Maak gebruik van de kracht van MCP-servers: maak netwerkautomatisering revolutionair met AI-gestuurde oplossingen

Inhoud

Referenties

Inleiding

Dit document beschrijft een uitgebreide referentiearchitectuur voor het bouwen van productierijpe Model Context Protocol (MCP)-servers met behulp van best practices uit de sector, aangetoond door middel van een implementatie in de praktijk waarbij Cisco Catalyst Center, ServiceNow en andere bedrijfssystemen worden geïntegreerd. De MCP vertegenwoordigt een paradigmaverschuiving in de manier waarop AI-systemen omgaan met externe diensten en gegevensbronnen. De overgang van prototype naar productie vereist echter het implementeren van bedrijfspatronen, waaronder authenticatie, autorisatie, bewaking en schaalbaarheid.

Achtergrondinformatie

Naarmate organisaties steeds meer AI-gestuurde automatisering toepassen, wordt de behoefte aan robuuste, veilige en schaalbare integratieplatforms van cruciaal belang. Traditionele point-to-point integraties zorgen voor onderhouds- en beveiligingsproblemen. Het Model Context Protocol (MCP) biedt een gestandaardiseerde benadering van AI-systeemintegraties, maar voor productie-implementaties zijn bedrijfsspecifieke mogelijkheden nodig die verder gaan dan basis-MCP-implementaties.

Dit artikel laat zien hoe u een productierijp MCP-serverplatform kunt bouwen dat het volgende omvat:

1. Enterprise Authentication: OpenID Connect (OIDC)-integratie met Cisco Duo
2. Fijnkorrelige autorisatie: policy-as-code met behulp van Open Policy Agent (OPA)
3. Secure Secret Management: HashiCorp Vault voor inloggegevens en configuratie
4. Uitgebreide bewaking: ELK Stack voor observeerbaarheid en probleemoplossing
5. Workflow Orchestration: Temporal.io voor complexe, langlopende processen
6. Legacy Integration: REST API-proxies voor bestaande systemen

Waarom dit belangrijk is

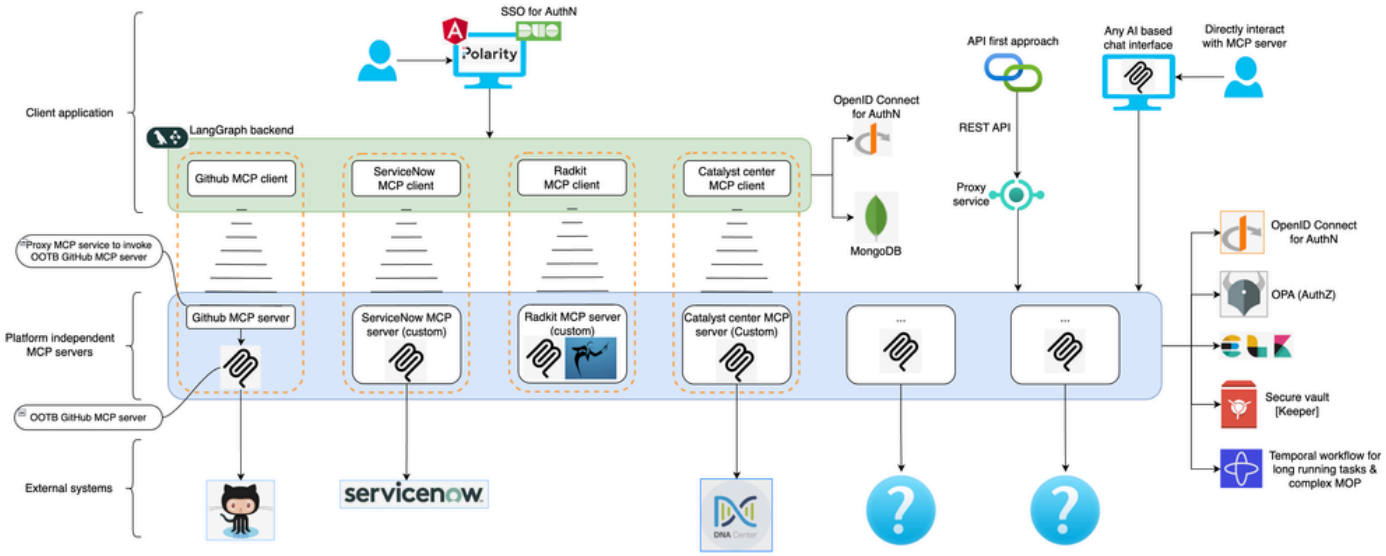
Traditionele integratiebenaderingen hebben te kampen met verschillende beperkingen:

1. Beveiligingsleemten: hard gecodeerde referenties en toegang met te veel rechten
2. Operationele complexiteit: moeilijk te controleren en problemen met gedistribueerde systemen op te lossen
3. Problemen met schaalbaarheid: Point-to-point-integraties worden niet geschaald uitgebreid met groeiende vereisten
4. Onderhoudskosten: voor elke integratie zijn aangepaste verificatie en foutafhandeling vereist

De MCP-aanpak met bedrijfspatronen pakt deze uitdagingen aan en biedt tegelijkertijd een gestandaardiseerde, herbruikbare basis voor AI-gedreven automatisering.

Architectuuroverzicht

De referentiearchitectuur implementeert een gelaagde benadering die clienttoepassingen scheidt van het MCP-serverplatform, waardoor meerdere toepassingen dezelfde MCP-infrastructuur van bedrijfsniveau kunnen benutten.



componentarchitectuur

1. Clienttoepassingslaag

De clientlaag biedt gebruikersinterfaces en orkestratielogica:

- Frontend: Angulaire toepassing met Cisco Polarity UI-framework
- Backend: LangGraph multi-agentsysteem voor workfloworkestratie
- Authenticatie: OIDC-integratie met aanbieders van bedrijfsidentiteiten

2. MCP-serverplatformlaag

De platformlaag implementeert MCP-servers van bedrijfsniveau met gedeelde services:

Core MCP-servers:

- `mcp-catalyst-center`: Cisco-netwerkapparaatbeheer
- `mcp-service-now`: ITSM-integratie en ticketbeheer
- `mcp-github`: Broncode en opslagplaatsbeheer
- `mcp-radkit`: netwerkanalyse en -monitoring
- `mcp-rest-api-proxy`: integratie van oudere systemen

Enterprise Services:

- Authenticatieservice: OIDC-tokenvalidatie en gebruikersbeheer
- Autorisatieservice: op OPA gebaseerde beleidshandhaving
- Geheim beheer: aanmeldingsgegevens en configuratieopslag op basis van een vault
- <Monitoring Stack: ELK voor logging, statistieken en waarschuwingen

- Workflow Engine: tijdelijk voor complexe procesorkestratie

Implementatie van bedrijfsbeveiliging

OpenID Connect-verificatie

Het platform implementeert verificatie op bedrijfsniveau met behulp van OpenID Connect, biedt naadloze integratie met bestaande identiteitsproviders en ondersteunt multi-factor authenticatie via Cisco Duo.

Belangrijkste voordelen

- Single Sign-On (SSO): gebruikers verifiëren eenmaal voor alle MCP-services
- Multi-Factor Authentication: geïntegreerd Cisco Duo voor verbeterde beveiliging
- Tokengebaseerde beveiliging: stateloze verificatie met JWT-tokens
- Gecentraliseerd beheer: provisioning en deprovisioning van gebruikers via bestaande IdP

Implementatieoverzicht

Bestand: `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")
```

Fijnkorrelige autorisatie met Open Policy Agent

OPA biedt flexibele, policy-as-code-autorisatie die fijnmazige toegangscontrole mogelijk maakt op basis van gebruikerskenmerken, brontypen en contextuele informatie.

structuur van het machtigingsbeleid

Bestand: `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-integratie

<Bestand: 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
```

Beveiligd geheim beheer met HashiCorp Vault

HashiCorp Vault biedt geheim beheer op bedrijfsniveau met codering, toegangscontrole en auditregistratie. Het MCP-platform integreert Vault om gevoelige informatie, waaronder API-referenties, databasewachtwoorden en configuratiegegevens, veilig op te slaan en op te halen.

Belangrijkste kenmerken

- Encryptie in rust en in transit: alle geheimen worden gecodeerd met behulp van AES 256-bits codering
- Dynamische geheimen: Credentials voor externe services genereren met beperkte tijd
- Toegangscontrole: Fijnkorrelig beleid bepaalt wie toegang heeft tot welke geheimen
- Controleregistratie: volledig controlespoor van alle geheime toegangsbewerkingen
- Geheime rotatie: geautomatiseerde rotatie van referenties en certificaten

uitvoering

Bestand: `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-serverstructuur

Elke MCP-server bevat een consistent patroon met integratie van bedrijfsbeveiliging:

Bestand: `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 voor integratie met oudere systemen

Het platform bevat een REST API-proxy om niet-MCP-clients te ondersteunen:

Bestand: `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}
```

Monitoring en observeerbaarheid

ELK Stack Integration

Het platform implementeert uitgebreide logboekregistratie met behulp van de ELK-stack:

Bestand: `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)
```

Belangrijkste controlestatistieken

Het platform volgt essentiële statistieken voor bedrijfsactiviteiten:

- Latentie aanvragen: uitvoertijd en percentielen per tool
- Verificatiegegevens: succespercentages/mislukkingen en responstijden
- Machtigingsbesluiten: frequentie en resultaten van de beleidsevaluatie
- Geheime toegang: vaultbewerkingen en gebruikspatronen voor referenties
- Foutpercentages: drempelwaarden voor het volgen van fouten en waarschuwingen op serviceniveau
- Bronnengebruik: CPU-, geheugen- en netwerkgebruik per service

Integratie van temporele workflow

Voor complexe, langlopende processen maakt het platform gebruik van Temporal.io:

Bestand: `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"}
```

Implementatie en schaalbaarheid

Containerorchestratie

Het platform gebruikt Docker Compose voor ontwikkeling. Kubernetes kan worden gebruikt voor de productie:

Bestand: `docker-compose.yml` (fragment)

```
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]
```

Prestatie- en veiligheidsoverwegingen

Best practices voor beveiliging

1. Zero-Trust Architecture: elk verzoek is geverifieerd en geautoriseerd
2. Geheime rotatie: geautomatiseerde geheime rotatie via Vault
3. Netwerksegmentatie: servicemassage met mTLS
4. Controleregistratie: uitgebreid controlespoor in ELK

Prestatieoptimalisaties

1. Verbindingspooling: HTTP-verbindingen hergebruiken voor externe API's
2. Caching: Redis-gebaseerde caching voor veelgebruikte gegevens
3. Async-verwerking: niet-blokkerende I/O in de hele stapel
4. Load Balancing: verdeel de belasting over meerdere MCP-serverinstanties

Monitoringstatistieken

Belangrijke bijgehouden statistieken zijn onder meer:

- Latentie aanvragen per MCP-tool
- Succespercentages voor verificatie/mislukkingen
- Machtigingsbesluiten per middel
- Geheime ophaalfrequentie
- Foutenpercentages per servicecomponent

Prestatiegegevens en -resultaten

Tijdens het testen van de prestaties bereikte het platform:

- Sub-100ms latentie voor authenticatiebeslissingen
- 99,9% uptime voor alle MCP-services
- Lineaire schaalbaarheid tot 1000 gelijktijdige gebruikers
- 90% reductie in integratieontwikkelingstijd

Geleerde lessen en best practices

Belangrijke succesfactoren

1. Standaardisatie: gemeenschappelijke bibliotheken verminderen duplicatie en bugs
2. Observeerbaarheid: uitgebreide logboekregistratie maakt snelle probleemoplossing mogelijk
3. Security by Design: authenticatie en autorisatie vanaf dag één
4. Modulariteit: onafhankelijke MCP-servers maken gerichte schaling mogelijk
5. Documentatie: duidelijke API-documentatie versnelt adoptie

Gemeenschappelijke valkuilen te vermijden

1. Over-Engineering: eenvoudig starten en complexiteit toevoegen indien nodig
2. Strakke koppeling: houd MCP-servers losjes gekoppeld
3. Security Afterthought: beveiliging vanaf het begin inbouwen
4. Onvoldoende testen: uitgebreide teststrategieën implementeren
5. Slechte foutafhandeling: zorg voor een gracieuze degradatie

Toekomstige verbeteringen

De routekaart voor het platform omvat:

1. AI-Driven Insights: op ML gebaseerde anomaliedetectie
2. Multi-Cloud Support: implementatie bij verschillende cloudproviders
3. Workflow Marketplace: herbruikbare werkstroomsjablonen
4. Geavanceerde analyse: realtime dashboards en rapportage

Conclusie

Voor het bouwen van MCP-servers die geschikt zijn voor de productie moet zorgvuldig rekening worden gehouden met bedrijfsvereisten, zoals beveiliging, schaalbaarheid, bewaking en onderhoudbaarheid. Deze referentiearchitectuur laat zien hoe deze mogelijkheden kunnen worden geïmplementeerd met behulp van industriestandaard tools en patronen.

Het modulaire ontwerp stelt organisaties in staat om MCP geleidelijk over te nemen en tegelijkertijd vanaf de eerste dag beveiliging en activiteiten op bedrijfsniveau te garanderen. Door gebruik te maken van bewezen technologieën zoals OIDC, OPA, Vault en ELK kunnen teams zich richten op bedrijfslogica in plaats van op infrastructurele problemen.

Over de auteurs

Dit artikel is ontwikkeld door het MCP Fusioners-team als onderdeel van de interne innovatie-initiatieven van Cisco, die praktische benaderingen van enterprise AI-systeemintegratie demonstreren.

Referenties

1. Specificatie van het modelcontextprotocol - https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. Documentatie van Open Policy Agent - https://www.openpolicyagent.org/docs
4. HashiCorp Vault-documentatie - https://www.vaultproject.io/docs
5. Documentatie van Temporal.io - https://docs.temporal.io/
6. ELK Stack Guide - https://www.elastic.co/elastic-stack/

Over deze vertaling

Cisco heeft dit document vertaald via een combinatie van machine- en menselijke technologie om onze gebruikers wereldwijd ondersteuningscontent te bieden in hun eigen taal. Houd er rekening mee dat zelfs de beste machinevertaling niet net zo nauwkeurig is als die van een professionele vertaler. Cisco Systems, Inc. is niet aansprakelijk voor de nauwkeurigheid van deze vertalingen en raadt aan altijd het oorspronkelijke Engelstalige document (link) te raadplegen.