descriptions
3. Poor reasoning → Better prompts
4. Hallucinated actions → Validate tool calls
5. Context loss → Better memory

Debug Tools

- Verbose logging
- Step-by-step inspection
- Replay functionality
- Metrics dashboard

Production Considerations

Before Deployment

- Extensive testing
- Rate limiting
- Cost monitoring
- Safety guardrails
- Rollback plan

Monitoring

- Success/failure rates
- Average steps
- Token usage
- Latency

Advanced Agent Patterns

Upcoming Topics

- RAG-enhanced agents
- Agents with persistent memory
- Multi-modal agents
- Agent-based evaluations
- Production deployment

Day 4: RAG + Evaluation

Day 5: Production systems

Resources

Frameworks

- LangChain/LangGraph
- LlamaIndex
- AutoGPT
- CrewAI
- AgentGPT

Papers

- ReAct (Yao et al., 2022)
- Reflexion (Shinn et al., 2023)
- Toolformer (Schick et al., 2023)

Q&A

Questions?

Tomorrow: Day 4 - RAG & Evaluation

- Retrieval-Augmented Generation
- Vector databases
- Evaluation frameworks
- Testing strategies

Thank You

Excellent progress!

Homework

- Complete all 3 exercises
- Experiment with different tools
- Read ReAct paper
- Design your own agent use case

See you tomorrow!