Building Agentic AI Systems with the BeeAI Framework

Estimated time needed: 120 minutes

Welcome to this comprehensive hands-on lab where you'll master the BeeAI Framework—a cutting-edge, open-source platform for building production-ready AI agents and multi-agent systems. Developed under the Linux Foundation AI & Data program and backed by IBM Research, BeeAI Framework represents the next generation of agent development tools, offering complete feature parity between Python and TypeScript with enterprise-grade stability.

Why BeeAI Framework?

In today's rapidly evolving AI landscape, building sophisticated agents that can reason, act, and collaborate requires more than just calling an LLM API. BeeAI Framework addresses the critical challenges of modern AI agent development by providing:

Production-ready architecture: Unlike experimental frameworks, BeeAI is built for real-world deployment with built-in caching, memory optimization, resource management, and OpenTelemetry integration for comprehensive observability.

Provider agnostic flexibility: Support for 10+ LLM providers including OpenAI, watsonx.ai, Groq, and Ollama, allowing you to choose the best model for your specific use case without vendor lock-in.

Advanced agent patterns: Implements proven patterns like ReAct (Reasoning and Acting), systematic thinking with ThinkTool, controlled execution through requirements systems, and sophisticated multi-agent coordination.

Performance and scalability: Optimized memory strategies, efficient tool integration, and workflow composition capabilities that scale from simple chatbots to complex enterprise systems.

Developer experience: Rich ecosystem with LangChain integration, Model Context Protocol (MCP) support, comprehensive documentation, and active community support via Discord and YouTube.

What you'll build

This lab takes a progressive approach, starting with fundamental concepts and advancing to sophisticated multi-agent architectures. You'll gain hands-on experience with real-world scenarios including cybersecurity analysis, business planning, and travel coordination systems. You'll do so by building systems using a range of foundational models, including IBM's Granite, OpenAI's GPT, and Meta's Llama. These models are provided by two large language model platforms: watsonx.ai and OpenAI, which demonstrates the flexibility of the BeeAI Framework.

Learning objectives

By the end of this lab, you will be able to learn:

- **Core concepts**: Understand chat models, memory management, and structured outputs
- **Agent development**: Build agents with custom tools and capabilities
- **Advanced patterns**: Implement multi-agent workflows, requirements control, and systematic reasoning
- **Real-world examples**: Create systems from basic calculators to sophisticated multi-agent travel planners
- **Production techniques**: Apply error handling, optimization, and best practices

Prerequisites

- Basic Python knowledge and async/await understanding
- Working knowledge of agentic AI
- Curiosity about building intelligent systems

1: Quick start—Installation and setup

Let's get you up and running with BeeAI framework in minutes! The BeeAI framework is designed to work with multiple AI providers, making it flexible for different enterprise environments and personal projects.

Task 1.1: Install required packages

BeeAI framework requires specific versions of packages to ensure compatibility and optimal performance. Execute these installation commands:

```
# Install BeeAI framework with Wikipedia integration
pip install openai==1.99.9
pip install beeai-framework[wikipedia]==0.1.35
pip install pydantic==2.11.7
pip install pydantic-core==2.33.2
```

Package overview:

- `openai`: Provides access to OpenAI models and compatible APIs
- `beeai-framework[wikipedia]`: Core BeeAI framework with Wikipedia search capabilities
- `pydantic`: Data validation and serialization for structured outputs
- `pydantic-core`: High-performance core for Pydantic operations

Task 1.2: Environment configuration

In order to connect to IBM's watsonx.ai, one of the LLM API providers used in this project, you must first set some environment variables. First, click the purple button below to create the `t1.py` file:

Open **t1.py** in IDE

Then, paste the following into the opened file and save:

```
import os
########################
### Set the watsonx.ai project ID
########################
# The following sets the watsonx.ai project ID for the
# Skills Network labs environment. Do not modify this setting
# unless you are running the code outside of the
# Skills Network labs environment. In that case, you will
# need to provide your own watsonx.ai project ID.
os.environ["WATSONX_PROJECT_ID"] = "skills-network"
########################
### Set up other watsonx.ai credentials
########################
# The following watsonx.ai API credentials are already preset
# for you in the Skills Network labs environment.
# Do not modify these settings unless you are running the code
# outside of the Skills Network labs environment. In that case,
# you will need to provide your own credentials.
# os.environ["WATSONX_API_KEY"] = "your-watsonx-api-key"
# If you are running this in your own environmment you would also need either:
# os.environ["WATSONX_URL"] =
# or:
# os.environ["WATSONX_REGION"] =
########################
### OpenAI specific configuration
########################
# OpenAI API credentials are already preset
# for you in the Skills Network labs environment.
# Do not modify these settings unless you are running the code
# outside of the Skills Network labs environment. In that case,
# you will need to provide your own credentials.
# OPENAI_API_KEY=your-openai-api-key
# OPENAI_API_HEADERS="secret-header=1234"
print("Environment configured successfully!")
```

The file above sets the required environment variables, such as the watsonx.ai project ID, needed to run the code in the Skills Network labs environment. Some variables, such as the watsonx.ai and OpenAI credentials, are already preconfigured for you in this environment. **Do not** modify this file when running the project on the Skills Network labs environment. However, if you plan to download the codes contained in this lab and run them outside the Skills Network labs environment, you will need to update the script with your own credentials.

To actually set the environment variables, run the following code in the terminal:

```
python3.11 t1.py
```

2: Your first AI conversation

The foundation of every AI agent is the ability to have natural conversations. Chat models form the core of BeeAI's conversational AI capabilities, enabling natural language interactions between users and AI systems.

Understanding chat models in BeeAI

BeeAI's **ChatModel** class provides a unified interface for working with different language models from various providers. This abstraction allows you to switch between OpenAI, WatsonX, and other providers without changing your application code.

Task 2.1: Create your first chat function

Let's build a basic chat function that demonstrates the core concepts of message handling and response generation. This function will showcase:

- **Model initialization**: Using `ChatModel.from_name()` with specific model and parameters
- **Message structure**: Combining SystemMessage (instructions) and UserMessage (user input)
- **Response generation**: Using the `create()` method for synchronous text generation
- **Content extraction**: Getting readable text from the response object

First, create the t2.py file:

Open **t2.py** in IDE

Then paste the following code into the file:

```
import asyncio
import logging
from beeai_framework.backend import ChatModel, ChatModelParameters, UserMessage, SystemMessage
# Initialize the chat model
async def basic_chat_example():
    # Create a chat model instance (works with OpenAI, WatsonX, etc.)
    llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))

    # Create a conversation about something everyone finds interesting
    messages = [
        SystemMessage(content="You are a helpful AI assistant and creative writing expert."),
        UserMessage(content="Help me brainstorm a unique business idea for a food delivery service that doesn't exist yet.")
    ]

    # Generate response using create() method
    response = await llm.create(messages=messages)

    print("User: Help me brainstorm a unique business idea for a food delivery service that doesn't exist yet.")
    print(f"Assistant: {response.get_text_content()}")

    return response
```

Task 2.2: Execute the chat example

Add the main execution function to run your chat example. This execution pattern will:

- Configure logging to reduce noise from async operations
- Run the chat example asynchronously
- Handle the async/await pattern required for AI model calls

Add this code to your t2.py file:

```
# Run the basic chat example
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL) # Suppress unwanted warnings
    response = await basic_chat_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run your first AI conversation:

```
python3.11 t2.py
```

Click the button below to see the complete first conversation script

▶ Complete script

3: Prompt templates

Dynamic prompt templates allow you to create reusable prompts with variable data - perfect for scenarios such as data science project evaluations, where you need consistent analysis across different projects. Templates enable you to maintain consistency while personalizing content for different contexts.

Understanding prompt templates

Prompt templates are reusable text patterns that can be filled with dynamic data. They're essential for:

- **Consistency**: Same structure across different requests
- **Efficiency**: Reuse patterns without rewriting prompts
- **Scalability**: Handle multiple scenarios with one template
- **Maintainability**: Update logic in one place

Task 3.1: Create a template class

First, let's build a simple but powerful template system using Python string formatting. This template class will:

- **Accept mustache-style syntax** (`{{variable}}`) for readability
- **Convert to Python format strings** (`{variable}`) for processing
- **Handle variable substitution** safely using string formatting
- **Support complex templates** with multiple variables

Create the t3.py file:

Open **t3.py** in IDE

Then paste the following code:

```
import asyncio
import logging
import string
from beeai_framework.backend import ChatModel, ChatModelParameters, UserMessage
class SimplePromptTemplate:
    """Simple prompt template using Python string formatting."""

    def __init__(self, template: str):
        self.template = template

    def render(self, variables: dict) -> str:
        """Render the template with provided variables."""
        # Replace mustache-style {{variable}} with Python format {variable}
        formatted_template = self.template
        for key, value in variables.items():
            formatted_template = formatted_template.replace(f"{{{{{key}}}}}", f"{{{key}}}")

        # Format the template with the variables
        return formatted_template.format(**variables)
```

Task 3.2: Design a business evaluation template

Create a comprehensive template for data science project evaluations. This template structure will include:

- **Clear role definition**: Set AI as "senior data scientist"
- **Structured input format**: Organized project details with variable placeholders
- **Specific output requirements**: 5-point analysis framework
- **Quality guidelines**: Request specific, actionable recommendations

Add this code to your t3.py file:

```
async def prompt_template_example():
    llm = ChatModel.from_name("watsonx:ibm/granite-3-3-8b-instruct", ChatModelParameters(temperature=0))

    # Create a prompt template for data science project evaluation
    template_content = """
    You are a senior data scientist evaluating a machine learning project proposal.

    Project Details:
    - Project Name: {{project_name}}
    - Business Problem: {{business_problem}}
    - Available Data: {{data_description}}
    - Timeline: {{timeline}}
    - Success Metrics: {{success_metrics}}

    Please provide:
    1. Feasibility assessment (1-10 scale)
    2. Key technical challenges
    3. Recommended approach
    4. Risk mitigation strategies
    5. Expected outcomes

    Be specific and actionable in your recommendations.
    """

    # Create the prompt template
    prompt_template = SimplePromptTemplate(template_content)
```

Task 3.3: Test with multiple scenarios

Execute the template with different project scenarios. This execution will demonstrate:

- **Template reuse**: Same structure for different projects
- **Variable substitution**: Dynamic content insertion
- **Consistent analysis**: Same evaluation criteria for all projects
- **Comparative results**: Easy to compare different project assessments

Continue adding this code to your t3.py file:

```
    # Test with different project scenarios
    project_scenarios = [
        {
            "project_name": "Smart Inventory Optimization",
            "business_problem": "Reduce inventory costs while maintaining 95% product availability",
            "data_description": "2 years of sales data, supplier lead times, seasonal patterns, 500K records",
            "timeline": "3 months development, 1 month testing",
            "success_metrics": "15% cost reduction, maintain 95% availability, <2% forecast error"
        },
        {
            "project_name": "Fraud Detection System",
            "business_problem": "Detect fraudulent transactions in real-time with minimal false positives",
            "data_description": "1M transaction records, user behavior data, device fingerprints",
            "timeline": "6 months development, 2 months validation",
            "success_metrics": "95% fraud detection rate, <1% false positive rate, <100ms response time"
        }
    ]

    for i, scenario in enumerate(project_scenarios, 1):
        print(f"\n=== Project Evaluation {i}: {scenario['project_name']} ===")

        # Render the template with scenario data
        rendered_prompt = prompt_template.render(scenario)
        print("\n  Rendered prompt:")
        print(rendered_prompt)

        # Generate evaluation using create() method
        messages = [UserMessage(content=rendered_prompt)]
        response = await llm.create(messages=messages)

        print("### LLM response: ###\n")
        print(response.get_text_content())

async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL) # Suppress unwanted warnings
    await prompt_template_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the prompt template example:

```
python3.11 t3.py
```

Click the button below to see the complete prompt template script

▶ Complete Script

4: Structured output magic

Instead of parsing unstructured text responses, you can use Pydantic schemas to get perfectly formatted, type-safe data structures. This is essential for building reliable applications that integrate with APIs, databases, or other systems. BeeAI's `create_structure()` method makes this effortless!

Understanding structured outputs

Structured outputs transform unpredictable AI text into reliable, typed data structures. They provide:

- **Type safety**: Guaranteed data types and structure validation
- **API integration**: Direct compatibility with databases and web services
- **Error prevention**: Catch data issues before they cause problems
- **Consistency**: Same output format regardless of AI model variations

Task 4.1: Define a business plan schema

Create a Pydantic model that defines your desired output structure. This schema will define:

- **Required fields**: All business plan components that must be present
- **Field descriptions**: Guide the AI on what content to generate
- **Data types**: Strings for text, List[str] for multiple items
- **Documentation**: Clear purpose for each field using Field() descriptions

Create the t4.py file:

Open **t4.py** in IDE

Then paste the following code:

```
import asyncio
import logging
from pydantic import BaseModel, Field
from typing import List
from beeai_framework.backend import ChatModel, ChatModelParameters, UserMessage, SystemMessage
# Define a structured output for business planning
class BusinessPlan(BaseModel):
    """A comprehensive business plan structure."""
    business_name: str = Field(description="Catchy name for the business")
    elevator_pitch: str = Field(description="30-second description of the business")
    target_market: str = Field(description="Primary target audience")
    unique_value_proposition: str = Field(description="What makes this business special")
    revenue_streams: List[str] = Field(description="Ways the business will make money")
    startup_costs: str = Field(description="Estimated initial investment needed")
    key_success_factors: List[str] = Field(description="Critical elements for success")
```

Task 4.2: Generate structured business plans

Use `create_structure()` to get perfectly formatted business plan data. Note that in this case, in order to highlight the flexibility of the BeeAI framework, we'll use one of OpenAI's GPT models. However, this code works just as well with IBM's Granite model served by watsonx.ai, the model used in previous examples.

Add this code to your t4.py file:

```
async def structured_output_example():
    llm = ChatModel.from_name("openai:gpt-5-nano", ChatModelParameters(temperature=0))

    messages = [
        SystemMessage(content="You are an expert business consultant and entrepreneur."),
        UserMessage(content="Create a business plan for a mobile app that helps people find and book unique local experiences in their c
    ]

    # Generate structured response using create_structure() method
    response = await llm.create_structure(
        schema=BusinessPlan,
        messages=messages
    )

    print("User: Create a business plan for a mobile app that helps people find and book unique local experiences in their city.")
    print("\n🚀 AI-Generated Business Plan:")
    print(f"💡 Business Name: {response.object['business_name']}")
    print(f"🎯 Elevator Pitch: {response.object['elevator_pitch']}")
    print(f"👥 Target Market: {response.object['target_market']}")
    print(f"⭐ Unique Value Proposition: {response.object['unique_value_proposition']}")
    print(f"💰 Revenue Streams: {', '.join(response.object['revenue_streams'])}")
    print(f"💵 Startup Costs: {response.object['startup_costs']}")
    print(f"🔑 Key Success Factors:")
    for factor in response.object['key_success_factors']:
        print(f"  - {factor}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL) # Suppress unwanted warnings
    await structured_output_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the structured output example:

```
python3.11 t4.py
```

Key benefits of this approach include:

- **Guaranteed structure**: Always get the same data fields
- **Type validation**: Pydantic ensures correct data types
- **Easy access**: Use dictionary-style access to structured data
- **Error handling**: Automatic validation catches malformed responses

Click the button below to see the complete structured output script

▶ Complete Script

5: Your first BeeAI agent

Agents in BeeAI are created using the RequirementAgent class, which is BeeAI's most powerful agent framework. RequirementAgent provides **declarative control** and **reliable behavior**. It combines LLMs, tools, middleware, and requirements in a predictable interface.

Understanding RequirementAgent architecture

RequirementAgent is the cornerstone of BeeAI's intelligent agent system. It provides:

- **Unified interface**: Consistent API regardless of complexity
- **Declarative control**: Specify behavior requirements, not implementation details
- **Tool integration**: Seamless connection with external capabilities
- **Memory management**: Persistent context across interactions
- **Middleware support**: Extensible execution pipeline

Task 5.1: Create your first minimal agent

Let's start with the basics—a RequirementAgent with no external tools, demonstrating pure LLM capabilities.

Create the t5.py file:

Open **t5.py** in IDE

Step 1: Set up imports and function structure

First, let's import the new RequirementAgent components and set up our function:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
async def minimal_tracked_agent_example():
    """
    Minimal RequirementAgent
    """
```

Step 2: Initialize the language model

Notice we're using the same model selection pattern, but now we're building an agent rather than calling the model directly. Again, to highlight the flexibility of the BeeAI framework, we'll now use one of Meta's Llama models served by watsonx.ai, which is different from the IBM Granite and OpenAI GPT models we used in previous examples:

```
    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))
```

Step 3: Define system instructions

We'll use consistent system instructions across all agent examples to demonstrate capability differences:

```
    # CONSISTENT SYSTEM PROMPT (used in all examples)
    SYSTEM_INSTRUCTIONS = """You are an expert cybersecurity analyst specializing in threat assessment and risk analysis.
Your methodology:
1. Analyze the threat landscape systematically
2. Research authoritative sources when available
3. Provide comprehensive risk assessment with actionable recommendations
4. Focus on practical, implementable security measures"""
```

Step 4: Create the RequirementAgent

Here's where we create our first agent. Notice the key parameters:

- **llm**: The language model to use
- **tools**: Empty list—no external tools yet
- **memory**: UnconstrainedMemory for flexible context handling
- **instructions**: The system prompt defining the agent's role

```
    # Minimal RequirementAgent
    minimal_agent = RequirementAgent(
        llm=llm,
        tools=[],  # No tools yet
        memory=UnconstrainedMemory(),
        instructions=SYSTEM_INSTRUCTIONS
    )
```

Step 5: Execute the agent

Finally, we run the agent with our standard query and add the main execution function:

```
    # CONSISTENT QUERY (used in all examples)
    ANALYSIS_QUERY = """Analyze the cybersecurity risks of quantum computing for financial institutions.
    What are the main threats, timeline for concern, and recommended preparation strategies?"""

    result = await minimal_agent.run(ANALYSIS_QUERY)
    print(f"\n💬 Pure LLM Analysis:\n{result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await minimal_tracked_agent_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run your first agent:

```
python3.11 t5.py
```

This setup demonstrates:

- **Model initialization**: Creating a ChatModel with specific parameters
- **System instructions**: Clear role and methodology definition
- **Agent construction**: Minimal RequirementAgent with no external tools
- **Memory configuration**: UnconstrainedMemory for flexible context handling
- **Query execution**: Running complex analysis tasks with the agent
- **Result access**: Easy access to the agent's response through `result.answer.text`

Click the button below to see the complete minimal agent script

▶ Complete Script

This minimal RequirementAgent demonstrates:

- **Clean, structured interface** for basic agent setup
- **Baseline for comparing tool enhancements** in later examples
- **Pure LLM knowledge** without external data sources
- **Foundation for agent architecture** that scales to complex multi-tool systems

6: Tool calling with RequirementAgent

Now let's add WikipediaTool, a built-in BeeAI tool, to our agent. Tools extend agent capabilities beyond pure text generation, enabling access to external data sources, APIs, and specialized functions.

Understanding tool integration

Tool calling transforms agents from isolated text generators into connected systems that can:

- **Access external data**: Retrieve real-time information from APIs and databases
- **Perform actions**: Execute functions, calculations, and operations
- **Research context**: Gather authoritative information to enhance responses
- **Track usage**: Monitor and control tool invocations through middleware

Task 6.1: Add Wikipedia research capability

Now we'll enhance our agent with Wikipedia research capabilities.

Create the t6.py file:

Open **t6.py** in IDE

Step 1: Import tool components

Notice the new imports for tool integration - these enable external capabilities beyond pure LLM reasoning:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from beeai_framework.tools import Tool
```

Step 2: Set up enhanced agent function

We'll use the same model and system instructions, but now add research capabilities:

```
async def wikipedia_enhanced_agent_example():
    """
    RequirementAgent with Wikipedia - Research Enhancement and tracking

    Adding WikipediaTool provides access to Wikipedia summaries for contextual research.
    Same query - but now with research capability.
    Moreover, middleware is used to track all tool usage.
    """
    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))

    # SAME SYSTEM PROMPT as Example 1
    SYSTEM_INSTRUCTIONS = """You are an expert cybersecurity analyst specializing in threat assessment and risk analysis.
Your methodology:
1. Analyze the threat landscape systematically
2. Research authoritative sources when available
3. Provide comprehensive risk assessment with actionable recommendations
4. Focus on practical, implementable security measures"""
```

Step 3: Create tool-enhanced agent

Here are the key new features being introduced:

- **tools=[WikipediaTool()]**: Gives the agent Wikipedia research capability
- **middlewares**: GlobalTrajectoryMiddleware tracks all tool usage for observability
- **requirements**: ConditionalRequirement limits Wikipedia calls to prevent excessive usage

```
    # RequirementAgent with Wikipedia research capability
    wikipedia_agent = RequirementAgent(
        llm=llm,
        tools=[WikipediaTool()],  # Added research capability
        memory=UnconstrainedMemory(),
        instructions=SYSTEM_INSTRUCTIONS,
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
        requirements=[ConditionalRequirement(WikipediaTool, max_invocations=2)]
    )
```

Step 4: Execute and compare results

Using the same query as Task 5, we can see how external research enhances the analysis:

```
    # SAME QUERY as Example 1
    ANALYSIS_QUERY = """Analyze the cybersecurity risks of quantum computing for financial institutions.
    What are the main threats, timeline for concern, and recommended preparation strategies?"""

    result = await wikipedia_agent.run(ANALYSIS_QUERY)
    print(f"\n📖 Research-Enhanced Analysis:\n{result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await wikipedia_enhanced_agent_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the Wikipedia-enhanced agent:

```
python3.11 t6.py
```

Click the button below to see the complete Wikipedia-enhanced agent script

▶ Complete script

Wikipedia enhancement benefits:

- **Access to current, authoritative information** through WikipediaTool integration
- **Tracking shows when Wikipedia is consulted** via GlobalTrajectoryMiddleware
- **More comprehensive, fact-based analysis** combining LLM reasoning with external data
- **Agent autonomously decides when to research** based on the query context

7: Add reasoning

Now let's add ThinkTool for systematic reasoning. The ThinkTool enables agents to engage in explicit reasoning processes, making their thought processes visible and more structured.

Understanding ThinkTool

ThinkTool provides agents with explicit reasoning capabilities:

- **Visible thinking**: Agents can show their reasoning process step-by-step
- **Structured analysis**: Break down complex problems into manageable parts
- **Planning**: Think before acting with other tools
- **Transparency**: Users can see how agents arrive at conclusions

Task 7.1: Add systematic reasoning to your agent

Now we'll add ThinkTool to enable explicit reasoning alongside research capabilities.

Create the t7.py file:

Open **t7.py** in IDE

Step 1: Import ThinkTool components

The key new import here is ThinkTool, which enables structured reasoning:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
```

```
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.tools.think import ThinkTool
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from beeai_framework.tools import Tool
```

Step 2: Define the reasoning-enhanced function

We'll combine explicit reasoning with research capabilities:

```
async def reasoning_enhanced_agent_example():
    """
    RequirementAgent with Systematic Reasoning - ThinkTool + WikipediaTool

    Adding ThinkTool enables structured reasoning alongside research.
    Same query, same tracking - now with visible thinking process.
    """
    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))

    # SAME SYSTEM PROMPT as previous examples
    SYSTEM_INSTRUCTIONS = """You are an expert cybersecurity analyst specializing in threat assessment and risk analysis.
Your methodology:
1. Analyze the threat landscape systematically
2. Research authoritative sources when available
3. Provide comprehensive risk assessment with actionable recommendations
4. Focus on practical, implementable security measures"""
```

Step 3: Create multi-tool agent

Notice we now have both tools available, with separate requirements for each:

- **ThinkTool**(): Enables explicit step-by-step reasoning
- **WikipediaTool**(): Provides external research capability
- **Separate ConditionalRequirements**: Control usage limits for each tool independently

```
    # RequirementAgent with reasoning + research capability
    reasoning_agent = RequirementAgent(
        llm=llm,
        tools=[ThinkTool(), WikipediaTool()],  # Thinking + Research
        memory=UnconstrainedMemory(),
        instructions=SYSTEM_INSTRUCTIONS,
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
        requirements=[
            ConditionalRequirement(ThinkTool, max_invocations=2),
            ConditionalRequirement(WikipediaTool, max_invocations=2)
        ]
    )
```

Step 4: Execute multi-tool analysis

With both reasoning and research tools, the agent can think systematically AND gather external information:

```
    # SAME QUERY as previous examples
    ANALYSIS_QUERY = """Analyze the cybersecurity risks of quantum computing for financial institutions.
    What are the main threats, timeline for concern, and recommended preparation strategies?"""

    result = await reasoning_agent.run(ANALYSIS_QUERY)
    print(f"\n🧠 Reasoning + Research Analysis:\n{result.answer.text}")
```

```
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await reasoning_enhanced_agent_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the reasoning-enhanced agent:

```
python3.11 t7.py
```

Click the button below to see the complete reasoning-enhanced agent script

▶ Complete Script

Systematic reasoning benefits:

- **Tracking shows both thinking and research steps** through middleware integration
- **More structured, methodical analysis approach** with ThinkTool guiding the process
- **Agent plans research strategy systematically** before executing external tool calls
- **Better synthesis of reasoning + external knowledge** for comprehensive responses

8: Explore requirements

Now let's explore **Requirements** to control exactly how and when tools are used. Requirements provide declarative control over agent behavior, ensuring predictable and reliable execution patterns.

Understanding the requirements system

Requirements in BeeAI provide precise control over tool execution:

- **Execution order**: Define when and in what sequence tools can be used
- **Invocation limits**: Control minimum and maximum tool usage
- **Dependencies**: Specify which tools must be used before others
- **Step control**: Force specific tools at exact execution steps

Task 8.1: Implement controlled execution

Now we'll explore advanced requirements to control exactly how and when tools execute.

Create the t8.py file:

Open **t8.py** in IDE

Step 1: Import requirements system

Same imports as before, but now we'll use advanced ConditionalRequirement features:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.tools.think import ThinkTool
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from beeai_framework.tools import Tool
```

Step 2: Function setup and model configuration

Now we'll create the function structure and configure the language model:

```
async def controlled_execution_example():
    """
    RequirementAgent with Controlled Execution - Requirements System

    Requirements provide precise control over tool execution order and behavior.
    Same query, same tracking - but now with strict execution rules.
    """
    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))

    # SAME SYSTEM PROMPT as previous examples
    SYSTEM_INSTRUCTIONS = """You are an expert cybersecurity analyst specializing in threat assessment and risk analysis.
Your methodology:
1. Analyze the threat landscape systematically
2. Research authoritative sources when available
3. Provide comprehensive risk assessment with actionable recommendations
4. Focus on practical, implementable security measures"""
```

Step 3: Create agent with advanced requirements

The key new concept here is the advanced ConditionalRequirement parameters that provide precise control over tool execution:

```
    # RequirementAgent with strict execution control
    controlled_agent = RequirementAgent(
        llm=llm,
        tools=[ThinkTool(), WikipediaTool()],
        memory=UnconstrainedMemory(),
        instructions=SYSTEM_INSTRUCTIONS,
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],

        # REQUIREMENTS: Declarative control over execution flow
        requirements=[
            # MUST start with systematic thinking
            ConditionalRequirement(
                ThinkTool,
                force_at_step=1,  # Thinking required first
                min_invocations=1,  # At least once
                max_invocations=3,  # Max number of invocations
                consecutive_allowed=False  # No repeated thinking
            ),
            # Wikipedia research only after thinking
            ConditionalRequirement(
                WikipediaTool,
                only_after=[ThinkTool],  # Only after thinking
                min_invocations=1,  # Must research at least once
                max_invocations=2  # Max number of research calls
            )
        ]
    )
```

Step 4: Execute agent and display results

Now we'll run the agent with our cybersecurity analysis query and display the controlled execution results:

```
    # SAME QUERY as all previous examples
    ANALYSIS_QUERY = """Analyze the cybersecurity risks of quantum computing for financial institutions.
    What are the main threats, timeline for concern, and recommended preparation strategies?"""

    result = await controlled_agent.run(ANALYSIS_QUERY)
    print(f"\n🔧 Controlled Execution Analysis:\n{result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await controlled_execution_example()
if __name__ == "__main__":
```

```
asyncio.run(main())
```

Now run the controlled execution agent:

```
python3.11 t8.py
```

Click the button below to see the complete controlled execution script

▶ Complete script

Controlled execution benefits:

- **Tracking shows enforced Think → Research → Answer pattern** through requirements
- **force_at_step guarantees systematic thinking first** before any tool usage
- **only_after creates strict tool dependencies** ensuring proper execution order
- **min/max_invocations prevent resource waste** while ensuring necessary operations
- **Predictable, reliable execution every time** through declarative requirements

While Requirements provide greater control, they don't guarantee optimal results unless configured properly. In the example above, the Wikipedia tool or the reasoning step can be triggered consecutively, which is inefficient. For instance, if a Wikipedia query fails to return results, the same query might be retried without any reasoning in between—resulting in redundant tool usage. In the next example, we'll explore a more efficient pattern that addresses these issues.

9: Create a ReAct agent

The ReAct (Reasoning and Acting) pattern is a fundamental approach in AI agent design that alternates between thinking and tool usage. Let's implement the classic ReAct trajectory: **Think → Tool → Think → Tool → Think → … → Answer**

Understanding ReAct pattern

ReAct is a powerful agent design pattern that:

- **Alternates reasoning and action**: Think before and after each tool usage
- **Provides transparency**: Shows the agent's reasoning process at each step
- **Improves reliability**: Systematic thinking reduces errors and improves outcomes
- **Enables self-correction**: Agent can adjust strategy based on tool results

Task 9.1: Implement ReAct pattern

Now we'll implement the classic ReAct pattern.

Create the t9.py file:

Open **t9.py** in IDE

Step 1: Import components for ReAct pattern

Same imports as before—we'll use the same tools but with different requirement patterns:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.tools.think import ThinkTool
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from beeai_framework.tools import Tool
```

Step 2: Define ReAct agent function

The ReAct pattern combines reasoning (thinking) with acting (tool usage) in a systematic cycle:

```
async def reasoning_enhanced_agent_example():
    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))

    # SAME SYSTEM PROMPT as previous examples
    SYSTEM_INSTRUCTIONS = """You are an expert cybersecurity analyst specializing in threat assessment and risk analysis.
Your methodology:
1. Analyze the threat landscape systematically
2. Research authoritative sources when available
3. Provide comprehensive risk assessment with actionable recommendations
4. Focus on practical, implementable security measures"""
```

Step 3: Create ReAct pattern agent

The key ReAct features:

- **force_at_step=1**: Must start with thinking (Reasoning)
- **force_after=Tool**: Think after every tool usage (Reflection)
- **consecutive_allowed=False**: Prevents thinking without acting

```
    # RequirementAgent with reasoning + research capability
    reasoning_agent = RequirementAgent(
        llm=llm,
        tools=[ThinkTool(), WikipediaTool()],  # Thinking + Research
        memory=UnconstrainedMemory(),
        instructions=SYSTEM_INSTRUCTIONS,
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
        requirements=[
            ConditionalRequirement(
                ThinkTool,
                force_at_step=1,  # Thinking required first
                force_after=Tool,  # Force reasoning after every tool call
                min_invocations=1,  # At least once
                max_invocations=5,  # Max number of invocations
                consecutive_allowed=False  # No repeated thinking
            ),
            #ConditionalRequirement(WikipediaTool, max_invocations=2)
        ]
    )
```

Step 4: Execute ReAct pattern

Watch for the Think → Act → Think → Act → … pattern in the output:

```
    # SAME QUERY as previous examples
    ANALYSIS_QUERY = """Analyze the cybersecurity risks of quantum computing for financial institutions.
    What are the main threats, timeline for concern, and recommended preparation strategies?"""

    result = await reasoning_agent.run(ANALYSIS_QUERY)
    print(f"\n🧠 Reasoning + Research Analysis:\n{result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await reasoning_enhanced_agent_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the ReAct pattern agent:

```
python3.11 t9.py
```

Click the button below to see the complete ReAct agent script

▶ Complete script

Key ReAct implementation details:

- **`force_at_step=1`**: Ensures reasoning starts first
- **`force_after=Tool`**: Forces thinking after each tool usage
- **`consecutive_allowed=False`**: Prevents repeated thinking steps
- **Flexible tool sequence**: Maintains ReAct structure without restricting which tool, if any, should be called following reasoning steps

10: Human-in-the-loop

Now, let's demonstrate usage of the **AskPermissionRequirement** call, which asks for permission from the human before taking an action. Human-in-the-loop systems are essential for production environments where oversight and approval are required.

Understanding human-in-the-loop systems

Human-in-the-loop (HITL) systems provide crucial oversight and control:

- **Security compliance**: Meet regulatory requirements for human oversight
- **Risk management**: Prevent unintended actions through approval workflows
- **Quality control**: Human verification before executing critical operations
- **Trust building**: Users feel more confident with transparent approval processes

Task 10.1: Implement permission-based agent

Now we'll add human oversight through permission requirements for production security.

Create the t10.py file:

Open **t10.py** in IDE

Step 1: Import permission components

The key new import is AskPermissionRequirement for human-in-the-loop approval:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.agents.experimental.requirements.ask_permission import AskPermissionRequirement
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.tools.think import ThinkTool
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from beeai_framework.tools import Tool
```

Step 2: Function setup and model configuration

Create the function structure and configure the language model for production security:

```
async def production_security_example():
    """
    Production-Ready RequirementAgent with Security Approval

    AskPermissionRequirement adds human-in-the-loop security controls.
    Same query, same tracking - but now with approval workflow.
    """
    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))

    # SAME SYSTEM PROMPT as all previous examples
    SYSTEM_INSTRUCTIONS = """You are an expert cybersecurity analyst specializing in threat assessment and risk analysis.
Your methodology:
1. Analyze the threat landscape systematically
2. Research authoritative sources when available
3. Provide comprehensive risk assessment with actionable recommendations
4. Focus on practical, implementable security measures"""
```

Step 3: Create agent with permission requirements

The key new concept is AskPermissionRequirement, which enforces human approval before tool execution:

```
    # Production-grade RequirementAgent with security approval
    secure_agent = RequirementAgent(
        llm=llm,
        tools=[ThinkTool(), WikipediaTool()],
        memory=UnconstrainedMemory(),
        instructions=SYSTEM_INSTRUCTIONS,
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],

        requirements=[
            # Same systematic thinking requirement
            ConditionalRequirement(
                ThinkTool,
                force_at_step=1,
                min_invocations=1,
                max_invocations=2,
                consecutive_allowed=False
            ),
            # SECURITY: Permission required for external access
            AskPermissionRequirement(
                WikipediaTool,
            ),
            # Same control after permission granted
            ConditionalRequirement(
                WikipediaTool,
                only_after=[ThinkTool],
                min_invocations=0,  # Optional after approval
                max_invocations=1  # Limited even after approval
            )
        ]
    )
```

Step 4: Execute agent with security approval

Now we'll run the agent with human-in-the-loop approval workflow:

```
    # SAME QUERY as all previous examples
    ANALYSIS_QUERY = """Analyze the cybersecurity risks of quantum computing for financial institutions.
    What are the main threats, timeline for concern, and recommended preparation strategies?"""

    result = await secure_agent.run(ANALYSIS_QUERY)
    print(f"\n🛡️ Security-Approved Analysis:\n{result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await production_security_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the permission-based agent:

```
python3.11 t10.py
```

Click the button below to see the complete human-in-the-loop agent script

▶ Complete script

Production security benefits:

- **Tracking shows Think → Approval → Research → Answer pattern** with human oversight
- **AskPermissionRequirement enforces human oversight** for sensitive tool operations
- **Compliance with security and regulatory requirements** through permission controls
- **Complete audit trail through middleware integration** for accountability
- **Production-ready security and approval workflows** for enterprise deployment

You've now explored the full progression of RequirementAgent capabilities, from basic agents to sophisticated, controlled systems. Next, let's learn to create custom tools that extend agent capabilities even further.

11: Creating your first custom tool

Tools extend agent capabilities beyond conversation—they enable actions such as calculations, API calls, database queries, and more. Custom tools allow you to integrate your agents with any system or service, making them truly practical for real-world applications.

Understanding custom tool creation

Custom tools in BeeAI provide unlimited extensibility:

- **Domain-specific operations**: Create tools for your specific business needs
- **External system integration**: Connect to APIs, databases, file systems, and more
- **Input validation**: Use Pydantic schemas for type-safe tool inputs
- **Error handling**: Implement robust error handling and user feedback
- **Agent integration**: Seamlessly integrate with RequirementAgent workflows

Task 11.1: Create a simple calculator tool

Now we'll learn to create custom tools by building a calculator.

Create the t11.py file:

Open **t11.py** in IDE

Step 1: Import tool creation components

Notice the new imports for custom tool creation—these enable building your own tools:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.tools import StringToolOutput, Tool, ToolRunOptions
from beeai_framework.context import RunContext
from beeai_framework.emitter import Emitter
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from pydantic import BaseModel, Field
from typing import Any
```

Step 2: Define tool input schema

Every BeeAI tool needs a Pydantic input schema for type safety and validation:

```
# === REAL TOOL CREATION WITH OFFICIAL BEEAI TOOLS ===
class CalculatorInput(BaseModel):
    """Input model for basic mathematical calculations."""
    expression: str = Field(description="Mathematical expression using +, -, *, / (e.g., '10 + 5', '20 - 8', '4 * 6', '15 / 3')")
```

Step 3: Create Tool class structure

Every BeeAI tool inherits from the base Tool class with type annotations for input, options, and output:

```
class SimpleCalculatorTool(Tool[CalculatorInput, ToolRunOptions, StringToolOutput]):
    """A simple calculator tool for basic arithmetic operations: add, subtract, multiply, divide."""
    name = "SimpleCalculator"
    description = "Performs basic arithmetic calculations: addition (+), subtraction (-), multiplication (*), and division (/)."
    input_schema = CalculatorInput
    def __init__(self, options: dict[str, Any] | None = None) -> None:
        super().__init__(options)
    def _create_emitter(self) -> Emitter:
        return Emitter.root().child(
            namespace=["tool", "calculator", "basic"],
            creator=self,
        )
```

Step 4: Implement safe calculation logic

The core calculation method with security and error handling:

```
    def _safe_calculate(self, expression: str) -> float:
        """Safely evaluate basic arithmetic expressions."""
        # Remove spaces for processing
        expr = expression.replace(' ', '')

        # Only allow numbers, basic operators, parentheses, and decimal points
        allowed_chars = set('0123456789+-*/().')
        if not all(c in allowed_chars for c in expr):
            raise ValueError("Only numbers and basic operators (+, -, *, /, parentheses) are allowed")

        try:
            # Use eval with restricted environment for basic arithmetic only
            result = eval(expr, {"__builtins__": {}}, {})
            return float(result)
        except ZeroDivisionError:
            raise ValueError("Division by zero is not allowed")
        except Exception as e:
            raise ValueError(f"Invalid arithmetic expression: {str(e)}")
```

Step 5: Implement the main tool execution method

The async _run method is called when the agent uses the tool:

```
    async def _run(
        self, input: CalculatorInput, options: ToolRunOptions | None, context: RunContext
    ) -> StringToolOutput:
        """Perform basic arithmetic calculations."""
        try:
            expression = input.expression.strip()
```

```
            # Perform calculation
            result = self._safe_calculate(expression)

            # Format result
            output = f"🧮 Simple Calculator\n"
            output += f"Expression: {expression}\n"
            output += f"Result: {result}\n"

            # Add operation type hint
            if '+' in expression:
                output += "Operation: Addition"
            elif '-' in expression:
                output += "Operation: Subtraction"
            elif '*' in expression:
                output += "Operation: Multiplication"
            elif '/' in expression:
                output += "Operation: Division"
            else:
                output += "Operation: Basic Arithmetic"

            return StringToolOutput(output)

        except ValueError as e:
            return StringToolOutput(f"❌ Calculation Error: {str(e)}")
        except Exception as e:
            return StringToolOutput(f"❌ Unexpected Error: {str(e)}")
```

Step 6: Create agent function with custom tool

Now we'll create an agent that uses our custom calculator tool:

```
async def calculator_agent_example():
    """RequirementAgent with SimpleCalculatorTool - Interactive Math Assistant"""

    llm = ChatModel.from_name("watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8", ChatModelParameters(temperature=0))

    # Create calculator agent with our custom tool
    calculator_agent = RequirementAgent(
        llm=llm,
        tools=[SimpleCalculatorTool()],
        memory=UnconstrainedMemory(),
        instructions="""You are a helpful math assistant. When users ask for calculations,
        use the SimpleCalculator tool to provide accurate results.
        Always show both the expression and the calculated result.""",
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
    )
```

Step 7: Test agent with multiple calculations

Finally, let's test our custom tool with various mathematical queries:

```
    # Interactive examples - simulating human input
    math_queries = [
        "What is 15 + 27?",
        "Calculate 144 divided by 12",
        "I need to know what 8 times 9 equals",
        "What's (10 + 5) * 3 - 7?"
    ]

    for query in math_queries:
        print(f"\n👤 Human: {query}")
        result = await calculator_agent.run(query)
        print(f"🤖 Agent: {result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await calculator_agent_example()
if __name__ == "__main__":
    asyncio.run(main())
```

Now run the custom calculator tool example:

```
python3.11 t11.py
```

Click the button below to see the complete custom tool implementation script

▶ Complete script

This example demonstrates creating a custom tool integrated with RequirementAgent:

- **SimpleCalculatorTool** performs basic arithmetic operations safely
- **Math assistant agent** uses the calculator tool to answer natural language math queries
- **Human input simulation** shows how agents respond to conversational math questions
- **Middleware tracking** provides visibility into tool usage patterns

Key custom tool components:

- **Input schema**: Pydantic model defines the tool's expected inputs with validation
- **Tool class**: Inherits from BeeAI's Tool base class with proper typing
- **Safe execution**: Implements secure calculation with comprehensive error handling
- **Rich output**: Provides formatted results with operation context
- **Agent integration**: Works seamlessly with RequirementAgent and middleware

12: Multi-agent travel planner with language expert

Multi-agent systems represent the pinnacle of AI orchestration! Multi-agent architecture allows you to create specialized AI teams where each agent has specific expertise, working together to solve complex problems that no single agent could handle alone.

Understanding multi-agent systems

Multi-agent systems provide powerful capabilities for complex problem-solving:

- **Specialization**: Each agent focuses on a specific domain of expertise
- **Collaboration**: Agents communicate and coordinate through HandoffTool
- **Scalability**: Add new agents with different capabilities as needed
- **Modularity**: Update or replace individual agents without affecting the system
- **Robustness**: System continues functioning even if individual agents fail

The HandoffTool: Enabling agent coordination

The **HandoffTool** is BeeAI's mechanism for multi-agent communication and coordination:

What is HandoffTool?

- A special tool that allows one agent to delegate tasks to another agent
- Creates a communication bridge between specialized agents
- Enables complex workflows by combining different agent expertise
- Maintains context and conversation flow across agent handoffs

How agent coordination works:

1. **Coordinator agent** receives a complex query requiring multiple specializations
2. **HandoffTool** identifies which expert agents are needed
3. **Task delegation** passes specific sub-queries to appropriate expert agents
4. **Expert response** specialized agents process their domain-specific tasks
5. **Result integration** coordinator synthesizes responses into comprehensive answer

This pattern enables sophisticated "AI teams" where each agent contributes their specialized knowledge to solve complex problems that no single agent could handle alone.

Task 12.1: Import multi-agent components

First, let's set up our imports including the crucial HandoffTool for agent coordination.

Create the t12.py file:

Open **t12.py** in IDE

Step 1: Import required components

Notice the key new import—**HandoffTool**—which enables inter-agent communication:

```
import asyncio
import logging
from beeai_framework.agents.experimental import RequirementAgent
from beeai_framework.agents.experimental.requirements.conditional import ConditionalRequirement
from beeai_framework.agents.experimental.requirements.ask_permission import AskPermissionRequirement
from beeai_framework.memory import UnconstrainedMemory
from beeai_framework.backend import ChatModel, ChatModelParameters
from beeai_framework.tools.search.wikipedia import WikipediaTool
from beeai_framework.tools.weather import OpenMeteoTool
from beeai_framework.tools.think import ThinkTool
from beeai_framework.tools.handoff import HandoffTool
from beeai_framework.middleware.trajectory import GlobalTrajectoryMiddleware
from beeai_framework.tools import Tool
```

Step 2: Initialize multi-agent function

Create the main function that will orchestrate our specialized AI team:

```
async def multi_agent_travel_planner_with_language():
    """
    Advanced Multi-Agent Travel Planning System with Language Expert

    This system demonstrates:
    1. Specialized agent roles and coordination
    2. Tool-based inter-agent communication
    3. Requirements-based execution control
    4. Language and cultural expertise integration
    5. Comprehensive travel planning workflow
    """

    # Initialize the language model
    llm = ChatModel.from_name(
        "watsonx:meta-llama/llama-4-maverick-17b-128e-instruct-fp8",
        ChatModelParameters(temperature=0)
    )
```

Task 12.2: Create specialized expert agents

In multi-agent systems, each agent has specialized expertise. We'll create three expert agents that will later communicate via HandoffTool:

Step 1: Create destination research expert

This agent specializes in destination information and travel logistics:

```
    # === AGENT 1: DESTINATION RESEARCH EXPERT ===
    destination_expert = RequirementAgent(
        llm=llm,
        tools=[WikipediaTool(), ThinkTool()],
        memory=UnconstrainedMemory(),
        instructions="""You are a Destination Research Expert specializing in comprehensive travel destination analysis.
        Your expertise:
        - Landmarks and tourist activities
        - Best times to visit and seasonal considerations
        - Transportation options and accessibility
        - Safety considerations and travel advisories
        Always provide detailed, factual information with clear source attribution.""",
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
        requirements=[
            ConditionalRequirement(
                ThinkTool,
                force_at_step=1,
```

```
                min_invocations=1,
                max_invocations=5,
                consecutive_allowed=False
            ),
            ConditionalRequirement(
                WikipediaTool,
                only_after=[ThinkTool],
                min_invocations=1,
                max_invocations=4,
                consecutive_allowed=False
            ),
        ]
    )
```

Step 2: Create travel meteorologist expert

This agent specializes in weather analysis and climate-based travel recommendations:

```
    # === AGENT 2: TRAVEL METEOROLOGIST ===
    travel_meteorologist = RequirementAgent(
        llm=llm,
        tools=[OpenMeteoTool(), ThinkTool()],
        memory=UnconstrainedMemory(),
        instructions="""You are a Travel Meteorologist specializing in weather analysis for travel planning.
        Your expertise:
        - Climate patterns and seasonal weather analysis
        - Travel-specific weather recommendations
        - Packing suggestions based on weather forecasts
        - Activity planning based on weather conditions
        - Regional climate variations and microclimates
        - Weather-related travel risks and precautions
        Focus on actionable weather guidance for travelers.""",
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
        requirements=[
            ConditionalRequirement(
                ThinkTool,
                force_at_step=1,
                min_invocations=1,
                max_invocations=2
            ),
            ConditionalRequirement(
                OpenMeteoTool,
                only_after=[ThinkTool],
                min_invocations=1,
                max_invocations=1
            )
        ]
    )
```

Step 3: Create language and cultural expert

This agent provides cultural guidance and language assistance for respectful travel:

```
    # === AGENT 3: LANGUAGE & CULTURAL EXPERT ===
    language_and_culture_expert = RequirementAgent(
        llm=llm,
        tools=[WikipediaTool(), ThinkTool()],
        memory=UnconstrainedMemory(),
        instructions="""You are a Language & Cultural Expert specializing in linguistic and cultural guidance for travelers.
        Your expertise:
        - Local languages and dialects spoken in destinations
        - Essential phrases and communication tips for travelers
        - Cultural etiquette, customs, and social norms
        - Religious and cultural sensitivities to be aware of
        - Local communication styles and business etiquette
        - Cultural festivals, events, and local celebrations
        - Dining customs, tipping practices, and social interactions
        Always emphasize cultural sensitivity and respectful travel practices.""",
        middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
        requirements=[
            ConditionalRequirement(
                ThinkTool,
                force_at_step=1,
```

```
                min_invocations=1,
                max_invocations=3,
                consecutive_allowed=False
            ),
        ]
    )
```

Task 12.3: Create HandoffTools and coordinator agent

Now we create the **HandoffTool** instances that enable inter-agent communication, plus the coordinator agent that orchestrates the entire system:

Step 1: Create HandoffTool instances

Each HandoffTool connects the coordinator to a specific expert agent:

```
# === AGENT 4: TRAVEL COORDINATOR (MAIN INTERFACE) ===
# Create handoff tools for coordination with unique names
handoff_to_destination = HandoffTool(
    destination_expert,
    name="DestinationResearch",
    description="Consult our Destination Research Expert for comprehensive information about travel destinations, attractions, and p
)
handoff_to_weather = HandoffTool(
    travel_meteorologist,
    name="WeatherPlanning",
    description="Consult our Travel Meteorologist for weather forecasts, climate analysis, and weather-appropriate travel recommenda
)
handoff_to_language = HandoffTool(
    language_and_culture_expert,
    name="LanguageCulturalGuidance",
    description="Consult our Language & Cultural Expert for essential phrases, cultural etiquette, and communication guidance for re
)
```

Step 2: Create the travel coordinator agent

The coordinator uses HandoffTools as regular tools to delegate tasks to expert agents:

```
travel_coordinator = RequirementAgent(
    llm=llm,
    tools=[handoff_to_destination, handoff_to_weather, handoff_to_language, ThinkTool()],
    memory=UnconstrainedMemory(),
    instructions="""You are the Travel Coordinator, the main interface for comprehensive travel planning.
    Your role:
    - Understand traveler requirements and preferences
    - Coordinate with specialized expert agents as needed
    - Synthesize information from multiple sources
    - Create comprehensive, actionable travel recommendations
    - Ensure all aspects of travel planning are covered
    Available Expert Agents:
    - Destination Expert: Practical destination information
    - Travel Meteorologist: Weather analysis and climate recommendations
    - Language Expert: Language tips, cultural etiquette, and communication guidance
    Coordination Process:
    1. Think about what information is needed for comprehensive travel planning
    2. Delegate specific queries to appropriate expert agents using handoff tools
    3. Gather insights from multiple specialists
    4. Synthesize information into cohesive travel recommendations
    5. Provide a complete travel planning summary
    Always ensure travelers receive well-rounded guidance covering destinations and landmarks, weather, and cultural considerations.
    middlewares=[GlobalTrajectoryMiddleware(included=[Tool])],
    requirements=[
        ConditionalRequirement(ThinkTool, consecutive_allowed=False),
        AskPermissionRequirement(["DestinationResearch", "WeatherPlanning", "LanguageCulturalGuidance"])
    ]
)
```

Task 12.4: Execute the multi-agent travel planning

Step 1: Test the multi-agent system

Now we'll run the complete system to see how HandoffTool enables coordination between specialized agents:

How multi-agent coordination works:

1. **Coordinator receives complex query** requiring multiple specializations
2. **HandoffTools enable task delegation** to appropriate expert agents
3. **Expert agents process specialized sub-tasks** using their tools and expertise
4. **Results flow back** through HandoffTools to the coordinator
5. **Coordinator synthesizes** all expert insights into comprehensive response

```
    query = """I'm planning a 2-week cultural immersion trip to Japan (Tokyo and Osaka) as a first-time visitor.
    I want to experience traditional culture, visit historical sites, and interact with locals.
    I speak only English and want to be respectful of Japanese customs.
    What should I know about the destination, weather expectations, and language/cultural tips?"""

    result = await travel_coordinator.run(query)
    print(f"\n Comprehensive Travel Plan:\n{result.answer.text}")
async def main() -> None:
    logging.getLogger('asyncio').setLevel(logging.CRITICAL)
    await multi_agent_travel_planner_with_language()
if __name__ == "__main__":
    asyncio.run(main())
```

Multi-agent benefits demonstrated:

- **Specialized agent roles**: Each agent focuses on their domain of expertise for maximum effectiveness
- **HandoffTool communication**: Agents collaborate through structured handoff mechanisms for seamless coordination
- **Requirements-based control**: Each agent follows systematic execution patterns for reliable performance
- **Human-in-the-loop**: AskPermissionRequirement ensures oversight of agent coordination for transparency
- **Comprehensive results**: Combined expertise produces better results than any single agent could achieve

Now run the complete multi-agent travel planner:

```
python3.11 t12.py
```

Click the button below to see the complete multi-agent travel planner script

▶ Complete script

This multi-agent travel planner demonstrates advanced AI orchestration.

Multi-agent system benefits:

- **Specialized expertise from dedicated agents** for focused domain knowledge
- **Coordinated information gathering and synthesis** through handoff tools
- **Requirements-based execution control** ensuring systematic workflows
- **Language and cultural expertise integration** for respectful travel
- **Comprehensive, well-rounded travel planning** covering all aspects
- **Modular, scalable agent architecture** for easy expansion
- **Clear separation of concerns and responsibilities** between agents

Key innovation: Language expert agent

- Provides essential phrases and pronunciation guidance
- Explains cultural etiquette and social norms

- Helps travelers communicate respectfully
- Reduces cultural misunderstandings and barriers
- Enhances overall travel experience through cultural awareness

Key features of this multi-agent travel planner:

Agent specializations

Agent	Role	Tools	Expertise
Destination Expert	Research destinations	Wikipedia, Think	Attractions, history, culture, safety
Travel Meteorologist	Weather analysis	OpenMeteo, Think	Climate patterns, seasonal recommendations
Language Expert	Cultural & linguistic guidance	Wikipedia, Think	Essential phrases, cultural etiquette
Travel Coordinator	Main interface	Handoff tools, Think	Coordination and synthesis

Multi-agent coordination patterns

Requirements-based execution:

- **Mandatory thinking**: All agents must think systematically first
- **Tool dependencies**: Research tools only used after planning
- **Invocation limits**: Prevents resource waste and ensures focus
- **Expert handoffs**: Coordinator delegates to appropriate specialists

Information flow:

1. **User query** → Travel Coordinator
2. **Coordinator analysis** → Identifies needed expertise
3. **Expert consultation** → Handoffs to specialized agents
4. **Information synthesis** → Comprehensive recommendations

Language agent innovation

The **Language Expert** provides:

- **Essential phrases**: Key words and expressions for travelers
- **Cultural etiquette**: Do's and don'ts for respectful interactions
- **Communication tips**: Overcoming language barriers
- **Cultural context**: Understanding local customs and sensitivities

BeeAI Framework reference

RequirementAgent feature summary

RequirementAgent provides a comprehensive framework for building reliable, controlled AI agents. Here's a summary of key capabilities you've learned:

Feature	Description	Use Case
ConditionalRequirement	Controls tool execution order and frequency	Enforce systematic workflows
AskPermissionRequirement	Requires human approval for tool usage	Security and compliance
GlobalTrajectoryMiddleware	Tracks complete execution flow	Debugging and transparency
force_at_step	Forces tool usage at specific steps	Critical initial operations
min/max_invocations	Controls tool usage frequency	Resource management
only_after	Creates tool dependencies	Logical execution sequences
consecutive_allowed	Prevents consecutive tool usage	Avoid repetitive operations

Production Benefits

RequirementAgent excels in production environments:

Predictable behavior: Requirements ensure consistent execution patterns
Error handling: Built-in controls prevent common failure modes
Transparency: Middleware provides complete execution visibility
Security: Permission requirements enable human oversight
Scalability: Declarative design scales across different models and tools
Maintainability: Clear separation between logic, tools, and control flow

Conclusion

Congratulations! You've successfully completed a comprehensive journey through the BeeAI Framework, mastering the essential skills for building production-ready AI agents and multi-agent systems. You've progressed from basic chat models to sophisticated multi-agent architectures, gaining hands-on experience with real-world applications in cybersecurity, business planning, and intelligent automation.

What you've accomplished

Against each learning objective, here's what you've mastered:

Core concepts: You can now confidently work with chat models, implement memory management strategies, and generate structured outputs using Pydantic schemas—the foundation of reliable AI systems.

Agent development: You've built agents with custom tools and capabilities, understanding how to create RequirementAgents that can reason, take actions, and maintain context across interactions.

Advanced patterns: You've implemented multi-agent workflows, mastered requirements-based control systems, and applied systematic reasoning patterns like ReAct (Reasoning and Acting) for transparent agent behavior.

Real-world systems: You've created practical applications including AI math assistants, multi-agent travel planners, systematic reasoning agents, and interactive business plan generators.

Production techniques: You've applied enterprise-grade patterns including error handling, human-in-the-loop approvals, middleware tracking, and optimization strategies essential for scalable AI systems.

The progression from simple conversations to complex multi-agent coordination demonstrates your readiness to tackle sophisticated AI challenges in professional environments. If you missed any concepts or want to reinforce your learning, you can always revisit specific sections of this lab.

Author(s)

Wojciech “Victor” Fulmyk