GAI-3101 Lesson Guide

Crafting Custom Agentic AI Solutions

ascendient learning

© 2025 Ascendient, LLC

Revision 4.1.0-asc published on 2025-11-24.

No part of this book may be reproduced or used in any form or by any electronic, mechanical, or other means, currently available or developed in the future, including photocopying, recording, digital scanning, or sharing, or in any information storage or retrieval system, without permission in writing from the publisher.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

To obtain authorization for any such activities (e.g., reprint rights, translation rights), to customize this book, or for other sales inquiries, please contact:

Ascendient, LLC
1 California Street Suite 2900
San Francisco, CA 94111
https://www.ascendientlearning.com/

USA: 1-877-517-6540, email: getinfousa@ascendientlearning.com
Canada: 1-877-812-8887 toll free, email: getinfo@ascendientlearning.com

Table of Contents

 © Ascendient, LLC

 © Ascendient, LLC

 © Ascendient, LLC

 © Ascendient, LLC

© Ascendient, LLC

GAI-3101 Introduction

MODULE 1

- ✓ Understand the benefits and characteristics of Agentic AI.
- ✓ Deploy agents to accomplish tasks a single GenAI model cannot.
- ✓ Identify and apply design patterns for Agentic AI applications.
- ✓ Develop agents that learn, adapt, and interact with other agents.
- ✓ Ensure security, safety, and robustness in Agentic AI applications.

1.1. Agenda

1

Introduction to Agentic AI

Defining Agentic AI & its characteristics, benefits, challenges, & real-world applications.

2

Architectures & Frameworks

Architectures, comparisons, reactive, deliberative, hybrid, & frameworks like AutoGen, LangGraph, CrewAI.

3

Memory & Context Management

The Role of Memory in Agentic AI, types, challenges, techniques.

4

Tool Use & Function Calling

Exploring tool use & function calls in Agentic AI, including OODA Loop & validation methods.

5

Planning & Reasoning

Role of Planning & Reasoning in Agentic AI, types, hierarchical planning, & reasoning in uncertain environments.

© Ascendient, LLC

Introduction to Agentic AI

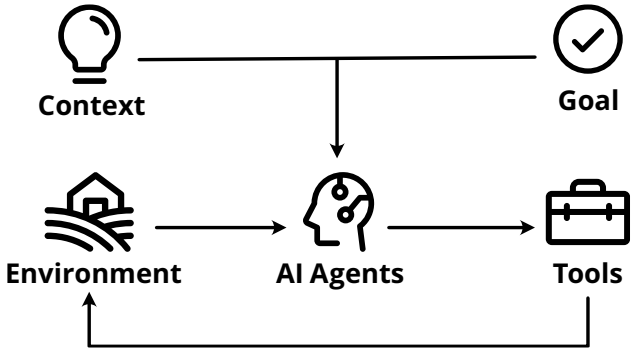
MODULE 2

- ✓ Define Agentic AI and its Key Characteristics
- ✓ Differentiate Agentic AI from Traditional AI and RPA
- ✓ Understand Benefits and Challenges of Agentic AI
- ✓ Review Real-World Applications of Agentic AI
- ✓ Discuss Potential Impact of Agentic AI on Key Sectors
- ✓ Agentic AI for Solving Complex Problems

2.1. Defining Agentic AI

Agentic AI is:

- AI agents
- Operating in an environment
- From a context
- Using tools
- To achieve goals



2.2. Components of Agentic AI: AI Agents

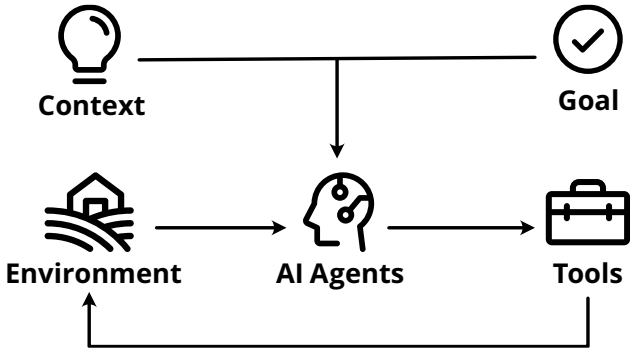
An **agent** is:

1. Authorized to act for another
2. Capable of action
3. A means to an end

An **agent** is:

1. A person authorized to act on behalf of another person
2. A being with the capacity to act
3. A means by which something is done or caused

An **AI agent** is an AI system with the capacity to act on behalf of a user.



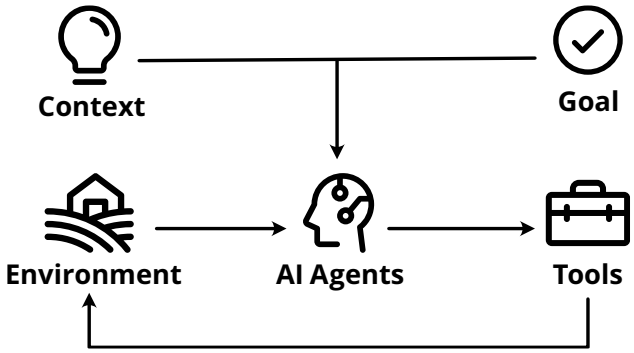
© Ascendient, LLC

2.3. Components of Agentic AI: Environment

The **environment** is the external information that the AI agents interact with:

- Databases
- APIs
- User interfaces
- Sensors
- etc.

The environment provides the necessary inputs for AI agents to perceive, analyze, and act upon.



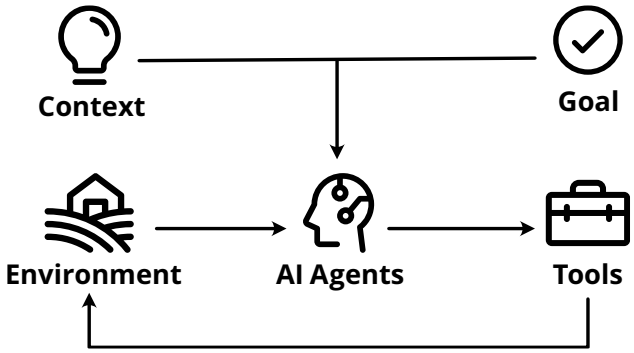
© Ascendient, LLC

2.4. Components of Agentic AI: Context

The **context** is the broader setting in which the AI agents operate:

- Industry
- Organization
- Market
- Regulatory environment
- etc.

The context shapes the goals, constraints, and opportunities for the AI agents.



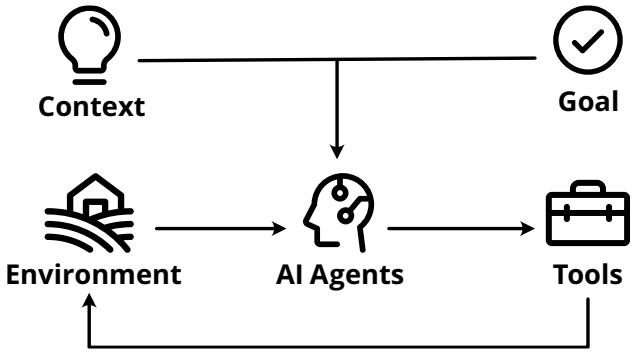
© Ascendient, LLC

2.5. Components of Agentic AI: Tools

The **tools** are the software and hardware components that enable the AI agents to perform tasks:

- APIs
- Databases
- Functions
- Actuators
- etc.

The tools provide the means for the AI agents to interact with the environment and achieve their goals.

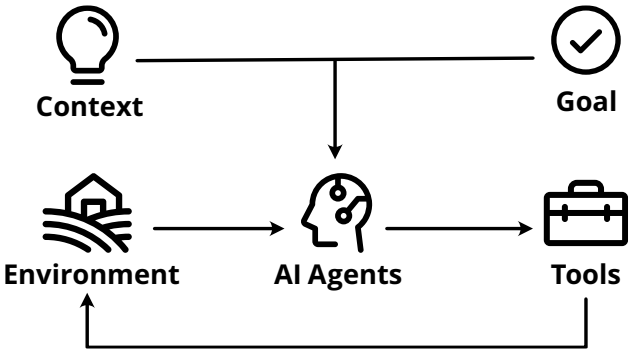


© Ascendient, LLC

2.6. Components of Agentic AI: Goals

The **goals** are the specific objectives that the AI agents aim to achieve:

- Create a report
- Make a recommendation
- Execute a transaction
- etc.



> ⓘ Note
>
> Goals can be predefined or learned through interaction with the environment.

© Ascendient, LLC

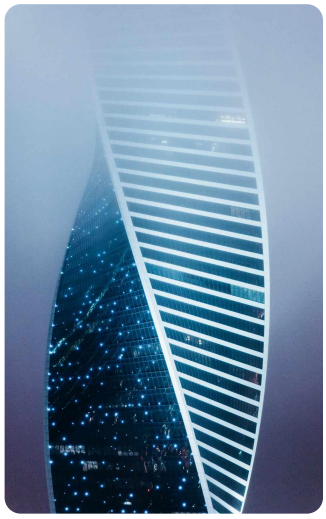
2.7. Components of Agentic AI: Example

Component	Example
AI Agents	Commit Analysis Agent
Environment	Issue Tracking System, Git Repository, Slack
Context	Software Development Team
Tools	APIs for Git, Slack, and Issue Tracking
Goals	Analyze git commits for code quality issues

- A team of software developers uses an AI agent to analyze git commits for code quality issues.
- When a commit is made, the agent is triggered.
- It analyzes the commit message, code changes, and issues.
- If a code quality issue is detected:
  - It comments on the issue - in the issue tracking system.
  - It sends a notification to the developer.

© Ascendient, LLC

2.8. Characteristics of Agentic AI



Agency

Agents perform tasks with minimal oversight or direction.

Specialization

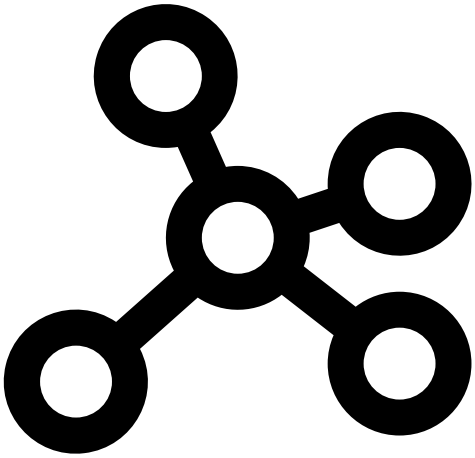
Agents perform tasks related to a specific role or function.

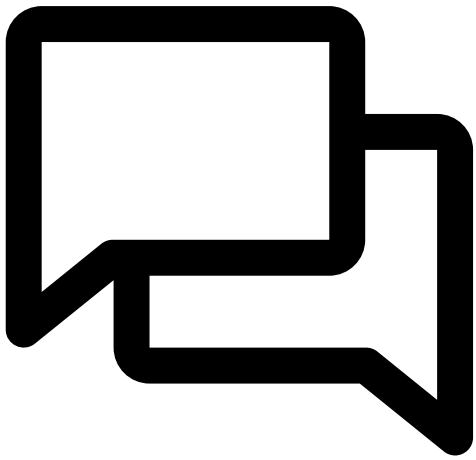
Goal

Agents plan and complete tasks to achieve a goal.

© Ascendient, LLC

2.9. Agentic AI vs. Generative AI





Agentic AI

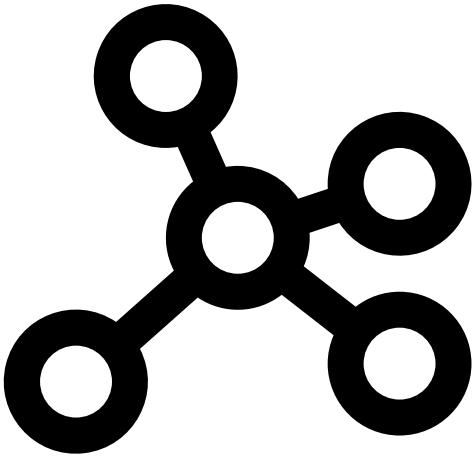
- Focuses on achieving goals.
- Responds to environmental stimuli.
- Makes decisions autonomously.
- Uses tools to interact with the environment.
- Examples: Email-flagging Agent, Commit Analysis Agent

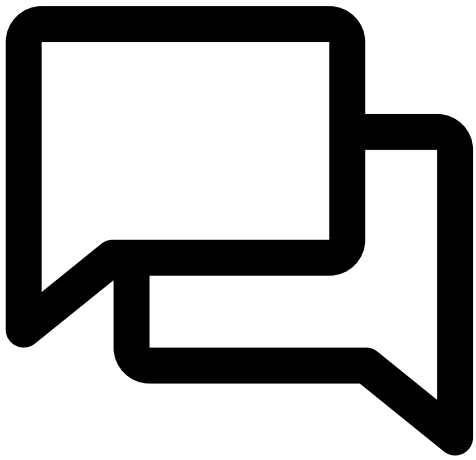
Generative AI

- Focuses on creating content
- Responds to direct user input
- Makes minimal decisions
- Uses tools sparingly
- Examples: ChatGPT, Gemini, DALL-E

© Ascendient, LLC

2.10. Agentic AI vs. Robotic Process Automation





Agentic AI

- Makes decisions autonomously
- Adapts to changing environments
- Handles complex, unstructured tasks
- Learns over tasks (optionally)
- Less human intervention

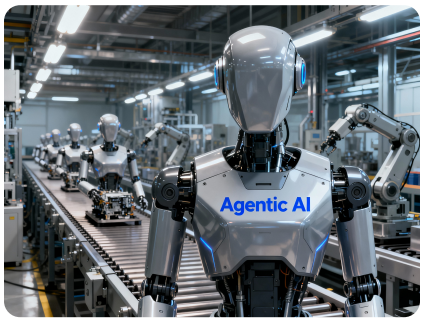
Robotic Process Automation

- Follows predefined workflow
- Often fails when environments change
- Excels at repetitive, structured tasks
- No learning from previous tasks
- Needs human intervention for exceptions

© Ascendient, LLC

2.11. Benefits of Agentic AI (1/3)

- **Agents can be created and rolled-out incrementally.**
  - Start small with well-defined tasks or workflows.
  - Lower initial investment and disruption to existing operations.
  - Teams can identify challenges before scaling up.
  - Adoption can begin with human oversight.



2.12. Benefits of Agentic AI (2/3)

- **Agents can be improved individually.**
  - Agents are inherently a modular design.
  - Agents interact based on a flexible contract (usually).
  - Agents can be tailored for specific tasks or domains.

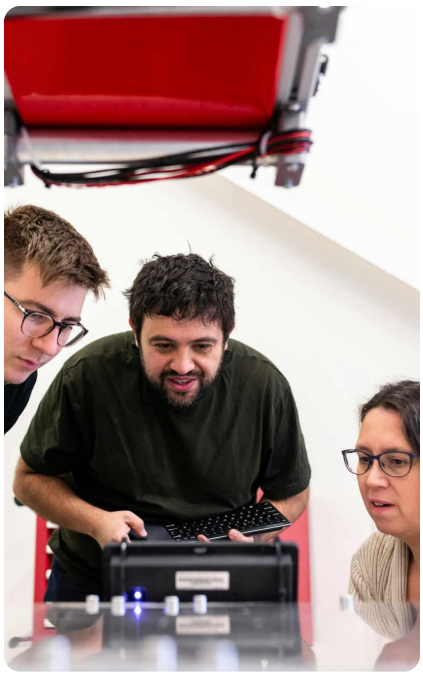
© Ascendient, LLC

2.13. Benefits of Agentic AI (3/3)

- **Even small systems can have a big impact.**
  - Complete a single step in a workflow.
  - Filter and prioritize tasks.
  - Provide a first-pass at the final output.
  - etc.

2.14. Challenges of Agentic AI (1/3)

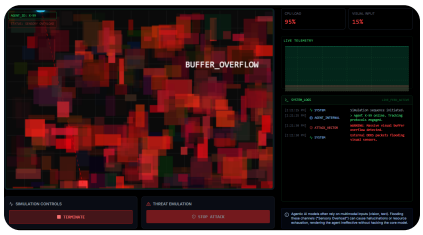
- **Agentic AI can be complex**
  - Predicting agent behavior can be difficult.
  - Agents may interact in unexpected ways.
  - Validating agent behavior can be challenging.



© Ascendient, LLC

2.15. Challenges of Agentic AI (2/3)

- **Agentic AI has security risks**
  - All AI is vulnerable to adversarial attacks.
  - Agentic systems often have access to sensitive data.
  - Agents may perform unexpected actions.



2.16. Challenges of Agentic AI (3/3)

- **Implementation and management can be difficult**
  - Poor quality data can lead to poor agent performance.
  - Agents may require significant computational resources or API costs.
  - Building trust in agent actions can be challenging.
  - Libraries and methodologies are still evolving.

© Ascendient, LLC

2.17. Real-world Applications of Agentic AI

- **Email** — reads incoming emails, flags urgent messages, and drafts responses.
- **Coding** — Link commits to issues, analyze code quality, suggest improvements, create pull requests.
- **Management** — Balance workloads, prioritize tasks, report on progress.
- **Helpdesk** — Respond to common queries, escalate complex issues, and provide status updates.
- **Sales** — Prioritize leads, schedule meetings, provide customer insights.

2.18. Potential Impact of Agentic AI

- **Healthcare** — Assist with patient care, analyze medical images, and manage patient records.
- **Finance** — Analyze company reports and disclosure, track news and social media for potential disruptions, predict market trends.
- **Consulting** — Analyze client data, provide recommendations, and automate reporting.
- **Insurance** — Convert plain-text inquiries into quotes and claims, analyze risk factors and trends, and review policies.
- **Government** — Provide real-time help to the public, analyze data for policy decisions, and automate administrative tasks.

© Ascendient, LLC

2.19. Agentic AI for Solving Complex Problems

Agentic AI solves complex problems by:

- **Reasoning and Plannning** — Agents break down complexity by analyzing problems and creating multi-step plans to achieve their goal.
- **Interaction & Collaboration** — Agents communicate with each other—and sometimes humans—to solve problems too complex for a single agent.
- **Proactive & Reactive Behavior** — Agents both initiate actions and respond to changes in the environment.
- **Memory & Context** — Agents with memory use past experiences and context to inform their decisions.
- **Tool-Use** — Agents use external data and tools to expand their capabilities.
- **Learning & Adaptation** — Agents with learning capabilities can improve their performance over time.

2.20. Conclusion

- **Agentic AI**
  - Is an AI agent operating within an environment from a context using tools to achieve goals.
  - Or a collection of AI agents operating within a context to achieve specific goals.
- Components include agents, environment, context, tools, and goals.
- Unique characteristics distinguish it from other AI paradigms.
- Benefits include incremental deployment, individual improvement, and significant impact.
- Challenges involve complexity, security, and implementation.
- Applications span various industries and functions.
- It solves complex problems through reasoning, planning, and collaboration.

© Ascendient, LLC

Agent Architectures and Frameworks

MODULE 3

- ✓ Identify the need for Agentic AI architecture.
- ✓ Describe the characteristics of reactive, deliberative, and hybrid architectures.
- ✓ Compare the strengths and weaknesses of different AI architectures.
- ✓ Implement a basic reactive agent in Python.
- ✓ Evaluate the suitability of different Agentic frameworks for AI development.

3.1. The Need for Agentic AI Architecture

- AI architecture is the foundational design that determines how an AI system functions.
- The right architecture is essential to how efficiently the AI can perform tasks, adapt to changes, and scale as needed.
- A good architecture is crucial for:
  - Making fast and efficient decisions.
  - Scaling the system to handle more complex tasks.
  - Adapting to changes in the environment.
  - The architecture type chosen will significantly impact the framework used to implement the AI system.

3.2. Introduction to Agentic AI Architectures

Agentic systems can be classified into three main architectures based on how they make decisions and act in the world:

Reactive

Reacts to events or inputs immediately, without planning ahead.

Deliberative

Makes decisions by considering long-term goals and potential future outcomes, which requires more processing power.

Hybrid

Combines reactive and deliberative methods, allowing the system to respond quickly while also planning for future actions.

© Ascendient, LLC

3.3. Architecture Strengths and Weaknesses

© Ascendient, LLC

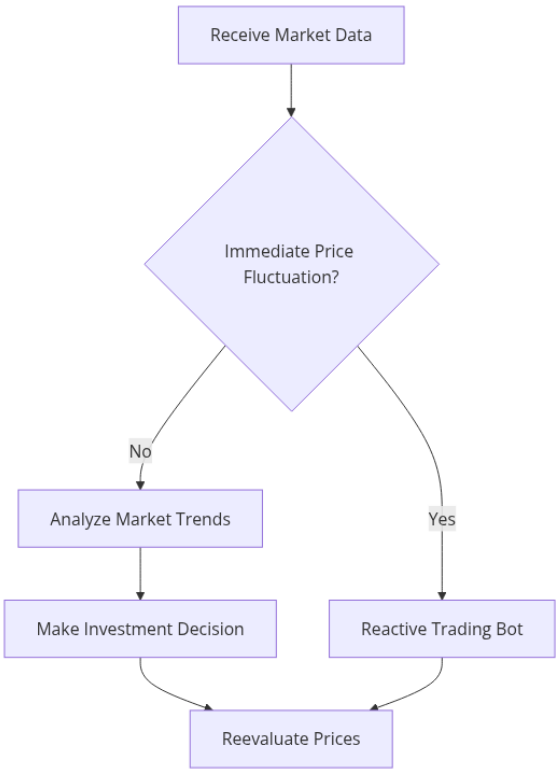
Architecture	Strengths	Weaknesses	Use Case Example
Reactive	• Fast response times • Simple design • Low compute requirements • Well-suited for real-time actions	• Lacks long-term planning • Limited adaptability • May not handle complex tasks well	• Reactive Agents in autonomous drones • High-frequency trading agents • Real-time monitoring
Deliberative	• Better decision-making • Considers long-term consequences • Can handle complex tasks • Adaptable to changing environments	• Higher compute requirements • Slower response times • More complex to design and implement • Requires more processing power for planning	• Chess-playing system • Market trend analysis • Autonomous exploration robots

© Ascendient, LLC

Architecture	Strengths	Weaknesses	Use Case Example
Hybrid	• Combines speed and planning • Balances immediate responses with strategy • Adapts to environment	• More complex to implement • Requires careful coordination • Higher compute requirements	• Self-driving cars • Healthcare Agents • Advanced robotics

3.4. Case Study: AI in Stock Trading

- **Reactive**:
  - High-frequency trading bots that react to price fluctuations in real time.
- **Deliberative**:
  - AI that analyzes market trends and forecasts before making investment decisions.
- **Hybrid**:
  - A system that reacts to price changes but also plans based on long-term predictions.



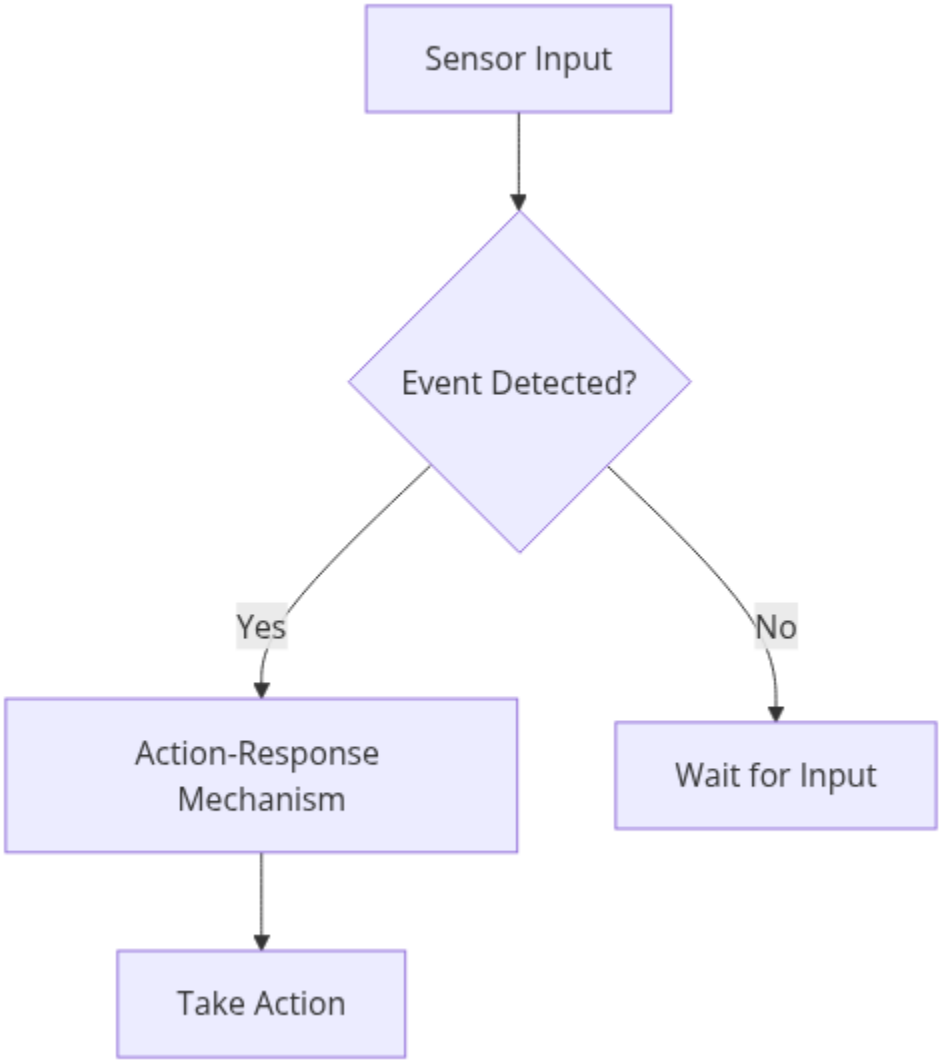
© Ascendient, LLC

3.5. Exploring Reactive Architectures

- Reactive architectures only respond to immediate inputs.
  - They do not store past data or plan for the future.
- Reactive architecture involves the following key components:
  - **Sensor modules**:
    - Detect external inputs, (e.g., obstacles, new data in a database)
  - **Action-response mechanisms**:
    - Decide what action to take based on inputs.
  - **Event-driven programming**:
    - The system only reacts when an event occurs (e.g., a customer places an order).

© Ascendient, LLC

3.6. Reactive Architecture Flow



© Ascendient, LLC

3.7. Example: Basic Reactive Agent in Python

This simple Python function demonstrates how a reactive agent reacts to different inputs like obstacles or clear paths.

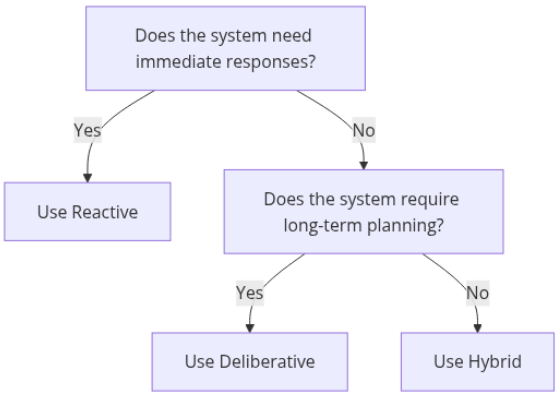
Simplistic Reactive Agent

```python
def reactive_agent(sensor_input):
    if sensor_input == "obstacle":
        return "Stop"
    elif sensor_input == "clear_path":
        return "Move forward"
    else:
        return "Stay still"

print(reactive_agent("obstacle"))  # Output: Stop
```

3.8. Exploring Deliberative Architectures

- Deliberative architectures use internal models and knowledge to plan future actions.
- They are best for tasks requiring long-term strategy and foresight.



© Ascendient, LLC

3.9. Example: Path Planning Agent in Python

The agent plans a path considering obstacles. It takes detours - when necessary - and moves forward when possible.

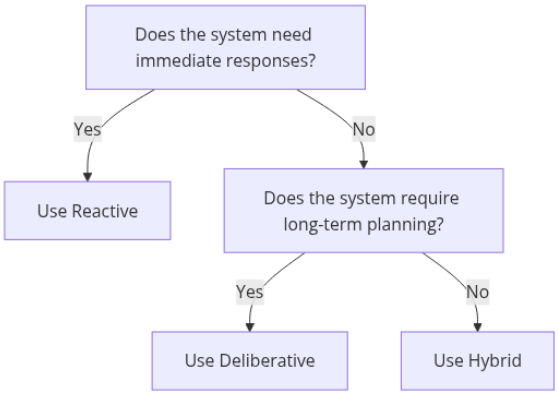
Simplistic Deliberative Agent

```python
def deliberative_agent(goal, obstacles):
    path = []
    for step in range(goal):
        if step in obstacles:
            path.append("Detour")
        else:
            path.append("Move forward")
    return path

print(deliberative_agent(5, [2]))
# Output: [
#   'Move forward',
#   'Move forward',
#   'Detour',
#   'Move forward',
#   'Move forward'
# ]
```

3.10. Hybrid Approach

- Hybrid architectures combine the fast, reactive nature with the strategic planning of deliberative systems.
- This approach allows AI to quickly adapt to changing circumstances while also considering long-term goals.



© Ascendient, LLC

3.11. Hybrid Approach: Combining Speed with Strategy

- Hybrid architectures integrate the best of reactive and deliberative approaches, enabling AI to make fast decisions while also considering long-term goals.
- **Why Use a Hybrid Architecture?**
  - Reactive systems respond instantly but lack long-term planning.
  - Deliberative systems plan ahead but may be slow to react in dynamic environments.
  - A hybrid system balances both, allowing AI to adapt in real-time while still following a structured goal.

3.12. How Hybrid Architectures Work

Hybrid systems typically use a layered or hierarchical structure:

- **Low-Level Reactive Layer**
  - Handles immediate responses (e.g., avoiding an obstacle).
- **High-Level Deliberative Layer**
  - Plans actions based on goals (e.g., optimizing the best route in advance).
- **Coordination Mechanism**
  - Ensures smooth transitions between reactive responses and planned actions.

3.13. Choosing the Right Architecture

When deciding which architecture to use, you need to think about:

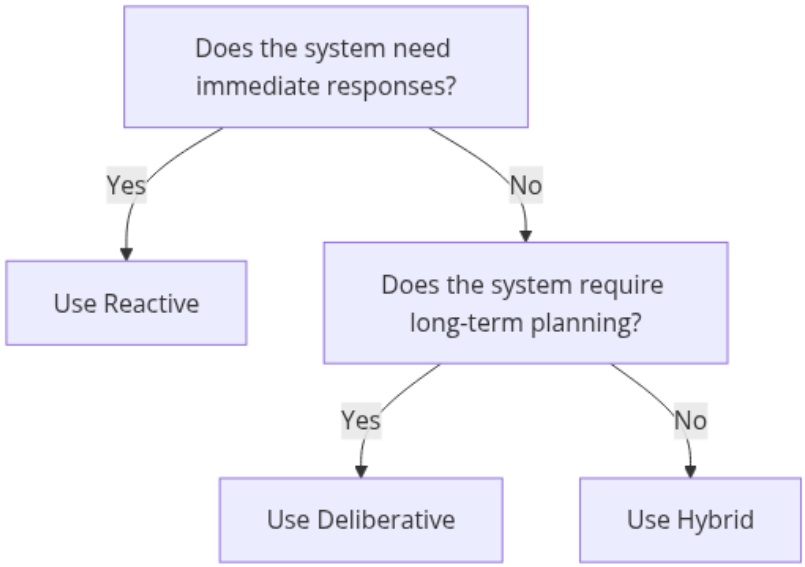
- **Problem domain**
  - What kind of tasks will the AI be performing?
- **Resource constraints**
  - Do you have limited processing power or memory?

© Ascendient, LLC

- **Scalability needs**
  - Will the system need to handle more complex tasks in the future?

3.14. Decision-Making Flowchart

This flowchart helps guide the choice of architecture based on whether the system needs quick responses or long-term planning.



3.15. Agentic Frameworks

Frameworks provide tools to build and manage AI systems with agentic capabilities.

- **AutoGen**:
  - Microsoft-backed Open Source project (CC BY 4.0)
  - Primarily for conversational single and multi-agent applications.
- **LangGraph**:
  - LangChain-backed Open Source Project (MIT)

© Ascendient, LLC

    - Graph-based reasoning and decision-making.
- **CrewAI**:
    - CrewAI-backed Open Source Project (MIT)
    - Allows both conversational and graph-based applications.

3.16. AutoGen

"**AutoGen** is a framework for building AI agents and applications."

- Primarily developed in Python, but has support for .NET.
- Less integrated with commercial tools than LangGraph and CrewAI.
- Strong support fo conversational agents.
- Documentation is somewhat sparse.

Libraries

- **Magentic-One CLI**: A console-based multi-agent assistant for web and file-based tasks.
- **Studio**: An app for prototyping and managing agents without writing code.
- **AgentChat**: A programming framework for building conversational single and multi-agent applications.
- **Core**: An event-driven programming framework for building scalable multi-agent AI systems.
- **Extensions**: Implementations of Core and AgentChat components that interface with external services/libraries.

© Ascendient, LLC

3.17. AutoGen: Example

AutoGen's Quick Start Example

```python
# Define a model client.
model_client = OpenAIChatCompletionClient(
    model="gpt-4.1",
    # api_key="YOUR_API_KEY",
)

# Define a simple function tool that the agent can use.
async def get_weather(city: str) -> str:
    return f"The weather in {city} is 73 degrees and Sunny."


# Define an AssistantAgent with the model, tool, system message, and reflection
enabled.
# The system message instructs the agent via natural language.
agent = AssistantAgent(
    name="weather_agent",
    model_client=model_client,
    tools=[get_weather],
    system_message="You are a helpful assistant.",
    reflect_on_tool_use=True,
    model_client_stream=True,  # Enable streaming tokens from the model client.
)

# Run the agent and stream the messages to the console.
await Console(agent.run_stream(task="What is the weather in New York?"))
```

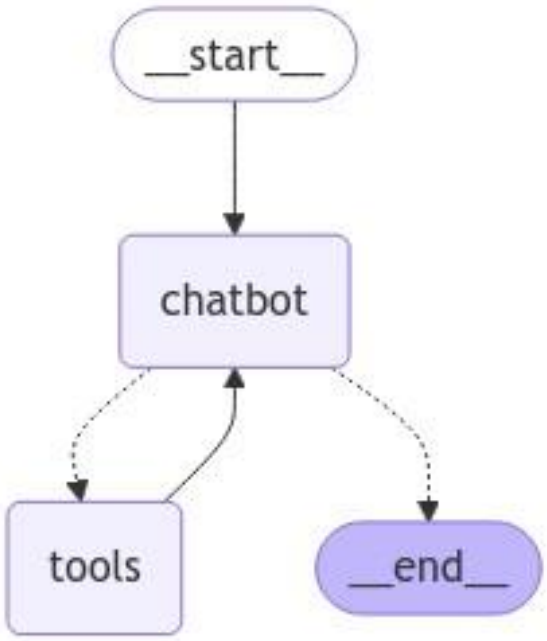

© Ascendient, LLC

3.18. LangGraph

"**LangGraph** is a low-level orchestration framework for building controllable agents."

- Supports Python and JavaScript.
- Heavily integrated with LangChain's commercial tools.
  - LangSmith, LangGraph Cloud
- Designed around graph-based reasoning and decision-making.
  - Conversational systems are more complex to build.
- Built for LangChain
  - Great documentation and tutorials.
  - Lots of community support.

> ⓘ Note
>
> LangGraph's graph-based approach simplifies managing complex agents but can complicate simple agents and quick prototypes.



3.19. LangGraph: Example

LangGraph's Quick Start Example (Greatly Abbreviated)

```
class State(TypedDict):
    messages: Annotated[list, add_messages]
```


© Ascendient, LLC

```
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools([tool])

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

def route_tools(state: State):
    """
    Use in the conditional_edge to route to the ToolNode if the last message
    has tool calls. Otherwise, route to the end.
    """
    # Removed for brevity. Code is available in the LangGraph documentation.

graph_builder = StateGraph(State)
graph_builder.add_node("chatbot", chatbot)
graph_builder.add_node("tools", ToolNode([tool]))
graph_builder.add_conditional_edges("chatbot", route_tools, {"tools": "tools", END: E
ND})
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")
graph = graph_builder.compile()

graph.invoke({"messages": [{"role": "user", "content": user_input}]}):
```


© Ascendient, LLC

3.20. CrewAI

"**CrewAI** is a fast and flexible multi-agent automation framework."

- Built from scratch in Python.
  - Not dependent on LangChain, AutoGen, or other commercial tools.
- Deployable using commercial offering from CrewAI.
- Supports both conversational and graph-based applications.
  - Called "Crews" and "Flows"
- Much of the configuration is done in YAML files.
  - Great for non-programmers.

Basic Project Workflow

1. Install CrewAI

```
pip install crewai crewai[tools]
```

2. Create a CrewAI project

```
crewai create crew latest-ai-development
```

3. Modify
   - `agents.yaml`
   - `tasks.yaml`
   - `crew.py`

4. Run the project

```
crewai run
```


© Ascendient, LLC

3.21. Framework Comparison

Feature	AutoGen	LangGraph	CrewAI
Ease of Use	Moderate	Moderate	High
Flexibility	High	Moderate	Low to Moderate
Workflow Approach	Conversational, Configurable	Graph-based, Explicit	Role-based, Structured
State Management	Message-based	Built-in graph-based	Role-based
Human-in-the-Loop	Strong support	Strong support with graph control	Moderate support
Enterprise Readiness	High	High	Moderate
Prototyping Speed	Moderate	Moderate	High
Community & Documentation	Strong	Growing	Growing
Integration with LangChain	Independent	Native	Optional
Best Use Cases	Complex multi-agent collaboration, code generation, research, highly customized applications	Intricate workflows, precise control, applications within LangChain ecosystem, human-intensive processes	Rapid prototyping, role-based business process automation, collaborative tasks, beginner-friendly projects

© Ascendient, LLC

3.22. Conclusion

- Agentic AI architectures include Reactive, Deliberative, and Hybrid approaches.
- Each has its strengths and weaknesses based on the task at hand.
- Frameworks like AutoGen, LangGraph, and OpenAI Gym enable the development of AI systems with these architectures.
- Experimenting with different architectures and frameworks will help develop smarter, more adaptable systems.

3.23. Reflection

- How do reactive architectures differ from deliberative architectures in decision-making?
- What are the advantages of using a hybrid architecture in AI systems?
- How can the choice of architecture impact the scalability of an AI system?
- In what scenarios would a reactive architecture be more beneficial than a deliberative one?
- How do agentic frameworks like AutoGen and LangGraph support different AI architectures?

© Ascendient, LLC

Agent Memory and Context Management

MODULE 4

- ✓ Understand the Role of Memory in Agentic AI
- ✓ Identify Types of Memory - Working Memory, Long-term Memory, Episodic Memory, etc.
- ✓ Analyze Challenges of Memory Management in Agentic AI
- ✓ Evaluate Limitations of Simple Memory Models
- ✓ Explain Memory as a Form of Context

4.1. About Memory and Context

Why is Memory Important?

- Memory allows agents to:
  - Learn from past experiences
  - Retain information
  - Make informed decisions
- It's crucial for:
  - Context awareness
  - Personalization
  - Complex problem-solving

What is Context?

- Context is the surrounding information or circumstances that help interpret and understand a situation.
- In AI, context can include:
  - User history
  - Environmental factors
  - Task-specific knowledge

4.2. The Role of Memory in Agentic AI

- **Context Retention**
  - Retain context in conversations
  - Provide coherent responses
- **Predictive Capabilities**
  - Anticipate future events
  - Make proactive decisions
- **Personalization**
  - Recall user preferences
- **Adaptive Learning**
  - Learn from past experiences
  - Improve over time

© Ascendient, LLC

4.3. Types of Memory in AI (1/2)

Memory Type	Stores	Examples
Working Memory	Short-term information	• **Human:** Remembering a phone number • **Computer:** Random access memory (RAM) • **Agent:** Recent chat history
Long-Term Memory	Persistent knowledge	• **Human:** Recalling childhood memories • **Computer:** Hard drive storage • **Agent:** User preferences
Episodic Memory	Specific events or experiences	• **Human:** Recalling a birthday party • **Computer:** Event logs • **Agent:** Customer interactions

© Ascendient, LLC

4.4. Types of Memory in AI (2/2)

Memory Type	Stores	Examples
🕮 **Semantic Memory**	General knowledge and facts	• **Human:** Knowing historical events • **Computer:** Knowledge bases • **Agent:** Factual information
⚙ **Procedural Memory**	Learned behaviors and patterns	• **Human:** Riding a bike • **Computer:** Algorithms • **Agent:** Task sequences
👥 **Entity Memory**	Information about specific entities	• **Human:** Remembering a friend's birthday • **Computer:** Database records • **Agent:** Customer details

© Ascendient, LLC

4.5. Types of Memory: Working Memory (Short-Term)

Definition

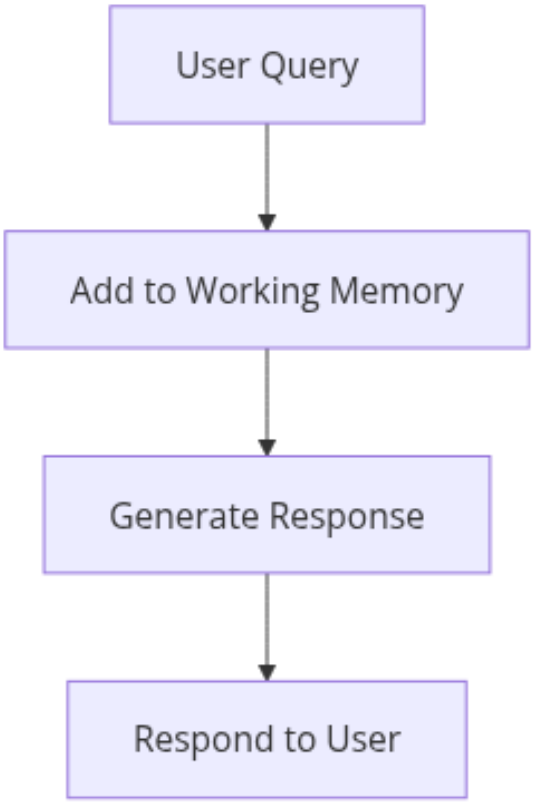
Stores temporary information for active tasks.

Purpose

Maintains immediate context for ongoing interactions.

Limitations

Forgetful after a session ends.



© Ascendient, LLC

4.6. Types of Memory: Long-Term Memory

Definition

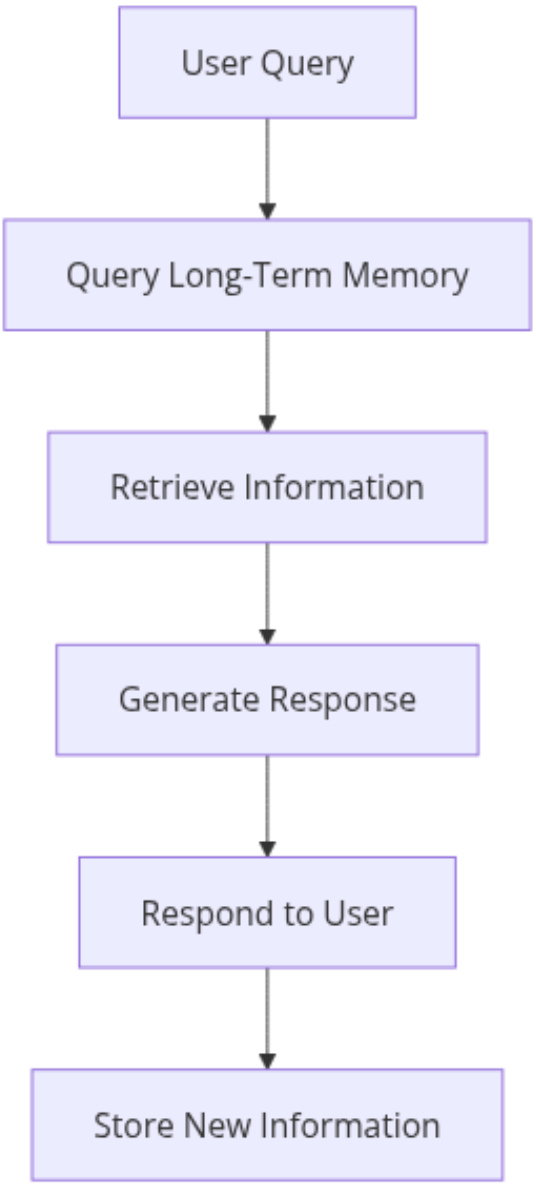
Retains knowledge over multiple interactions or sessions.

Purpose

Improves personalization and learning.

Limitations

Requires efficient retrieval mechanisms.



© Ascendient, LLC

4.7. Types of Memory: Semantic Memory

Definition

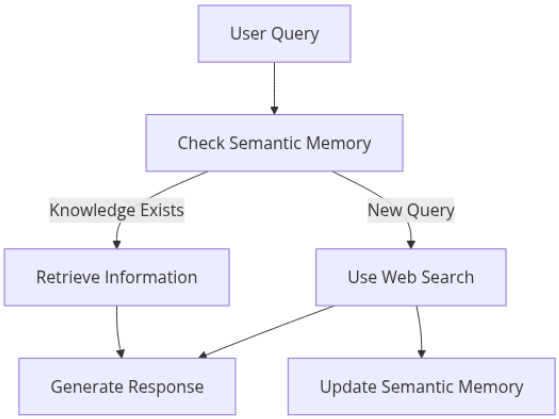
Stores general knowledge and facts.

Purpose

Provides a knowledge base for the AI to draw upon.

Limitations

Requires constant updating and curation.



© Ascendient, LLC

4.8. Types of Memory: Procedural Memory

Definition

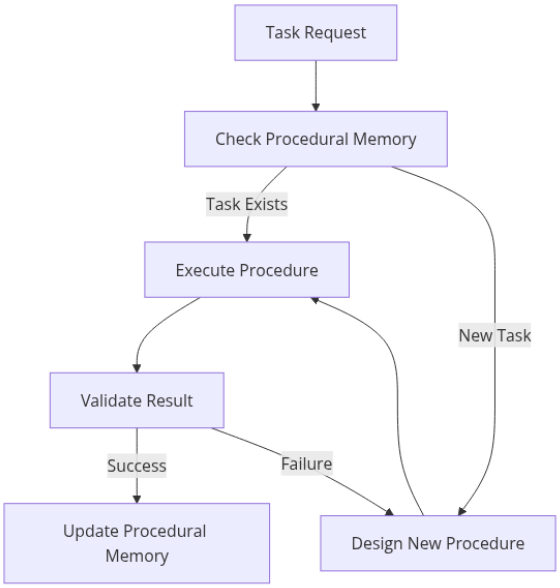
Stores learned behaviors and patterns.

Purpose

Guides the AI in executing tasks and sequences.

Limitations

Requires continuous refinement and adaptation.



© Ascendient, LLC

4.9. Example: Sales Agents Entity Memory

1. Create a profile for each customer

```
customer:
    id: 12345
    likes: []
    dislikes: []
    relationships: []
```

2. Agent reviews transcripts of meetings with customer.

```
Sales: Did you have a nice weekend?
Customer: Yes! I went canoeing with my niece.
```

3. Agent updates customer profile with new information.

```
customer:
    id: 24601
    name: John
    likes: [canoeing]
    dislikes: []
    relationships: [niece]
```

Agents don't just remember things! Memory must be coded into their systems.

© Ascendient, LLC

4.10. Including Memory in Prompts

- Memories can be included in prompts to guide agent responses.
- Here, the agent is prompted with information about John's interests.
- The agent uses this information to start a conversation with John.

Sales Agent Prompt

```
Prompt:
    <system prompt>

    Here is what you know about John:
        - John likes canoeing
        - John has a niece.

    Begin the conversation.

Sales Agent:
    Hi John! Did you spend the weekend on the water?
```

4.11. Memory in Autogen: The Memory Interface

AutoGen Memory Interface

```python
class Memory(ABC, ComponentBase[BaseModel]):
    # Update the model context using relevant memory content
    async def update_context(self, model_context: ChatCompletionContext) -> UpdateCon
textResult:

    # Query the memory store and return relevant entries
    async def query(self,
        query: str | MemoryContent,
        cancellation_token: CancellationToken | None = None,
        **kwargs: Any,
    ) -> MemoryQueryResult:

    # Add a new content to memory
    async def add(self,
        content: MemoryContent,
```


© Ascendient, LLC

```
        cancellation_token: CancellationToken | None = None
    ) -> None:

    # Clear all entries from memory
    async def clear(self) -> None:

    # Clean up any resources used by the memory implementation
    async def close(self) -> None:
```

4.12. Memory in Autogen: Sample Memory Implementation

Simplified ListMemory implementation

```
class ListMemory(Memory):
    def __init__():
        self._contents = []

    async def update_context(self, model_context: ChatCompletionContext) -> UpdateCon
textResult:
        memory_strings = [f"{i}. {str(mem.content)}" for i, mem in enumerate(self._co
ntents, 1)]

        if memory_strings:
            memory_context = textwrap.dedent("""
                Relevant memory content (chronological):
                {"\n".join(memory_strings)}
            """)
            await model_context.add_message(SystemMessage(content=memory_context))

        return UpdateContextResult(memories=MemoryQueryResult(results=self._contents))

    async def query(self, query, cancellation_token, **kwargs):
        return MemoryQueryResult(results=self._contents)

    async def add(self, content, cancellation_token):
        self._contents.append(content)

    ...
```


© Ascendient, LLC

4.13. Memory in Autogen: Usage

Using memory in a Sales Agent

```
user_memory = ListMemory() (1)

await user_memory.add(MemoryContent("John likes canoeing")) (2)
await user_memory.add(MemoryContent("John has a niece"))

sales_agent = SalesAgent(
    name="sales_agent",
    model_client=OpenAiChatCompletionClient(),
    memory=[user_memory] (3)
)
```

1	Create a new instance of ListMemory.
2	Add relevant memory content to the memory store.
3	Initialize the SalesAgent with the memory interface.

4.14. Challenges of Memory in Agentic AI

- **Data Quality and Relevance:**
  - Ensuring the accuracy and relevance of stored information is crucial.
  - Garbage in, garbage out principle applies.
  - Consider additional memory consolidation steps to increase information density.
- **Storage Limitations:**
  - Managing the storage and retrieval of vast amounts of data can be challenging.
  - Consider storing only essential information.
  - In-memory databases offer fast access, but require more resources.
- **Context Window Limitations:**
  - LLMs have a finite context window, limiting the amount of information they can process at once.
  - This is becoming less of an issue with advancements in model architecture.

© Ascendient, LLC

    - Consider selecting only relevant memories.

4.15. Best Practices for Agentic Memory

- Provide Agents with various memory types for different purposes.
- Allow the Agents to forget irrelevant or outdated information.
- Provide opportunities for the Agents to review and consolidate memories.
- Store memories of successes and failures and rationale behind decisions.
- Use a tiered memory system to retrieve and prioritize information.

> ⓘ Note
>
> Memory quickly becomes a RAG system!

4.16. Conclusion

- Memory is a crucial component of agentic AI.
- Different types of memory serve different purposes.
- Memory management is essential for context retention and personalization.
- Memory models need to be adaptive and scalable.
- Memory is a form of context that guides agent behavior.

4.17. Reflection

- How does memory enhance the capabilities of agentic AI?
- What challenges arise in managing memory for AI agents?
- How can different types of memory improve AI personalization?
- Why is context important in AI memory management?

© Ascendient, LLC

- How can memory limitations impact AI performance?

© Ascendient, LLC

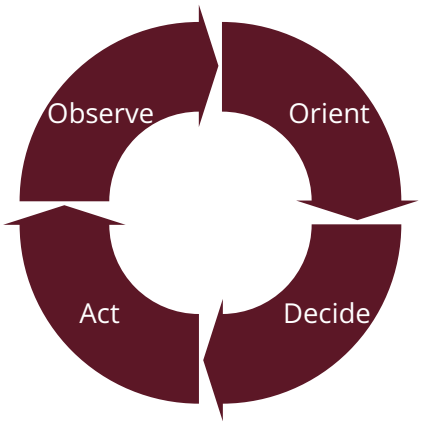
Tool Use and Function Calling in Agentic AI

MODULE 5

- ✓ Investigate the OODA loop in the context of Agentic AI.
- ✓ Distinguish between observation and action tools.
- ✓ Employ methods of tool use (direct, indirect).
- ✓ Integrate Agentic AI with programming using function calls.
- ✓ Verify inputs and outputs of tools.

5.1. The OODA Loop

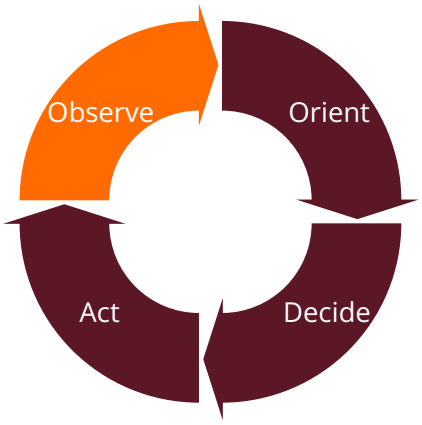
- A Fundamental Decision-Making Cycle:
  - Originates from military strategy (John Boyd).
  - Applicable to various decision-making contexts.
- Stages of the OODA Loop:
  - **O**bserve: Gather information from the environment.
  - **O**rient: Analyze the information, understand the context.
  - **D**ecide: Choose the best course of action.
  - **A**ct: Execute the chosen action.
- Iterative Nature:
  - The loop repeats continuously.
  - Actions lead to new observations, restarting the cycle.



© Ascendient, LLC

5.2. OODA Loop: Observation

- Agents use various inputs to perceive their surroundings.
  - This is the foundation for informed decision-making.
- **Examples:**
  - Reading data from a sensor.
  - Querying a database.
  - Accessing information through an API.
  - Retrieving data from a web service.

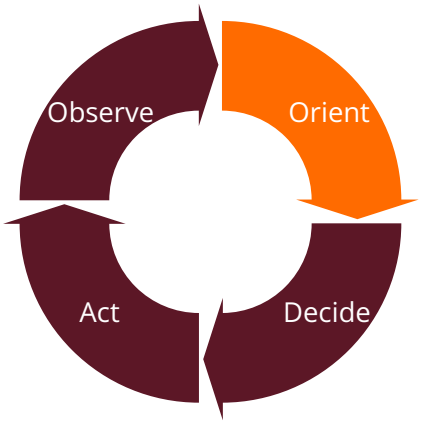


> ⓘ Note
>
> The quality and completeness of observations directly affect the subsequent stages of the OODA loop.

© Ascendient, LLC

5.3. OODA Loop: Orient

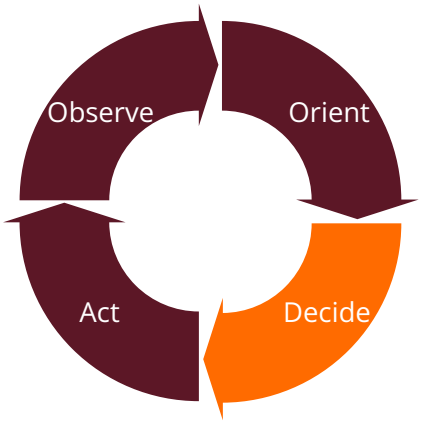
- **Analyzing and Interpreting Data:**
  - Processing raw observations into meaningful information.
  - Understanding the context and significance of the data.
- **Key Processes:**
  - Contextualization: Relating observations to the agent's goals and state.
  - Pattern Recognition: Identifying trends and relationships.
  - Inference: Drawing conclusions from available information.
  - Reasoning: Applying logic to understand the situation.
- **Goal:** Develop a coherent understanding of the current situation.



© Ascendient, LLC

5.4. OODA Loop: Decision

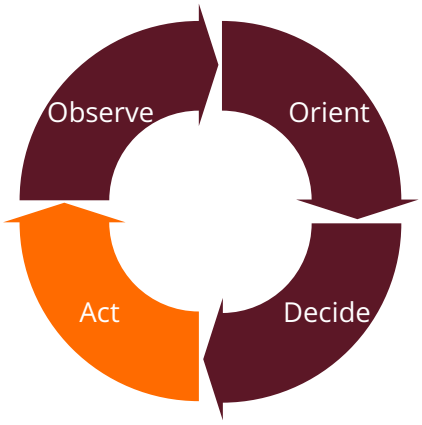
- **Choosing a Course of Action:**
  - Evaluating potential actions based on orientation.
  - Considering potential outcomes and risks.
- **Decision-Making Factors:**
  - Goals: Actions should align with objectives.
  - Resources: Constraints on feasible actions.
  - Consequences: Predicting impacts.
  - Risk: Evaluating potential negatives.
- **Output:** Selecting the most appropriate action.



© Ascendient, LLC

5.5. OODA Loop: Act

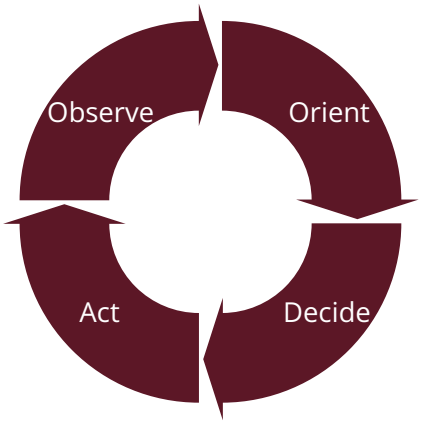
- **Executing the Decision:**
  - Taking the chosen action within the environment.
  - This often involves interacting with external systems or tools.
- **Tool Use:**
  - Action frequently requires the use of specific tools.
  - Tools are the means by which agents effect change.
- **Feedback Loop:**
  - The outcome of the action becomes a new observation.
  - This feedback informs the next iteration of the OODA loop.



© Ascendient, LLC

5.6. OODA Loop and Tool Use

- The OODA loop and tool use are tightly coupled.
- Tools in Each Stage:
  - **Observation:** Gather data (e.g., web scrapers, APIs).
  - **Orientation:** Analyze data (e.g., statistical analysis, visualization).
  - **Decision:** Evaluate options (e.g., simulation, prediction).
  - **Action:** Perform actions (e.g., sending emails, controlling devices).
- Framework and Functionality:
  - The OODA loop provides the decision-making framework.
  - Tools execute those decisions.



© Ascendient, LLC

5.7. Tool Categories: Observation and Action

Observation Tools

Tools that allow agents to gather information from the environment.

- Web search engines.
- Database query tools.
- API calls to retrieve data
- etc.

Action Tools

Tools that enable agents to interact with the environment.

- Sending emails or messages.
- Controlling physical devices
- Making purchases or transactions.
- etc.

© Ascendient, LLC

5.8. Observation Tools

- **Functionality:**
  - Accessing external data sources.
  - Transforming raw data into usable formats.
  - Filtering and summarizing information.
- **Challenges:**
  - Handling incomplete or noisy data.
  - Combining data from multiple sources.
  - Processing large data volumes efficiently.
  - Ensuring access controls and protecting sensitive information.

Examples

- **Web Scrapers:** Extracting data from websites.
  - Targeting specific elements on a page.
  - Handling different website structures.
- **APIs:** Accessing structured data from services.
  - Using API keys for authentication.
  - Handling API rate limits.
- **Database Connectors:** Querying structured data in databases.
  - Using SQL or other query languages.
  - Managing database connections.

© Ascendient, LLC

5.9. Action Tools

- **Functionality:**
  - Performing actions in the real world or digital systems.
  - Modifying the state of the environment.
  - Triggering events or processes.
- **Challenges:**
  - Ensuring actions are performed correctly.
  - Preventing unauthorized actions.
  - Interfacing with various systems and devices.
  - Handling responses and outcomes of actions.

Examples

- **Email Sending:** Automating communication.
  - Composing and sending emails.
  - Handling email responses.
- **Device Control:** Interacting with physical devices (IoT).
  - Sending commands to devices.
  - Receiving feedback from devices.
- **Software Automation:** Performing tasks within applications.
  - Automating workflows.
  - Interacting with user interfaces.

© Ascendient, LLC

5.10. Methods of Tool Use: Direct and Indirect

Direct Tool Use

The agent selects a tool to achieve a specific outcome.

- Requires precise knowledge of the tool's interface.
- Provides precise control over tool execution.
- Simpler to implement for small-scale systems.
- Less flexible when dealing with a large number of tools.

Indirect Tool Use

The agent specifies the desired outcome, another system selects the tool.

- Relies on a tool selection mechanism.
- Scales well to a large number of tools.
- The agent doesn't need to know about all available tools.
- Allows for dynamic tool discovery and integration.

© Ascendient, LLC

5.11. Function Calling: The Basics

- **Defining Tools:**
  - Depends on the library or framework used.
  - Usually specified by a function.
  - Requires documentation on the tool's interface.
  - Include security in your tool specifications.
- **Calling Tools:**
  - Tools are often "bound" to the LLMs.
  - Libraries help manage tool calls and responses.

Required Information

- **Name:** A unique identifier for the tool.
- **Description:** A clear explanation of the tool's purpose and functionality.
  - Helps the agent understand when to use the tool.
- **Parameters:** The inputs the tool requires.
  - Each parameter should have a defined data type (e.g., string, number, boolean).
- **Return Value:** The output the tool produces.

5.12. Example: Forecast Tool

A Tool to Retrieve Forecast Info from the US National Weather Service API

```python
def get_forecast(latitude: str, longitude: str) -> dict:
    """Retrieves the forecast for a given latitude and longitude."""

    # Construct the API endpoint
    endpoint = f"https://api.weather.gov/points/{latitude},{longitude}"

    # Make the HTTP GET request
    response = requests.get(endpoint)
```


© Ascendient, LLC

```
    # Check if the request was successful
    if response.status_code == 200:
        data = response.json()
        forecast_url = data['properties']['forecast']

        # Make another request to get the forecast
        forecast_response = requests.get(forecast_url)

        if forecast_response.status_code == 200:
            forecast_data = forecast_response.json()
            return forecast_data
        else:
            return {"error": "Failed to retrieve forecast data."}
    else:
        return {"error": "Failed to retrieve point data."}
```

5.13. Example: Agent Using the Forecast Tool

An Agent Using the Forecast Tool in AutoGen

```
# Create a function tool.
forecast_tool = FunctionTool(get_forecast, description="Get the stock price.")

# Create an agent that can use the tool.
model_client = OpenAIChatCompletionClient(model="gpt-4.1")
agent = AssistantAgent(
    name="forecaster",
    model_client=model_client,
    tools=[forecast_tool], # <-- You might want a Lat/Long lookup tool too!
    reflect_on_tool_use=True # <-- This lets the agent react to tool's output
)

# Let the agent fetch the content of a URL and summarize it.
result = await agent.run(task="What's the forecast for Charlevoix?")
print(result.messages[-1].content)
```

5.14. Forcing Tool Use

- In some situations, it's critical to guarantee that an agent uses a specific tool.
  - This is important for tasks requiring external data or actions.

© Ascendient, LLC

- Techniques:
  - **Prompt Engineering**
    - Requires modifying the agent's prompt to encourage or require tool use.
    - Not always effective if the agent doesn't understand the prompt.
  - **Explicit Parameters**
    - Some frameworks allow forcing the agent to use any tool or a specific tool.

LangChain/LangGraph

```
llm.bind_tools(tools, tool_choice="any")
```

5.15. Input Validation

- Standard Input Validation:
  - **Type Checking:** Ensuring parameters are of the expected data type.
  - **Range Checking:** Verifying numerical values fall within acceptable limits.
  - **Format Checking:** Enforcing specific formats (e.g., email address, date).
  - **Required Parameter Checking:** Ensuring all mandatory parameters are provided.
  - **Allowed Value Checking:** Ensuring a parameter is within a set of allowed values.
- Agent Input Validation:
  - Will this tool help accomplish the task?
  - Does the tool require specific parameters?
  - Were those parameters provided from the user's input or another tool?

5.16. Output Validation

- Standard Output Validation:
  - **Type Checking:** Ensuring the output matches the expected data type.

© Ascendient, LLC

    - **Format Checking:** Verifying the output structure and format.
    - **Value Checking:** Ensuring the output values are within acceptable ranges.
    - **Error Handling:** Detecting and handling errors in the output.
- Agent Output Validation:
    - Is the output relevant to the task?
    - Does the output contain the expected information?
    - Is the output safe and secure for the user?

5.17. Conclusion

Key Takeaways

- The OODA loop is a continuous decision-making cycle.
- Observation and action tools are crucial for agentic AI.
- Direct and indirect tool use methods have distinct advantages.
- The OODA loop is a continuous decision-making cycle.
- Observation and action tools are crucial for agentic AI.
- Direct and indirect tool use methods have distinct advantages.
- Function calls integrate tools with programming.
- Input and output validation ensures tool effectiveness.

Next Steps

- Apply the OODA loop in a practical scenario.
- Experiment with different observation and action tools.
- Implement direct and indirect tool use in projects.
- Practice input and output validation techniques.
- Explore advanced function calling methods.

© Ascendient, LLC

5.18. Reflection

- How does the OODA loop enhance decision-making in Agentic AI?
- What are the key differences between observation and action tools?
- How can direct and indirect tool use impact an agent's flexibility?
- Why is input and output validation crucial for tool effectiveness?
- How do function calls integrate tools with programming in Agentic AI?

© Ascendient, LLC

Planning and Reasoning in Agentic AI

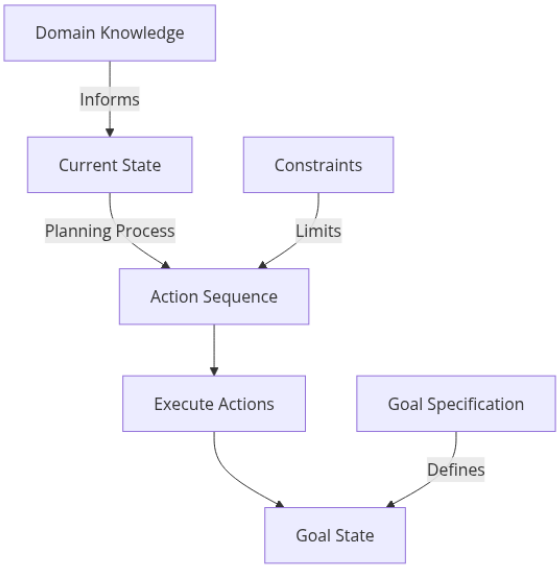
MODULE 6

- ✓ Explain the role of planning in agentic AI systems
- ✓ Compare different planning approaches (rule-based, goal-based, utility-based)
- ✓ Detail hierarchical planning and task decomposition mechanisms
- ✓ Define and categorize reasoning types in agentic AI
- ✓ Demonstrate reasoning in uncertain environments
- ✓ Show how reasoning agents function in multi-agent systems

6.1. The Role of Planning in Agentic AI

Planning in agentic AI is the process of determining a sequence of actions to achieve specific goals. Effective planning enables agents to:

- Move from current state to desired goal state
- Handle complex, multi-step tasks
- Anticipate and prepare for contingencies
- Optimize resource utilization
- Coordinate with other agents



© Ascendient, LLC

6.2. Planning vs. Reactive Approaches

Planning-Based Agents

- Consider future states
- Create action sequences
- Handle complex scenarios
- Higher computational cost

Reactive Agents

- Immediate stimulus-response
- Simple lookup tables
- Fast response time
- Lower computational cost

When to Use Planning

- Complex, multi-step tasks
- Resource optimization needed
- Collaborative coordination
- Long-term goal achievement

When to Use Reactive

- Time-critical responses
- Simple, well-defined tasks
- Stable, predictable environments
- Limited computational resources

© Ascendient, LLC

6.3. The Planning Process

1. **Problem Formulation**
    - Define initial state
    - Specify goal state(s)
    - Identify available actions
    - Determine constraints
2. **Plan Generation**
    - Search for action sequences
    - Evaluate alternatives
    - Optimize for objectives
3. **Execution and Monitoring**
    - Implement plan actions
    - Monitor for deviations
    - Replan when necessary

```python
# Example: Simple planning process
def generate_plan(
        initial_state, goal_state, ava
ilable_actions
    ):
    """
    Generate a sequence of actions to
reach goal state.
    """
    current_state = initial_state
    plan = []

    while current_state != goal_state:
        # Find best next action
        best_action = select_best_acti
on(
            current_state,
            goal_state,
            available_actions
        )

        # Update state based on action
        current_state = \
            apply_action(current_state,
best_action)

        # Add to plan
        plan.append(best_action)

    return plan
```


© Ascendient, LLC

6.4. Planning Horizons

Planning Time Horizons

- **Short-term planning:** Immediate actions (seconds to minutes)
- **Medium-term planning:** Intermediate goals (minutes to hours)
- **Long-term planning:** Strategic objectives (hours to months)

Horizon Selection Factors

- Environment predictability
- Computational resources
- Task complexity
- Real-time requirements
- Uncertainty levels

Balancing Planning Horizons

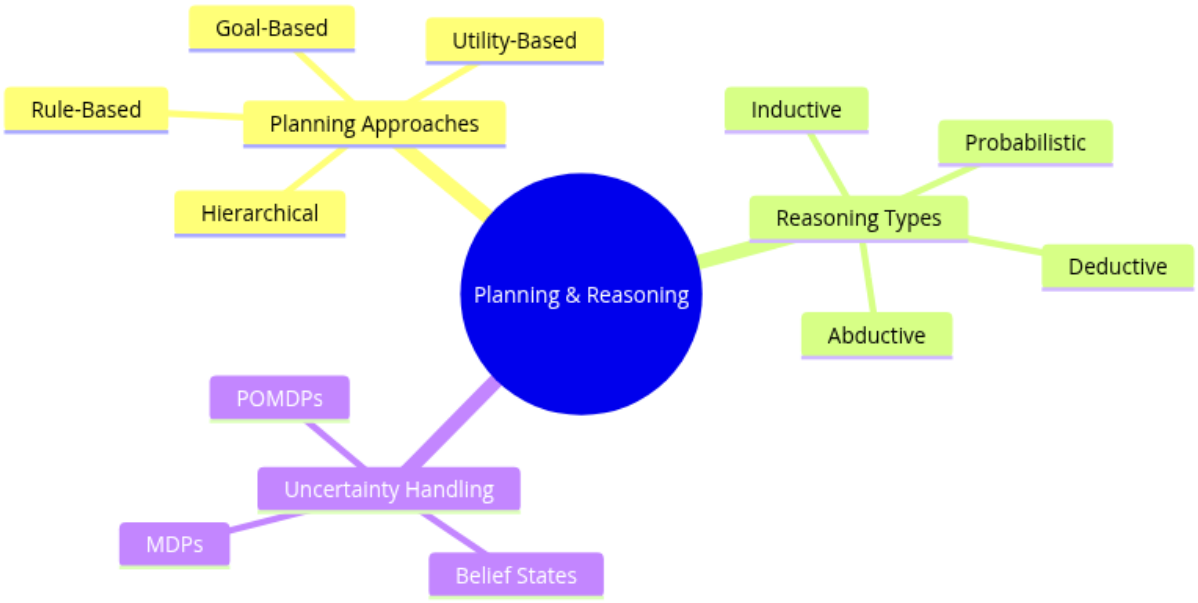
Effective agents often use multi-horizon planning:

1. Long-term plans establish strategic direction
2. Medium-term plans break down strategic goals
3. Short-term plans handle immediate execution

This hierarchical approach balances computational efficiency with strategic vision.

© Ascendient, LLC

6.5. Planning Approaches



© Ascendient, LLC

6.6. Approaches: Rule-Based Planning

Characteristics

- Uses predefined if-then rules
- Simple to implement and understand
- Deterministic and transparent
- Fast execution for known scenarios

Limitations

- Struggles with novel situations
- Rules can become unmanageable
- Limited learning capability
- Requires expert knowledge

Implementation Approaches

- Decision trees
- Rule engines
- Expert systems
- Production systems

6.7. Approaches: Rule-Based Planning — Example

```
# Rule-based planning example for a smart home
def temperature_control_plan(current_temp, target_temp):
    """
    Generate temperature control actions based on rules.
    """
    plan = []

    # Rule 1: If too cold, turn on heating
    if current_temp < target_temp - 2:
        plan.append("activate_heating")

    # Rule 2: If too hot, turn on cooling
    elif current_temp > target_temp + 2:
```


© Ascendient, LLC

```
        plan.append("activate_cooling")

    # Rule 3: If near target, maintain current state
    else:
        plan.append("maintain_current_settings")

    # Rule 4: If significant delta, increase fan speed
    if abs(current_temp - target_temp) > 5:
        plan.append("increase_fan_speed")

    return plan
```

6.8. Approaches: Goal-Based Planning

Characteristics

- Focused on achieving specific goal states
- Plans backwards from goals to current state
- Considers multiple action sequences
- Evaluates based on goal achievement

Common Approaches

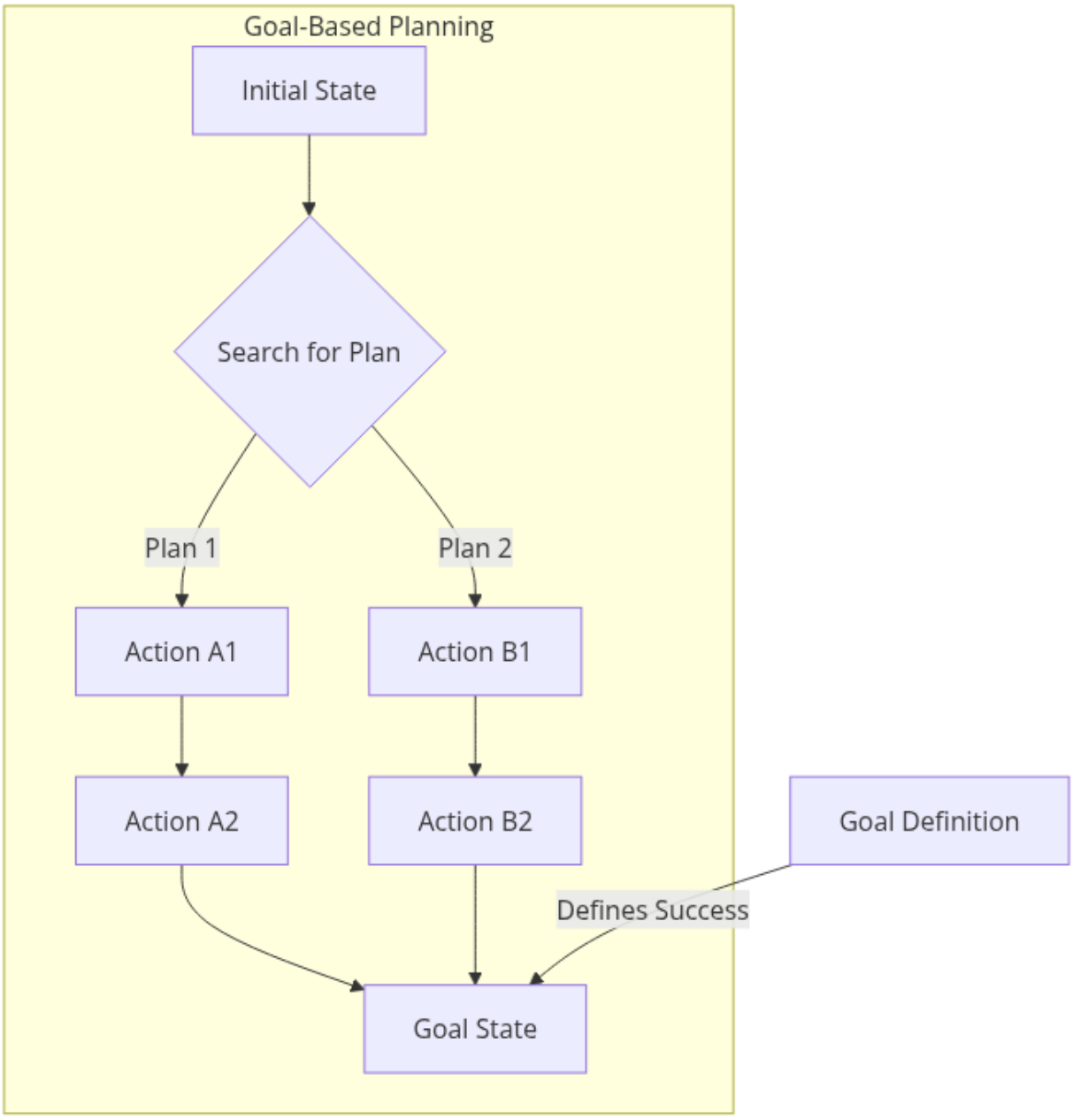
- STRIPS and PDDL planners
- Regression planning
- Partial-order planning
- GraphPlan algorithms

Key Advantages

- Works in unfamiliar situations
- Handles complex goal specifications
- Finds novel action sequences
- Adapts to changing environments

© Ascendient, LLC

6.9. Approaches: Goal-Based Planning — Example



© Ascendient, LLC

6.10. Approaches: Utility-Based Planning

Characteristics

- Optimizes value or utility functions
- Considers costs, benefits, and risks
- Can handle multiple competing objectives
- Plans for preference maximization

Applications

- Resource allocation
- Portfolio optimization
- Autonomous vehicle routing
- Energy management

Approaches

- Markov Decision Processes (MDPs)
- Monte Carlo Tree Search
- Stochastic optimization
- Multi-objective planning

© Ascendient, LLC

6.11. Approaches: Utility-Based Planning — Example

- Example utility calculation for different plan options.
- Plan B has the highest utility score when considering multiple factors.

Plan Option	Cost	Time	Risk	Utility Score
Plan A	High	Low	Medium	75
Plan B	Medium	Medium	Low	82
Plan C	Low	High	High	60
Plan D	Medium	Low	High	65

© Ascendient, LLC

6.12. Classification of Planning Approaches

Deterministic Planning

- Assumes actions have fixed outcomes
- Classical planning algorithms
- STRIPS, PDDL formulations
- Complete and optimal solutions

Probabilistic Planning

- Incorporates outcome probabilities
- Markov Decision Processes
- Partially Observable MDPs
- Plans for contingencies

Single-Agent Planning

- One agent optimizing its actions
- Centralized control
- No need for negotiation
- Full information available

Multi-Agent Planning

- Multiple coordinated agents
- Distributed planning
- Negotiation and conflict resolution
- Information sharing protocols

6.13. Hierarchical Planning: Hierarchical Task Networks (HTN)

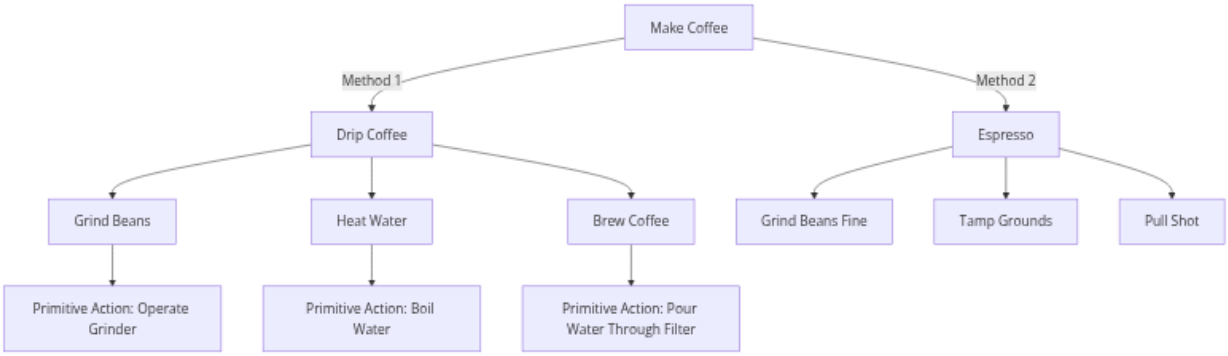
HTNs decompose complex tasks into simpler subtasks:

- **Tasks:** Activities to be performed
- **Methods:** Ways to break down tasks
- **Operators:** Primitive, executable actions
- **Control Knowledge:** Rules for decomposition

HTN planning searches for decompositions that satisfy all constraints.

© Ascendient, LLC

6.14. Hierarchical Task Networks (HTN) - Example



6.15. Hierarchical Planning: Benefits

Computational Efficiency

- Reduces search space by focusing on relevant abstraction levels
- Enables solutions for otherwise intractable problems
- Allows incremental planning at different detail levels
- Improves performance by 1-2 orders of magnitude for complex tasks

Explainability and Modularity

- Creates understandable plans with clear structure
- Facilitates maintenance and updates to specific components
- Enables reuse of subplans across different scenarios
- Supports human oversight and intervention

© Ascendient, LLC

6.16. Hierarchical Planning: Abstraction

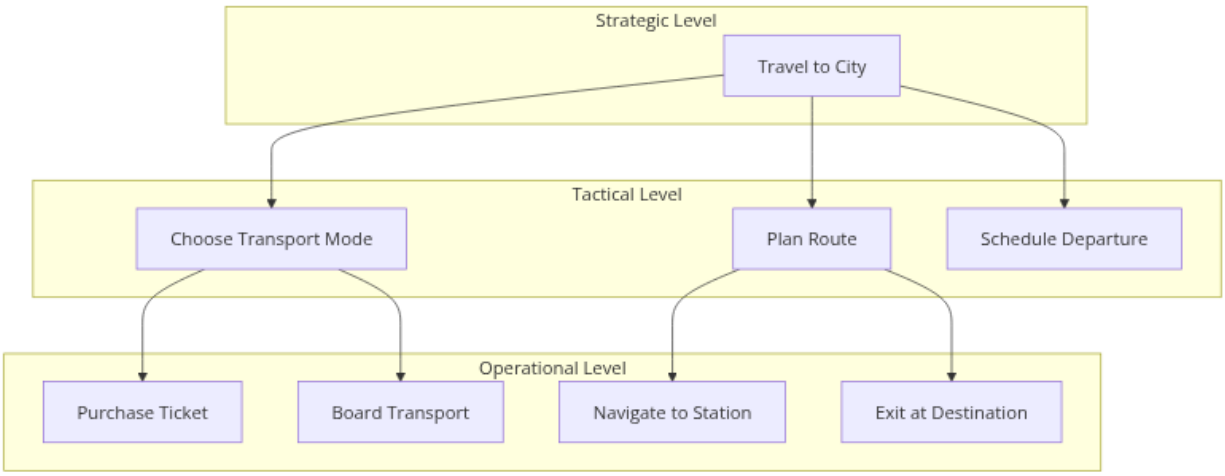
Principles

- **State Abstraction:** Simplify state representations at higher levels
- **Temporal Abstraction:** Group sequences of actions into single abstract actions
- **Component Abstraction:** Treat subsystems as integrated units
- **Value Abstraction:** Simplify evaluation criteria at higher levels

Abstraction Benefits

- Focuses attention on relevant details
- Enables reasoning at appropriate levels
- Facilitates communication between agents
- Matches human conceptual models

6.17. Hierarchical Planning: Abstraction - Example



© Ascendient, LLC

6.18. Task Decomposition Methods

Top-Down Decomposition

- Starts with high-level goal
- Recursively breaks into subtasks
- Natural for hierarchical planning
- Domain knowledge guides decomposition

Bottom-Up Composition

- Starts with primitive actions
- Combines into higher-level tasks
- Useful for learning from experience
- Builds reusable macro-operators

Method	Strengths	Weaknesses
Goal Decomposition	Directly tied to objectives	May miss efficient solutions
Functional Decomposition	Mirrors system functions	Can create artificial divisions
Object-Centered	Natural for physical domains	May complicate cross-object tasks
Temporal Decomposition	Clear sequencing	May obscure parallel opportunities

© Ascendient, LLC

6.19. Reasoning: Types

Deductive Reasoning

- Applies general rules to specific cases
- Guaranteed valid conclusions
- Logic programming approaches

Inductive Reasoning

- Generalizes from specific to general
- Probabilistic conclusions
- Machine learning approaches

Abductive Reasoning

- Infers best explanation for observations
- Diagnostic problem solving
- Plausible but not certain

Analogical Reasoning

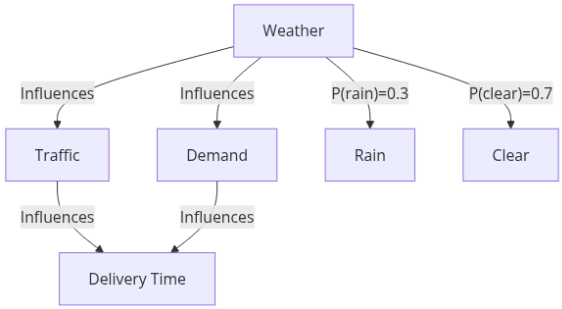
- Transfers knowledge from familiar to new
- Case-based reasoning
- Similarity-based inference

© Ascendient, LLC

6.20. Reasoning: Probabilistic Reasoning

Handling Uncertainty

- World knowledge is often incomplete
- Actions may have probabilistic outcomes
- Observations may be noisy or partial
- Multiple hypotheses may be plausible



Approaches

- Bayesian networks
- Markov models
- Monte Carlo methods
- Probabilistic logic

© Ascendient, LLC

6.21. Reasoning: Belief State Representation

Belief State Approaches

- **Probabilistic:** Distributions over possible states
- **Set-based:** Collections of possible states
- **Fuzzy:** Degree of membership in state sets
- **Dempster-Shafer:** Belief and plausibility measures

Key Properties

- Captures uncertainty explicitly
- Updates with new information
- Propagates through reasoning

```
# Example: Simple belief state for a
robot's location
class LocationBeliefState:
    def __init__(self, grid_size):
        # Initialize uniform
probability across all cells
        self.grid_size = grid_size
        self.probabilities = np.ones(
            (grid_size, grid_size))
        self.probabilities /= self.pro
babilities.sum()

    def update_from_sensor(
        self, sensor_reading, sensor_m
odel):
        """Update belief based on new
sensor reading."""
        for x in range(self.grid_size):
            for y in range(self.grid_s
ize):
                likelihood = sensor_mo
del(
                    sensor_reading, (x,
y))
                self.probabilities[x,
y] *= likelihood

        # Normalize probabilities
        self.probabilities /= self.pro
babilities.sum()
```

6.22. Reasoning: Markov Decision Processes (MDPs)

MDP Framework

© Ascendient, LLC

MDPs model sequential decision-making under uncertainty:

- **States:** Possible situations
- **Actions:** Available choices
- **Transition Model:** P(s'|s,a)
- **Reward Function:** R(s,a,s')
- **Discount Factor:** γ (future value)

Solution Methods

- Value Iteration
- Policy Iteration
- Q-Learning
- Monte Carlo methods

6.23. Reasoning: Markov Decision Processes (MDPs) — Example

```
# Example: Value iteration for solving an MDP
def value_iteration(states, actions, transitions, rewards, gamma=0.9):
    # Initialize value function
    V = {s: 0 for s in states}

    # Convergence threshold
    theta = 0.01

    while True:
        delta = 0
        # Update value of each state
        for s in states:
            v = V[s]
            # Compute new value using Bellman equation
            V[s] = max([
                sum([
                    transitions(s, a, s_prime) *
                    (rewards(s, a, s_prime) + gamma * V[s_prime])
                    for s_prime in states
                ])
                for a in actions(s)
```

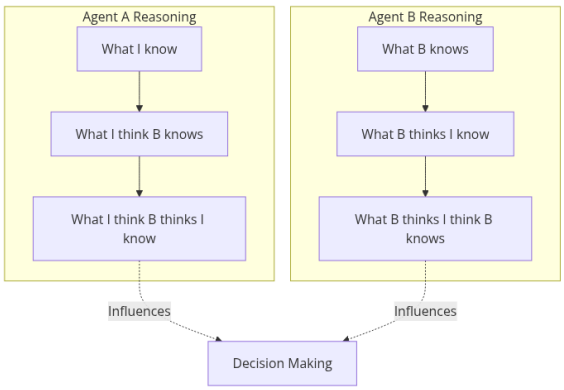

© Ascendient, LLC

```
            ])
            delta = max(delta, abs(v - V[s]))
            if delta < theta:
                break
    return V
```

6.24. Reasoning: Multi-Agent Systems

Key Challenges

- **Strategic Reasoning:** Accounting for other agents' decisions
- **Belief Modeling:** Understanding others' knowledge
- **Coordination:** Aligning actions with other agents
- **Negotiation:** Resolving conflicts and sharing resources
- **Learning:** Adapting to other agents' behaviors



6.25. Reasoning: Implementing Rule-Based Reasoning

Rule-Based System Components

- **Knowledge Base:** Collection of rules
- **Working Memory:** Current facts and assertions
- **Inference Engine:** Mechanism for applying rules
- **Explanation Component:** Reasoning trace

© Ascendient, LLC

Implementation Approaches

- Forward chaining (data-driven)
- Backward chaining (goal-driven)
- Hybrid chaining
- Truth maintenance systems

6.26. Reasoning: Implementing Rule-Based Reasoning — Example

```
# Simple rule-based system
class RuleBasedAgent:
    def __init__(self):
        self.rules = []
        self.facts = set()

    def add_rule(self, condition, action):
        """Add a rule to the knowledge base."""
        self.rules.append((condition, action))

    def add_fact(self, fact):
        """Add a fact to working memory."""
        self.facts.add(fact)

    def forward_chain(self):
        """Apply rules using forward chaining."""
        changes = True
        while changes:
            changes = False
            for condition, action in self.rules:
                if condition(self.facts) and not action in self.facts:
                    self.facts.add(action)
                    changes = True
                    print(f"Applied rule to derive: {action}")

        return self.facts
```


© Ascendient, LLC

6.27. Conclusion

- Planning enables agents to achieve goals through action sequences
- Different planning approaches suit different requirements
- Hierarchical planning manages complexity through decomposition
- Reasoning under uncertainty requires probabilistic approaches
- Multi-agent reasoning considers strategic interactions
- Integration of planning, reasoning, and learning creates robust agents

6.28. Reflection

- How does planning enhance the autonomy of AI agents?
- What are the trade-offs between planning and reactive approaches?
- How can hierarchical planning improve computational efficiency?
- What challenges arise in multi-agent planning scenarios?
- How does probabilistic reasoning handle uncertainty in planning?

© Ascendient, LLC

Building Single-Agent Applications

MODULE 7

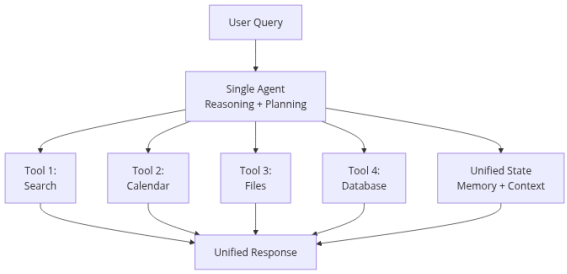
- ✓ Design single-agent architectures for complex, multi-capability applications
- ✓ Select appropriate frameworks and tools for single-agent development
- ✓ Implement best practices for state management and tool integration
- ✓ Explore testing and debugging strategies for agent systems
- ✓ Learn about agent performance and resource usage
- ✓ Integrate agents into production environments

- ✓ Handle errors and implement resilience patterns

7.1. What is a Single-Agent Application?

A single-agent application is a unified AI system that:

- Coordinates multiple capabilities through one reasoning loop
- Maintains centralized state and memory
- Orchestrates tools and external integrations
- Provides coherent, conversational user experience



© Ascendient, LLC

7.2. Single-Agent vs Multi-Agent Design

Aspect	Single-Agent	Multi-Agent
Reasoning	Centralized in one agent	Distributed across multiple agents
State Management	Unified state dictionary	Shared state or message passing
Coordination	Internal orchestration	Inter-agent communication
Debugging	Simpler - single execution trace	Complex - multiple interacting traces
Scalability	Limited by single reasoning loop	Scales with parallel execution
Best For	Related tasks sharing context	Independent specialized tasks

7.3. When to Use Single-Agent Architecture

Single-agent architecture is ideal when:

- Tasks are **related** and share context
  - Example: Research workflow (search papers → organize → schedule → cite)
- User expects **conversational** interaction
  - Example: "Find papers on LLMs" → "Schedule time to read the top 3"
- **Centralized state** simplifies the problem
  - Example: All tools need access to user preferences
- **Coordination overhead** of multiple agents isn't justified
  - Example: Simple workflows with 3-5 tools

© Ascendient, LLC

> ⚠ Warning
>
> For highly specialized or parallel tasks, consider transitioning to multi-agent architecture.

7.4. Core Components of Single-Agent Applications

State Management • Typed state structure • Message history tracking • Context persistence • User preferences	**Tool Integration** • External API calls • File system operations • Database queries • Custom functions
Reasoning Loop • Intent understanding • Tool selection • Sequential execution • Result synthesis	**Memory & Learning** • Conversation history • Adaptive preferences • Error recovery • Session persistence

© Ascendient, LLC

7.5. Designing Single-Agent State

State Design Principles

1. **Type everything** - Use TypedDict or Pydantic
2. **Track conversation** - Messages with add_messages reducer
3. **Context matters** - Store task-relevant information
4. **Preferences persist** - Learn from user behavior
5. **Errors are data** - Track failures for recovery

```
class ResearchAssistantState(TypedDict):
    # Conversation history
    messages: Annotated[list, add_messages]

    # Task context
    research_context: Dict[str, Any]
    current_task: str

    # Tool outputs
    tool_results: List[Dict[str, Any]]

    # Learned preferences
    preferences: Dict[str, str]
```

7.6. Tool Integration Best Practices

Tool Design Patterns:

```
from langchain_core.tools import tool

@tool
def search_papers_tool(query: str, limit: int = 5) -> str:
    """Search for academic papers on a given topic.

    Args:
        query: Search query
        limit: Maximum results to return

    Returns:
        Formatted string with paper titles, authors, years
```


© Ascendient, LLC

```
    """
    # Implementation with error handling
    try:
        results = call_external_api(query, limit)
        return format_results(results)
    except APIError as e:
        logger.error(f"Search failed: {e}")
        return f"Error: {e}. Please try a different query."
```

- **Clear docstrings** - Agent uses them to understand tool purpose
- **Type hints** - Enforce correct argument types
- **Error handling** - Return user-friendly error messages
- **Structured output** - Consistent format for agent to parse

7.7. Building the Agent Workflow

LangGraph Pattern:

```
# Define nodes
def agent_node(state):
    """Agent decides what to do."""
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}

def tool_node(state):
    """Execute tool calls."""
    # ... execute tools, return results

# Build graph
builder = StateGraph(State)
builder.add_node("agent", agent_node)
builder.add_node("tools", tool_node)

# Add conditional routing
builder.add_conditional_edges(
    "agent",
    lambda s: "tools" if has_tool_calls(s) else END
)
builder.add_edge("tools", "agent")  # Loop back

agent = builder.compile(checkpointer=MemorySaver())
```


© Ascendient, LLC

7.8. Multi-Turn Conversation Handling

Context Tracking Across Turns:

Turn 1

User: "Find papers on transformers"
Agent: "I found 8 papers. Would you like me to organize them?"

Turn 2

User: "Yes, just the top 3. Schedule 30 minutes to read each."
Agent: Uses stored topic + preferences to organize and schedule.

- **Context is key** - Store task-relevant information in state
- **Agent infers** - "the top 3" refers to previously found papers
- **Preferences apply** - Duration and timing become preferences

7.9. Adaptive Learning from User Feedback

Learning Pattern:

```python
def detect_correction(message: str) -> dict:
    """Detect simple preference corrections."""
    corrections = {}

    text = message.lower()
    if "morning" in text:
        corrections["meeting_time"] = "morning"
    elif "afternoon" in text:
        corrections["meeting_time"] = "afternoon"

    return corrections
```

Example: "No, I prefer mornings" → updates `preferences["meeting_time"]`.

© Ascendient, LLC

7.10. Framework Selection: LangGraph vs AutoGen vs Custom

Framework	Best For	Pros	Cons
LangGraph	Complex workflows, state management	Explicit graph control, great debugging, checkpointing	Steeper learning curve
AutoGen	Multi-agent systems, code execution	Easy multi-agent setup, built-in code exec	Less control over flow
Custom (LangChain primitives)	Simple linear chains	Maximum flexibility, minimal abstraction	More code to write, less structure
CrewAI	Role-based multi-agent	Intuitive role assignments, good for teams	Newer, smaller ecosystem

7.11. Testing Single-Agent Applications

Test Categories:

1. **Unit Tests** - Test individual tools

```python
def test_search_papers_tool():
    result = search_papers_tool.invoke({"query": "LLMs"})
    assert "LLM" in result
```

2. **Integration Tests** - Test agent workflow

3. **Validation Tests** - Test error handling (Lab 2)

4. **Resilience Tests** - Test retry and circuit breakers (Lab 3)

© Ascendient, LLC

7.12. Debugging Strategies

LangGraph Debugging Tools:

1. **State Inspection** - Check state at any node

```
result = agent.invoke(input_state, config)
print("Final state:", result)
print("Research context:", result["research_context"])
print("Tool results:", result["tool_results"])
```

2. **Execution Traces** - See node-by-node execution

```
for chunk in agent.stream(input_state, config):
    print(f"Node: {chunk}")
```

Use state inspection and traces first; add visualization and logging as needed.

7.13. Performance Optimization

Optimization Strategies:

Strategy	Impact	When to Apply
Cache tool results	Often significantly faster for repeated queries	Tools with expensive API calls
Reduce token usage	Commonly reduces token costs	Prompts with large context
Parallel tool calls	Often much faster for independent tools	Multiple tools don't depend on each other
Streaming responses	Better UX, feels significantly faster	Long-running agent workflows
Smaller models	Often much cheaper and faster	Non-critical tool selection

© Ascendient, LLC

Example: Parallel Tool Calls

```
# Sequential (slow)
papers = search_papers("AI")
events = search_calendar("tomorrow")
# Total time: 2s + 1s = 3s

# Parallel (fast)
results = await asyncio.gather(
    search_papers_async("AI"),
    search_calendar_async("tomorrow")
)
# Total time: max(2s, 1s) = 2s
```

7.14. Production Integration Patterns

Four common deployment patterns for single-agent applications:

- **Web API** - RESTful endpoints for any frontend
- **Gradio UI** - Rapid prototyping with built-in chat interface
- **Slack Bot** - Integration where users already work
- **Batch Processing** - Background jobs and scheduled tasks

7.15. Web API Deployment (FastAPI)

Use Cases:

- Production web applications
- Mobile app backends
- Microservices architecture
- Multi-channel support

Advantages:

- Maximum flexibility

© Ascendient, LLC

- Works with any frontend
- Standard REST interface
- Easy to scale horizontally

See code example next slide

7.16. Web API Deployment (FastAPI) - Code Example

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class ChatRequest(BaseModel):
    user_id: str
    message: str
// Detailed examples for Gradio, Slack, and batch deployments are provided in labs and
// reference materials to keep this deck focused and within slide length guidelines.
# Schedule daily at 2am
schedule.every().day.at("02:00").do(process_daily_tasks)

while True:
    schedule.run_pending()
    time.sleep(60)
```

7.17. Error Handling and Resilience

Error Recovery Patterns (from Lab 3):

1. **Error Classification**

```
if error_type == "transient":
    retry_with_backoff()
elif error_type == "permanent":
    fail_fast()
elif error_type == "partial":
    return_available_results()
```


© Ascendient, LLC

2. **Retry with Exponential Backoff**

```
Attempt 1: immediate → fail
Attempt 2: wait 1s → retry → fail
Attempt 3: wait 2s → retry → fail
Attempt 4: wait 4s → retry → success
```

3. **Circuit Breaker**

```
Closed (normal) → failures exceed threshold → Open (fail fast)
                                                    ↓
                                            wait timeout period
                                                    ↓
                                            Half-Open (test recovery)
```

4. **Graceful Degradation**

```
5 data sources queried
3 succeed, 2 fail
→ Return 3 results (not total failure)
```

7.18. Monitoring and Observability

What to Monitor:

Metric	Why It Matters	How to Track
Latency	User experience	Log timestamp at each node
Tool call distribution	Usage patterns	Count calls per tool
Error rate	System health	Log errors by type and source
State size	Memory usage	Measure state dict size
Token usage	Cost control	Track input/output tokens per call
Circuit breaker state	Service health	Log state transitions

© Ascendient, LLC

Logging Best Practices:

```
logger.info(f"Agent invoked: topic={topic}, user={user_id}")
logger.debug(f"Tool called: {tool_name}, args={tool_args}")
logger.warning(f"Retry attempt {attempt}: {error}")
logger.error(f"Permanent failure: {error}")
```

7.19. Best Practices Summary

Design: ✓ Start with single agent, split to multi-agent only when necessary ✓ Type your state structure completely ✓ Design tools with clear, descriptive docstrings ✓ Track conversation context in state

Implementation: ✓ Use LangGraph for complex workflows ✓ Implement error handling in every tool ✓ Add logging at decision points ✓ Write tests for tools (unit) and workflows (integration)

Production: ✓ Monitor latency, errors, and token usage ✓ Implement retry logic for transient failures ✓ Use circuit breakers for external dependencies ✓ Cache expensive operations

Optimization: ✓ Profile before optimizing ✓ Use smaller models where possible ✓ Parallelize independent tool calls ✓ Stream responses for better UX

7.20. Lab Overview

Lab 1: Building a Personal Assistant Agent

- Design multi-tool single-agent architecture
- Implement state management with persistence
- Build multi-turn conversational interface
- Integrate multiple tools (calendar, search, files, bibliography)
- Apply adaptive learning from user corrections

© Ascendient, LLC

Lab 2: Creating a Data Analysis Agent

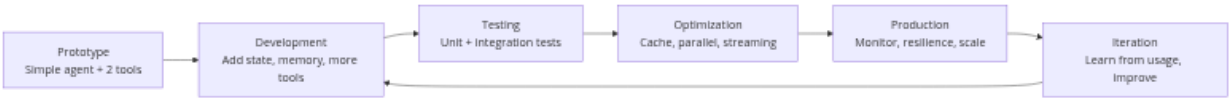
- Generate code safely with AST validation
- Implement sandboxed execution environment
- Apply reflexion for result validation
- Build iterative refinement loops
- Handle data quality issues programmatically

Lab 3: Implementing Error Recovery and Resilience

- Classify error types and recovery strategies
- Implement retry with exponential backoff
- Build circuit breaker patterns
- Handle partial failures gracefully
- Add production-grade logging and monitoring

7.21. From Single-Agent to Production

The Journey:



Key Milestones:

1. **Prototype** (1 week): Core agent loop + 2-3 tools
2. **Development** (2-3 weeks): Full tool suite + state + memory
3. **Testing** (1 week): Comprehensive test coverage
4. **Optimization** (1 week): Performance tuning
5. **Production** (Ongoing): Deploy + monitor + improve

> ⓘ Note

© Ascendient, LLC

> This timeline is for a team of 2-3 developers. Adjust based on complexity and team size.

7.22. Key Takeaways

Single-agent applications are powerful when:

✓ Tasks share context and state ✓ User expects conversational interaction ✓ Coordination overhead isn't justified ✓ You need clear, debuggable execution traces

Critical success factors:

✓ Well-designed state structure ✓ Clear, tested tools with error handling ✓ Proper framework selection (LangGraph recommended) ✓ Comprehensive testing strategy ✓ Production-grade resilience patterns ✓ Monitoring and observability from day one

Remember:

> "Start simple, add complexity only when needed. A single agent that works beats a multi-agent system that doesn't."

7.23. Next Steps

In the labs, you will:

1. Build a **personal assistant** with multi-tool integration (Lab 1)
2. Create a **data analysis agent** with code generation (Lab 2)
3. Implement **error recovery** patterns for production (Lab 3)

After this module:

© Ascendient, LLC

→ **Capstone Project**: Integrate all patterns into Bean Machine support agent

Resources:

- LangGraph Documentation: https://langchain-ai.github.io/langgraph/
- LangSmith (Monitoring): https://smith.langchain.com/
- LangChain Community: https://discord.gg/langchain

Questions?

© Ascendient, LLC