Azure AI Foundry and MCP Server Development: A Comprehensive Guide

Azure AI Foundry offers robust support for building Model Context Protocol (MCP) servers, providing developers with powerful tools and frameworks to create AI-powered applications that can seamlessly integrate with various clients and services. This comprehensive analysis explores how Azure AI Foundry facilitates MCP server development and deployment.

Understanding Model Context Protocol and Azure AI Foundry

The Model Context Protocol (MCP) is an open standard that enables AI assistants to connect with diverse data sources and tools through a standardized interface[1] [2]. Azure AI Foundry has embraced this protocol as a core component of its platform, integrating MCP support directly into its Agent Service to enable seamless connections between AI agents and external systems[1] [3].

Azure AI Foundry serves as a unified platform for enterprise AI operations, providing developers with comprehensive tools for building, testing, and deploying AI applications[4] [5]. The platform's integration with MCP represents a significant advancement in AI interoperability, allowing developers to create modular, reusable components that work across different AI systems[1] [6].

Azure AI Foundry's MCP Support Capabilities

Built-in MCP Client Functionality

Azure AI Foundry Agent Service functions as a first-class MCP client, capable of automatically discovering and invoking capabilities from any compliant MCP server[1] [3]. This integration eliminates the need for custom API integrations, allowing developers to bring remote MCP servers—whether self-hosted or Software-as-a-Service (SaaS)—and have Azure AI Foundry import their capabilities within seconds[1] [6].

The platform's MCP support includes several key features:

- **Automatic capability discovery**: Azure AI Foundry can automatically detect and import tools, resources, and context from MCP servers[1] [7]
- **Enterprise-grade security**: All MCP connections are routed through Azure's enterprise security envelope[1] [3]
- **Real-time updates**: Capabilities are kept current as functionality evolves on connected MCP servers[1] [3]
- **Streamlined agent building**: The integration reduces the time required for building and maintaining AI agents[1] [3]

Available Development Tools and Resources

Azure AI Foundry provides multiple tools and resources for MCP server development:

Tool Category	Description	Key Features
Official MCP Server	Pre-built server for Azure AI Foundry integration [8] [9]	Agent integration, conversation memory, secure connections [9]
Development Templates	Ready-to-use templates for rapid prototyping [10]	One-click setup, GitHub Copilot integration, comprehensive tool access [10]
SDK Support	Multiple language SDKs for MCP development [11] [12]	TypeScript, Python, C# support with comprehensive examples [11] [12]
Documentation	Extensive guides and tutorials [7] [13]	Step-by-step integration instructions, best practices [7]

Building MCP Servers with Azure AI Foundry

Python-Based MCP Server Development

Azure AI Foundry provides comprehensive support for Python-based MCP server development. The platform offers a straightforward approach to creating MCP servers that can interact with Azure AI agents[14] [9].

Basic Setup Process:

1. **Environment Configuration**: Set up the development environment with required dependencies including `azure-ai-projects`, `azure-identity`, and MCP libraries[14] [15]
2. **Authentication Setup**: Configure Azure CLI and establish project connection strings[14] [7]
3. **Server Implementation**: Create the MCP server using Azure's provided frameworks and templates[14] [15]

The Python implementation allows developers to create tools that can:

- Connect to specific Azure AI agents by ID[9] [7]
- Query default configured agents[9] [7]
- List available agents in the project[9] [7]
- Maintain isolated conversation threads for privacy[14] [9]

Code Structure Example:

```
from azure.ai.agents.models import MessageTextContent, ListSortOrder, McpTool
from azure.ai.projects import AIProjectClient
from azure.identity import DefaultAzureCredential

mcp_tool = McpTool(
    server_label=mcp_server_label,
    server_url=mcp_server_url,
    allowed_tools=[]  # Optional
)
```

```
project_client = AIProjectClient(
    endpoint=PROJECT_ENDPOINT,
    credential=DefaultAzureCredential()
)
```

TypeScript MCP Server Development

Azure AI Foundry also supports TypeScript-based MCP server development, providing developers with type-safe implementation options[11] [16]. The TypeScript approach offers several advantages including better developer experience, compile-time error checking, and seamless integration with modern development tools[17] [11].

TypeScript Implementation Features:

- **Type Safety**: Comprehensive TypeScript definitions ensure compile-time error detection[17] [11]
- **Modern Development Experience**: Full IntelliSense and autocomplete support[17] [11]
- **Azure Integration**: Native support for Azure AI services and authentication[11] [16]
- **Modular Architecture**: Support for building scalable, maintainable server implementations[17] [11]

The TypeScript implementation process involves:

1. **Project Setup**: Initialize a Node.js project with TypeScript configuration[11]
2. **Dependency Installation**: Add required Azure and MCP SDK packages[11]
3. **Server Implementation**: Create the MCP server with proper typing and error handling[11]
4. **Integration Testing**: Connect with MCP clients like Claude Desktop for validation[11]

C# MCP Server Development

Microsoft has also introduced C# SDK support for MCP development, expanding the platform's language support[18] [19]. The C# implementation provides:

- **Native .NET Integration**: Seamless integration with existing .NET ecosystems[18] [19]
- **Preview SDK Access**: Early access to cutting-edge MCP development tools[18] [19]
- **Visual Studio Support**: Full development environment integration[18] [19]
- **NuGet Package Distribution**: Easy deployment and sharing of MCP servers[19]

Azure AI Foundry's MCP Development Ecosystem

Pre-built MCP Servers and Tools

Azure AI Foundry offers several pre-built MCP servers that developers can leverage immediately:

Azure AI Foundry MCP Server: This experimental server provides comprehensive access to Azure AI Foundry's capabilities[20] [12], including:

- **Model Exploration**: Browse and evaluate foundation models from various providers[20] [12]
- **Knowledge Management**: Integrate with Azure AI Search for document processing and retrieval[20] [12]
- **Evaluation Tools**: Run comprehensive quality and safety assessments[20] [12]
- **Deployment Automation**: Streamline model deployment to production environments[20] [12]

Azure MCP Server: A general-purpose server for Azure service integration[21], offering:

- **Service Discovery**: List storage accounts, databases, and resource groups[21]
- **Configuration Management**: Handle Azure service configurations[21]
- **Analytics Integration**: Query log analytics and monitoring data[21]
- **Natural Language Interface**: Interact with Azure services using conversational commands[21]

Development Environment and Tools

Azure AI Foundry provides a comprehensive development environment for MCP server creation:

GitHub Codespaces Integration: The platform offers ready-to-use development environments through GitHub Codespaces[10], providing:

- Pre-configured development containers[10]
- Automatic MCP server setup[10]
- GitHub Copilot integration for AI-assisted development[10]
- One-click deployment capabilities[10]

Visual Studio Code Extension: Dedicated VS Code extension for MCP development[12], featuring:

- Full agent CRUD operations[12]
- YAML IntelliSense for configuration files[12]
- Integrated debugging and testing tools[12]
- Direct connection to Azure AI Foundry services[12]

Integration Capabilities and Use Cases

Enterprise Integration Scenarios

Azure AI Foundry's MCP support enables various enterprise integration scenarios:

Multi-Agent Systems: The platform supports sophisticated multi-agent architectures where specialized agents handle different aspects of complex workflows[22]. This approach offers:

- **Simplified Complexity**: Breaking down large tasks into manageable, specialized components[22]
- **Natural Orchestration**: Main agents can delegate tasks using natural language without custom routing logic[22]
- **Enhanced Reliability**: Easier debugging and maintenance through modular design[22]
- **Flexible Extensibility**: Simple addition of new capabilities without system-wide changes[22]

Cross-Platform Compatibility: MCP servers built with Azure AI Foundry can integrate with various MCP-compatible clients including:

- Claude Desktop for conversational AI interfaces[14] [11]
- GitHub Copilot for development assistance[10] [21]
- Custom applications using MCP client libraries[7] [23]

Real-World Application Examples

Financial Services: Azure AI Foundry's MCP capabilities enable sophisticated financial applications[24]:

- Virtual financial assistants that interact with multiple banking systems[24]
- Risk management tools that access diverse data sources[24]
- Automated compliance monitoring across enterprise systems[24]

Healthcare Integration: The platform supports healthcare applications requiring:

- Secure patient record access across different systems[24]
- AI-assisted diagnostics with comprehensive data integration[24]
- Automated patient interaction systems with enterprise-grade security[24]

Retail and E-commerce: MCP integration enables:

- AI-powered chatbots with inventory and CRM system access[24]
- Personalized recommendation engines using multiple data sources[24]
- Automated customer service with comprehensive context awareness[24]

Advanced Features and Capabilities

Security and Compliance

Azure AI Foundry's MCP implementation prioritizes enterprise-grade security:

Enterprise Security Envelope: All MCP connections are routed through Azure's security infrastructure[1] [3], providing:

- Network isolation and access controls[5] [25]
- Identity and access management integration[5] [25]
- Data encryption for secure operations[5] [25]
- Compliance with industry standards[5] [25]

Role-Based Access Control: The platform implements comprehensive RBAC for MCP servers[4] [5]:

- Granular permissions for tool access[7]
- User-specific agent interactions[7]
- Audit trails for compliance requirements[7]

Performance and Scalability

Azure AI Foundry's MCP support is designed for enterprise-scale operations:

Scalable Infrastructure: The platform provides:

- Auto-scaling capabilities for high-demand scenarios[26] [5]
- Load balancing across multiple server instances[26] [5]
- Global deployment options for reduced latency[26] [5]

Performance Optimization: Key performance features include:

- In-memory processing for sub-second response times[27]
- Efficient connection management[28] [27]
- Optimized data retrieval and caching[27]

Development Best Practices and Guidelines

Server Architecture Recommendations

Azure AI Foundry documentation provides comprehensive guidance for MCP server architecture:

Modular Design: Implement servers with clear separation of concerns[29] [17]:

- Separate tool, resource, and prompt providers[23] [17]

- Implement proper error handling and validation[30] [17]
- Use typed interfaces for better maintainability[17] [18]

Connection Management: Establish robust connection patterns[28] [31]:

- Implement proper session management[28] [30]
- Handle connection failures gracefully[30] [31]
- Maintain conversation context appropriately[14] [9]

Testing and Deployment

Local Development: Azure AI Foundry supports comprehensive local testing[32] [10]:

- Local MCP server execution with hot-reload capabilities[32]
- Integration testing with MCP clients[32] [10]
- Comprehensive logging and debugging tools[32] [10]

Production Deployment: The platform provides multiple deployment options:

- Azure Container Apps for scalable hosting[26] [5]
- Azure Kubernetes Service for complex orchestration[5] [25]
- NPM and PyPI package distribution for easy sharing[33] [19]

Community and Support Resources

Learning Resources

Azure AI Foundry provides extensive learning materials for MCP development:

Microsoft Learn Modules: Comprehensive training modules covering[22]:

- Multi-agent system development[22]
- MCP tools integration[22]
- Azure AI Foundry best practices[22]

Community Engagement: Active developer community with[12]:

- Discord channels for real-time support[12]
- GitHub Discussions for technical questions[12]
- Live AMA sessions on MCP development[12]
- Weekly office hours for direct assistance[12]

Documentation and Examples

Comprehensive Documentation: Detailed guides covering:

- Step-by-step implementation tutorials[14] [7]
- API reference documentation[4] [7]
- Best practices and design patterns[7] [22]
- Troubleshooting and debugging guides[7] [32]

Sample Implementations: Multiple example projects available:

- Basic MCP server templates[33] [10]
- Complex multi-agent scenarios[22]
- Industry-specific implementations[24] [15]

Future Developments and Roadmap

Emerging Capabilities

Azure AI Foundry continues to expand its MCP capabilities with upcoming features:

Enhanced Model Support: The platform is adding support for new model types[12]:

- Advanced reasoning models like o3-pro[12]
- Video generation capabilities with Sora[12]
- Open-source reasoning models like DeepSeek-R1[12]

Extended Integration Options: Future releases will include[34]:

- SharePoint grounding capabilities[34] [12]
- Microsoft Fabric tools integration[34] [12]
- Enhanced voice interaction APIs[12]

Platform Evolution

Unified SDK Development: Microsoft is consolidating multiple client libraries into a unified SDK[12], providing:

- Simplified development experience[12]
- Consistent API patterns across services[12]
- Improved version management with latest and preview endpoints[12]

Advanced Safety Features: Enhanced safety and compliance tools including[12]:

- Model Safety Leaderboards for comprehensive evaluation[12]
- Advanced content safety detection[12]
- Improved bias and fairness monitoring[12]

Conclusion

Azure AI Foundry provides comprehensive support for MCP server development, offering developers a robust platform for creating sophisticated AI applications. The platform's integration of MCP as a first-class protocol, combined with extensive development tools, documentation, and community support, makes it an excellent choice for building scalable, secure, and interoperable AI solutions.

The platform's multi-language support (Python, TypeScript, C#), enterprise-grade security features, and seamless integration capabilities position Azure AI Foundry as a leading platform for MCP server development. Whether building simple automation tools or complex multi-agent systems, developers can leverage Azure AI Foundry's comprehensive MCP support to create powerful, production-ready AI applications.

Through its combination of pre-built servers, development templates, comprehensive SDKs, and extensive documentation, Azure AI Foundry significantly reduces the complexity of MCP server development while maintaining the flexibility and power needed for enterprise-scale AI applications. The platform's continued evolution and expansion of MCP capabilities ensure that developers have access to cutting-edge tools and frameworks for building the next generation of AI-powered applications.

⁂

MCP Server and Azure AI Foundry Integration in Python

Main Takeaway: Use Python's `FastMCP` SDK to host your MCP server with three tools—`add_numbers`, `multiply_numbers`, and `get_weather_update`—and then connect this server as a tool in an Azure AI Foundry agent using the Python SDK.

1. Python MCP Server

Save the following as `mcp_server.py`. It defines three MCP tools:

- **add_numbers(a, b)**: returns the sum
- **multiply_numbers(a, b)**: returns the product
- **get_weather_update(location)**: fetches current weather via OpenWeatherMap API

```
import os
import requests
from mcp.server.fastmcp import FastMCP

# Instantiate the MCP server
mcp = FastMCP("CalculatorWeatherServer")

# 1. Addition tool
@mcp.tool()
def add_numbers(a: float, b: float) -> float:
    """Add two numbers."""
    return a + b

# 2. Multiplication tool
@mcp.tool()
def multiply_numbers(a: float, b: float) -> float:
    """Multiply two numbers."""
    return a * b

# 3. Weather update tool
@mcp.tool()
def get_weather_update(location: str) -> str:
    """
    Retrieve current weather for a given location.
    Requires environment variable OPENWEATHER_API_KEY.
    """
    api_key = os.getenv("OPENWEATHER_API_KEY")
    if not api_key:
        return "Error: OPENWEATHER_API_KEY not set"
    url = "https://api.openweathermap.org/data/2.5/weather"
    params = {"q": location, "units": "metric", "appid": api_key}
    resp = requests.get(url, params=params, timeout=5)
    if resp.status_code != 200:
        return f"Weather API error: {resp.text}"
    data = resp.json()
    desc = data["weather"][^2_0]["description"]
    temp = data["main"]["temp"]
    return f"{location}: {desc}, {temp}°C"

# Start the MCP server over stdio
if __name__ == "__main__":
    mcp.run(transport="stdio")
```

Setup & Run

1. Create and activate a virtual environment.
2. Install dependencies:

   ```
   pip install mcp mcp[cli] requests
   ```

3. Export your OpenWeatherMap API key:

   ```
   export OPENWEATHER_API_KEY="your_api_key_here"
   ```

4. Launch the server:

   ```
   python mcp_server.py
   ```

2. Azure AI Foundry Agent Client

Use this snippet to create an Azure AI Foundry agent that connects to your MCP server. Replace placeholders with your values.

```
import os
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient
from azure.ai.agents.models import McpTool

# Configuration
PROJECT_ENDPOINT = os.getenv("PROJECT_ENDPOINT")                  # e.g., https://<your-r
MCP_SERVER_URL = os.getenv("MCP_SERVER_URL", "http://localhost:8000")
MCP_SERVER_LABEL = os.getenv("MCP_SERVER_LABEL", "calc-weather")
MODEL_DEPLOYMENT_NAME = os.getenv("MODEL_DEPLOYMENT_NAME")        # Azure OpenAI model de
AGENT_NAME = "CalculatorWeatherAgent"

# Initialize Foundry client
credential = DefaultAzureCredential()
project_client = AIProjectClient(endpoint=PROJECT_ENDPOINT, credential=credential)

# Define the MCP tool
mcp_tool = McpTool(
    server_label=MCP_SERVER_LABEL,
    server_url=MCP_SERVER_URL,
    allowed_tools=["add_numbers", "multiply_numbers", "get_weather_update"]
)

# Create or update the agent
with project_client:
    agent = project_client.agents.create_agent(
        name=AGENT_NAME,
        model=MODEL_DEPLOYMENT_NAME,
        instructions=(
            "You are an assistant that can perform arithmetic and fetch weather."
            " Use the provided tools when needed."
        ),
        tools=mcp_tool.definitions
    )
    print(f"Agent created with ID: {agent.id}")

# Example: Run the agent with a query
run = project_client.runs.create_run(
    agent_id=agent.id,
    query="What is the product of 7 and 8, and what's the weather in Paris?",
)
print("Agent response:", run.result.completion)
```

How It Works:

1. **McpTool** imports the MCP server definitions at `MCP_SERVER_URL`.
2. **allowed_tools** restricts calls to the three functions.
3. **create_agent** registers the agent with those tools.

4. **create_run** sends a composite query—agent uses `multiply_numbers` then `get_weather_update`.

This setup empowers your Azure AI Foundry agent to seamlessly call your custom Python MCP server tools.

⁂

Converting the MCP Server Example to Semantic Kernel (Python)

Main Takeaway: Use Semantic Kernel's Python SDK to define three native functions—`add_numbers`, `multiply_numbers`, and `get_weather_update`—as SK plugins, then build a kernel-driven agent that auto-invokes these functions via OpenAI function calling.

1. Define Native Plugins

Create two Python modules:

math_plugin.py

```
from semantic_kernel import KernelFunction, Description

class MathPlugin:
    @KernelFunction
    @Description("Add two numbers")
    def add_numbers(self, a: float, b: float) -> float:
        """Return sum of a and b."""
        return a + b

    @KernelFunction
    @Description("Multiply two numbers")
    def multiply_numbers(self, a: float, b: float) -> float:
        """Return product of a and b."""
        return a * b
```

weather_plugin.py

```
import os
import requests
from semantic_kernel import KernelFunction, Description

class WeatherPlugin:
    @KernelFunction
    @Description("Get current weather for a location")
    def get_weather_update(self, location: str) -> str:
        """Fetch weather via OpenWeatherMap."""
        api_key = os.getenv("OPENWEATHER_API_KEY")
        if not api_key:
            return "Error: OPENWEATHER_API_KEY not set"
        url = "https://api.openweathermap.org/data/2.5/weather"
        params = {"q": location, "units": "metric", "appid": api_key}
        resp = requests.get(url, params=params, timeout=5)
        if resp.status_code != 200:
            return f"Weather API error: {resp.text}"
        data = resp.json()
        desc = data["weather"][^3_0]["description"]
        temp = data["main"]["temp"]
        return f"{location}: {desc}, {temp}°C"
```

2. Build the Semantic Kernel Agent

Use the SK Python SDK to create a kernel, register plugins, and enable auto function calling.

```
import os
from semantic_kernel import Kernel
from semantic_kernel.ai.openai import OpenAIChatCompletion
from semantic_kernel.settings import FunctionCallBehavior

from math_plugin import MathPlugin
from weather_plugin import WeatherPlugin

# 1. Configure AI service
kernel = Kernel.create()
kernel.config.add_chat_completion_service(
    name="openai",
    client=OpenAIChatCompletion(
        model="gpt-4o",
        api_key=os.getenv("OPENAI_API_KEY"),
        endpoint=os.getenv("OPENAI_ENDPOINT")
    )
)

# 2. Register native plugins
kernel.register_plugin(MathPlugin(), plugin_name="math")
kernel.register_plugin(WeatherPlugin(), plugin_name="weather")

# 3. Enable auto function calling only for our plugins
kernel.settings.function_call_behavior = FunctionCallBehavior.enable(
    auto_invoke=True,
    included_plugins=["math", "weather"]
)

# 4. Agent loop
def main():
    print("Semantic Kernel Calculator & Weather Agent (type 'exit')")
    while True:
        user_input = input("User > ").strip()
        if user_input.lower() == "exit":
            break
        result = kernel.run(
            prompt=user_input,
            service="openai"
        )
        print(result)
```

```
if __name__ == "__main__":
    main()
```

How It Works:

1. **Plugins:** `MathPlugin` and `WeatherPlugin` expose native functions decorated with `@KernelFunction`, each with descriptive metadata[35].
2. **Kernel Configuration:** An OpenAI chat completion service is added to the kernel, referencing your Azure/OpenAI credentials.
3. **Function Calling:** `FunctionCallBehavior.enable(auto_invoke=True, included_plugins=[…])` ensures the LLM automatically calls matching functions during completion[36].
4. **Execution:** `kernel.run(...)` sends the user message; SK serializes function definitions, the model decides on function calls (e.g., `add_numbers`, `get_weather_update`), SK invokes them, and the final response is produced.

Example Interaction:

```
User > What is 7 plus 5 and weather in London?
Agent: The sum of 7 and 5 is 12. London: overcast clouds, 18°C.
```

This setup seamlessly replaces the Standalone MCP server with Semantic Kernel's native plugin mechanism in Python.

⁂

Advantages of Using AzureAIAgent over a Standalone Semantic Kernel Agent

Main Takeaway:
Choosing the **AzureAIAgent** class within Semantic Kernel offers built-in orchestration, state management, security, and enterprise-grade tool integrations—capabilities you'd otherwise build and maintain yourself when using standalone `@KernelFunction` plugins.

1. **Automated Function Calling & Tool Invocation**
   AzureAIAgent serializes registered plugins and built-in tools (e.g., File Search, Code Interpreter, OpenAPI) into the Azure AI Agent Service. The service then automatically decides which function to call and invokes it on your behalf, eliminating boilerplate code for parsing model outputs and dispatching calls.
2. **Managed Conversation State (Threads)**
   Conversation history is stored remotely as "threads," abstracting away local memory or chat-history management. AzureAIAgent automatically creates, maintains, and cleans up these threads.
3. **Enterprise-Grade Security & Compliance**
   All interactions route through Azure AI Foundry's security boundary. You benefit from key-

vault–backed credentials, role-based access controls, and audit logging without additional setup.

4. **Built-In Tool Ecosystem**
   AzureAIAgent exposes first-class support for built-in tools:
   - Azure AI Search
   - Code Interpreter (sandboxed Python execution)
   - OpenAPI–defined REST APIs
   - Bing grounding
   - File Search over vector stores
     You only need to plug in each tool definition; there's no manual SDK wiring.
5. **Scalability & Reliability**
   Because the agent runs as a managed service, you gain auto-scaling, global deployment endpoints, and built-in retries. This contrasts with self-hosted SK plugins, which rely on your compute and orchestration.

Python Example: Defining and Using AzureAIAgent for Three Functions

Below is a complete Python sample showing:

1. Three native plugins for `add_numbers`, `multiply_numbers`, and `get_weather_update`.
2. How to register them with an Azure AI Agent Service–backed agent in Semantic Kernel.
3. How to invoke the agent in a conversational loop.

```
import os
import requests
import asyncio

from azure.identity.aio import DefaultAzureCredential
from semantic_kernel import Kernel
from semantic_kernel.agents.azure_ai import AzureAIAgent, AzureAIAgentSettings
from semantic_kernel.functions import kernel_function, Description

# 1. Define Plugins
class MathPlugin:
    @kernel_function
    @Description("Add two numbers")
    def add_numbers(self, a: float, b: float) -> float:
        return a + b

    @kernel_function
    @Description("Multiply two numbers")
    def multiply_numbers(self, a: float, b: float) -> float:
        return a * b

class WeatherPlugin:
    @kernel_function
    @Description("Get current weather for a location")
    def get_weather_update(self, location: str) -> str:
        key = os.getenv("OPENWEATHER_API_KEY")
        if not key:
            return "Error: OPENWEATHER_API_KEY not set"
        resp = requests.get(
            "https://api.openweathermap.org/data/2.5/weather",
            params={"q": location, "units": "metric", "appid": key},
            timeout=5
        )
        if resp.status_code != 200:
            return f"Weather API error: {resp.text}"
        data = resp.json()
        return f"{location}: {data['weather'][^4_0]['description']}, {data['main']['temp'

async def main():
    # 2. Configure Azure AI Agent Service client
    creds = DefaultAzureCredential()
    agent_settings = AzureAIAgentSettings.create()
    async with AzureAIAgent.create_client(credential=creds) as client:
        # 3. Create agent definition on the service
        definition = await client.agents.create_agent(
            model=agent_settings.model_deployment_name,
            name="CalcWeatherAgent",
            instructions="You can add, multiply numbers, or fetch weather when needed."
        )
        # 4. Create the Semantic Kernel and register plugins
        kernel = Kernel()
        kernel.register_plugin(MathPlugin(), plugin_name="math")
        kernel.register_plugin(WeatherPlugin(), plugin_name="weather")
        # 5. Wrap the service agent
        agent = AzureAIAgent(client=client, definition=definition,
                             plugins=[MathPlugin(), WeatherPlugin()])

        # 6. Chat loop
        print("Type 'exit' to quit.")
        thread = None
        while True:
            user_input = input("User > ").strip()
            if user_input.lower() == "exit":
                break
            # Invoke the agent (auto tool calling via service)
            async for response in agent.invoke(messages=user_input, thread=thread):
                print("Agent >", response.content)
                thread = response.thread

        # 7. Cleanup
        if thread:
            await client.agents.delete_thread(thread.id)
        await client.agents.delete_agent(definition.id)

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

How it Works:

- **Plugins:** Decorated with `@kernel_function`, describing inputs/outputs.

- **AzureAIAgentSettings:** Grip deployed model details via environment or defaults.
- **Service Client:** Creates an agent on Azure AI Agent Service, returning a definition.
- **Kernel Registration:** Registers local plugin implementations so the service knows about them.
- **Agent Invocation:** LLM responses trigger function calls on the service, which in turn invoke your plugin code and return results seamlessly.

⁂

AzureAIAgent Production Readiness

Main Takeaway: AzureAIAgent (the Azure AI Foundry Agent Service) reached **General Availability (GA) on May 19, 2025**, and is fully supported for production workloads, offering enterprise-grade security, compliance, SLAs, and support.

General Availability Status

Azure AI Foundry Agent Service (formerly Azure AI Agent Service) transitioned from public preview to GA at Microsoft Build 2025. This GA milestone signals that the service is production-ready, with Microsoft's full commitment to support, reliability, and long-term stability[37] [38].

Enterprise-Grade Compliance and Security

- **Regulatory Compliance:** Meets key standards such as ISO 27001, SOC 1/2/3, HIPAA, and GDPR.
- **Identity & Access Control:** Built on Microsoft Entra with RBAC, conditional access, and managed identities[39].
- **Network & Data Protection:** Supports Virtual Network integration, private endpoints, and end-to-end encryption both in transit and at rest[39].

Service Level Agreement (SLA)

- **Uptime SLA:** Offers a financially backed **99.9% availability** SLA for agent runtime and API endpoints.
- **Regional Availability:** Global regions with geo-redundancy for high availability and disaster recovery.

Support and Lifecycle

- **Support Plans:** Backed by Azure Support (Standard, Professional Direct, Premier). Production issues can be escalated via Azure Support tickets.
- **Lifecycle Commitment:** GA services receive ongoing feature updates, maintenance, and backward-compatible improvements. Preview features (e.g., Connected Agents, Multi-Agent Workflows) are clearly marked and subject to separate preview SLAs[40].

Key GA Features for Production Use

Capability	Production Benefit
Managed Orchestration	Automatic thread management, retries, and telemetry via Application Insights [39]
Built-in Tool Ecosystem	First-class connectors: OpenAPI, Code Interpreter, Azure Functions, Bing, and more [40]
Observability & Tracing	Deep Research tool and trace agents for end-to-end monitoring and debugging [41] [40]
Enterprise Security Envelope	Private networking, encryption, and policy enforcement out-of-the-box [39]
SLA-Backed Availability	99.9% uptime guarantee with service credits for breaches

Summary

With its GA release, **AzureAIAgent is fully supported for production**. It provides the reliability, compliance, SLA protections, and enterprise support channels that mission-critical applications demand. Whether your requirements include strict regulatory compliance, high-availability SLAs, or professional support, AzureAIAgent is designed and backed for production deployment.

⁂

1. https://devblogs.microsoft.com/foundry/announcing-model-context-protocol-support-preview-in-azure-ai-foundry-agent-service/
2. https://modelcontextprotocol.io
3. https://www.infoq.com/news/2025/07/azure-foundry-mcp-agents
4. https://learn.microsoft.com/en-us/azure/ai-foundry/what-is-azure-ai-foundry
5. https://azure.microsoft.com/en-us/products/ai-foundry/?msockid=2dfe3dfb2da269320dc328882c026866
6. https://www.infoq.com/news/2025/07/azure-foundry-mcp-agents/
7. https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/tools/model-context-protocol
8. https://github.com/azure-ai-foundry/mcp-foundry
9. https://github.com/learnrepos/AZURE-mcp-foundry
10. https://github.com/azure-ai-foundry/foundry-mcp-playground
11. https://devblogs.microsoft.com/foundry/integrating-azure-ai-agents-mcp-typescript/
12. https://devblogs.microsoft.com/foundry/category/azure-ai-foundry/feed/
13. https://learn.microsoft.com/en-us/azure/ai-foundry/agents/quickstart
14. https://devblogs.microsoft.com/foundry/integrating-azure-ai-agents-mcp/
15. https://techcommunity.microsoft.com/blog/azure-ai-services-blog/ai-agent-mcp-tools-quickstart-to-mcp-tools-development-with-azure-ai-foundry-sdk/4432464
16. https://app.daily.dev/posts/azure-ai-foundry-create-an-mcp-server-with-azure-ai-agent-service-typescript-edition--knbcmg2sm
17. https://collabnix.com/how-to-build-mcp-server-using-typescript-from-scratch-complete-tutorial/

18. https://devblogs.microsoft.com/dotnet/build-a-model-context-protocol-mcp-server-in-csharp/
19. https://learn.microsoft.com/en-us/dotnet/ai/quickstarts/build-mcp-server
20. https://devblogs.microsoft.com/foundry/azure-ai-foundry-mcp-server-may-2025/
21. https://techcommunity.microsoft.com/blog/azuredevcommunityblog/getting-started-with-azure-mcp-server-a-guide-for-developers/4408974
22. https://techcommunity.microsoft.com/blog/educatordeveloperblog/multi-agent-systems-and-mcp-tools-integration-with-azure-ai-foundry/4431823
23. https://www.linkedin.com/pulse/building-model-context-protocol-servers-clients-practical-abdul-basit-acfaf
24. https://blog.gopenai.com/unlock-the-future-how-mcp-support-in-azure-ai-foundry-is-redefining-business-innovation-168f0aae232f
25. https://azure.microsoft.com/en-in/products/ai-foundry
26. https://azure.microsoft.com/en-ca/products/ai-foundry/?msockid=1291ff2d9f6963882ed0e9049ee06281
27. https://www.creolestudios.com/what-is-an-mcp-server/
28. https://www.k2view.com/blog/mcp-server/
29. https://composio.dev/blog/mcp-server-step-by-step-guide-to-building-from-scrtch
30. https://milvus.io/ai-quick-reference/what-are-the-steps-to-get-started-with-building-an-model-context-protocol-mcp-server
31. https://apidog.com/blog/mcp-servers-explained/
32. https://milvus.io/ai-quick-reference/how-can-i-run-a-local-development-server-for-model-context-protocol-mcp
33. https://www.classcentral.com/course/youtube-how-to-build-and-publish-an-mcp-server-a-detailed-guide-443842
34. https://www.microsoft.com/de-de/techwiese/news/neues-model-context-protocol-ermoeglicht-verbesserte-ki-assistenten-in-azure-ai-foundry.aspx
35. https://learn.microsoft.com/en-us/semantic-kernel/concepts/plugins/
36. https://devblogs.microsoft.com/semantic-kernel/semantic-kernel-time-plugin-with-python/
37. https://www.infoq.com/news/2025/05/azure-ai-foundry-agents-ga/
38. https://techcommunity.microsoft.com/blog/azure-ai-services-blog/announcing-general-availability-of-azure-ai-foundry-agent-service/4414352
39. https://learn.microsoft.com/en-us/azure/ai-foundry/agents/overview
40. https://learn.microsoft.com/en-us/azure/ai-services/agents/whats-new
41. https://azure.microsoft.com/en-us/blog/introducing-deep-research-in-azure-ai-foundry-agent-service/