MCP-Server und Tools mit Python entwickeln

Professionelle MCP-Integrationen mit Python implementieren, testen und bereitstellen

01 INTRO

Model Context Protocol ermöglicht es, Funktionen und Datenquellen über eine standardisierte Schnittstelle für KI-Anwendungen bereitzustellen. Entscheidend für den praktischen Einsatz ist jedoch nicht allein das Protokoll, sondern die Qualität der implementierten Server und Tools: Schnittstellen müssen verständlich, robust, testbar und sicher sein.

Der Kurs ist ein praxisorientierter Entwicklungsworkshop. Die Teilnehmer entwickeln mit Python einen vollständigen MCP-Server, implementieren eigene Tools und Resources, integrieren vorhandene Python-Komponenten sowie externe Systeme und bauen schrittweise Fehlerbehandlung, Tests, Logging und Sicherheitsmechanismen ein. Am Ende steht keine Demo, sondern die Struktur einer MCP-Integration, wie sie auch in realen Projekten weiterentwickelt und betrieben werden kann.

DAUER

2 Tage

FORMAT

Präsenz · Live Online · Inhouse

VORKENNTNISSE

Sichere Python-Grundkenntnisse, praktische Entwicklungserfahrung und grundlegendes MCP-Verständnis

02 TRAININGSPROGRAMM

Inhalte

MCP-Architektur kompakt

- Host, Client und Server
- Fähigkeiten eines MCP-Servers
- Tools
- Resources
- Prompts
- Kommunikationswege
- Lifecycle
- Verantwortlichkeiten sauber trennen

Python-Entwicklungsumgebung

- Projektstruktur
- virtuelle Umgebung
- Abhängigkeiten
- MCP SDK
- Entwicklungswerkzeuge
- Konfiguration
- lokaler Start
- reproduzierbares Setup

Einen ersten MCP-Server entwickeln

- Server initialisieren
- Capability bereitstellen
- erstes Tool implementieren
- Server starten
- Client verbinden
- Tool Discovery
- Tool aufrufen
- Request und Response untersuchen

Tools in Python implementieren

- Python-Funktion als Tool
- Name und Beschreibung
- Parameter
- Typinformationen
- Rückgabewerte
- synchrone und asynchrone Funktionen
- Exceptions
- Tool-Implementierung von MCP-spezifischem Code trennen

Tool-Schnittstellen entwerfen

- eine klar definierte Aufgabe pro Tool
- fachliche statt technische Operationen
- geeignete Granularität
- aussagekräftige Namen
- verständliche Beschreibungen
- explizite Parameter
- vorhersehbare Seiteneffekte
- stabile Schnittstellen

Python-Typen und Schemas

- Type Hints
- strukturierte Parameter
- optionale Werte
- Enums
- komplexe Eingaben
- verschachtelte Strukturen
- Validierung
- Schema und Python-Modell konsistent halten

Eingaben validieren

- syntaktische Validierung
- fachliche Validierung
- Wertebereiche
- erlaubte Werte
- Pfade
- Identifikatoren
- ungültige Kombinationen
- Fehler früh abweisen

Strukturierte Ergebnisse

- Text und strukturierte Daten
- Python-Datenstrukturen
- Ergebnisobjekte
- maschinenlesbare Antworten
- Metadaten
- große Ergebnisse
- stabile Ausgabeformate
- nur relevante Informationen zurückgeben

Fehlerbehandlung

- fachliche Fehler
- technische Fehler
- ungültige Eingaben
- externe Systeme
- Timeouts
- Exceptions kapseln
- verständliche Fehlermeldungen
- interne Details nicht unnötig exponieren

Asynchrone Tools

- async und await
- I/O-lastige Operationen
- HTTP Requests
- Datenbankzugriffe
- parallele externe Operationen
- Timeouts
- Fehler
- wann asynchrone Implementierung sinnvoll ist

Resources implementieren

- Resources gegenüber Tools
- statische Informationen
- dynamische Informationen
- Resource URIs
- Resource Templates
- Dokumente
- Konfigurationen
- Datenquellen als Resources modellieren

Prompts bereitstellen

- wiederverwendbare Prompt-Vorlagen
- Parameter
- projektspezifische Arbeitsabläufe
- server-seitige Bereitstellung
- Prompts gegenüber Tools und Resources
- Client-Unterstützung
- geeignete Einsatzfälle
- Prompts nicht für Zugriffskontrolle missbrauchen

Vorhandenen Python-Code integrieren

- bestehende Module
- Libraries
- Services
- Geschäftslogik
- Adapter entwickeln
- MCP-Code dünn halten
- Domänenlogik unabhängig testen
- vorhandene Komponenten wiederverwenden

REST APIs als MCP-Tools

- HTTP Client
- Authentifizierung
- Requests
- Responses
- Parameter abbilden
- Timeouts
- Rate Limits
- API-Fehler in sinnvolle Tool-Fehler übersetzen

Datenbanken anbinden

- Connection Management
- parametrisierte Abfragen
- lesende Tools
- fachlich definierte Abfragen
- strukturierte Ergebnisse
- Transaktionen
- Schreiboperationen
- Datenbankrechte begrenzen

Warum kein Universal-SQL-Tool?

- beliebige Queries
- Zugriff auf unbekannte Tabellen
- Schreiboperationen
- Datenexfiltration
- Schemaänderungen
- Berechtigungen
- fachlich begrenzte Tools
- sichere Alternativen

Dateisystemzugriffe

- Dateien lesen
- Dateien schreiben
- erlaubte Verzeichnisse
- Pfade normalisieren
- Path Traversal verhindern
- Dateitypen
- Größenlimits
- kontrollierte Dateioperationen

Kommandozeilenwerkzeuge kapseln

- bestehende CLI-Tools verwenden
- Prozessaufrufe mit subprocess
- Argumente strukturiert übergeben
- Shell vermeiden, wenn nicht erforderlich
- Exit Codes
- stdout und stderr
- Timeouts
- erlaubte Kommandos begrenzen

MCP-Tools für Entwicklungswerkzeuge

- Build-Werkzeuge
- Testframeworks
- Linter
- statische Analyse
- Repository-Werkzeuge
- Browser-Automatisierung
- vorhandene Engineering-Tools kapseln
- maschinenlesbare Ergebnisse bevorzugen

Toolsets strukturieren

- zusammengehörige Fähigkeiten
- Server mit klarer Verantwortung
- ein großer oder mehrere kleine Server
- Namenskonventionen
- gemeinsame Hilfsfunktionen
- wiederverwendbare Komponenten
- Abhängigkeiten
- Wartbarkeit

Konfiguration

- Konfigurationsdateien
- Umgebungsvariablen
- Entwicklungs- und Produktionsumgebung
- URLs
- Timeouts
- Feature-Konfiguration
- Credentials
- Konfiguration nicht im Code verteilen

Secrets

- API Keys
- Tokens
- Passwörter
- Datenbankzugänge
- Secret Stores
- Umgebungsvariablen
- Secrets nicht an das Modell zurückgeben
- Logging sensibler Werte verhindern

Berechtigungen serverseitig erzwingen

- Modellinstruktionen sind keine Zugriffskontrolle
- Benutzer- und Serviceidentitäten
- Autorisierung
- Tool-spezifische Rechte
- Lese- und Schreibzugriffe
- externe Systeme
- Least Privilege
- Zugriff am tatsächlichen System prüfen

Sichere Tool-Implementierung

- Eingaben grundsätzlich validieren
- Fähigkeiten begrenzen
- sichere Defaults
- kritische Operationen explizit machen
- Seiteneffekte dokumentieren
- Ressourcenlimits
- externe Ergebnisse nicht blind vertrauen
- Defense in Depth

Read und Write trennen

- reine Abfragen
- verändernde Operationen
- unterschiedliche Tool-Namen
- unterschiedliche Rechte
- explizite Schreiboperationen
- unbeabsichtigte Änderungen vermeiden
- reversible Operationen bevorzugen
- kritische Funktionen isolieren

Tool-Komposition und unerwartete Fähigkeiten

- einzeln harmlose Tools
- Kombination mehrerer Fähigkeiten
- Lesen und externes Schreiben
- Dateisystem und Netzwerk
- Informationsflüsse
- indirekte Seiteneffekte
- effektive Gesamtberechtigung
- Toolset als Ganzes analysieren

Unit Tests

- Tool-Funktion ohne MCP testen
- normale Python-Tests
- gültige Eingaben
- ungültige Eingaben
- Grenzfälle
- Fehlerfälle
- externe Abhängigkeiten mocken
- schnelle deterministische Tests

Schema- und Contract-Tests

- angebotene Tool-Schnittstelle
- Parameter
- Typen
- Pflichtfelder
- strukturierte Ergebnisse
- stabile Contracts
- unbeabsichtigte Schnittstellenänderungen
- Regressionen erkennen

MCP-Integrationstests

- Server starten
- Client verbinden
- Fähigkeiten entdecken
- Tool aufrufen
- Ergebnis prüfen
- Resources abrufen
- Fehlerfälle
- vollständigen Kommunikationsweg testen

Tests ohne LLM

- MCP ist eine Softwareschnittstelle
- deterministische Tests bevorzugen
- Modellverhalten vom Serververhalten trennen
- Tool-Logik isolieren
- Fehler reproduzierbar machen
- schnelle Testausführung
- CI-taugliche Tests
- LLM erst für End-to-End-Szenarien einsetzen

End-to-End-Tests mit einem AI-Client

- Server konfigurieren
- Tools sichtbar machen
- typische Aufgaben stellen
- Tool-Auswahl beobachten
- Parameter überprüfen
- Ergebnisse kontrollieren
- Fehlverhalten untersuchen
- End-to-End zusätzlich zu deterministischen Tests

Logging

- Serverstart
- Requests
- Tool-Aufrufe
- Laufzeiten
- Fehler
- externe Systeme
- Korrelationsinformationen
- sensible Informationen filtern

Observability

- Erfolgs- und Fehlerraten
- Laufzeiten
- Timeouts
- externe Abhängigkeiten
- Ressourcennutzung
- auffällige Tool-Nutzung
- Betriebsdiagnose
- Audit-Anforderungen berücksichtigen

Debugging

- Server startet nicht
- Client verbindet sich nicht
- Tool fehlt
- Schema stimmt nicht
- falsche Parameter
- Exceptions
- externe Integration schlägt fehl
- Protokollproblem von Anwendungsfehler unterscheiden

Lokaler Betrieb

- stdio-basierter Server
- Prozesslebenszyklus
- Konfiguration im Client
- mehrere lokale Server
- Log-Ausgabe
- Updates
- Entwicklungswerkzeuge
- typische lokale Einsatzszenarien

Remote-Betrieb

- Netzwerktransport
- Serverprozess
- Authentifizierung
- mehrere Benutzer
- zentrale Konfiguration
- Skalierung
- Logging und Monitoring
- zusätzliche Trust Boundary

Packaging

- Python-Projekt paketieren
- pyproject.toml
- Abhängigkeiten
- Entry Points
- Versionsnummern
- reproduzierbare Installation
- interne Distribution
- Updates

Containerisierung

- MCP-Server im Container
- minimale Images
- Abhängigkeiten
- Konfiguration
- Secrets
- Dateisystem
- Netzwerk
- eingeschränkte Laufzeitumgebung

CI/CD

- Tests automatisch ausführen
- Linting und Typprüfung
- Security Checks
- Packaging
- Container Build
- Versionierung
- Deployment
- MCP-Server wie normale Software behandeln

Versionierung von Tools

- Tool-Schnittstellen als Vertrag
- kompatible Änderungen
- inkompatible Änderungen
- neue Parameter
- geänderte Semantik
- alte Clients
- Migration
- Deprecation

Einen produktionsnahen MCP-Server strukturieren

- dünne MCP-Schicht
- Domänenlogik
- Integrationsadapter
- Konfiguration
- Security
- Tests
- Observability
- klare Modulgrenzen

Praktische Übungen

- Python-Projekt für einen MCP-Server aufsetzen
- ersten Server implementieren
- mehrere typisierte Tools entwickeln
- Eingabevalidierung ergänzen
- strukturierte Ergebnisse zurückgeben
- Resource implementieren
- vorhandene Python-Funktion integrieren
- REST API als Tool kapseln
- Datenzugriff implementieren
- Fehlerfälle behandeln
- Berechtigungen und Grenzen ergänzen
- Unit Tests schreiben
- MCP-Integrationstest erstellen
- Server mit einem AI-Client verwenden
- Server paketieren und für Deployment vorbereiten

03 DURCHFÜHRUNG

Rahmen und Organisation

Dauer	2 Tage
Durchführung	Präsenz oder Live Online, vorzugsweise als Inhouse-Schulung
Schwerpunkt	Inhouse-Schulungen für Unternehmen und Teams
Sprache	Deutsch, englische Fachbegriffe und Dokumentation
Unterlagen	Kursunterlagen, Beispiele und Übungsaufgaben
Technik	Eigener Rechner mit Python, Entwicklungsumgebung und einem MCP-fähigen Client

> **Inhouse-Schulungen**
> Vorhandene Python-Komponenten, APIs, Datenbanken oder interne Werkzeuge können als Grundlage für die MCP-Implementierungen verwendet werden.

04 ZIELGRUPPE

Für wen ist der Kurs gedacht?

Der Kurs richtet sich an Python-Entwickler, AI Engineers, Softwarearchitekten und technische Consultants, die eigene MCP-Server und Tools entwickeln oder vorhandene Python-Systeme über MCP für KI-Anwendungen verfügbar machen möchten.

Er ist insbesondere für Teilnehmer geeignet, die MCP nicht nur konfigurieren, sondern robuste und wartbare Integrationen für reale Entwicklungs- und Unternehmensumgebungen implementieren wollen.

> **Voraussetzungen**
> Sichere Python-Grundkenntnisse und praktische Erfahrung in der Softwareentwicklung werden vorausgesetzt. Die Teilnehmer sollten Funktionen, Klassen, Exceptions, Type Hints, Module und grundlegende Python-Projektstrukturen kennen; Grundkenntnisse zu HTTP, APIs und automatisierten Tests sind hilfreich. Grundlegendes Verständnis von MCP wird vorausgesetzt und zu Beginn kompakt aufgefrischt.

05 LERNZIELE

Was Sie nach dem Kurs können

Nach dem Kurs können die Teilnehmer:

- MCP-Server mit Python strukturiert entwickeln
- Tools mit klaren und typisierten Schnittstellen implementieren
- Eingaben und strukturierte Ergebnisse validieren
- synchrone und asynchrone Tools entwickeln
- Resources und Prompts bereitstellen
- vorhandenen Python-Code in MCP-Server integrieren
- REST APIs und Datenbanken kontrolliert anbinden
- Dateisystem- und CLI-Funktionen sicher kapseln
- Toolsets sinnvoll strukturieren
- Konfiguration und Secrets sauber behandeln
- Berechtigungen serverseitig erzwingen
- gefährliche universelle Tool-Schnittstellen vermeiden
- Unit-, Contract- und Integrationstests entwickeln
- MCP-Server ohne LLM deterministisch testen
- End-to-End-Tests mit einem AI-Client durchführen
- Logging und Observability integrieren
- MCP-Server paketieren und containerisieren
- MCP-Server in CI/CD-Prozesse integrieren
- eine wartbare produktionsnahe MCP-Server-Architektur aufbauen

06 FAQ

Häufige Fragen

Was unterscheidet diesen Kurs vom MCP-Grundlagenkurs?

Der Grundlagenkurs behandelt MCP als Technologie und Integrationsarchitektur: Konzepte, Einsatzmöglichkeiten, Tools, Resources, Clients, Server, Security und grundlegende Implementierung.

Dieser Kurs ist ein Entwicklungsworkshop. Der Schwerpunkt liegt auf der konkreten Implementierung professioneller MCP-Server und Tools mit Python einschließlich Softwarearchitektur, Validierung, Tests, Security, Packaging und Betrieb.

Muss ich den Grundlagenkurs vorher besucht haben?

Nein, sofern die grundlegenden MCP-Konzepte bekannt sind. Teilnehmer, die MCP noch nicht kennen, erhalten zu Beginn einen kompakten Architekturüberblick.

Wie viel wird im Kurs programmiert?

Ein erheblicher Teil des Kurses besteht aus praktischer Entwicklung. Die Teilnehmer bauen schrittweise einen eigenen MCP-Server auf und erweitern ihn um Tools, externe Integrationen, Tests und Sicherheitsmechanismen.

Wird ein bestimmtes MCP-Python-SDK verwendet?

Für die Übungen wird ein aktuelles Python-SDK verwendet. Der Kurs konzentriert sich jedoch nicht ausschließlich auf dessen API, sondern auf die dahinterliegenden Entwurfs- und Implementierungsprinzipien, damit das Wissen auch bei Änderungen des SDKs nutzbar bleibt.

Entwickeln wir eigene Tools?

Ja. Das ist der Kern des Kurses. Von einfachen Python-Funktionen geht es bis zu Tools für externe APIs, Datenbanken, Dateisysteme oder vorhandene Entwicklungswerkzeuge.

Werden Resources und Prompts ebenfalls implementiert?

Ja. Der Schwerpunkt liegt auf Tools, weil dort die meisten Implementierungs- und Sicherheitsfragen entstehen. Resources und Prompts werden ebenfalls praktisch umgesetzt und gegenüber Tools abgegrenzt.

Wird Security praktisch behandelt?

Ja. Eingabevalidierung, Berechtigungen, Secrets, Dateisystemgrenzen und die Gestaltung begrenzter Tool-Schnittstellen werden direkt in der Implementierung berücksichtigt.

Werden die MCP-Server automatisiert getestet?

Ja. Unit-, Contract- und Integrationstests gehören zum Kurs. Ein wichtiger Grundsatz ist, die Serverlogik zunächst deterministisch und ohne Sprachmodell zu testen.

Wird auch Deployment behandelt?

Ja. Packaging, Containerisierung und die Einbindung in CI/CD werden behandelt. Der Schwerpunkt bleibt allerdings auf Entwicklung und Qualität der MCP-Server, nicht auf einer allgemeinen DevOps-Schulung.

Wird ein bestimmter Coding Agent benötigt?

Nein. Ein MCP-fähiger AI-Client wird für End-to-End-Tests verwendet. Die Bedienung eines bestimmten Coding Agents ist nicht Gegenstand dieses Kurses.

Warum dauert der Kurs zwei Tage?

Der Kurs setzt die grundlegenden MCP-Konzepte voraus und kann deshalb unmittelbar in die Implementierung einsteigen. Zwei intensive Tage reichen für die Entwicklung eines vollständigen MCP-Servers einschließlich Tools, externer Integration, Security und automatisierter Tests. Umfangreiche Agentenarchitektur oder allgemeine Python-Grundlagen gehören bewusst nicht in diesen Kurs.

Kann der Kurs mit unseren eigenen Systemen durchgeführt werden?

Ja. Bei Inhouse-Schulungen können vorhandene Python-Komponenten, APIs, Datenbanken oder interne Werkzeuge als Grundlage für die MCP-Implementierungen verwendet werden.

UC IT Service · Ulrich Cuber · kontakt@uc-it.de
https://www.uc-it.de/coaching/mcp-mit-python/