# Advanced Topic: Integrate MCP Tools with Azure AI Agents

## 1. Concept Explanation

Integrating Model Context Protocol (MCP) tools with Azure AI Agents represents a significant advancement in building sophisticated and dynamic AI solutions. MCP is a standardized way for AI models and agents to interact with external tools and services, providing a common language and framework for tool invocation and data exchange. This protocol allows AI agents to extend their capabilities beyond their inherent knowledge by dynamically discovering and utilizing a wide array of external functionalities, effectively turning them into highly versatile problem-solvers.

**Key Concepts:**

* **Azure AI Agents:** These are intelligent software entities designed to perform tasks, make decisions, and interact with users or other systems. They can be built using frameworks like Semantic Kernel and leverage large language models (LLMs) for reasoning and natural language understanding.
* **Tools (or Functions/Skills):** These are external functionalities that an AI agent can call upon to perform specific actions. Examples include searching the web, sending emails, querying databases, interacting with APIs, or performing complex calculations. Tools provide agents with access to real-world data and actions.
* **Model Context Protocol (MCP):** MCP is a specification that defines how AI models and agents can describe, discover, and invoke tools. It provides a structured way for tools to expose their capabilities (e.g., input parameters, expected output, description of what they do) to an AI agent. This standardization enables agents to dynamically select and use the most appropriate tool for a given task without being explicitly programmed for each tool.
* **Dynamic Tool Invocation:** Instead of having a fixed set of pre-programmed actions, AI agents can, through MCP, reason about a user’s request, identify the need for an external tool, discover available tools that can fulfill that need, and then invoke the chosen tool with the correct parameters. This makes agents highly adaptable and capable of handling a broader range of complex tasks.

**Benefits of Integrating MCP Tools:**

* **Enhanced Capabilities:** Agents can perform actions beyond their core LLM capabilities, such as fetching real-time data, executing business logic, or interacting with external systems.
* **Increased Accuracy and Relevance:** By accessing up-to-date and specific information through tools, agents can provide more accurate and contextually relevant responses, reducing hallucinations.
* **Automation of Complex Workflows:** Agents can orchestrate multi-step processes by chaining together various tools, automating complex business workflows that previously required human intervention.
* **Modularity and Extensibility:** Tools can be developed independently and then exposed to agents via MCP, promoting modular design and making it easy to extend agent functionalities.
* **Reduced Development Effort:** Developers can focus on creating specialized tools, and agents can dynamically leverage them, reducing the need for hardcoding complex logic within the agent itself.

## 2. Relevant Use Cases and Implementation Scenarios

Integrating MCP tools with Azure AI Agents unlocks innovative solutions across diverse industries:

* **Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) Automation:**
  + **Use Case:** An AI agent assists sales teams by automatically retrieving customer information from CRM, generating personalized sales proposals based on product catalogs (via a tool), and updating sales records in the ERP system.
  + **Scenario:** A sales representative asks the agent, “What is the current order status for customer ‘Acme Corp’ and can you generate a draft email to them with the update?” The agent uses a ‘CRM\_Lookup’ tool to get the order status and customer email, then a ‘Email\_Drafting’ tool to compose the email, and finally a ‘ERP\_Update’ tool to log the interaction.
* **Financial Services and Investment Analysis:**
  + **Use Case:** An AI agent provides real-time market insights and executes trades based on user commands and predefined rules. It can access live stock data, financial news feeds, and trading platforms through various tools.
  + **Scenario:** An investor asks the agent, “What is the current price of AAPL and should I buy or sell based on recent news?” The agent uses a ‘Stock\_Price\_Lookup’ tool, a ‘News\_Sentiment\_Analysis’ tool, and then, based on its analysis, might suggest using a ‘Trading\_Platform\_Execute\_Order’ tool.
* **Healthcare and Patient Management:**
  + **Use Case:** An AI agent assists medical staff with patient information retrieval, appointment scheduling, and medication management. It can interact with Electronic Health Record (EHR) systems and pharmacy databases.
  + **Scenario:** A nurse asks the agent, “What are the latest lab results for patient Jane Doe and schedule her next follow-up appointment for next Tuesday.” The agent uses an ‘EHR\_Query’ tool to retrieve lab results and an ‘Appointment\_Scheduler’ tool to book the appointment.
* **Manufacturing and Supply Chain Optimization:**
  + **Use Case:** An AI agent monitors production lines, manages inventory, and optimizes logistics by interacting with IoT sensors, inventory management systems, and logistics platforms.
  + **Scenario:** A plant manager asks the agent, “Are there any anomalies on Line 3, and if so, order replacement parts for the faulty component.” The agent uses an ‘IoT\_Sensor\_Monitor’ tool to check anomalies, then an ‘Inventory\_Lookup’ tool to check part availability, and finally a ‘Procurement\_Order’ tool to place an order.
* **Legal Research and Document Management:**
  + **Use Case:** An AI agent assists legal professionals by performing legal research, summarizing case law, and managing legal documents. It can access legal databases and document management systems.
  + **Scenario:** A lawyer asks the agent, “Find all precedents related to intellectual property infringement in the last five years and summarize the key rulings.” The agent uses a ‘Legal\_Database\_Search’ tool, then a ‘Document\_Summarizer’ tool to process the results, and finally a ‘Document\_Management\_System\_Save’ tool to store the summaries.

## 3. Key Considerations and Best Practices

Implementing MCP tools with Azure AI Agents effectively requires careful planning and adherence to best practices:

* **Tool Design and Granularity:**
  + **Single Responsibility:** Each tool should ideally perform a single, well-defined function. This makes tools easier to develop, test, and maintain, and improves the agent’s ability to select the correct tool.
  + **Clear Descriptions:** Provide clear, concise, and accurate descriptions for each tool, including its purpose, input parameters, and expected output. This metadata is crucial for the agent’s reasoning and tool selection process.
  + **Error Handling:** Tools should be robust and include comprehensive error handling. Agents need to be able to gracefully handle failures in tool execution and potentially retry or escalate.
* **Security and Access Control:**
  + **Least Privilege:** Tools should operate with the principle of least privilege, having only the necessary permissions to perform their designated function.
  + **Authentication and Authorization:** Implement robust authentication and authorization mechanisms for accessing tools, especially those interacting with sensitive data or critical systems. Azure Active Directory and Managed Identities are key here.
  + **Input Validation:** Validate all inputs received by tools to prevent injection attacks or unexpected behavior.
* **Observability and Monitoring:**
  + **Logging:** Implement comprehensive logging for both agent decisions (tool selection, reasoning) and tool execution. This is vital for debugging, auditing, and understanding agent behavior.
  + **Monitoring:** Monitor tool performance, latency, and success rates. Set up alerts for failures or performance degradation.
  + **Tracing:** Implement distributed tracing to follow the flow of a request through the agent and multiple tool invocations.
* **Scalability and Performance:**
  + **Stateless Tools:** Design tools to be stateless where possible, making them easier to scale horizontally.
  + **Asynchronous Operations:** Leverage asynchronous programming for tool execution to prevent blocking the agent’s main thread, especially for long-running operations.
  + **Caching:** Implement caching mechanisms for frequently accessed data to reduce redundant tool calls and improve response times.
* **Version Control and Deployment:**
  + **Version Tools:** Treat tools as independent software components with their own versioning. This allows for independent updates and rollbacks.
  + **Automated Deployment:** Automate the deployment of tools and agents using CI/CD pipelines to ensure consistency and reliability.
* **Human-in-the-Loop:**
  + **Escalation Paths:** For complex or sensitive tasks, design clear escalation paths to human operators when the agent cannot confidently complete a task or encounters an unhandled error.
  + **Transparency:** Provide users with visibility into the agent’s reasoning and the tools it is using, fostering trust and understanding.
* **Cost Management:**
  + **Resource Optimization:** Optimize the underlying resources for hosting tools and agents to manage operational costs.
  + **Usage Monitoring:** Monitor API calls and resource consumption related to tool invocation to identify and address cost inefficiencies.

## 4. Sample Implementations

To illustrate the practical application of integrating MCP tools with Azure AI Agents, we will provide sample implementations in both C# and Python. Our scenario is inspired by a real customer case where a company leverages AI agents to automate aspects of their IT support, specifically by interacting with an external system to manage user accounts.

### Customer Case Inspiration: Automated IT Support with Tool Integration

Many large enterprises, like those using Microsoft’s internal IT systems, face challenges in managing user accounts, permissions, and common IT requests efficiently. Manually handling these requests can be time-consuming and prone to delays. An AI agent capable of understanding natural language requests and then executing specific IT operations (like resetting passwords, checking user status, or creating new accounts) by integrating with existing IT management tools can significantly improve operational efficiency and user satisfaction. This represents a highly innovative and disruptive approach to IT service management.

### Scenario: IT Support Agent with User Management Tools

Our sample implementation will demonstrate an IT support agent that can perform user management tasks by invoking external tools. The agent will be able to:

1. **Check User Status:** Determine if a user account is active or locked.
2. **Reset User Password:** Reset a user’s password.

These actions will be exposed as tools that the AI agent can dynamically call based on user requests.

**Components:**

* **Azure OpenAI Service:** To host the large language model for agent reasoning.
* **Semantic Kernel:** To orchestrate the agent’s behavior and tool invocation.
* **Custom Tools:** Simulated external functions for user management (e.g., CheckUserStatusTool, ResetUserPasswordTool). In a real-world scenario, these would interact with Active Directory, identity management systems, or other IT infrastructure APIs.

**Workflow:**

1. User asks the IT support agent a question (e.g., “Can you check the status of John Doe’s account?” or “Reset the password for Jane Smith.”).
2. The Semantic Kernel, leveraging the LLM, interprets the user’s intent and identifies the appropriate tool to use.
3. The agent extracts necessary parameters (e.g., username) from the user’s request.
4. The Semantic Kernel invokes the identified tool with the extracted parameters.
5. The tool executes the IT operation and returns a result.
6. The agent formulates a natural language response based on the tool’s output and communicates it to the user.

### C# Implementation Example

This C# example demonstrates an IT support agent that uses Semantic Kernel to integrate and invoke custom tools for user management. We will simulate the tools as simple C# methods.

First, ensure you have the necessary NuGet package installed:

<ItemGroup>  
 <PackageReference Include="Microsoft.SemanticKernel" Version="1.0.0-beta10" />  
 <PackageReference Include="Microsoft.SemanticKernel.Connectors.OpenAI" Version="1.0.0-beta10" />  
</ItemGroup>

using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.Connectors.OpenAI;  
using Microsoft.SemanticKernel.ChatCompletion;  
using System;  
using System.Threading.Tasks;  
using System.ComponentModel;  
  
// Define a simple class to represent our IT User Management Tools  
public class UserManagementTools  
{  
 [KernelFunction, Description("Checks the status of a user account (active or locked).")]  
 public string CheckUserStatus([Description("The username of the account to check.")] string username)  
 {  
 Console.WriteLine($"[Tool Call: CheckUserStatus for {username}]");  
 // Simulate interaction with an external system  
 if (username.ToLower() == "johndoe")  
 {  
 return $"User {username} is active.";  
 }  
 else if (username.ToLower() == "janesmith")  
 {  
 return $"User {username} is locked.";  
 }  
 else  
 {  
 return $"User {username} not found.";  
 }  
 }  
  
 [KernelFunction, Description("Resets the password for a specified user account.")]  
 public string ResetUserPassword([Description("The username of the account to reset password for.")] string username)  
 {  
 Console.WriteLine($"[Tool Call: ResetUserPassword for {username}]");  
 // Simulate interaction with an external system  
 if (username.ToLower() == "johndoe" || username.ToLower() == "janesmith")  
 {  
 return $"Password for user {username} has been successfully reset.";  
 }  
 else  
 {  
 return $"Could not reset password for user {username}. User not found.";  
 }  
 }  
}  
  
public class ITSupoportAgent  
{  
 private readonly Kernel \_kernel;  
 private readonly ChatHistory \_chatHistory;  
  
 public ITSupoportAgent(string openAiEndpoint, string openAiKey, string openAiDeploymentName)  
 {  
 \_kernel = Kernel.CreateBuilder()  
 .AddAzureOpenAIChatCompletion(  
 deploymentName: openAiDeploymentName,  
 endpoint: openAiEndpoint,  
 apiKey: openAiKey)  
 .Build();  
  
 // Import the custom tools into the kernel  
 \_kernel.ImportPluginFromObject(new UserManagementTools(), "UserManagementPlugin");  
  
 \_chatHistory = new ChatHistory();  
 \_chatHistory.AddSystemMessage("You are an IT support assistant. You can check user status and reset user passwords using the available tools. Respond concisely and helpfully.");  
 }  
  
 public async Task ChatWithAgent(string userMessage)  
 {  
 Console.WriteLine($"\nUser: {userMessage}");  
 \_chatHistory.AddUserMessage(userMessage);  
  
 // Enable auto-invocation of tools  
 OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new() { ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions };  
  
 var result = await \_kernel.GetChatCompletionAsync(  
 \_chatHistory,  
 openAIPromptExecutionSettings  
 );  
  
 Console.WriteLine($"Agent: {result.Content}");  
 }  
  
 public static async Task Main(string[] args)  
 {  
 // Replace with your actual Azure OpenAI credentials  
 string openAiEndpoint = Environment.GetEnvironmentVariable("AZURE\_OPENAI\_ENDPOINT") ?? "YOUR\_AZURE\_OPENAI\_ENDPOINT";  
 string openAiKey = Environment.GetEnvironmentVariable("AZURE\_OPENAI\_KEY") ?? "YOUR\_AZURE\_OPENAI\_KEY";  
 string openAiDeploymentName = Environment.GetEnvironmentVariable("AZURE\_OPENAI\_DEPLOYMENT\_NAME") ?? "YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME";  
  
 if (new[] { openAiEndpoint, openAiKey, openAiDeploymentName }.Any(string.IsNullOrWhiteSpace))  
 {  
 Console.WriteLine("Please set the environment variables for Azure OpenAI credentials.");  
 return;  
 }  
  
 ITSupoportAgent agent = new ITSupoportAgent(openAiEndpoint, openAiKey, openAiDeploymentName);  
  
 Console.WriteLine("IT Support Agent Ready. Ask for assistance (type 'exit' to quit):\n");  
  
 while (true)  
 {  
 Console.Write("You: ");  
 string userQuery = Console.ReadLine();  
  
 if (userQuery.ToLower() == "exit")  
 {  
 break;  
 }  
  
 await agent.ChatWithAgent(userQuery);  
 }  
 }  
}

**Explanation:**

1. **UserManagementTools Class:**
   * This class defines our custom tools as methods (CheckUserStatus, ResetUserPassword).
   * The [KernelFunction] attribute marks these methods as functions that the Semantic Kernel can invoke.
   * The [Description] attribute provides a natural language description of the function and its parameters. This description is crucial for the LLM to understand when and how to use the tool.
   * Inside the methods, we simulate interaction with an external IT system. In a real-world scenario, these would contain actual API calls to identity management systems (e.g., Microsoft Graph API for Azure AD).
2. **ITSupportAgent Class:**
   * **Kernel Initialization:** The Kernel is initialized with an Azure OpenAI chat completion service. This is the LLM that will power the agent’s reasoning.
   * **Tool Import:** \_kernel.ImportPluginFromObject(new UserManagementTools(), "UserManagementPlugin") registers our UserManagementTools class as a plugin with the Semantic Kernel. The methods within this class become available as tools for the agent.
   * **Chat History:** A ChatHistory object maintains the conversation context, including a system message that defines the agent’s role and capabilities.
   * **ChatWithAgent Method:**
     + The user’s message is added to the chat history.
     + OpenAIPromptExecutionSettings is configured with ToolCallBehavior.AutoInvokeKernelFunctions. This setting instructs the Semantic Kernel to automatically invoke any tools that the LLM decides are necessary to fulfill the user’s request.
     + \_kernel.GetChatCompletionAsync is called. The LLM processes the chat history, and if it determines a tool is needed, it generates a tool call. Semantic Kernel intercepts this, executes the tool, and then feeds the tool’s output back to the LLM for a final natural language response.
3. **Main Method:** This is the entry point for the console application. It retrieves Azure OpenAI credentials from environment variables and then enters a loop to continuously interact with the user.

**Before running this code:**

* **Azure OpenAI Service:** Ensure you have an Azure OpenAI service and a chat model deployed (e.g., gpt-3.5-turbo or gpt-4).
* **Environment Variables:** Set the following environment variables with your actual credentials:
  + AZURE\_OPENAI\_ENDPOINT
  + AZURE\_OPENAI\_KEY
  + AZURE\_OPENAI\_DEPLOYMENT\_NAME

This C# example provides a clear demonstration of how to build an intelligent agent that can dynamically interact with external systems (simulated here as C# methods) by leveraging Semantic Kernel’s tool integration capabilities. This pattern is highly extensible for various IT automation or business process automation scenarios.

### How to Test the C# Implementation Example

This section provides step-by-step instructions on how to set up, run, and verify the C# code example for the IT Support Agent.

#### Prerequisites

* .NET SDK (version 6.0 or higher) installed.
* An Azure OpenAI Service instance with a chat model deployed (e.g., gpt-3.5-turbo or gpt-4). You will need the endpoint, API key, and the deployment name of your chat model.

#### Setup

1. **Create a new C# Console Project:**

* dotnet new console -n ITSupportAgentApp  
  cd ITSupportAgentApp

1. **Add NuGet Packages:** Open the ITSupportAgentApp.csproj file and add the following ItemGroup:

* <ItemGroup>  
   <PackageReference Include="Microsoft.SemanticKernel" Version="1.0.0-beta10" />  
   <PackageReference Include="Microsoft.SemanticKernel.Connectors.OpenAI" Version="1.0.0-beta10" />  
  </ItemGroup>
* Then, restore the packages:
* dotnet restore

1. **Create Program.cs:** Replace the content of Program.cs with the C# code provided above in the “C# Implementation Example” section. Ensure the using statements and class definitions are correct.

#### Configuration

Set the following environment variables with your Azure OpenAI credentials. Replace the placeholder values with your actual information.

* AZURE\_OPENAI\_ENDPOINT="YOUR\_AZURE\_OPENAI\_ENDPOINT"
* AZURE\_OPENAI\_KEY="YOUR\_AZURE\_OPENAI\_KEY"
* AZURE\_OPENAI\_DEPLOYMENT\_NAME="YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME"

**Example (for PowerShell on Windows):**

$env:AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
$env:AZURE\_OPENAI\_KEY="your-openai-api-key"  
$env:AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"

**Example (for Bash/Zsh on Linux/macOS):**

export AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
export AZURE\_OPENAI\_KEY="your-openai-api-key"  
export AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"

#### Execution

Run the application from the ITSupportAgentApp directory:

dotnet run

#### Expected Results

The application will start an interactive chat session with the IT Support Agent. You can ask questions related to user status and password resets. The agent will use the simulated tools to respond.

**Example Interaction:**

IT Support Agent Ready. Ask for assistance (type 'exit' to quit):  
  
You: Check the status of JohnDoe  
[Tool Call: CheckUserStatus for JohnDoe]  
Agent: User JohnDoe is active.  
  
You: Reset password for JaneSmith  
[Tool Call: ResetUserPassword for JaneSmith]  
Agent: Password for user JaneSmith has been successfully reset.  
  
You: What about a user named Alice?  
Agent: User Alice not found.  
  
You: exit

### Python Implementation Example

This Python example mirrors the C# implementation, demonstrating an IT support agent that uses Semantic Kernel to integrate and invoke custom tools for user management. We will simulate the tools as simple Python functions.

First, ensure you have the necessary Python package installed:

pip install semantic-kernel openai

import os  
import asyncio  
from semantic\_kernel import Kernel  
from semantic\_kernel.connectors.ai.open\_ai import AzureChatCompletion  
from semantic\_kernel.functions import kernel\_function  
from semantic\_kernel.contents.chat\_history import ChatHistory  
  
# Define a simple class to represent our IT User Management Tools  
class UserManagementTools:  
 @kernel\_function(description="Checks the status of a user account (active or locked).")  
 def check\_user\_status(self, username: str) -> str:  
 print(f"[Tool Call: check\_user\_status for {username}]")  
 # Simulate interaction with an external system  
 if username.lower() == "johndoe":  
 return f"User {username} is active."  
 elif username.lower() == "janesmith":  
 return f"User {username} is locked."  
 else:  
 return f"User {username} not found."  
  
 @kernel\_function(description="Resets the password for a specified user account.")  
 def reset\_user\_password(self, username: str) -> str:  
 print(f"[Tool Call: reset\_user\_password for {username}]")  
 # Simulate interaction with an external system  
 if username.lower() == "johndoe" or username.lower() == "janesmith":  
 return f"Password for user {username} has been successfully reset."  
 else:  
 return f"Could not reset password for user {username}. User not found."  
  
class ITSupportAgent:  
 def \_\_init\_\_(self):  
 self.kernel = Kernel()  
 self.kernel.add\_service(  
 AzureChatCompletion(  
 service\_id="azure\_openai\_chat\_completion",  
 deployment\_name=os.environ.get("AZURE\_OPENAI\_DEPLOYMENT\_NAME"),  
 endpoint=os.environ.get("AZURE\_OPENAI\_ENDPOINT"),  
 api\_key=os.environ.get("AZURE\_OPENAI\_KEY"),  
 )  
 )  
  
 # Import the custom tools into the kernel  
 self.kernel.import\_plugin\_from\_object(UserManagementTools(), "UserManagementPlugin")  
  
 self.chat\_history = ChatHistory()  
 self.chat\_history.add\_system\_message("You are an IT support assistant. You can check user status and reset user passwords using the available tools. Respond concisely and helpfully.")  
  
 async def chat\_with\_agent(self, user\_message: str):  
 print(f"\nUser: {user\_message}")  
 self.chat\_history.add\_user\_message(user\_message)  
  
 # Enable auto-invocation of tools  
 # In Semantic Kernel for Python, tool invocation is often handled implicitly  
 # when the model suggests a tool call and the kernel is configured to execute it.  
 # For chat completions, the model's response will include tool calls if it decides to use them.  
 # The kernel's `invoke` or `get\_chat\_message\_content` methods will handle the execution.  
  
 result = await self.kernel.invoke(  
 self.kernel.plugins["UserManagementPlugin"],  
 self.chat\_history,  
 settings=self.kernel.get\_service("azure\_openai\_chat\_completion").get\_prompt\_execution\_settings(tool\_choice="auto")  
 )  
  
 # The result from invoke might be a tool call or a direct response.  
 # For chat scenarios, we typically get the chat completion directly.  
 # Let's use the chat completion method for simplicity in this chat-based agent.  
 chat\_completion\_service = self.kernel.get\_service("azure\_openai\_chat\_completion")  
 response = await chat\_completion\_service.get\_chat\_message\_content(  
 self.chat\_history,  
 settings=chat\_completion\_service.get\_prompt\_execution\_settings(tool\_choice="auto")  
 )  
  
 print(f"Agent: {response.content}")  
  
async def main():  
 # Replace with your actual Azure OpenAI credentials  
 # It's recommended to set these as environment variables  
 required\_env\_vars = [  
 "AZURE\_OPENAI\_ENDPOINT",  
 "AZURE\_OPENAI\_KEY",  
 "AZURE\_OPENAI\_DEPLOYMENT\_NAME",  
 ]  
  
 for var in required\_env\_vars:  
 if not os.environ.get(var):  
 print(f"Please set the environment variable: {var}")  
 exit(1)  
  
 agent = ITSupportAgent()  
  
 print("IT Support Agent Ready. Ask for assistance (type 'exit' to quit):\n")  
  
 while True:  
 user\_query = input("You: ")  
  
 if user\_query.lower() == "exit":  
 break  
  
 await agent.chat\_with\_agent(user\_query)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

**Explanation:**

1. **UserManagementTools Class:**
   * This class defines our custom tools as methods (check\_user\_status, reset\_user\_password).
   * The @kernel\_function decorator marks these methods as functions that the Semantic Kernel can invoke. The description argument provides a natural language description of the function, which is crucial for the LLM to understand its purpose.
   * Inside the methods, we simulate interaction with an external IT system. In a real-world scenario, these would contain actual API calls to identity management systems (e.g., Microsoft Graph API for Azure AD).
2. **ITSupportAgent Class:**
   * **Kernel Initialization:** The Kernel is initialized with an Azure OpenAI chat completion service, which serves as the LLM for the agent’s reasoning.
   * **Tool Import:** self.kernel.import\_plugin\_from\_object(UserManagementTools(), "UserManagementPlugin") registers our UserManagementTools class as a plugin with the Semantic Kernel. The methods within this class become available as tools for the agent.
   * **Chat History:** A ChatHistory object maintains the conversation context, including a system message that defines the agent’s role and capabilities.
   * **chat\_with\_agent Method:**
     + The user’s message is added to the chat history.
     + chat\_completion\_service.get\_chat\_message\_content is called with tool\_choice="auto". This setting allows the LLM to decide whether to call a tool based on the user’s input. If the LLM determines a tool is needed, Semantic Kernel intercepts this, executes the tool, and then feeds the tool’s output back to the LLM for a final natural language response.
3. **main Function:** This is the asynchronous entry point for the script. It retrieves Azure OpenAI credentials from environment variables and then enters a loop to continuously interact with the user.

**Before running this code:**

* **Azure OpenAI Service:** Ensure you have an Azure OpenAI service and a chat model deployed (e.g., gpt-3.5-turbo or gpt-4).
* **Environment Variables:** Set the following environment variables with your actual credentials:
  + AZURE\_OPENAI\_ENDPOINT
  + AZURE\_OPENAI\_KEY
  + AZURE\_OPENAI\_DEPLOYMENT\_NAME

This Python example provides a clear demonstration of how to build an intelligent agent that can dynamically interact with external systems (simulated here as Python functions) by leveraging Semantic Kernel’s tool integration capabilities. This pattern is highly extensible for various IT automation or business process automation scenarios.

### How to Test the Python Implementation Example

This section provides step-by-step instructions on how to set up, run, and verify the Python code example for the IT Support Agent.

#### Prerequisites

* Python 3.8 or higher installed.
* An Azure OpenAI Service instance with a chat model deployed (e.g., gpt-3.5-turbo or gpt-4). You will need the endpoint, API key, and the deployment name of your chat model.

#### Setup

1. **Create a new Python file:** Create a file named it\_support\_agent.py.
2. **Install required packages:**

* pip install semantic-kernel openai

1. **Add the Python code:** Copy the Python code provided above in the “Python Implementation Example” section into it\_support\_agent.py.

#### Configuration

Set the following environment variables with your Azure OpenAI credentials. Replace the placeholder values with your actual information.

* AZURE\_OPENAI\_ENDPOINT="YOUR\_AZURE\_OPENAI\_ENDPOINT"
* AZURE\_OPENAI\_KEY="YOUR\_AZURE\_OPENAI\_KEY"
* AZURE\_OPENAI\_DEPLOYMENT\_NAME="YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME"

**Example (for Bash/Zsh on Linux/macOS):**

export AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
export AZURE\_OPENAI\_KEY="your-openai-api-key"  
export AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"

**Example (for PowerShell on Windows):**

$env:AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
$env:AZURE\_OPENAI\_KEY="your-openai-api-key"  
$env:AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"

#### Execution

Run the Python script:

python it\_support\_agent.py

#### Expected Results

The application will start an interactive chat session with the IT Support Agent. You can ask questions related to user status and password resets. The agent will use the simulated tools to respond.

**Example Interaction:**

IT Support Agent Ready. Ask for assistance (type 'exit' to quit):  
  
You: Check the status of JohnDoe  
[Tool Call: check\_user\_status for JohnDoe]  
Agent: User JohnDoe is active.  
  
You: Reset password for JaneSmith  
[Tool Call: reset\_user\_password for JaneSmith]  
Agent: Password for user JaneSmith has been successfully reset.  
  
You: What about a user named Alice?  
Agent: User Alice not found.  
  
You: exit