

AI Application Development with Python and Azure

From async APIs to a bounded agent prototype in five days

FA-0571 · 5 days

Software Development · Agentic & Generative AI · DevOps, Cloud & Infrastructure

Develop a small AI application using Python, async HTTP, web APIs, local/remote models, retrieval, MCP tools and a prepared Linux/Azure deployment workflow.

Introduction

This five-day course is for developers with basic Python, HTTP, shell and Git skills. It combines a focused refresher with web APIs, model integration, retrieval, bounded agents and deployment choices.

Participants build and review a small application in prepared environments. FastAPI is the core service example, with Flask compared or demonstrated. Local inference depends on available hardware; advanced orchestration, Linux operations and cloud scaling are selected clinics rather than production-readiness guarantees.

The final day connects a sandbox deployment to Microsoft Foundry (formerly Azure AI Foundry) concepts and supported deployment routes. Moving a prototype into production requires additional architecture, security, operational and service-specific review.

Learning outcomes

The course teaches participants to:

- Apply selected Python and async/concurrency patterns to HTTP and streaming calls.
- Build a small AI-facing web API with validation, limits and error handling.
- Run a suitable local model where resources allow and compare remote model integration.
- Test prompt/context patterns and evaluate latency, cost and reliability.
- Implement a small retrieval workflow and bounded tool-using agent.

- Explain MCP client/server integration and inspect tool permissions and failure paths.
- Deploy the example to a prepared Linux sandbox and review service/CI/CD controls.
- Compare Microsoft Foundry agent/deployment routes and identify further production requirements.

Prerequisites

- Basic Python knowledge (functions, classes, data structures).
- Basic familiarity with REST, JSON, HTTP.
- Basic command-line / shell competence (Unix commands, file navigation).
- Experience with version control (Git).

Refresh these foundations before attending; the course is not a first programming course.

A laptop and prepared Python/Linux environment, authorised model/API access and an Azure sandbox with required roles and selected service availability. Confirm usage costs and local-model resources in advance; a GPU-dependent exercise may be demonstrated or adapted when hardware is unsuitable.

Training outline

1. Day 1 – Python, async patterns and HTTP

Core Python refresher (compact but rigorous)

- Data structures, comprehensions, slicing
- Functions, closures, decorators, context managers
- Modules, packages, pip/venv, import mechanics
- Type hints, dataclasses, exception patterns

Concurrency, I/O, asynchronous patterns

- Blocking vs non-blocking I/O; threads, processes
- asyncio primitives: event loop, coroutines, async/await
- Tasks, futures, gather, wait, cancellation, timeout
- Interfacing blocking code using executors
- Backpressure, throttling, retry strategies

Async HTTP / network integration

- Using httpx (async) or aiohttp for REST calls

- Streaming responses, chunked transfers, backpressure
- WebSockets or Server-Sent Events basics
- Authentication (Bearer tokens, OAuth), headers, retry logic

Error handling, instrumentation, observability

- Logging, tracing, correlation IDs
- Retry/backoff, circuit-breaker patterns in async code
- Metrics: latency, error rates

2. Day 2 — Web APIs and AI service endpoints

Foundations of web service design

- HTTP semantics: verbs, status codes, headers, payloads
- JSON, serialization, validation

Flask primer

- App structure, routes, request/response lifecycle
- Blueprints, error handlers, middlewares
- Handling input validation, JSON payloads, exceptions

FastAPI core service example

- Defining path operations, request/response models (Pydantic)
- Dependency injection and background-task basics; distinguish in-process tasks from a durable job system.
- Async handlers, streaming responses
- Auto-generated OpenAPI / docs, versioning
- Sub-application mounting as an optional architecture example.

Wrapping AI endpoints

- Designing your service interface to LLM calls (e.g. /generate, /chat)
- Throttling, batching, fallback logic
- Asynchronous chaining of multiple calls
- Rate-limiting, concurrency controls

Service architecture considerations

- Clean layering (handlers, services, utils)
- Configuration management, secrets (e.g. storing API keys)
- Error propagation and fallback paths

3. Day 3 — Local inference, prompting and remote models

Intro to local inference

- Local-inference trade-offs: data handling, latency, resources and offline requirements.
- Limitations (GPU, VRAM, quantization)
 - Simple inference: transformers library usage
 - Use a compatible causal-language-model class and tokenizer for the selected model.
- For an appropriate unquantized model, select a supported CPU/GPU device; CUDA requires a compatible installed runtime.
- Accelerate/device mapping and supported 8-bit/4-bit quantization; backend and model compatibility matter.
- A suitable small model example, such as an appropriately sized Qwen-family model; use a prepared compatible environment rather than spend the lab installing GPU drivers.

Quantization / memory optimizations (overview)

- Tradeoffs: precision vs memory vs throughput
- Supported reduced-precision loading and memory-management options for the selected runtime; not every technique applies to every model.
- Offloading, CPU fallback for unsupported operations
- Handling OOM / memory overflow gracefully

Prompt engineering fundamentals

- Clear templates, selected examples and task decomposition where useful; test model-appropriate prompts rather than assume chain-of-thought is universally beneficial.
- Guardrails, prompt injection risks, fallback prompts
- Prompt versioning and tracking (e.g. prompt logs, prompt unit tests)
- Code-assistant prompts and output verification; generated code still needs review/testing.

Remote LLM API integration

- Using OpenAI / Azure OpenAI / other LLM-as-a-service endpoints
- Authentication, batching, backoff, error handling
- Prompt wrapping, fallback to local inference (hybrid)
- Latency tradeoffs, cost modeling

4. Day 4 — Retrieval, agents and MCP

Chaining & reasoning pipelines

- Concept: chains, nodes, memory, branching
- Current LangChain model/tool workflows, retrievers and create_agent where appropriate; compare fixed chains with agent loops.
- Building simple chains: e.g. summarization→follow-up→analysis
- Debugging chains, instrumentation

Retrieval Augmented Generation (RAG)

- Indexing documents: embeddings, vector stores (FAISS, Pinecone, etc.)
- Similarity search, re-ranking
- Grounding prompts with retrieved context
- Dealing with hallucination, context window limits
- Hybrid: local + remote retrieval

Agent architectures & tool invocation

- Agent as controller: planning, tool selection, re-plan, loop
- Scoped wrappers for selected tools; code, browser or database actions require appropriate sandbox and permissions.
- Single-step router, planner agents, hierarchical agents
- Fallback, abort, error paths

Model Context Protocol (MCP)

- What is MCP: a standard for tool + context interaction (exposing data/tools via MCP server, connecting with AI clients)
- MCP server / client architecture, JSON RPC contract

- Integrating MCP with agent frameworks (e.g. LangChain adapters)
- Use case: agent calling your own MCP-exposed tools (e.g. internal search, domain logic)

Interaction between RAG and MCP: retrieval context vs tool invocation

- Example: expose an authorised data-access tool through MCP and use retrieved evidence as context; the protocol alone does not guarantee deterministic or correct results.

Multi-agent orchestration & safety

- Agent-to-agent coordination, delegation
- Logging, tracing, fallback, safe stop
- Observability, debugging strategies

Hands-on lab

Build a small agent that:

- retrieves context via RAG
- Invoke a selected authorised domain tool through a compatible MCP server/client.
- returns answer, handles fallback
- Instrument, test, log, debug

5. Day 5 — Linux deployment and Microsoft Foundry

POSIX / Linux basics

- File structure, users, permissions, networking basics
- Process management: systemd, supervisord, logs
- Reverse proxy (Nginx), SSL/TLS, firewalls
- Monitoring basics: logs, top, resource usage

Containerization & deployment (if time permits)

- Dockerizing the app (Flask/FastAPI + agent logic)
- Container management, orchestration (optional lightweight)

CI/CD overview and prepared pipeline example

- GitHub Actions / Azure Pipelines: build, test, deploy
- Environment variables, secrets management

- Separate sandbox/staging settings from production requirements.

Deploying to Ubuntu VM / server

- Transfer, installation, virtual environment, dependencies
- Systemd services wrapping the app
- Logs, metrics, restarts, health checks

Microsoft Foundry architecture and migration clinic

- Microsoft Foundry: current platform for agents, models and tools, with management and governance capabilities.
- Distinguish current Foundry projects from classic/hub-based experiences and confirm the selected surface.
- Foundry Agent Service: compare managed prompt agents and hosted/code-based routes; capabilities and preview status vary.
- Compare local prototype components with supported Foundry runtime/deployment choices; migration is not automatic.
- Review selected model/version, identity, evaluation, monitoring and governance capabilities.
- Explore supported packaging/deployment for the selected agent type in a prepared example.
- Distinguish a web API service from a Foundry agent; choose the appropriate supported integration route.
- Plan local-to-cloud migration requirements and gaps.
- Best practices, pitfalls, performance considerations (latency, model swap, cost)

Prompting and coding assistants during deployment

- Prompts for chain orchestration, fallback logic
- Strategies to use Copilot: prompting, verifying, scaffolding
- Code review, unit tests around Copilot output