Semantic Kernel Multi-Agent Orchestration Python Hands-On Lab - Azure Government

This comprehensive hands-on lab will guide you through building advanced multi-agent systems with Semantic Kernel in Python. You'll learn to create single agents, manage conversation threads, implement sequential orchestration, and handle agent response callbacks using Azure OpenAI in Azure Government Cloud.

Prerequisites

Development Environment

- Visual Studio Code
- Python 3.10 or later (recommended for compatibility with latest Semantic Kernel features)
- Python extension for VS Code

Required Azure OpenAI Resources

- Azure OpenAI resource deployed in Azure Government Cloud
- Azure OpenAI service endpoint (e.g., `https://your-resource.openai.usgovcloudapi.net/`)
- Azure OpenAI API key
- Deployed models: `gpt-4` or `gpt-35-turbo` (for optimal agent performance)

Required Python Packages

Before starting this multi-agent lab, you'll need to install the following Python packages. These packages provide the agent orchestration capabilities beyond the basic Semantic Kernel framework:

Package Name	Version	Purpose
`semantic-kernel[azure]`	Latest stable	Core Semantic Kernel with Azure connectors including agent support
`python-dotenv`	Latest	Environment variable management for secure configuration
`pydantic`	Latest	Data validation and serialization for structured data

Installation Command:

```
# Run this single command to install all required packages
pip install semantic-kernel[azure] python-dotenv pydantic
```

Note on Package Simplification:

- The `semantic-kernel[azure]` extra automatically includes `azure-identity`, `azure-core`, and other Azure-specific dependencies that are officially tested with Semantic Kernel
- `asyncio` is a built-in Python module (Python 3.4+) and doesn't need separate installation

- `typing-extensions` is typically included as a dependency of modern Python packages when needed
- Advanced agent features like OpenAI Responses Agent require Semantic Kernel Python 1.27.0 or later

Alternative Individual Installation:

```
pip install semantic-kernel[azure]
pip install python-dotenv
pip install pydantic
```

Step 1: Python Project Setup

1. **Create a new folder** for the Python multi-agent project:

```
mkdir SemanticKernelMultiAgentPython
cd SemanticKernelMultiAgentPython
code .
```

2. **Create a virtual environment** in the VS Code terminal:

```
python -m venv sk-multiagent-env
# On Windows:
sk-multiagent-env\Scripts\activate
# On macOS/Linux:
source sk-multiagent-env/bin/activate
```

3. **Install required packages**:

```
pip install semantic-kernel[azure] python-dotenv pydantic
```

> **What you're doing:** These packages provide the core multi-agent capabilities, Azure Government integration, and supporting functionality needed for advanced agent orchestration.

4. **Create a `requirements.txt` file**:

```
pip freeze > requirements.txt
```

Step 2: Environment Configuration

5. **Create a `.env` file** for secure credential storage:

```
# Azure Government OpenAI Configuration
AZURE_OPENAI_ENDPOINT=https://your-resource.openai.usgovcloudapi.net/
AZURE_OPENAI_API_KEY=your-azure-openai-api-key
AZURE_OPENAI_CHAT_DEPLOYMENT=gpt-4
AZURE_OPENAI_MODEL_ID=gpt-4
```

6. **Create a `.env.example` file** as a template:

```
# Example environment variables for Azure Government
AZURE_OPENAI_ENDPOINT=https://your-resource.openai.usgovcloudapi.net/
AZURE_OPENAI_API_KEY=your-azure-openai-api-key-here
AZURE_OPENAI_CHAT_DEPLOYMENT=gpt-4
AZURE_OPENAI_MODEL_ID=gpt-4
```

> **What you're doing:** Environment variables provide secure credential management for Azure Government OpenAI resources, ensuring sensitive information is not hardcoded in your application.

Step 3: Create Base Agent Infrastructure

7. **Create** `agent_base.py` to define the base agent structure:

```python
from abc import ABC, abstractmethod
from typing import Optional, Dict, Any
from datetime import datetime
import logging

class AgentBase(ABC):
    """Abstract base class for all agent implementations."""

    def __init__(self, agent_id: str, name: str, description: str):
        self.agent_id = agent_id
        self.name = name
        self.description = description
        self.created_at = datetime.utcnow()
        self.logger = logging.getLogger(f"Agent.{name}")

        self.log_agent_action("Initialized", f"Agent created with ID: {agent_id}")

    @abstractmethod
    async def process_message_async(self, message: str) -> str:
        """Process a message and return a response."""
        pass

    def log_agent_action(self, action: str, details: str = ""):
        """Log agent actions with consistent formatting."""
        log_message = f"🤖 [{self.name}] {action}"
        if details:
            log_message += f"\n   └─ {details}"
        print(log_message)
        self.logger.info(f"{action}: {details}")

    def get_agent_info(self) -> Dict[str, Any]:
        """Get agent information."""
        return {
            "agent_id": self.agent_id,
            "name": self.name,
            "description": self.description,
            "created_at": self.created_at.isoformat(),
            "type": self.__class__.__name__
        }
```

> **What you're doing:** This creates a base class that all your agents will inherit from, providing common functionality, logging capabilities, and a consistent interface for agent operations.

Step 4: Create Chat Completion Agent

8. **Create** `chat_completion_agent.py`:

```
import asyncio
from typing import List, Optional
from datetime import datetime
from semantic_kernel import Kernel
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion
from semantic_kernel.contents import ChatHistory, ChatMessageContent
from agent_base import AgentBase

class ChatCompletionAgent(AgentBase):
    """A chat completion agent that maintains conversation history."""

    def __init__(
        self,
        agent_id: str,
        name: str,
        description: str,
        system_prompt: str,
        kernel: Kernel
    ):
        super().__init__(agent_id, name, description)
        self.kernel = kernel
        self.system_prompt = system_prompt
        self.chat_history = ChatHistory()
        self.chat_service = kernel.get_service(AzureChatCompletion)

        # Initialize with system prompt
        if system_prompt:
            self.chat_history.add_system_message(system_prompt)

        self.log_agent_action("Chat agent initialized", f"System prompt: {system_prompt}")
    async def process_message_async(self, message: str) -> str:
        """Process a message and return a response."""
        self.log_agent_action("Processing message", message)

        try:
            # Add user message to history
            self.chat_history.add_user_message(message)

            # Get response from Azure OpenAI
            response = await self.chat_service.get_chat_message_content(
                chat_history=self.chat_history,
                settings=None,
                kernel=self.kernel
            )

            # Add assistant response to history
            response_content = str(response) if response else ""
            self.chat_history.add_assistant_message(response_content)

            self.log_agent_action("Generated response", response_content)
            return response_content

        except Exception as ex:
            error_msg = f"Error processing message: {str(ex)}"
            self.log_agent_action("Error", error_msg)
            raise

    def clear_history(self):
        """Clear chat history and reinitialize with system prompt."""
        self.chat_history.clear()
        if self.system_prompt:
            self.chat_history.add_system_message(self.system_prompt)
        self.log_agent_action("Chat history cleared")

    def get_message_count(self) -> int:
        """Get the number of messages in chat history."""
        return len(self.chat_history)

    def get_chat_history(self) -> ChatHistory:
        """Get the current chat history."""
        return self.chat_history
```

> **What you're doing:** This creates a Chat Completion Agent that maintains conversation history and can process messages using Azure OpenAI. The agent remembers the conversation context and applies a system prompt to guide its behavior.

Step 5: Create Azure AI Agent (Enhanced Agent)

9. **Create** `azure_ai_agent.py` for more advanced agent capabilities:

```
import json
import asyncio
from typing import Dict, List, Any, Optional
from datetime import datetime
from semantic_kernel import Kernel
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion
from
semantic_kernel.connectors.ai.open_ai.prompt_execution_settings.open_ai_prompt_exe
cution_settings import OpenAIChatPromptExecutionSettings
from semantic_kernel.contents import ChatHistory
from agent_base import AgentBase
from pydantic import BaseModel

class AgentAction(BaseModel):
    """Model for tracking agent actions."""
    timestamp: datetime
    input_message: str
    output_message: str = ""
    agent_id: str
    success: bool = True
    error_message: Optional[str] = None

class AzureAIAgent(AgentBase):
    """Enhanced AI agent with context management and action tracking."""

    def __init__(
        self,
        agent_id: str,
        name: str,
        description: str,
        system_prompt: str,
        kernel: Kernel
    ):
        super().__init__(agent_id, name, description)
        self.kernel = kernel
        self.system_prompt = system_prompt
        self.chat_history = ChatHistory()
        self.chat_service = kernel.get_service(AzureChatCompletion)
        self.context: Dict[str, Any] = {}
        self.action_history: List[AgentAction] = []

        # Initialize with enhanced system prompt
        enhanced_prompt = (f"{system_prompt}\n\n"
                          f"You are an AI agent named '{name}' with ID
'{agent_id}'. "
                          f"{description}")
        self.chat_history.add_system_message(enhanced_prompt)

        self.log_agent_action("Enhanced AI agent initialized", enhanced_prompt)

    async def process_message_async(self, message: str) -> str:
        """Process a message with enhanced capabilities."""
        self.log_agent_action("Processing message with context", message)
        action = AgentAction(
            timestamp=datetime.utcnow(),
            input_message=message,
            agent_id=self.agent_id
        )

        try:
            # Build contextual message if context is available
            contextual_message = self._build_contextual_message(message)

            self.chat_history.add_user_message(contextual_message)

            # Configure Azure OpenAI execution settings for Azure Government
            execution_settings = OpenAIChatPromptExecutionSettings(
                temperature=0.7,
                max_tokens=1000,
                top_p=0.9,
                frequency_penalty=0.0,
                presence_penalty=0.0
            )

            response = await self.chat_service.get_chat_message_content(
                chat_history=self.chat_history,
                settings=execution_settings,
                kernel=self.kernel
            )

            response_content = str(response) if response else ""
            self.chat_history.add_assistant_message(response_content)

            action.output_message = response_content
            action.success = True

            self.log_agent_action("Generated enhanced response", response_content)
            return response_content

        except Exception as ex:
            error_msg = str(ex)
            action.success = False
            action.error_message = error_msg
            self.log_agent_action("Error in enhanced processing", error_msg)
            raise
        finally:
            self.action_history.append(action)

    def _build_contextual_message(self, message: str) -> str:
        """Build a message with context information."""
        if not self.context:
            return message

        context_json = json.dumps(self.context, indent=2)
        return f"Context: {context_json}\n\nUser Message: {message}"
    def add_context(self, key: str, value: Any):
        """Add context information."""
        self.context[key] = value
        self.log_agent_action("Context updated", f"{key} = {value}")

    def remove_context(self, key: str):
        """Remove context information."""
        if key in self.context:
            del self.context[key]
            self.log_agent_action("Context removed", key)

    def get_context(self) -> Dict[str, Any]:
        """Get current context."""
        return self.context.copy()

    def get_action_history(self) -> List[AgentAction]:
        """Get action history."""
        return self.action_history.copy()
```

> **What you're doing:** This creates an enhanced Azure AI Agent that includes context management, action history tracking, and more sophisticated Azure OpenAI integration specifically configured for Azure Government Cloud.

Step 6: Create Conversation Thread Manager

10. **Create** `conversation_thread.py`:

```
import json
from typing import List, Dict, Any, Optional
from datetime import datetime
from enum import Enum
from pydantic import BaseModel

class ThreadMessageRole(Enum):
    """Enum for thread message roles."""
    USER = "user"
    AGENT = "agent"
    SYSTEM = "system"

class ThreadMessage(BaseModel):
    """Model for thread messages."""
    id: str
    agent_id: str
    agent_name: str
    content: str
    role: ThreadMessageRole
    timestamp: datetime

class ConversationThread:
    """Manages conversation threads with multiple agents."""
    def __init__(self, thread_id: Optional[str] = None):
        import uuid
        self.thread_id = thread_id or str(uuid.uuid4())
        self.messages: List[ThreadMessage] = []
        self.metadata: Dict[str, Any] = {}
        self.created_at = datetime.utcnow()
        self.last_updated = self.created_at

        print(f"🧵 Thread created: {self.thread_id}")

    def add_message(
        self,
        agent_id: str,
        agent_name: str,
        content: str,
        role: ThreadMessageRole
    ):
        """Add a message to the thread."""
        import uuid
        message = ThreadMessage(
            id=str(uuid.uuid4()),
            agent_id=agent_id,
            agent_name=agent_name,
            content=content,
            role=role,
            timestamp=datetime.utcnow()
        )

        self.messages.append(message)
        self.last_updated = datetime.utcnow()

        print(f"📝 [{agent_name}] {role.value}: {content}")

    def add_user_message(self, content: str):
        """Add a user message to the thread."""
        self.add_message("user", "User", content, ThreadMessageRole.USER)

    def add_agent_message(self, agent_id: str, agent_name: str, content: str):
        """Add an agent message to the thread."""
        self.add_message(agent_id, agent_name, content, ThreadMessageRole.AGENT)

    def set_metadata(self, key: str, value: Any):
        """Set metadata for the thread."""
        self.metadata[key] = value
        self.last_updated = datetime.utcnow()

    def get_metadata(self, key: str, default: Any = None) -> Any:
        """Get metadata from the thread."""
        return self.metadata.get(key, default)

    def get_messages_by_agent(self, agent_id: str) -> List[ThreadMessage]:
        """Get messages from a specific agent."""
        return [msg for msg in self.messages if msg.agent_id == agent_id]
    def get_thread_summary(self) -> str:
        """Get a summary of the thread."""
        summary = (f"Thread {self.thread_id}: {len(self.messages)} messages, "
                  f"Created: {self.created_at.strftime('%Y-%m-%d %H:%M:%S')}, "
                  f"Last Updated: {self.last_updated.strftime('%Y-%m-%d 
%H:%M:%S')}")
        return summary

    def export_to_json(self) -> str:
        """Export thread to JSON format."""
        export_data = {
            "thread_id": self.thread_id,
            "created_at": self.created_at.isoformat(),
            "last_updated": self.last_updated.isoformat(),
            "messages": [
                {
                    "id": msg.id,
                    "agent_id": msg.agent_id,
                    "agent_name": msg.agent_name,
                    "content": msg.content,
                    "role": msg.role.value,
                    "timestamp": msg.timestamp.isoformat()
                }
                for msg in self.messages
            ],
            "metadata": self.metadata
        }

        return json.dumps(export_data, indent=2)
```

> **What you're doing:** This creates a conversation thread manager that tracks all messages, maintains metadata, and provides a structured way to manage multi-agent conversations with full history and context preservation.

Step 7: Create Sequential Orchestration

11. **Create** `sequential_orchestrator.py`:

```
import asyncio
import json
from typing import List, Dict, Any, Optional, Callable, Awaitable, Tuple
from datetime import datetime
from pydantic import BaseModel
from agent_base import AgentBase
from conversation_thread import ConversationThread

class OrchestrationStep(BaseModel):
    """Model for orchestration steps."""
    step_number: int
    agent_id: str
    agent_name: str
    input_message: str
    output_message: str = ""
    start_time: datetime
    end_time: Optional[datetime] = None
    success: bool = True
    error_message: Optional[str] = None

    @property
    def duration(self) -> float:
        """Get step duration in seconds."""
        if self.end_time:
            return (self.end_time - self.start_time).total_seconds()
        return 0.0

class OrchestrationResult(BaseModel):
    """Model for orchestration results."""
    start_time: datetime
    end_time: Optional[datetime] = None
    initial_message: str
    final_response: str = ""
    success: bool = True
    error_message: Optional[str] = None
    steps: List[OrchestrationStep] = []

    @property
    def duration(self) -> float:
        """Get total duration in seconds."""
        if self.end_time:
            return (self.end_time - self.start_time).total_seconds()
        return 0.0

class SequentialOrchestrator:
    """Orchestrates multiple agents in sequential order."""

    # Type aliases for callbacks
    AgentResponseCallback = Callable[[str, str, str, bool], Awaitable[None]]
    HumanResponseFunction = Callable[[str], Awaitable[Tuple[bool, str]]]

    def __init__(self, thread: Optional[ConversationThread] = None):
        self.agents: List[AgentBase] = []
        self.thread = thread or ConversationThread()
        self.orchestration_history: List[OrchestrationStep] = []

        # Callbacks
        self.on_agent_response: Optional[self.AgentResponseCallback] = None
        self.on_human_response_required: Optional[self.HumanResponseFunction] =
None

        print("🎭 Sequential Orchestrator initialized")

    def add_agent(self, agent: AgentBase):
        """Add an agent to the orchestrator."""
        self.agents.append(agent)
        print(f"➕ Agent added to orchestrator: {agent.name} (ID:
```

```
{agent.agent_id})")

    async def execute_sequential_async(self, initial_message: str) -> OrchestrationResult:
        """Execute sequential orchestration."""
        print(f"\n🚀 Starting sequential orchestration with {len(self.agents)} agents")
        print(f"Initial message: {initial_message}")

        result = OrchestrationResult(
            start_time=datetime.utcnow(),
            initial_message=initial_message
        )

        # Add initial user message to thread
        self.thread.add_user_message(initial_message)

        current_message = initial_message

        try:
            for i, agent in enumerate(self.agents):
                step = OrchestrationStep(
                    step_number=i + 1,
                    agent_id=agent.agent_id,
                    agent_name=agent.name,
                    input_message=current_message,
                    start_time=datetime.utcnow()
                )

                print(f"\n🔄 Step {step.step_number}: Processing with {agent.name}")

                try:
                    # Process message with current agent
                    agent_response = await agent.process_message_async(current_message)

                    step.output_message = agent_response
                    step.success = True
                    step.end_time = datetime.utcnow()

                    # Add agent response to thread
                    self.thread.add_agent_message(agent.agent_id, agent.name, agent_response)

                    # Trigger agent response callback
                    if self.on_agent_response:
                        await self.on_agent_response(
                            agent.agent_id,
                            agent.name,
                            agent_response,
                            True
                        )
                    # Check if human response is required before proceeding to next agent
                    if i < len(self.agents) - 1 and self.on_human_response_required:
                        prompt = (f"Agent {agent.name} responded: {agent_response}\n\n"
                                f"Would you like to modify the input for the next agent "
                                f"({self.agents[i + 1].name})? Current input will be: '{agent_response}'")

                        should_modify, human_response = await self.on_human_response_required(prompt)

                        if should_modify and human_response.strip():
                            current_message = human_response
                            self.thread.add_user_message(f"Human intervention: {human_response}")
                            print(f"👤 Human provided modified input: {human_response}")
                        else:
                            current_message = agent_response
                    else:
                        current_message = agent_response

                except Exception as ex:
                    step.success = False
                    step.error_message = str(ex)
                    step.end_time = datetime.utcnow()

                    print(f"❌ Error in step {step.step_number}: {ex}")

                    # Trigger agent response callback with error
                    if self.on_agent_response:
                        await self.on_agent_response(
                            agent.agent_id,
                            agent.name,
                            f"Error: {ex}",
                            False
                        )

                    result.success = False
                    result.error_message = str(ex)

                result.steps.append(step)
                self.orchestration_history.append(step)

            result.end_time = datetime.utcnow()
            result.final_response = current_message

            if result.success:
                print(f"\n✅ Sequential orchestration completed successfully")
                print(f"Final response: {result.final_response}")
```

```
        except Exception as ex:
            result.success = False
            result.error_message = str(ex)
            result.end_time = datetime.utcnow()

            print(f"❌ Orchestration failed: {ex}")

        return result

    def get_orchestration_summary(self) -> str:
        """Get orchestration summary as JSON."""
        summary = {
            "thread_id": self.thread.thread_id,
            "agent_count": len(self.agents),
            "total_steps": len(self.orchestration_history),
            "agents": [
                {
                    "agent_id": agent.agent_id,
                    "name": agent.name,
                    "description": agent.description
                }
                for agent in self.agents
            ],
            "thread_summary": self.thread.get_thread_summary()
        }

        return json.dumps(summary, indent=2)
```

> **What you're doing:** This creates a sophisticated sequential orchestrator that manages multiple agents in sequence, handles callbacks, supports human-in-the-loop scenarios, and provides detailed tracking of the orchestration process.

Step 8: Complete Application Implementation

12. **Create** `main.py` with the complete multi-agent implementation:

```
import asyncio
import os
import logging
from typing import Tuple
from dotenv import load_dotenv
from semantic_kernel import Kernel
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion

# Import our custom modules
from chat_completion_agent import ChatCompletionAgent
from azure_ai_agent import AzureAIAgent
from conversation_thread import ConversationThread
from sequential_orchestrator import SequentialOrchestrator

# Load environment variables
```

```
load_dotenv()

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def create_kernel() -> Kernel:
    """Create and configure the Semantic Kernel for Azure Government."""
    # Get Azure OpenAI configuration from environment variables
    endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
    api_key = os.getenv("AZURE_OPENAI_API_KEY")
    deployment_name = os.getenv("AZURE_OPENAI_CHAT_DEPLOYMENT")

    if not all([endpoint, api_key, deployment_name]):
        raise ValueError("Missing required Azure OpenAI environment variables")

    print(f"🔧 Configuring kernel with Azure Government endpoint: {endpoint}")

    # Create kernel
    kernel = Kernel()

    # Add Azure OpenAI chat completion for Azure Government
    chat_completion = AzureChatCompletion(
        service_id="default",
        deployment_name=deployment_name,
        endpoint=endpoint,
        api_key=api_key
    )
    kernel.add_service(chat_completion)

    print("✅ Kernel created successfully with Azure Government configuration!")
    return kernel

async def demonstrate_single_agent(kernel: Kernel):
    """Demonstrate single agent creation and usage."""
    print("\n ◇ DEMONSTRATION 1: Single Agent Creation and Usage")
    print("====================================================")

    # Create a Chat Completion Agent
    chat_agent = ChatCompletionAgent(
        agent_id="chat-agent-001",
        name="Chat Assistant",
        description="A general-purpose chat assistant",
        system_prompt="You are a helpful AI assistant that provides clear,
concise, and accurate responses.",
        kernel=kernel
    )

    # Create an Azure AI Agent
    azure_agent = AzureAIAgent(
        agent_id="azure-agent-001",
        name="Azure Specialist",
        description="An AI agent specialized in Azure services and solutions",
        system_prompt="You are an Azure expert who helps users with Azure
```

```
services, best practices, and solutions. Always provide practical, actionable
advice.",
        kernel=kernel
    )

    # Test Chat Completion Agent
    print("\n🧪 Testing Chat Completion Agent:")
    chat_response = await chat_agent.process_message_async("What is artificial
intelligence?")
    print(f"Response: {chat_response}\n")

    # Test Azure AI Agent with context
    print("🧪 Testing Azure AI Agent with context:")
    azure_agent.add_context("user_role", "Cloud Architect")
    azure_agent.add_context("project_type", "Government Compliance")

    azure_response = await azure_agent.process_message_async(
        "How should I design a secure Azure architecture for a government
application?"
    )
    print(f"Response: {azure_response}\n")

    # Show agent capabilities
    print("📊 Agent Information:")
    print(f"Chat Agent Messages: {chat_agent.get_message_count()}")
    print(f"Azure Agent Context: {len(azure_agent.get_context())} items")
    print(f"Azure Agent Actions: {len(azure_agent.get_action_history())}
actions\n")

async def demonstrate_thread_management(kernel: Kernel):
    """Demonstrate conversation thread management."""
    print(" ◇  DEMONSTRATION 2: Conversation Thread Management")
    print("===================================================")

    # Create a conversation thread
    thread = ConversationThread()
    print(f"Thread Details: {thread.get_thread_summary()}\n")

    # Create agents for the thread
    analyzer_agent = ChatCompletionAgent(
        agent_id="analyzer-001",
        name="Data Analyzer",
        description="Analyzes data and provides insights",
        system_prompt="You are a data analyst who examines information and
provides clear insights and recommendations.",
        kernel=kernel
    )

    reviewer_agent = AzureAIAgent(
        agent_id="reviewer-001",
        name="Content Reviewer",
        description="Reviews and validates content",
        system_prompt="You are a content reviewer who ensures accuracy, clarity,
and completeness of information.",
        kernel=kernel
    )

    # Simulate a conversation in the thread
    print("💬 Simulating conversation in thread:")

    user_message = "Please analyze the benefits of using Azure Government for
sensitive workloads"
    thread.add_user_message(user_message)

    analysis_response = await analyzer_agent.process_message_async(user_message)
    thread.add_agent_message(analyzer_agent.agent_id, analyzer_agent.name,
analysis_response)

    review_response = await reviewer_agent.process_message_async(
        f"Please review this analysis: {analysis_response}"
    )
    thread.add_agent_message(reviewer_agent.agent_id, reviewer_agent.name,
review_response)

    # Show thread information
    print(f"\n📊 Thread Summary: {thread.get_thread_summary()}")
    print(f"Total Messages: {len(thread.messages)}")

    thread.set_metadata("topic", "Azure Government Analysis")
    thread.set_metadata("participants", ["User", analyzer_agent.name,
reviewer_agent.name])

    exported_json = thread.export_to_json()
    print("💾 Thread exported to JSON:")
    print(exported_json[:500] + "...\n")

async def demonstrate_multi_agent_orchestration(kernel: Kernel):
    """Demonstrate multi-agent sequential orchestration."""
    print("◇ DEMONSTRATION 3: Multi-Agent Sequential Orchestration")
    print("=========================================================")

    # Create specialized agents for the orchestration
    requirements_agent = ChatCompletionAgent(
        agent_id="requirements-agent",
        name="Requirements Analyst",
        description="Analyzes and structures requirements",
        system_prompt="You are a requirements analyst. Take user input and
structure it into clear, detailed requirements. Format your response as a numbered
list.",
        kernel=kernel
    )

    architect_agent = AzureAIAgent(
        agent_id="architect-agent",
        name="Solutions Architect",
        description="Designs technical solutions",
        system_prompt="You are a solutions architect. Based on requirements,
design a high-level technical solution. Focus on Azure services and architecture
```

```
patterns.",
        kernel=kernel
    )

    security_agent = ChatCompletionAgent(
        agent_id="security-agent",
        name="Security Specialist",
        description="Reviews solutions for security compliance",
        system_prompt="You are a security specialist focused on Azure Government
compliance. Review the proposed solution and add security recommendations,
especially for government/compliance requirements.",
        kernel=kernel
    )

    # Create orchestrator and add agents
    orchestrator = SequentialOrchestrator()
    orchestrator.add_agent(requirements_agent)
    orchestrator.add_agent(architect_agent)
    orchestrator.add_agent(security_agent)

    # Implement agent response callback
    async def agent_response_callback(agent_id: str, agent_name: str, response:
str, is_success: bool):
        print(f"🔔 Agent Response Callback triggered:")
        print(f"   Agent: {agent_name} (ID: {agent_id})")
        print(f"   Success: {is_success}")
        print(f"   Response Length: {len(response)} characters")

        if not is_success:
            print(f"   ⚠️ Agent encountered an error")

        # You could log to external systems, trigger notifications, etc.
        await asyncio.sleep(0.1)  # Simulate some callback processing

    orchestrator.on_agent_response = agent_response_callback

    # Implement human response function
    async def human_response_function(prompt: str) -> Tuple[bool, str]:
        print(f"\n🤔 Human input requested:")
        print(f"Prompt: {prompt}")
        print("\nOptions:")
        print("1. Press ENTER to continue with agent output")
        print("2. Type custom input to modify the flow")

        # For demo purposes, we'll simulate automatic continuation
        # In a real application, you would get actual user input
        print("💭 Simulating automatic continuation for demo...")
        await asyncio.sleep(1)

        # Return False to indicate no human modification
        return False, ""

    orchestrator.on_human_response_required = human_response_function
    # Execute the orchestration
    initial_request = ("I need to build a secure document management system for a government agency "
                      "that handles classified documents. The system needs to be compliant with FedRAMP "
                      "and support multiple users with role-based access.")

    print(f"🚀 Starting orchestration with request:")
    print(f"'{initial_request}'\n")

    result = await orchestrator.execute_sequential_async(initial_request)

    # Display results
    print("\n📋 ORCHESTRATION RESULTS:")
    print("=========================")
    print(f"Success: {result.success}")
    print(f"Duration: {result.duration:.2f} seconds")
    print(f"Steps Completed: {len(result.steps)}")

    if not result.success and result.error_message:
        print(f"Error: {result.error_message}")

    print(f"\n📄 Final Response:")
    print(f"{result.final_response}")

    # Show detailed step information
    print(f"\n🔍 Detailed Step Information:")
    for step in result.steps:
        print(f"Step {step.step_number}: {step.agent_name}")
        print(f"  Duration: {step.duration:.2f}s")
        print(f"  Success: {step.success}")
        if not step.success and step.error_message:
            print(f"  Error: {step.error_message}")

    # Export orchestration summary
    print(f"\n💾 Orchestration Summary:")
    print(orchestrator.get_orchestration_summary())

async def main():
    """Main application entry point."""
    print("🤖 Semantic Kernel Multi-Agent Orchestration Lab")
    print("==================================================\n")

    try:
        # Create kernel with Azure OpenAI for Azure Government
        kernel = create_kernel()

        # Demonstrate single agent creation and usage
        await demonstrate_single_agent(kernel)

        # Demonstrate thread management
        await demonstrate_thread_management(kernel)

        # Demonstrate multi-agent orchestration
        await demonstrate_multi_agent_orchestration(kernel)

    except Exception as ex:
        print(f"❌ Application error: {ex}")
        logger.error(f"Application failed: {ex}")

    print("\n🎉 Multi-Agent Lab completed!")

if __name__ == "__main__":
    asyncio.run(main())
```

> **What you're doing:** This complete application demonstrates all aspects of multi-agent orchestration including single agent creation, thread management, sequential orchestration with callbacks, and human-in-the-loop functionality, all configured for Azure Government compliance.

Running the Multi-Agent Lab

1. **Navigate to your Python project folder** in VS Code

2. **Activate the virtual environment:**

   ```
   # Windows:
   sk-multiagent-env\Scripts\activate
   # macOS/Linux:
   source sk-multiagent-env/bin/activate
   ```

3. **Set your Azure OpenAI credentials in the `.env` file:**

   ```
   AZURE_OPENAI_ENDPOINT=https://your-actual-resource.openai.usgovcloudapi.net/
   AZURE_OPENAI_API_KEY=your-actual-api-key
   AZURE_OPENAI_CHAT_DEPLOYMENT=gpt-4
   AZURE_OPENAI_MODEL_ID=gpt-4
   ```

4. **Run the application:**

   ```
   python main.py
   ```

5. **Follow the interactive prompts** during the multi-agent orchestration demonstration

Key Learning Points

What You've Built

Single Agent Creation:

- **Chat Completion Agent:** Basic conversational AI with history management

- **Azure AI Agent:** Enhanced agent with context management and action tracking
- Both agents configured specifically for Azure Government Cloud compliance

Thread Management:

- **Conversation Threads:** Structured conversation management with metadata
- **Message Tracking:** Complete history of all agent and user interactions
- **Export Capabilities:** JSON export for persistence and analysis

Multi-Agent Orchestration:

- **Sequential Processing:** Agents process messages in a defined order
- **Agent Response Callbacks:** Real-time notifications of agent actions
- **Human Response Functions:** Human-in-the-loop capability for guided orchestration
- **Error Handling:** Comprehensive error management and recovery

Advanced Features:

- **Context Management:** Agents can maintain and share context information
- **Action History:** Detailed tracking of all agent actions and decisions
- **Metadata Management:** Rich metadata support for threads and conversations
- **Azure Government Integration:** All components designed for government compliance requirements

Development Patterns

Agent Architecture:

```
AgentBase (Abstract)
├── ChatCompletionAgent (Basic conversational agent)
└── AzureAIAgent (Enhanced agent with context and tracking)
```

Orchestration Flow:

```
User Input → Agent 1 → [Human Intervention?] → Agent 2 → ... → Final Response
     ↓            ↓                                ↓
  Thread      Callback                        Callback
```

File Organization:

```
Multi-Agent Python Project:
├── main.py (main application)
├── .env (Azure Government environment variables)
├── agent_base.py (base agent class)
├── chat_completion_agent.py (basic agent implementation)
├── azure_ai_agent.py (enhanced agent implementation)
├── conversation_thread.py (thread management)
└── sequential_orchestrator.py (orchestration engine)
```

Best Practices Demonstrated

Security:

- Environment-based configuration management
- Azure Government specific endpoints and compliance
- Secure credential handling through environment variables
- Context isolation between agents

Error Handling:

- Comprehensive try-catch blocks in all async operations
- Graceful degradation when agents fail
- Detailed error logging and reporting
- Recovery mechanisms in orchestration

Performance:

- Asynchronous processing throughout
- Efficient memory management with type hints
- Cancellation support for long-running operations
- Minimal object allocation in hot paths

Maintainability:

- Clear separation of concerns between components
- Extensible agent architecture through inheritance
- Configurable orchestration patterns
- Comprehensive logging and diagnostics

Troubleshooting Tips

Common Issues:

1. **Agent Creation Failures:** Verify Azure OpenAI deployment names and endpoints
2. **Orchestration Errors:** Check agent system prompts for clarity and completeness
3. **Callback Issues:** Ensure async/await patterns are properly implemented
4. **Thread Management:** Verify proper message ordering and metadata handling
5. **Azure Government Access:** Confirm subscription has access to required services
6. **Virtual Environment:** Ensure you're working within the activated virtual environment

Performance Considerations:

- Use appropriate Azure OpenAI model sizes for your use case
- Implement proper cancellation for long-running orchestrations
- Monitor token usage across multiple agents
- Consider implementing agent response caching for repeated queries

Government Compliance:

- All data processing occurs within Azure Government boundaries
- Maintain audit trails of all agent interactions
- Implement proper access controls for sensitive conversations
- Follow government data handling and retention policies

This comprehensive multi-agent lab demonstrates advanced Semantic Kernel capabilities for building sophisticated AI agent systems that can handle complex, multi-step processes while maintaining Azure Government compliance and security requirements.