Nutzen Sie das Potenzial von MCP-Servern: Revolutionieren Sie die Netzwerkautomatisierung mit KI-basierten Lösungen

Inhalt

Einleitung

Dieses Dokument beschreibt eine umfassende Referenzarchitektur für den Aufbau von produktionsfähigen Model Context Protocol (MCP)-Servern unter Verwendung von Best Practices der Branche, die durch eine reale Implementierung demonstriert wird, die Cisco Catalyst Center, ServiceNow und andere Unternehmenssysteme integriert. Das MCP stellt einen Paradigmenwechsel in der Interaktion von KI-Systemen mit externen Services und Datenquellen dar. Der Übergang vom Prototyp zur Produktion erfordert jedoch die Implementierung von Mustern der Enterprise-Klasse, einschließlich Authentifizierung, Autorisierung, Überwachung und Skalierbarkeit.

Hintergrundinformationen

Durch die zunehmende Einführung künstlicher Intelligenz (KI) wird die Notwendigkeit robuster, sicherer und skalierbarer Integrationsplattformen immer wichtiger. Herkömmliche Point-to-Point-Integrationen verursachen Wartungsaufwand und Sicherheitsschwachstellen. Das Model Context Protocol (MCP) bietet einen standardisierten Ansatz für die Integration von KI-Systemen, für Produktionsbereitstellungen sind jedoch Funktionen der Enterprise-Klasse erforderlich, die über einfache MCP-Implementierungen hinausgehen.

Dieser Artikel zeigt, wie eine produktionsfähige MCP-Serverplattform erstellt wird, die Folgendes umfasst:

1. Enterprise-Authentifizierung: OpenID Connect (OIDC)-Integration mit Cisco Duo
2. Detaillierte Autorisierung: Richtlinien als Code mit Open Policy Agent (OPA)
3. Sicheres geheimes Management: HashiCorp Vault für Anmeldeinformationen und Konfiguration
4. Umfassende Überwachung: ELK-Stack für Beobachtbarkeit und Fehlerbehebung
5. Workflow-Orchestrierung: Temporal.io für komplexe, langlebige Prozesse
6. Legacy-Integration: REST-API-Proxys für vorhandene Systeme

Warum ist das wichtig?

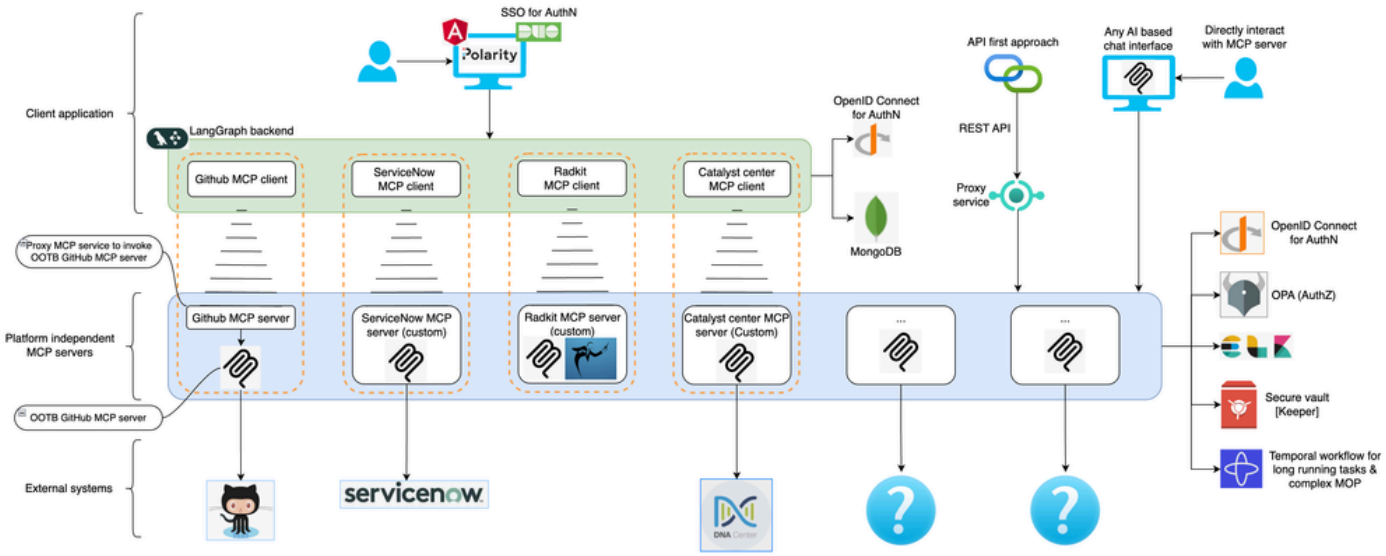
Herkömmliche Integrationsansätze unterliegen verschiedenen Einschränkungen:

1. Sicherheitslücken: Hartcodierte Anmeldedaten und überprivilegierter Zugriff
2. Betriebliche Komplexität: Überwachung und Fehlerbehebung bei verteilten Systemen schwierig
3. Probleme mit der Skalierbarkeit: Point-to-Point-Integration lässt sich nicht mit wachsenden Anforderungen skalieren
4. Wartungsaufwand: Jede Integration erfordert eine benutzerdefinierte Authentifizierung und Fehlerbehandlung.

Der MCP-Ansatz mit Unternehmensmustern bewältigt diese Herausforderungen und bietet gleichzeitig eine standardisierte, wiederverwendbare Grundlage für KI-gesteuerte Automatisierung.

Architekturübersicht

Die Referenzarchitektur implementiert einen mehrschichtigen Ansatz, der Client-Anwendungen von der MCP-Serverplattform trennt, sodass mehrere Anwendungen dieselbe MCP-Infrastruktur der Enterprise-Klasse nutzen können.



Komponentenarchitektur

1. Client-Anwendungsebene

Die Client-Ebene stellt Benutzeroberflächen und Orchestrierungslogik bereit:

- Frontend: Angular-Anwendung mit Cisco Polarity UI-Framework
- Back-End: Multi-Agent-System für die Workflow-Orchestrierung von LangGraph
- Authentifizierung: OIDC-Integration mit Enterprise Identity Providern

2. MCP-Server-Plattformebene

Die Plattformebene implementiert MCP-Server der Enterprise-Klasse mit Shared Services:

Core-MCP-Server:

- `MCP-Catalyst-Center`: Cisco Management von Netzwerkgeräten
- `mcp-service-now`: ITSM-Integration und Ticketverwaltung
- `mcp-github`: Quellcode- und Repository-Management
- `MCP-Radkit`: Netzwerkanalysen und -überwachung
- `mcp-rest-api-proxy`: Legacy-Systemintegration

Enterprise Services:

- Authentifizierungsdienst: OIDC-Tokenvalidierung und Benutzerverwaltung
- Autorisierungsdienst: OPA-basierte Richtliniendurchsetzung
- Geheime Verwaltung: Vault-basierte Anmeldeinformationen und Konfigurationsspeicher
- <Überwachungs-Stack: ELK für Protokollierung, Metriken und Warnungen
- Workflow-Modul: Temporär für komplexe Prozessorchestrierung

Enterprise Security-Implementierung

OpenID Connect-Authentifizierung

Die Plattform implementiert mithilfe von OpenID Connect eine Authentifizierung der Enterprise-Klasse, die eine nahtlose Integration in bestehende Identitätsanbieter ermöglicht und gleichzeitig die mehrstufige Authentifizierung durch Cisco Duo unterstützt.

Wichtigste Vorteile

- Single Sign-On (SSO): Benutzer authentifizieren sich einmal für alle MCP-Services
- Multi-Factor-Authentifizierung: Integriertes Cisco Duo für mehr Sicherheit
- Tokenbasierte Sicherheit: Stateless-Authentifizierung mit JWT-Token
- Zentralisierte Verwaltung: Bereitstellung und Aufhebung der Bereitstellung durch Benutzer über vorhandene IDp

Übersicht der Implementierung

Datei: `mcp-common-app/src/mcp_common/oidc_auth.py`

```
"""OIDC Authentication Module - Enterprise-grade token validation with Vault integration"""

import requests
from typing import Dict, Any, Optional
from fastapi import HTTPException

def get_oidc_config_from_vault() -> Dict[str, Any]:
    """Retrieve OIDC configuration from Vault with caching."""
    vault_client = get_vault_client_with_retry()
    config = vault_client.get_secret("oidc/config")

    if not config:
        raise ValueError("OIDC configuration not found in Vault")

    # Validate required fields
    required_fields = ["issuer", "client_id", "user_info_endpoint"]
    missing_fields = [field for field in required_fields if field not in config]

    if missing_fields:
        raise ValueError(f"Missing required OIDC config fields: {missing_fields}")

    return config
```

```
def verify_token_with_oidc(token: str) -> Dict[str, Any]:
    """Verify OIDC token and extract user information."""
    config = get_oidc_config_from_vault()

    response = requests.get(
        config["user_info_endpoint"],
        headers={"Authorization": f"Bearer {token}"},
        timeout=10
    )

    if response.status_code == 200:
        user_info = response.json()
        if "sub" not in user_info:
            raise HTTPException(status_code=401, detail="Invalid token: missing subject")
        return user_info
    else:
        raise HTTPException(status_code=401, detail="Token validation failed")
```

Detaillierte Autorisierung mit Open Policy Agent

OPA ermöglicht eine flexible, richtlinienbasierte Autorisierung, die eine detaillierte Zugriffskontrolle auf der Grundlage von Benutzerattributen, Ressourcentypen und Kontextinformationen ermöglicht.

Struktur der Autorisierungsrichtlinie

Datei: `common-services/opa/config/policy.rego`

```
# Authorization Policy for MCP Server Platform - RBAC Implementation
package authz

default allow = false

# Administrative access - full permissions
allow {
    group := input.groups[_]
    group == "admin"
}

# Network engineers - Catalyst Center access
allow {
    group := input.groups[_]
    group == "network-engineers"
    input.resource == "catalyst-center"
    allowed_actions := ["read", "write", "execute"]
    allowed_actions[_] == input.action
}

# Service desk - ServiceNow and read-only network access
allow {
    group := input.groups[_]
    group == "service-desk"
    input.resource in ["servicenow", "catalyst-center"]
    input.resource == "servicenow" or input.action == "read"
}
```

```
# Developers - GitHub and REST API proxy access
allow {
    group := input.groups[_]
    group == "developers"
    input.resource in ["github", "rest-api-proxy"]
}
```

Python-OPA-Integration

<Datei: `mcp-common-app/src/mcp_common/opa.py`

```
"""OPA Integration - Centralized authorization with audit logging"""

import os
import json
import requests
from typing import List, Dict, Any
from dataclasses import dataclass

@dataclass
class AuthorizationRequest:
    """Structure for authorization requests to OPA."""
    user_groups: List[str]
    resource: str
    action: str
    context: Dict[str, Any] = None

class OPAClient:
    """Client for interacting with Open Policy Agent (OPA) for authorization decisions."""

    def __init__(self, opa_addr: str = None):
        self.opa_addr = opa_addr or os.getenv("OPA_ADDR", "http://opa:8181")
        self.opa_url = f"{self.opa_addr}/v1/data/authz/allow"

    def check_permission(self, auth_request: AuthorizationRequest) -> bool:
        """Check if a user has permission to perform an action on a resource."""
        try:
            opa_input = {
                "input": {
                    "groups": auth_request.user_groups,
                    "resource": auth_request.resource,
                    "action": auth_request.action
                }
            }

            if auth_request.context:
                opa_input["input"]["context"] = auth_request.context

            response = requests.post(self.opa_url, json=opa_input, timeout=5)

            if response.status_code == 200:
                result = response.json()
                allowed = result.get("result", False)
                self._audit_log(auth_request, allowed)
                return allowed
            else:
                print(f"OPA authorization check failed: {response.status_code}")
```

```
                return False  # Fail secure

        except requests.RequestException as e:
            print(f"OPA connection error: {e}")
            return False  # Fail secure

    def _audit_log(self, auth_request: AuthorizationRequest, allowed: bool):
        """Log authorization decisions for audit purposes."""
        log_entry = {
            "user_groups": auth_request.user_groups,
            "resource": auth_request.resource,
            "action": auth_request.action,
            "allowed": allowed
        }
        print(f"Authorization Decision: {json.dumps(log_entry)}")

# Usage decorator for MCP server methods
def require_permission(resource: str, action: str):
    """Decorator for MCP server methods that require authorization."""
    def decorator(func):
        async def wrapper(self, *args, **kwargs):
            user_groups = getattr(self, 'user_groups', [])
            if not user_groups:
                raise Exception("User groups not found in request context")

            opa_client = OPAClient()
            auth_request = AuthorizationRequest(
                user_groups=user_groups, resource=resource, action=action
            )

            if not opa_client.check_permission(auth_request):
                raise Exception(f"Access denied for {action} on {resource}")

            return await func(self, *args, **kwargs)
        return wrapper
    return decorator
```

Sicheres geheimes Management mit HashiCorp Vault

HashiCorp Vault bietet ein geheimes Management der Enterprise-Klasse mit Verschlüsselung, Zugriffskontrolle und Audit-Protokollierung. Die MCP-Plattform integriert Vault zum sicheren Speichern und Abrufen vertraulicher Informationen wie API-Anmeldeinformationen, Datenbankkennwörter und Konfigurationsdaten.

Wichtigste Funktionen

- Verschlüsselung im Ruhe- und Transit: Alle Geheimnisse werden mit AES 256-Bit-Verschlüsselung verschlüsselt
- Dynamische Geheimnisse: Generieren zeitlich begrenzter Anmeldedaten für externe Services
- Zugriffskontrolle: Detaillierte Richtlinien steuern, wer auf welche Geheimnisse zugreifen kann
- Audit-Protokollierung: Vollständiger Prüfpfad für alle geheimen Zugriffsvorgänge
- Geheime Drehung: Automatisierte Rotation von Zertifikaten und Zertifikaten

Implementierung

Datei: mcp-common-app/src/mcp_common/vault.py

```
"""HashiCorp Vault Integration - Secure secret management with audit logging"""

import os
import json
import requests
from typing import Dict, Any, Optional, List
from datetime import datetime

class VaultClient:
    """Enterprise HashiCorp Vault client for secure secret management."""

    def __init__(self, vault_addr: str = None, vault_token: str = None,
                 mount_point: str = "secret"):
        self.vault_addr = vault_addr or os.getenv("VAULT_ADDR", "http://vault:8200")
        self.vault_token = vault_token or os.getenv("VAULT_TOKEN")
        self.mount_point = mount_point
        self.headers = {"X-Vault-Token": self.vault_token}

        if not self.vault_token:
            raise ValueError("Vault token must be provided or set in VAULT_TOKEN")

    def set_secret(self, path: str, secret_data: Dict[str, Any]) -> bool:
        """Store a secret in Vault KV store."""
        try:
            response = requests.post(
                f"{self.vault_addr}/v1/{self.mount_point}/data/{path}",
                headers=self.headers,
                json={"data": secret_data},
                timeout=10
            )

            success = response.status_code in [200, 204]
            self._audit_log("set_secret", path, success)
            return success

        except requests.RequestException as e:
            self._audit_log("set_secret", path, False, error=str(e))
            return False

    def get_secret(self, path: str) -> Optional[Dict[str, Any]]:
        """Retrieve a secret from Vault KV store."""
        try:
            response = requests.get(
                f"{self.vault_addr}/v1/{self.mount_point}/data/{path}",
                headers=self.headers,
                timeout=10
            )

            success = response.status_code == 200
            self._audit_log("get_secret", path, success)

            if success:
                return response.json()["data"]["data"]
            return None
```

```
        except requests.RequestException as e:
            self._audit_log("get_secret", path, False, error=str(e))
            return None

    def _audit_log(self, operation: str, path: str, success: bool, error: str = None):
        """Log secret operations for audit purposes."""
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "operation": operation,
            "path": f"{self.mount_point}/{path}",
            "success": success
        }
        if error:
            log_entry["error"] = error

        print(f"Vault Audit: {json.dumps(log_entry)}")

# Usage mixin for MCP servers
class MCPSecretMixin:
    """Mixin class for MCP servers to easily access Vault secrets."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._vault_client = None

    @property
    def vault_client(self) -> VaultClient:
        if self._vault_client is None:
            self._vault_client = VaultClient()
        return self._vault_client

    def get_api_credentials(self, service_name: str) -> Optional[Dict[str, Any]]:
        """Get API credentials for a specific service."""
        return self.vault_client.get_secret(f"api/{service_name}")
```

Core-MCP-Serverstruktur

Jeder MCP-Server enthält ein konsistentes Muster mit Enterprise Security-Integration:

Datei: `mcp-catalyst-center/src/main.py`

```
"""Cisco Catalyst Center MCP Server - Enterprise implementation"""

from mcp_common import VaultClient, OPAClient, get_logger, require_permission
from fastmcp import FastMCP
import os

app = FastMCP("Cisco Catalyst Center MCP Server")
logger = get_logger(__name__)

# Initialize enterprise services
vault_client = VaultClient()
opa_client = OPAClient()

@app.tool()
@require_permission("catalyst-center", "read")
async def get_all_templates(request) -> str:
```

```
    """Fetch all configuration templates from Catalyst Center."""

    # Get credentials from Vault
    credentials = vault_client.get_secret("api/catalyst-center")
    if not credentials:
        raise Exception("Catalyst Center credentials not found")

    try:
        # API call implementation
        templates = await fetch_templates_from_api(credentials)
        logger.info(f"Retrieved {len(templates)} templates")

        return {
            "templates": templates,
            "status": "success",
            "count": len(templates)
        }

    except Exception as e:
        logger.error(f"Failed to fetch templates: {e}")
        raise Exception(f"Template fetch failed: {str(e)}")

@app.tool()
@require_permission("catalyst-center", "write")
async def deploy_template(template_id: str, device_id: str) -> str:
    """Deploy configuration template to network device."""
    credentials = vault_client.get_secret("api/catalyst-center")

    # Implementation details...
    logger.info(f"Deployed template {template_id} to device {device_id}")
    return {"status": "deployed", "template_id": template_id, "device_id": device_id}
```

REST-API-Proxy für Legacy-Integration

Die Plattform umfasst einen REST-API-Proxy zur Unterstützung von Nicht-MCP-Clients:

Datei: `mcp-rest-api-proxy/main.py`

```
"""REST API Proxy - Bridge between REST clients and MCP servers"""

from fastapi import FastAPI, HTTPException, Request
from langchain_mcp_adapters.client import MultiServerMCPClient

app = FastAPI()

# MCP server configurations
MCP_SERVERS = {
    "servicenow": "http://mcp-servicenow:8080/mcp/",
    "catalyst-center": "http://mcp-catalyst-center:8002/mcp/",
    "github": "http://mcp-github:8000/mcp/"
}

client = MultiServerMCPClient({
    server_name: {"url": url, "transport": "streamable_http"}
    for server_name, url in MCP_SERVERS.items()
})
```

```
@app.post("/api/v1/mcp/{server_name}/tools/{tool_name}")
async def execute_tool(server_name: str, tool_name: str, request: Request):
    """Execute MCP tool via REST API for legacy clients."""
    try:
        body = await request.json()

        result = await client.call_tool(
            server_name=server_name,
            tool_name=tool_name,
            arguments=body.get("arguments", {})
        )

        return {
            "status": "success",
            "result": result,
            "server": server_name,
            "tool": tool_name
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Tool execution failed: {str(e)}")

@app.get("/api/v1/mcp/{server_name}/tools")
async def list_tools(server_name: str):
    """List available tools for a specific MCP server."""
    tools = await client.list_tools(server_name)
    return {"server": server_name, "tools": tools}
```

Überwachung und Beobachtbarkeit

ELK-Stack-Integration

Die Plattform implementiert eine umfassende Protokollierung mithilfe des ELK-Stacks:

Datei: `mcp-common-app/src/mcp_common/logger.py`

```
"""Structured Logging for ELK Stack Integration"""

import logging
import json
from datetime import datetime
from pythonjsonlogger import jsonlogger

class StructuredLogger:
    def __init__(self, name: str, level: str = "INFO"):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(getattr(logging, level.upper()))

        # JSON formatter for ELK ingestion
        formatter = jsonlogger.JsonFormatter(
            fmt='%(asctime)s %(name)s %(levelname)s %(message)s'
        )

        handler = logging.StreamHandler()
        handler.setFormatter(formatter)
```

```
        self.logger.addHandler(handler)

    def log_mcp_call(self, tool_name: str, user: str, duration: float, status: str):
        """Log MCP tool invocation with structured data."""
        self.logger.info("MCP tool executed", extra={
            "tool_name": tool_name,
            "user": user,
            "duration_ms": duration,
            "status": status,
            "service_type": "mcp_server"
        })

def get_logger(name: str) -> StructuredLogger:
    """Get configured logger instance."""
    return StructuredLogger(name)
```

Wichtige Metriken zur Überwachung

Die Plattform verfolgt wichtige Kennzahlen für den Unternehmensbetrieb nach:

- Anforderungslatenz: Ausführungszeit und Perzentile pro Tool
- Authentifizierungsmetriken: Erfolgs-/Fehlerquoten und Reaktionszeiten
- Genehmigungsentscheidungen: Häufigkeit und Ergebnisse der Politikbewertung
- Geheimer Zugriff: Nutzungsmuster für Vault-Vorgänge und Zertifikate
- Fehlerquoten: Schwellenwerte für die Nachverfolgung von Fehlern und Warnungen auf Service-Ebene
- Ressourcenauslastung: CPU-, Arbeitsspeicher- und Netzwerknutzung pro Service

Integration von temporären Workflows

Für komplexe, langwierige Prozesse nutzt die Plattform Temporal.io:

Datei: `temporal-service/src/workflows/template_deployment.py`

```
"""Template Deployment Workflow - Orchestrated automation with error handling"""

from temporalio import workflow, activity
from datetime import timedelta

@workflow.defn
class TemplateDeploymentWorkflow:
    @workflow.run
    async def run(self, deployment_request: dict) -> dict:
        """Orchestrate template deployment with proper error handling."""

        # Step 1: Validate template and device
        validation_result = await workflow.execute_activity(
            validate_deployment, deployment_request,
            start_to_close_timeout=timedelta(minutes=5)
        )

        if not validation_result["valid"]:
```

```
            return {"status": "failed", "reason": "Validation failed"}

        # Step 2: Create ServiceNow ticket
        ticket_result = await workflow.execute_activity(
            create_servicenow_ticket, validation_result,
            start_to_close_timeout=timedelta(minutes=2)
        )

        # Step 3: Deploy template
        deployment_result = await workflow.execute_activity(
            deploy_template, {
                **deployment_request,
                "ticket_id": ticket_result["ticket_id"]
            },
            start_to_close_timeout=timedelta(minutes=30)
        )

        # Step 4: Close ticket
        await workflow.execute_activity(
            close_servicenow_ticket, {
                "ticket_id": ticket_result["ticket_id"],
                "deployment_result": deployment_result
            },
            start_to_close_timeout=timedelta(minutes=2)
        )

        return {
            "status": "completed",
            "ticket_id": ticket_result["ticket_id"],
            "deployment_id": deployment_result["deployment_id"]
        }

@activity.defn
async def validate_deployment(request: dict) -> dict:
    """Validate deployment request against business rules."""
    # Validation logic implementation
    return {"valid": True, "validated_request": request}

@activity.defn
async def deploy_template(request: dict) -> dict:
    """Execute template deployment via Catalyst Center."""
    # Template deployment logic
    return {"deployment_id": "deploy_123", "status": "success"}
```

Bereitstellung und Skalierbarkeit

Containerorchestrierung

Die Plattform verwendet Docker Compose für die Entwicklung. Kubernetes kann für die Produktion verwendet werden:

Datei: `docker-compose.yml` (Auszug)

```
version: '3.8'
services:
```

```
  mcp-catalyst-center:
    build: ./mcp-catalyst-center
    environment:
      - VAULT_ADDR=http://vault:8200
      - OPA_ADDR=http://opa:8181
      - ELASTICSEARCH_URL=http://elasticsearch:9200
    depends_on: [vault, opa, elasticsearch]
    networks: [mcp-network]

  vault:
    image: hashicorp/vault:latest
    environment:
      VAULT_DEV_ROOT_TOKEN_ID: myroot
      VAULT_DEV_LISTEN_ADDRESS: 0.0.0.0:8200
    cap_add: [IPC_LOCK]
    networks: [mcp-network]

  opa:
    image: openpolicyagent/opa:latest-envoy
    command: ["run", "--server", "--config-file=/config/config.yaml", "/policies"]
    volumes: ["./common-services/opa/config:/policies"]
    networks: [mcp-network]
```

Überlegungen zu Leistung und Sicherheit

Best Practices für die Sicherheit

1. Zero-Trust-Architektur: Jede Anforderung wird authentifiziert und autorisiert.
2. Geheime Drehung: Automatisierte geheime Rotation über Tresor
3. Netzwerksegmentierung: Service-Mesh mit mTLS
4. Audit-Protokollierung: Umfassender Prüfpfad in ELK

Leistungsoptimierung

1. Verbindungspooling: Wiederverwendung von HTTP-Verbindungen zu externen APIs
2. Zwischenspeicherung: Redis-basiertes Caching für häufig genutzte Daten
3. Asynchrone Verarbeitung: Non-Blocking-E/A im gesamten Stack
4. Lastenausgleich: Lastverteilung auf mehrere MCP-Serverinstanzen

Metriken überwachen

Zu den wichtigsten verfolgten Metriken gehören:

- Anforderungslatenz pro MCP-Tool
- Erfolgs-/Fehlerraten der Authentifizierung
- Autorisierungsentscheidungen pro Ressource
- Häufigkeit des geheimen Abrufs
- Fehlerquoten nach Servicekomponente

Leistungsmetriken und -ergebnisse

Während der Leistungstests erzielte die Plattform folgende Ergebnisse:

- Latenz unter 100 ms für Authentifizierungsentscheidungen
- 99,9 % Verfügbarkeit für alle MCP-Services
- Lineare Skalierbarkeit bis zu 1000 gleichzeitige Benutzer
- 90 % kürzere Integrationsentwicklungszeit

Erkenntnisse und Best Practices

Wichtigste Erfolgsfaktoren

1. Standardisierung: Gemeinsame Bibliotheken reduzieren Duplizierung und Fehler
2. Beobachtbarkeit: Umfassende Protokollierung für schnelle Fehlerbehebung
3. Sicherheit durch Design: Authentifizierung und Autorisierung von Anfang an
4. Modularität: Unabhängige MCP-Server ermöglichen eine gezielte Skalierung
5. Dokumentation: Eine klare API-Dokumentation beschleunigt die Einführung

Häufige Stolpersteine

1. Überentwicklung: Einfacher Einstieg, bei Bedarf Erhöhung der Komplexität
2. Enge Kopplung: Lockere Anbindung von MCP-Servern
3. Sicherheit im Nachhinein: Sicherheit von Anfang an
4. Unzureichende Tests: Implementierung umfassender Teststrategien
5. Schlechte Fehlerbehandlung: Gewährleistung einer sanften Verschlechterung

Künftige Erweiterungen

Die Plattform-Roadmap umfasst:

1. KI-gestützte Erkenntnisse: ML-basierte Erkennung von Anomalien
2. Multi-Cloud-Unterstützung: Bereitstellung über Cloud-Anbieter hinweg
3. Workflow-Marktplatz: Wiederverwendbare Workflow-Vorlagen
4. Erweiterte Analysen: Dashboards und Berichte in Echtzeit

Schlussfolgerung

Bei der Erstellung von MCP-Servern der Produktionsklasse müssen die Unternehmensanforderungen, einschließlich Sicherheit, Skalierbarkeit, Überwachung und Wartungsfreundlichkeit, sorgfältig berücksichtigt werden. Diese Referenzarchitektur zeigt, wie diese Funktionen mithilfe von branchenüblichen Tools und Mustern implementiert werden.

Das modulare Design ermöglicht die schrittweise Einführung von MCP und gewährleistet gleichzeitig Sicherheit und Betrieb der Enterprise-Klasse vom ersten Tag an. Durch den Einsatz bewährter Technologien wie OIDC, OPA, Vault und ELK können sich Teams auf die Geschäftslogik statt auf Infrastrukturprobleme konzentrieren.

Informationen zu den Autoren

Dieser Artikel wurde vom MCP Fusioners-Team im Rahmen der internen Innovationsinitiativen von Cisco entwickelt und zeigt praktische Ansätze für die Integration von KI-Systemen für Unternehmen auf.

Referenzen

1. Model Context Protocol-Spezifikation - https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. Open Policy Agent-Dokumentation - https://www.openpolicyagent.org/docs
4. HashiCorp Vault-Dokumentation - https://www.vaultproject.io/docs
5. Dokumentation für Temporal.io - https://docs.temporal.io/
6. ELK Stack-Leitfaden: https://www.elastic.co/elastic-stack/

Informationen zu dieser Übersetzung

Cisco hat dieses Dokument maschinell übersetzen und von einem menschlichen Übersetzer editieren und korrigieren lassen, um unseren Benutzern auf der ganzen Welt Support-Inhalte in ihrer eigenen Sprache zu bieten. Bitte beachten Sie, dass selbst die beste maschinelle Übersetzung nicht so genau ist wie eine von einem professionellen Übersetzer angefertigte. Cisco Systems, Inc. übernimmt keine Haftung für die Richtigkeit dieser Übersetzungen und empfiehlt, immer das englische Originaldokument (siehe bereitgestellter Link) heranzuziehen.