ChatGPT

Comprehensive Training Resource for Implementing a Foundry-Azure MCP Server in .NET

Executive summary

The **Model Context Protocol (MCP)** is a JSON-RPC 2.0–based, stateful protocol designed to standardize how AI host applications connect to external *servers* that expose tools, resources, and prompts. MCP sits between an "AI host" (for example, an IDE assistant) and your integration surface (your MCP server), providing capability negotiation, structured tool schemas, and explicit lifecycle phases (initialize → operate → shutdown). [1]

In the **Microsoft Foundry ecosystem**, there are two distinct "MCP server" realities you need to train a team on:

- **Foundry MCP Server (preview)**: a *Microsoft-hosted*, public endpoint (`https://mcp.ai.azure.com`) that exposes curated operations over Foundry resources and enforces authentication/authorization with Microsoft Entra ID (including On-Behalf-Of behavior). This is not something your team "implements," but something they *use* and must understand for governance, security, and operational safety. [2]
- **Bring-your-own remote MCP server**: a server you build (for private/internal APIs or custom tooling) and connect to **Foundry Agent Service** as a remote MCP endpoint. Microsoft's guidance explicitly notes that Agent Service connects only to **publicly accessible MCP endpoints**, and provides patterns for registering servers in an organizational catalog (Azure API Center) and configuring authentication choices (key-based, Entra, OAuth identity passthrough). [3]

For a team "implementing a Foundry Azure MCP server in .NET," the practical training objective is therefore: **build and operate a standards-compliant MCP server (HTTP transport) that can be safely consumed by Foundry Agent Service and other MCP clients, integrate securely with Azure services, and remain interoperable as MCP evolves.** [4]

Key unspecified items (that materially affect design and training labs) and recommended options:

- **Target .NET version**: unspecified. Recommended baseline is **.NET 8+ for production services**, but note Microsoft's MCP server project templates currently require **.NET 10 SDK** (template installation) and the MCP C# SDK is still in **preview**. Provide tracks for both ".NET 8 LTS production" and ".NET 10 SDK / template-driven" development in training. [5]
- **Transport**: unspecified. For Foundry Agent Service remote servers, plan on **HTTP transport** (not stdio), because Agent Service is remote and expects an HTTPS-accessible endpoint. [6]
- **Authentication**: unspecified. Training should cover (a) **Microsoft Entra auth** (preferred when possible), (b) **OAuth identity passthrough** when user context must persist, and (c) **key-based** when you need a single shared identity and can govern secret exposure. [7]

- **Expected load/concurrency**: unspecified. Training should include load testing and profiling labs and teach the team how to select hosting (Functions vs Container Apps vs AKS) based on latency, cold-start tolerance, and scaling needs. [8]

MCP protocol specification essentials

Core protocol model and roles

MCP defines a client-server model where an AI **host** initiates connections via one or more MCP **clients** to MCP **servers** that provide capabilities (tools/resources/prompts). The protocol uses **JSON-RPC 2.0 messages** and is explicitly designed for **stateful connections** with **capability negotiation**. [9]

At the wire level, MCP uses JSON-RPC request/response/notification objects. The base protocol specifies standard JSON-RPC fields (`jsonrpc`, `id`, `method`, `params`, `result`, `error`) and notes that requests require an `id` to correlate responses, while notifications have no `id`. [10]

Lifecycle phases, state machines, and negotiation

MCP defines a strict lifecycle with three phases: **Initialization**, **Operation**, and **Shutdown**. Initialization must be first: the client sends an `initialize` request that includes the protocol version it supports, client capabilities, and client info; the server responds with agreed protocol version, server capabilities, and server info; then the client sends an `initialized` notification to complete the handshake. [11]

A training-friendly abstraction is to teach both a **protocol-level state machine** and an **implementation-level state machine**:

```
stateDiagram-v2
    [*] --> AwaitInitialize
    AwaitInitialize --> Initialized: initialize request/response + initialized
notification
    Initialized --> Running: normal operation
    Running --> ShuttingDown: client/server shutdown
    ShuttingDown --> Closed
    Closed --> [*]
```

The lifecycle spec emphasizes version negotiation and capability negotiation as first-class responsibilities (not optional niceties). Version negotiation is centered around the protocol revision string (for example `2025-06-18`) exchanged during initialization. [11]

Transports and connection semantics

MCP supports two primary transport mechanisms:

- **stdio transport**: for local, same-machine process communication (fast, no network overhead). [12]
- **Streamable HTTP transport**: client messages are HTTP POSTs containing single JSON-RPC objects; responses can be returned as JSON or streamed via **Server-Sent Events (SSE)**. MCP's HTTP transport

supports standard HTTP authentication approaches (bearer tokens, API keys, custom headers), and MCP recommends OAuth token acquisition patterns for robust auth. [13]

The streamable HTTP spec is operationally nuanced: clients must send `Accept: application/json, text/event-stream`; servers may return `application/json` for a single response or `text/event-stream` for streaming. The spec explicitly allows the server to send notifications/requests on the SSE stream before the final JSON-RPC response, and it discusses resumability and reconnection behavior (including `Last-Event-ID` and server-provided `retry` guidance). [14]

A simplified end-to-end sequence to teach in labs (HTTP + SSE):

```
sequenceDiagram
    participant C as MCP Client
    participant S as MCP Server (HTTP endpoint)
    C->>S: POST initialize (JSON-RPC)
    S-->>C: 200 JSON (initialize result)
    C->>S: POST initialized (notification)
    C->>S: POST tools/list (request)
    S-->>C: 200 JSON or SSE stream
    C->>S: POST tools/call (request)
    S-->>C: SSE: progress/notifications (optional)
    S-->>C: SSE or JSON: tools/call response (result or error)
```

This is especially important for Foundry-connected remote servers because your server must behave correctly under real-world network conditions (disconnects, retries, reconnections) rather than assuming a single long-lived TCP stream. [6]

MCP primitives and message types

MCP centers on *server primitives* and *client primitives*:

- **Server primitives**: **tools**, **resources**, **prompts**. • Tools are model-invocable functions with schema metadata. • Resources expose contextual data. • Prompts provide reusable templates/workflows. [15]
- **Client primitives**: features that the client can offer the server, including **roots**, **sampling**, and **elicitation**, enabling servers to request scoped filesystem context, request LLM sampling, or ask for user input through the host. [16]

A critical training point: MCP does not mandate a user interaction model for tool execution, but it is designed so that hosts *may* insert guardrails (approvals, confirmations, policy checks) before running tools. That host-mediated safety model shows up strongly in Microsoft's MCP guidance and in the Azure/Foundry tool ecosystems. [17]

Error handling, failure modes, and standards alignment

MCP uses JSON-RPC errors, and the spec provides examples and recommended error code usage for common cases in different primitives:

- Resources: "resource not found" is shown as `-32002` and internal errors as `-32603`. [18]
- Prompts: invalid prompt name or missing required arguments should map to `-32602` (Invalid params). [19]
- Roots: if the client does not support roots, `-32601` (Method not found) is recommended. [20]

Training should include a failure-mode catalog that aligns protocol errors to operational root causes (auth failure vs not found vs validation vs downstream dependency outage), because MCP errors become the "surface area" that the LLM host reasons about. [21]

Versioning and extensions

MCP uses a **protocol revision** model (date-stamped revisions such as `2025-06-18`) and expects clients and servers to negotiate compatibility during initialization. [11]

Extensions are managed via a documented extensions process and "SEP" proposals; the MCP community describes both **official extensions** and **experimental extensions** as part of an incubation pathway. Training should treat extensions as *versioned specs* with explicit governance rather than ad-hoc custom fields. [22]

Microsoft implementation landscape and how it relates to the spec

Foundry MCP Server versus bring-your-own MCP servers

Foundry MCP Server (preview) is a cloud-hosted MCP implementation that exposes curated tools for operations across Foundry resources. Microsoft's documentation highlights Entra ID enforcement and On-Behalf-Of behavior, and it explicitly warns the feature is preview and not recommended for production workloads. [2]

In contrast, **Foundry Agent Service** supports connecting to remote MCP servers that are "bring your own endpoint." Microsoft's guidance emphasizes:

- You are responsible for choosing trusted remote servers.
- When connecting to non-Microsoft MCP servers, prompt content and other data may flow to those servers; Microsoft does not test/verify third-party servers.
- The MCP tool supports custom headers, and headers passed at runtime are only valid for the current run (not persisted). [23]

Microsoft also states that Agent Service connects only to **publicly accessible MCP endpoints**, which strongly influences Azure network design and security posture (you often need a public endpoint with tight auth, rather than private networking only). [24]

Authentication models supported in Foundry for MCP tools

Foundry's MCP tool authentication guidance frames two scenarios: **shared authentication** (one identity for all users) and **individual authentication** (each user keeps their context). It recommends Microsoft Entra authentication "when in doubt" (eliminates secret management and provides token rotation), and it distinguishes OAuth identity passthrough as the user-context-preserving option. [25]

This informs your training resources: you must teach not only "how to validate a bearer token," but also **how to choose** an auth model, document it for governance, and validate it in end-to-end tests with Foundry Agent Service. [7]

Azure MCP Server and Microsoft-provided MCP servers

The **Azure MCP Server** is an MCP server implementation designed to manage Azure resources via natural language–driven tool invocation. Microsoft documents that it implements MCP, supports Entra ID authentication (via Azure Identity), and is compatible with MCP clients including GitHub Copilot agent mode and Semantic Kernel. [26]

Microsoft's Azure MCP Server documentation explicitly covers multi-service workflows and "server modes" (namespace mode, consolidated mode, all mode), highlighting practical constraints such as tool limits in some clients and the need to filter exposed tools for focused workflows. [27]

Microsoft also announced a stable release of Azure MCP Server with emphasis on trust: sensitive tools requiring user confirmation, threat modeling and SDL processes, and performance improvements (including .NET AOT compilation). [28]

.NET SDKs, templates, and sample ecosystems

The "official" MCP C# SDK ecosystem centers on **modelcontextprotocol/csharp-sdk**, described as maintained in collaboration with Microsoft. It splits into:

- `ModelContextProtocol`: hosting + DI extensions (non-HTTP focus)
- `ModelContextProtocol.AspNetCore`: HTTP-based MCP server support
- `ModelContextProtocol.Core`: low-level client/server APIs with fewer dependencies [29]

For **HTTP MCP servers in ASP.NET Core**, the SDK shows a concrete pattern: `AddMcpServer().WithHttpTransport().WithToolsFromAssembly()` plus `app.MapMcp()` to map endpoints. [30]

Microsoft's .NET quickstart provides an "MCP Server App template" workflow and includes both stdio and HTTP `mcp.json` configuration examples for VS Code MCP server registration, but it also notes template tooling is preview and requires .NET 10 SDK for template installation. [31]

Deployment patterns Microsoft documents for Foundry-connected MCP endpoints

Microsoft provides an explicit pattern for deploying a remote MCP server endpoint and connecting it in Foundry:

- An Azure Developer CLI (`azd`) template deploys the Azure MCP Server to **Azure Container Apps** over HTTPS with **managed identity** for outbound calls (for example, to Storage) and with **Entra app registration / roles** for incoming OAuth authentication. [32]
- An On-Behalf-Of variant template for hosted Azure MCP Server describes exchanging incoming user access tokens for downstream Azure service calls using OBO flow, positioning it as suitable when you want a centralized server for users with different permissions. [33]

From a training perspective, these templates are valuable reference deployments to ensure your team understands the "Microsoft-native" approach to identity, RBAC, and observability for MCP endpoints. [34]

Reference architecture for a Foundry-connected Azure MCP server in .NET

Architectural goals and common tool patterns

For a remote MCP server intended to be consumed by Foundry Agent Service, the architecture must reconcile:

- MCP's **stateful, negotiated** protocol surface with
- a **cloud-hosted, horizontally scalable** service design (often stateless per request, with externalized state), and
- Foundry's requirement for **public reachability** plus authentication/authorization controls. [35]

A practical pattern to teach:

- Keep MCP server operations **fast and deterministic** when possible (tool calls that map directly to a single downstream API call).
- For long-running operations, design tools to be **asynchronous workflows** backed by messaging (Service Bus) or events (Event Grid), and use **status query tools** (polling) rather than blocking until completion. MCP supports utility behaviors "like notifications for real-time updates and progress tracking for long-running operations," which allows a host to reflect progress when the transport supports streaming. [13]

Azure services selection map

A common "production-grade" reference stack for MCP servers includes:

- **Compute/hosting**: Azure Functions, Azure Container Apps, Azure App Service, or AKS
- **Secrets and identity**: Key Vault + Managed Identity
- **Messaging for async tools**: Service Bus (durable queues/topics) and/or Event Grid (pub/sub eventing)
- **State and artifacts**: Azure Storage (blobs/files/queues/tables)

- **Observability**: Application Insights + Azure Monitor (metrics/logs/alerts)
- **Deployment automation**: Azure DevOps Pipelines / GitHub Actions plus Bicep/Terraform/Helm for IaC and rollout strategies (blue/green, canary) [36]

Hosting model comparison

Hosting model	When it fits MCP servers	Key trade-offs for MCP
Azure Functions	Tool calls are bursty; desire scale-to-zero; small surface; strong dev velocity	Cold start risks if scaled to zero; careful with long-lived SSE connections; networking/auth patterns differ by plan [37]
Azure Container Apps	Container-first, revisioned deployments; straightforward HTTPS endpoints; traffic splitting for progressive delivery	Some protocol features (e.g., streaming) require careful config; must design for multi-revision behavior [38]
Azure App Service	Familiar web hosting; deployment slots enable staging/blue-green swaps	Less Kubernetes-native; scaling model differs; still need robust auth and secrets story [39]
AKS	Complex workloads; need service mesh/ ingress control; high scale; custom networking	Highest operational overhead; best for mature platform teams; progressive delivery patterns require routing strategy [40]

The training resource should explicitly teach teams how to map “tool latency expectations” to hosting selection (e.g., serverless vs always-warm), and it should include at least one lab that demonstrates a scale-to-zero cold start impact for tool calls. [41]

Messaging choice comparison for asynchronous tools

Service	Best for	Why it matters for MCP tools
Azure Service Bus	Durable command/work queues; transactional workflows; dead-lettering	Useful when tool calls must enqueue work reliably and later provide tool status/results; supports enterprise broker features [42]
Azure Event Grid	Pub/sub event distribution; event-driven architectures	Useful when tools react to events (resource changes, blob created, etc.) and trigger downstream processing; highly scalable event routing [43]

Reference architecture diagram

```
flowchart LR
  A[Foundry Agent Service\n(MCP client)] -->|HTTPS + auth headers| B[MCP Server
(.NET)\nHTTP transport]
  B -->|Managed Identity / Entra| KV[Key Vault]
```

```
  B -->|Managed Identity / Entra| ST[Azure Storage]
  B -->|enqueue long jobs| SB[Service Bus]
  B -->|publish events| EG[Event Grid]
  B -->|metrics/logs/traces| AI[Application Insights]
  AI --> AM[Azure Monitor\nalerts/dashboards]

  SB --> W[Worker / Processor\n(Functions or Container Apps)]
  W --> ST
  W --> AI
```

This model aligns with Microsoft guidance for securing endpoints, using least privilege for downstream calls, and instrumenting services for troubleshooting. [44]

.NET implementation best practices with concrete examples

Recommended solution structure in .NET

Because team experience level is unspecified, the training should teach a layered structure that scales from "simple demo" to "production service":

- **Protocol layer** (MCP): exposes tools/prompts/resources and handles MCP transport.
- **Application layer**: implements tool behaviors and orchestration (validation, routing, permissions).
- **Infrastructure layer**: Azure SDK calls, messaging, storage, secrets.
- **Observability & ops layer**: logging, metrics, tracing, health checks.
- **Policy layer**: authentication/authorization, throttling, rate limiting, allowlists.

This maps well to ASP.NET Core DI and service container patterns. [45]

Minimal HTTP MCP server in ASP.NET Core

The MCP C# SDK demonstrates an ASP.NET Core server using `WithHttpTransport()` and `MapMcp()`. A training lab should start here and then extend into auth, logging, and Azure integration. [30]

```
// Program.cs
using ModelContextProtocol.Server;
using System.ComponentModel;

var builder = WebApplication.CreateBuilder(args);

// Register MCP server + HTTP transport + tool discovery.
builder.Services.AddMcpServer()
    .WithHttpTransport()
    .WithToolsFromAssembly();

var app = builder.Build();
```

```
// Map MCP endpoints (HTTP + optional SSE streaming depending on client
behavior).
app.MapMcp();

// Example tool class discovered by WithToolsFromAssembly()
[McpServerToolType]
public static class EchoTool
{
    [McpServerTool, Description("Echoes the message back to the client.")]
    public static string Echo(string message) => $"hello {message}";
}

app.Run("http://localhost:3001");
```

This aligns with the SDK's recommended bootstrap and tool-discovery conventions. [46]

Explicit handler control, input schema, and protocol-level errors

For rigorous correctness and interop testing, teach how to implement handlers directly and throw **protocol-aware exceptions**. The SDK example uses `McpProtocolException` with `McpErrorCode` to represent invalid params or invalid requests. [47]

```
using ModelContextProtocol;
using ModelContextProtocol.Protocol;
using ModelContextProtocol.Server;
using System.Text.Json;

var options = new McpServerOptions
{
    ServerInfo = new Implementation { Name = "Contoso.McpServer", Version =
"1.0.0" },
    Handlers = new McpServerHandlers
    {
        ListToolsHandler = (request, ct) =>
            ValueTask.FromResult(new ListToolsResult
            {
                Tools =
                [
                    new Tool
                    {
                        Name = "echo",
                        Description = "Echoes the input back to the client.",
                        InputSchema =
JsonSerializer.Deserialize<JsonElement>("""
                        {
                          "type": "object",
```

```
                        "properties": {
                          "message": { "type": "string", "description":
"Message to echo" }
                        },
                        "required": ["message"]
                      }
                      """)
                  }
              ]
          }),

      CallToolHandler = (request, ct) =>
      {
          if (request.Params?.Name != "echo")
              throw new McpProtocolException(
                  $"Unknown tool: '{request.Params?.Name}'",
                  McpErrorCode.InvalidRequest);

          if (request.Params.Arguments?.TryGetValue("message", out var
message) is not true)
              throw new McpProtocolException(
                  "Missing required argument 'message'",
                  McpErrorCode.InvalidParams);

          return ValueTask.FromResult(new CallToolResult
          {
              Content = [ new TextContentBlock { Type = "text", Text =
$"Echo: {message}" } ]
          });
      }
  }
};
```

This “manual handler” style is pedagogically useful for teaching message validation, versioning, and compatibility testing because it makes control points explicit. [48]

Async/await, cancellation, and DI patterns

Training should reinforce “async all the way,” especially because MCP servers are typically I/O bound (HTTP calls, storage operations, queue writes). For ASP.NET Core, dependency injection is built-in and lifecycle-aware; .NET also provides DI guidelines and patterns for lifetimes and constructor injection. [49]

A recommended pattern for tool methods:

- accept `CancellationToken`
- avoid blocking calls (`.Result`, `.Wait()`)
- validate inputs early

- keep tool handlers thin by pushing work into injected services

Serialization: reduce overhead using System.Text.Json source generation

MCP tools frequently pass JSON payloads; correctness and performance hinge on predictable serialization. .NET's `System.Text.Json` supports source generation via `JsonSerializerContext` to reduce runtime reflection and improve performance (also relevant for AOT scenarios). [50]

```
using System.Text.Json.Serialization;

[JsonSerializable(typeof(MyToolRequest))]
[JsonSerializable(typeof(MyToolResponse))]
internal partial class McpJsonContext : JsonSerializerContext;

public sealed record MyToolRequest(string Message);
public sealed record MyToolResponse(string Echo);
```

Resilience: retries, circuit breakers, timeouts for downstream calls

A production MCP server must behave predictably when downstream dependencies fail. Microsoft's guidance for resilient microservice HTTP calls recommends using `IHttpClientFactory` and proven resilience libraries like Polly (Retry, Circuit Breaker, Timeout, etc.). [51]

In newer .NET guidance, **Microsoft.Extensions.Http.Resilience** provides HttpClient-focused resilience mechanisms; its package description explicitly includes retry and circuit breaker behaviors. [52]

Practical training lab: implement typed HttpClient + retry + circuit breaker for a downstream API your tool calls, then verify behavior under fault injection.

Health probes and readiness/liveness

Health checks are essential for container orchestrators and traffic managers (AKS, Container Apps) and for operational runbooks. ASP.NET Core health checks can probe downstream dependencies (e.g., database probe patterns) and expose a health endpoint. [53]

Azure integration patterns in MCP tools

A realistic Foundry-connected MCP server typically needs:

- Managed identity authentication to access Azure services without secrets. Managed identities remove credential management overhead and let apps authenticate to Entra-protected resources. [54]
- Key Vault for secrets/certificates where secrets are unavoidable; Key Vault centralizes secret storage and integrates with RBAC. [55]
- Durable state via Storage accounts; Storage accounts are durable, secure, and support Entra ID authorization via Azure RBAC (recommended). [56]

Security, governance, and threat modeling for MCP servers

Trust boundaries and “MCP as an attack surface”

An MCP server is an externally invocable integration surface where:

- the *caller* may be an LLM host acting on user instructions,
- tool arguments may include untrusted text,
- and the server may have privileged access to internal APIs.

Microsoft explicitly warns that when connecting to non-Microsoft MCP servers, data (including prompt content) can flow to those servers, and Microsoft does not verify third-party servers. This frames governance and vendor trust as security requirements, not optional best practices. [23]

Authentication and authorization models

Foundry’s authentication guidance distinguishes shared identity vs individual identity persistence:

- **Key-based** and **Microsoft Entra** methods are “shared-auth” patterns (user context does not persist).
- **OAuth identity passthrough** preserves user identity and permissions across calls.
- Foundry recommends starting with Microsoft Entra authentication when supported to avoid secret management and rely on built-in token rotation. [25]

In the **Foundry MCP Server (preview)** offering, Microsoft emphasizes that operations run according to the authenticated user’s permissions via Azure RBAC, and it documents Conditional Access control patterns. [57]

TLS, certificates, and endpoint security

For HTTP MCP servers, treat TLS as mandatory for any real deployment:

- Kestrel requires a TLS certificate for HTTPS, and production requires explicit certificate configuration (development certificates are not for nondevelopment environments). [58]
- ASP.NET Core recommends HTTPS redirection and HSTS for production web apps. [59]

Secrets management and least privilege

Microsoft’s “build your own MCP server” guidance explicitly calls out a baseline:

- require authentication,
- treat credentials as secrets (avoid hard-coding, avoid source control exposure),
- store secrets in Key Vault,
- and implement least privilege for downstream calls. [60]

Key Vault documentation reinforces the value proposition: centralize storage of secrets/keys/certificates, reduce accidental leakage, and enforce authentication/authorization via Entra ID with Azure RBAC or Key Vault access policy. [61]

Input validation and injection resistance

MCP servers must validate tool arguments and resource URIs. The MCP spec explicitly requires validating resource URIs and recommends access controls for sensitive resources; it also cautions about binary encoding and permission checks. [18]

For broader web application security posture, the OWASP [62] Top Ten is the standard awareness baseline; it explicitly calls out classes of risk (broken access control, injection, security misconfiguration, and cryptographic failures) that are directly relevant to MCP endpoints (which are often "API-like" surfaces). [63]

Foundry MCP Server operational security constraints (important for training)

Microsoft's Foundry MCP Server security guidance includes constraints that should shape enterprise training, even if you are building your own MCP server:

- Foundry MCP Server currently does **not** support network isolation and uses a public endpoint; it can't reach Foundry resources behind Private Links.
- Microsoft describes a "global stateless proxy architecture," where it does not store data, but requests/responses may be processed in EU or US datacenters (encrypted in transit), and strict in-region requirements may require restricting usage. [57]

This is a concrete example of why "network isolation" and "data residency" must be explicit topics in MCP training modules. [64]

Operations, performance, CI/CD, testing, migration, and the training checklist

Performance and scalability engineering

Because expected load is unspecified, training should focus on measurement-driven performance work:

- **Benchmark the protocol paths**: initialize, tools/list, tools/call for representative payloads; capture latency distributions under load.
- **Connection handling**: implement correct streamable HTTP behavior (SSE vs JSON), and ensure graceful handling of disconnect/reconnect per spec. [14]
- **Memory and GC awareness**: .NET GC is automatic memory management; you must still design for allocation behavior, and you may need to understand workstation vs server GC trade-offs and runtime configuration settings. [65]
- **Diagnostics**: use `dotnet-counters` for first-level monitoring and `dotnet-trace` for trace collection via EventPipe. [66]

For load testing, Microsoft provides **Azure Load Testing** as a managed service to generate high-scale load for applications regardless of where hosted; it is positioned for developers/testers/QA engineers to optimize performance, scalability, and capacity. [67]

Observability and troubleshooting

A production MCP server should implement the “three pillars”:

- **Traces + metrics + logs** using OpenTelemetry patterns and export to Application Insights / Azure Monitor.
- Azure Monitor documentation frames the platform as aggregating and analyzing metrics/logs/traces and responding via alerts and automated actions. [68]
- Application Insights supports OpenTelemetry-based instrumentation with both automatic and manual approaches, and Microsoft positions OpenTelemetry as a future direction while maintaining older SDK support. [69]

Training should include: building dashboards for tool call rate/latency/error codes, adding alerts for failure spikes, and writing runbooks that map “MCP errors” to “downstream incidents + mitigation steps.” [70]

Deployment and CI/CD

Infrastructure-as-code and progressive delivery are critical because MCP servers are integration surfaces where regressions impact many tools/agents.

- **Bicep** is a declarative DSL for deploying Azure resources; Microsoft recommends Bicep as a simpler syntax alternative to ARM JSON. [71]
- **Terraform on Azure** is documented for provisioning Azure infrastructure; it provides a separate IaC workflow and ecosystem integration. [72]
- For **AKS**, Microsoft documents Helm-based application installation and lifecycle management. [73]
- For **Azure Container Apps**, Microsoft documents revisions and traffic splitting as mechanisms used for blue/green and phased rollouts. [38]
- For **Azure App Service**, deployment slots provide staging environments and slot swapping (blue/green-like). [74]
- In **Azure DevOps Pipelines**, Microsoft documents deployment jobs and explicitly lists supported strategies including runOnce, rolling, and canary. [75]

Containerization should be a first-class training module because it affects hosting, scaling, and rollouts. Microsoft provides a .NET tutorial on containerizing apps with Docker and highlights portability/immutability/scalability benefits of containers. [76]

Testing and QA strategy

A comprehensive testing curriculum should cover:

- **Unit tests + coverage**: Microsoft’s .NET guidance describes code coverage workflows using Coverlet and report generation tooling. [77]
- **Integration tests**: ASP.NET Core integration testing guidance uses `WebApplicationFactory<TEntryPoint>` and a TestServer to exercise the full pipeline. [78]
- **Contract testing**: MCP tool interfaces are effectively contracts: tools include JSON schema for inputs. Use schema validation and snapshot testing of tool catalogs; the C# SDK examples show tools with an `InputSchema` represented as JSON. [79]

- **Fuzzing**: introduce structured fuzzing of tool argument parsers and JSON-RPC payload boundaries (malformed JSON, huge payloads, missing required keys) as part of robustness testing; align results with OWASP injection and misconfiguration concerns. [80]
- **Chaos engineering**: use **Azure Chaos Studio** to run controlled fault injection experiments and validate resilience behaviors and runbooks. [81]

Migration and interoperability

Interoperability is largely about staying faithful to:

- lifecycle rules (initialize first),
- version negotiation,
- negotiated capabilities,
- and transport correctness.

Because MCP uses explicit protocol revisions and initialization negotiation, your server should be engineered for **backward compatibility** (support N and N-1 revisions where feasible) and for graceful negotiation failure modes. [11]

Additionally, Microsoft's ecosystem includes multiple MCP client surfaces (IDE agent modes, Foundry Agent Service, and other SDKs), and Azure MCP Server documentation explicitly positions interop with different clients as a design goal. This makes conformance testing against multiple clients a required training deliverable. [82]

Prioritized checklist for training modules and deliverables

Priority	Training module	Outcomes	Concrete deliverables
Highest	MCP fundamentals and spec conformance	Team can explain lifecycle, transports, message types, and error strategy	MCP "conformance notebook": initialize/tools/list/tools/call examples + error catalog mapped to JSON-RPC codes [83]
Highest	Foundry integration model	Team can connect a remote MCP endpoint to Foundry Agent Service and reason about public endpoint requirements	End-to-end lab: deploy MCP server, connect in Foundry, run tool call, capture logs/traces [84]
Highest	Authentication and authorization	Team can choose auth model and implement/operate it safely	Auth decision record + implementation guide for Entra vs OAuth passthrough vs key-based [7]
High	.NET implementation patterns	Team can build tools with DI, async, validation, structured errors	Reference codebase with HTTP MCP server + DI + resilience + health checks [85]

Priority	Training module	Outcomes	Concrete deliverables
High	Azure service integrations	Team can use managed identities, Key Vault, Storage, and messaging for async tools	Sample tools: “store artifact,” “enqueue job,” “query status,” with MI + Key Vault [86]
High	Observability & incident response	Team can debug failures and build dashboards/alerts/runbooks	Dashboards for tool calls + alert rules + on-call runbooks linked to common MCP errors [87]
Medium	Deployment & progressive delivery	Team can deploy safely with rollouts and rollbacks	IaC (Bicep/Terraform) + blue/green or canary rollout examples for chosen host [88]
Medium	Testing, fuzzing, chaos	Team can validate correctness and resilience beyond happy paths	Integration test suite + contract tests + chaos experiment plan + load test plan [89]

Recommended reading list and official references

Protocol and core learning: - MCP specification and overview (protocol revision `2025-06-18` and lifecycle/transports). [90]
- MCP architecture overview (roles, transports, auth patterns). [91]

Microsoft + Foundry: - Build and register a custom MCP server and connect it to Foundry Agent Service (includes public endpoint constraint, request flow, Azure Functions sample/template approach). [24]
- Foundry MCP tool authentication guidance (key-based vs Entra vs OAuth passthrough). [25]
- Foundry MCP Server best practices/security guidance (identity, RBAC, conditional access, network isolation constraints, data residency notes). [57]

.NET SDKs and examples: - Official MCP C# SDK packages and examples (HTTP server with `ModelContextProtocol.AspNetCore`). [92]
- Microsoft .NET quickstart for MCP server template and publishing (noting preview/template requirements). [31]

Azure services for MCP server implementations: - Managed identities (credential-free auth to Entra-protected resources). [93]
- Key Vault overview (secrets/keys/certs, RBAC integration). [61]
- Azure Storage intro (Entra ID + RBAC recommended for data authorization). [94]
- Service Bus overview (enterprise message broker) and queues/topics/subscriptions. [95]
- Event Grid overview (managed pub/sub for event-driven architectures). [96]
- Azure Monitor + Application Insights OpenTelemetry guidance. [97]
- Azure Load Testing overview for performance testing at scale. [98]
- Azure Chaos Studio for resilience/chaos experiments. [81]

Security baseline: - OWASP [62] Top Ten (current and prior editions) as a training baseline for MCP endpoint risk classes. [99]

Source repositories and templates to include in labs: - Azure Container Apps + managed identity template for deploying an MCP endpoint used by Foundry agents (reference `azd` template). [32]
- Azure MCP Server On-Behalf-Of remote hosting template (OBO flow reference). [33]
- MCP C# SDK repository and samples folder. [100]

---

[1] [9] [15] [90] https://modelcontextprotocol.io/specification/2025-06-18
https://modelcontextprotocol.io/specification/2025-06-18

[2] https://learn.microsoft.com/en-us/azure/ai-foundry/mcp/get-started?view=foundry
https://learn.microsoft.com/en-us/azure/ai-foundry/mcp/get-started?view=foundry

[3] [24] [36] [44] [60] [84] https://learn.microsoft.com/en-us/azure/ai-foundry/mcp/build-your-own-mcp-server?view=foundry
https://learn.microsoft.com/en-us/azure/ai-foundry/mcp/build-your-own-mcp-server?view=foundry

[4] [12] [13] [91] https://modelcontextprotocol.io/docs/learn/architecture
https://modelcontextprotocol.io/docs/learn/architecture

[5] [31] https://learn.microsoft.com/en-us/dotnet/ai/quickstarts/build-mcp-server
https://learn.microsoft.com/en-us/dotnet/ai/quickstarts/build-mcp-server

[6] [14] https://modelcontextprotocol.io/specification/draft/basic/transports
https://modelcontextprotocol.io/specification/draft/basic/transports

[7] [25] https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/mcp-authentication?view=foundry
https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/mcp-authentication?view=foundry

[8] [37] [41] https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale
https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale

[10] (no title)

[11] [16] [35] [83] https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle
https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle

[17] https://modelcontextprotocol.io/specification/2025-06-18/server/tools
https://modelcontextprotocol.io/specification/2025-06-18/server/tools

[18] [21] https://modelcontextprotocol.io/specification/2025-06-18/server/resources
https://modelcontextprotocol.io/specification/2025-06-18/server/resources

[19] https://modelcontextprotocol.io/specification/2025-06-18/server/prompts
https://modelcontextprotocol.io/specification/2025-06-18/server/prompts

[20] https://modelcontextprotocol.io/specification/2025-06-18/client/roots
https://modelcontextprotocol.io/specification/2025-06-18/client/roots

[22] https://modelcontextprotocol.io/docs/extensions/overview
https://modelcontextprotocol.io/docs/extensions/overview

[23] https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/tools-classic/model-context-protocol?view=foundry-classic
https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/tools-classic/model-context-protocol?view=foundry-classic

26 82 https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/overview
https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/overview

27 https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/concepts
https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/concepts

28 https://devblogs.microsoft.com/azure-sdk/announcing-azure-mcp-server-stable-release/
https://devblogs.microsoft.com/azure-sdk/announcing-azure-mcp-server-stable-release/

29 92 100 https://github.com/modelcontextprotocol/csharp-sdk
https://github.com/modelcontextprotocol/csharp-sdk

30 46 62 85 https://raw.githubusercontent.com/modelcontextprotocol/csharp-sdk/main/src/ModelContextProtocol.AspNetCore/README.md
https://raw.githubusercontent.com/modelcontextprotocol/csharp-sdk/main/src/ModelContextProtocol.AspNetCore/README.md

32 34 https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/how-to/deploy-remote-mcp-server-microsoft-foundry
https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/how-to/deploy-remote-mcp-server-microsoft-foundry

33 https://github.com/Azure-Samples/azmcp-obo-template
https://github.com/Azure-Samples/azmcp-obo-template

38 https://learn.microsoft.com/en-us/azure/container-apps/revisions
https://learn.microsoft.com/en-us/azure/container-apps/revisions

39 74 https://learn.microsoft.com/en-us/azure/app-service/deploy-staging-slots
https://learn.microsoft.com/en-us/azure/app-service/deploy-staging-slots

40 https://learn.microsoft.com/en-us/azure/architecture/guide/aks/blue-green-deployment-for-aks
https://learn.microsoft.com/en-us/azure/architecture/guide/aks/blue-green-deployment-for-aks

42 95 https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview
https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview

43 96 https://learn.microsoft.com/en-us/azure/event-grid/overview
https://learn.microsoft.com/en-us/azure/event-grid/overview

45 49 https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-10.0
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-10.0

47 48 79 https://raw.githubusercontent.com/modelcontextprotocol/csharp-sdk/main/README.md
https://raw.githubusercontent.com/modelcontextprotocol/csharp-sdk/main/README.md

50 https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/source-generation
https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/source-generation

51 https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests
https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests

52 https://learn.microsoft.com/en-us/dotnet/core/resilience/http-resilience
https://learn.microsoft.com/en-us/dotnet/core/resilience/http-resilience

53 https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-10.0
https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-10.0

54 86 93 https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview
https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview

55 61 https://learn.microsoft.com/en-us/azure/key-vault/general/overview
https://learn.microsoft.com/en-us/azure/key-vault/general/overview

56 https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview
https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview

57 64 https://learn.microsoft.com/en-us/azure/ai-foundry/mcp/security-best-practices?view=foundry
https://learn.microsoft.com/en-us/azure/ai-foundry/mcp/security-best-practices?view=foundry

58 https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/endpoints?view=aspnetcore-10.0
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/endpoints?view=aspnetcore-10.0

59 https://learn.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-10.0
https://learn.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-10.0

63 99 https://owasp.org/www-project-top-ten/
https://owasp.org/www-project-top-ten/

65 https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals
https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

66 https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-counters
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-counters

67 98 https://learn.microsoft.com/en-us/azure/app-testing/load-testing/overview-what-is-azure-load-testing
https://learn.microsoft.com/en-us/azure/app-testing/load-testing/overview-what-is-azure-load-testing

68 87 97 https://learn.microsoft.com/en-us/azure/azure-monitor/
https://learn.microsoft.com/en-us/azure/azure-monitor/

69 https://learn.microsoft.com/en-us/azure/azure-monitor/app/opentelemetry-overview
https://learn.microsoft.com/en-us/azure/azure-monitor/app/opentelemetry-overview

70 https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-overview
https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-overview

71 88 https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview
https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview

72 https://learn.microsoft.com/en-us/azure/developer/terraform/
https://learn.microsoft.com/en-us/azure/developer/terraform/

73 https://learn.microsoft.com/en-us/azure/aks/kubernetes-helm
https://learn.microsoft.com/en-us/azure/aks/kubernetes-helm

75 https://learn.microsoft.com/en-us/azure/devops/pipelines/process/deployment-jobs?view=azure-devops
https://learn.microsoft.com/en-us/azure/devops/pipelines/process/deployment-jobs?view=azure-devops

76 https://learn.microsoft.com/en-us/dotnet/core/docker/build-container
https://learn.microsoft.com/en-us/dotnet/core/docker/build-container

77 https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-code-coverage
https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-code-coverage

78 89 https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-10.0
https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-10.0

80 https://owasp.org/Top10/2021/A03_2021-Injection/
https://owasp.org/Top10/2021/A03_2021-Injection/

81 https://learn.microsoft.com/en-us/azure/chaos-studio/
https://learn.microsoft.com/en-us/azure/chaos-studio/

94 https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction
https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction