MCP 서버의 기능 활용: AI 기반 솔루션으로 네트워크 자동화 혁신

목차

소개

이 문서에서는 Cisco Catalyst Center, ServiceNow 및 기타 엔터프라이즈 시스템을 통합하는 실제 구현을 통해 입증된 업계 모범 사례를 사용하여 프로덕션 지원 MCP(Model Context Protocol) 서버를 구축하기 위한 포괄적인 참조 아키텍처에 대해 설명합니다. MCP는 AI 시스템이 외부 서비스 및 데이터 소스와 상호 작용하는 방식의 패러다임 변화를 나타냅니다. 그러나 프로토타입에서 프로덕션으로 전환하려면 인증, 권한 부여, 모니터링, 확장성과 같은 엔터프라이즈급 패턴을 구현해야 합니다.

배경 정보

AI 기반 자동화를 채택하는 기업이 늘어남에 따라, 강력하고 안전하며 확장 가능한 통합 플랫폼에 대한 필요성이 더욱 중요해지고 있습니다. 기존의 포인트-투-포인트 통합으로 유지 관리 오버헤드와 보안 취약성이 발생합니다. MCP(Model Context Protocol)는 AI 시스템 통합에 대한 표준화된 접근 방식을 제공하지만 프로덕션 구축에는 기본적인 MCP 구현을 뛰어넘는 엔터프라이즈급 기능이 필요합니다.

이 문서에서는 다음을 포함하는 프로덕션 지원 MCP 서버 플랫폼을 구축하는 방법을 설명합니다.

1. 엔터프라이즈 인증: Cisco Duo와의 OIDC(OpenID Connect) 통합
2. 세분화된 권한 부여: OPA(Open Policy Agent)를 사용하는 Policy-as-Code
3. 보안 암호 관리: 자격 증명 및 구성을 위한 HashiCorp 자격 증명 모음
4. 포괄적인 모니터링: 관찰 가능성 및 문제 해결을 위한 ELK 스택
5. 워크플로 오케스트레이션: 복잡한 장기 실행 프로세스를 위한 Temporal.io
6. 레거시 통합: 기존 시스템용 REST API 프록시

이 점이 중요한 이유

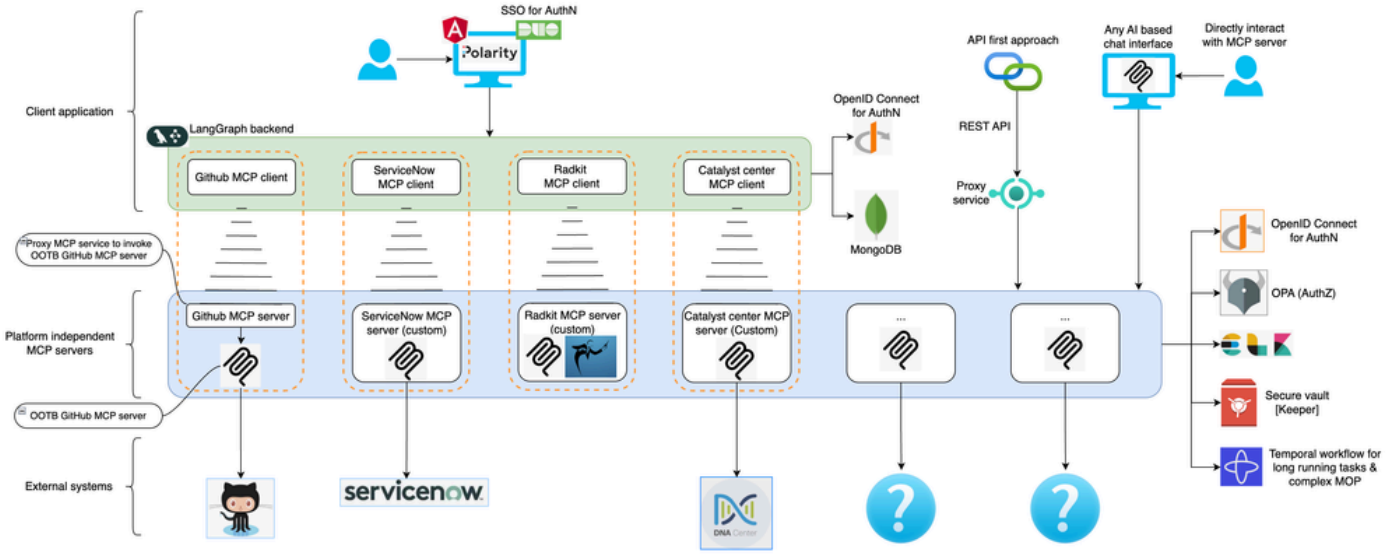
기존의 통합 방식은 다음과 같은 몇 가지 제약으로 인해 어려움을 겪습니다.

1. 보안 허점: 하드 코딩된 자격 증명 및 특권 초과 액세스
2. 운영 복잡성: 분산 시스템을 모니터링하고 문제를 해결하는 데 어려움
3. 확장성 문제: 증가하는 요구 사항에 따라 포인트-투-포인트 통합이 확장되지 않음
4. 유지 관리 오버헤드: 각 통합에는 맞춤형 인증 및 오류 처리가 필요합니다.

기업 패턴이 적용된 MCP 접근 방식은 이러한 과제를 해결하는 동시에 표준화되고 재사용 가능한 AI 기반 자동화의 기반을 제공합니다.

아키텍처 개요

참조 아키텍처는 클라이언트 애플리케이션을 MCP 서버 플랫폼으로부터 분리하는 계층화된 접근 방식을 구현하여 여러 애플리케이션이 동일한 엔터프라이즈급 MCP 인프라를 활용할 수 있도록 합니다.



구성 요소 아키텍처

1. 클라이언트 애플리케이션 계층

클라이언트 레이어는 사용자 인터페이스 및 오케스트레이션 논리를 제공합니다.

- 프런트 엔드: Cisco Polarity UI 프레임워크를 사용한 각형 애플리케이션
- 백엔드: 워크플로 오케스트레이션을 위한 LangGraph 멀티 에이전트 시스템
- 인증: 엔터프라이즈 ID 제공업체와 OIDC 통합

2. MCP 서버 플랫폼 레이어

플랫폼 레이어는 공유 서비스와 함께 엔터프라이즈급 MCP 서버를 구현합니다.

코어 MCP 서버:

- `mcp-catalyst-center`: Cisco 네트워크 디바이스 관리
- `mcp-service-now`: ITSM 통합 및 티켓 관리
- `mcp-github`: 소스 코드 및 저장소 관리
- `mcp-radkit`: 네트워크 분석 및 모니터링
- `mcp-rest-api-proxy`: 레거시 시스템 통합

엔터프라이즈 서비스:

- 인증 서비스: OIDC 토큰 검증 및 사용자 관리
- 권한 부여 서비스: OPA 기반 정책 시행
- 비밀 관리: 자격 증명 모음 기반 자격 증명 및 구성 저장소
- <모니터링 스택: 로깅, 메트릭 및 알림을 위한 ELK
- 워크플로 엔진: 복잡한 프로세스 오케스트레이션을 위한 임시

엔터프라이즈 보안 구현

OpenID 연결 인증

이 플랫폼은 OpenID Connect를 사용하여 엔터프라이즈급 인증을 구현함으로써 기존 ID 공급자와 원활하게 통합되는 동시에 Cisco Duo를 통한 다단계 인증을 지원합니다.

주요 이점

- SSO(Single Sign-On): 사용자가 모든 MCP 서비스에 대해 한 번 인증
- 다단계 인증: 통합 Cisco Duo로 보안 강화
- 토큰 기반 보안: JWT 토큰을 사용한 상태 비저장 인증
- 중앙 집중식 관리: 사용자 프로비저닝 및 기존 IdP를 통한 디프로비저닝

구현 개요

파일: `mcp-common-app/src/mcp_common/oidc_auth.py`을 참조하십시오.

```
"""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")
```

개방형 정책 에이전트를 통한 세분화된 권한 부여

OPA는 사용자 특성, 리소스 유형 및 상황 정보를 기반으로 세분화된 액세스 제어를 가능하게 하는 유연한 Policy-as-Code 권한 부여를 제공합니다.

권한 부여 정책 구조

파일: common-services/opa/config/policy.rego을 참조하십시오.

```
# 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 통합

<파일: mcp-common-app/src/mcp_common/opa.py을 참조하십시오.

```
"""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
```

HashiCorp Vault로 보안 비밀 관리

HashiCorp Vault는 암호화, 액세스 제어 및 감사 로깅을 통해 엔터프라이즈급 비밀 관리를 제공합니다. MCP 플랫폼은 Vault를 통합하여 API 자격 증명, 데이터베이스 비밀번호, 컨피그레이션 데이터 등의 중요한 정보를 안전하게 저장하고 검색합니다.

주요 기능

- 유휴 및 전송 중인 암호화: 모든 암호는 AES 256비트 암호화를 사용하여 암호화됩니다
- 동적 암호: 외부 서비스에 대한 시간 제한 자격 증명 생성
- 액세스 제어: 세분화된 정책으로 누가 어떤 기밀에 액세스할 수 있는지 제어
- 감사 로깅: 모든 비밀 액세스 작업에 대한 완벽한 감사 추적
- 암호 순환: 자격 증명 및 인증서 자동 순환

구현

파일: `mcp-common-app/src/mcp_common/vault.py`을 참조하십시오.

```
"""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}")
```

코어 MCP 서버 구조

각 MCP 서버는 엔터프라이즈 보안 통합을 통해 일관된 패턴을 제공합니다.

파일: mcp-catalyst-center/src/main.py을 참조하십시오.

```
"""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}
```

레거시 통합을 위한 REST API 프록시

이 플랫폼에는 비 MCP 클라이언트를 지원하는 REST API 프록시가 포함되어 있습니다.

파일: `mcp-rest-api-proxy/main.py`을 참조하십시오.

```
"""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}
```

모니터링 및 관찰 가능성

ELK 스택 통합

이 플랫폼은 ELK 스택을 사용하여 포괄적인 로깅을 구현합니다.

파일: mcp-common-app/src/mcp_common/logger.py을 참조하십시오.

```
"""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)
```

주요 모니터링 메트릭

이 플랫폼은 기업 운영에 필요한 필수 메트릭을 추적합니다.

- 요청 대기 시간: 툴별 실행 시간 및 백분위수
- 인증 메트릭: 성공/실패율 및 응답 시간
- 권한 부여 결정: 정책 평가 빈도 및 결과
- 비밀 액세스: 자격 증명 모음 작업 및 자격 증명 사용 패턴
- 오류율: 서비스 수준 오류 추적 및 경고 임계값
- 리소스 사용률: 서비스당 CPU, 메모리 및 네트워크 사용량

임시 워크플로 통합

복잡한 장기 실행 프로세스의 경우, 이 플랫폼은 Temporal.io를 활용합니다.

파일: `temporal-service/src/workflows/template_deployment.py`을 참조하십시오.

```
"""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"}
```

구축 및 확장성

컨테이너 오케스트레이션

이 플랫폼은 개발을 위해 Docker Compose를 사용합니다. Kubernetes는 다음과 같은 용도로 사용할 수 있습니다.

파일: `docker-compose.yml`(발췌)

```
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]
```

성능 및 보안 고려 사항

보안 모범 사례

1. 제로 트러스트 아키텍처: 모든 요청이 인증되고 승인됨
2. 암호 순환: 볼트를 통한 자동 비밀 순환
3. 네트워크 세그멘테이션: mTLS를 사용하는 서비스 메시
4. 감사 로깅: ELK의 포괄적인 감사 추적

성능 최적화

1. 연결 풀링: 외부 API에 대한 HTTP 연결 재사용
2. 캐싱: 자주 액세스하는 데이터를 위한 Redis 기반 캐싱
3. 비동기 처리: 스택 전체에서 I/O를 차단하지 않음
4. 로드 밸런싱: 여러 MCP 서버 인스턴스 전반에 부하 분산

측정 단위 모니터링

추적되는 주요 메트릭은 다음과 같습니다.

- MCP 툴당 요청 레이턴시
- 인증 성공/실패율
- 리소스별 권한 부여 결정
- 비밀 검색 빈도
- 서비스 구성 요소별 오류 비율

성능 메트릭 및 결과

성능 테스트 과정에서 이 플랫폼은 다음을 달성했습니다.

- 인증 결정을 위한 100ms 미만의 대기 시간
- 모든 MCP 서비스에서 99.9% 업타임
- 동시 사용자 최대 1,000명의 선형 확장성
- 통합 개발 시간 90% 단축

습득한 교훈 및 모범 사례

주요 성공 요인

1. 표준화: 공통 라이브러리로 중복 및 버그 감소
2. 관찰 가능성: 포괄적인 로깅을 통해 신속한 문제 해결 가능
3. 설계별 보안: 첫날부터 인증 및 권한 부여
4. 모듈 구조: 독립적인 MCP 서버를 통해 타겟 확장 가능
5. 설명서: 명확한 API 문서를 통해 채택 가속화

피해야 할 일반적인 위험

1. 오버엔지니어링: 단순하게 시작하고 필요에 따라 복잡성 추가
2. 긴밀한 연결: MCP 서버를 느슨하게 연결
3. 향후 보안: 처음부터 보안 구축
4. 불충분한 테스트: 포괄적인 테스트 전략 구현
5. 잘못된 오류 처리: 정상적인 성능 저하 보장

향후 개선 사항

플랫폼 로드맵에는 다음이 포함됩니다.

1. AI 중심의 인사이트: ML 기반 이상 징후 탐지
2. 멀티 클라우드 지원: 클라우드 사업자 전반에 걸친 구축
3. 워크플로 마켓플레이스: 재사용 가능한 워크플로 템플릿
4. 고급 분석: 실시간 대시보드 및 보고

결론

프로덕션 등급 MCP 서버를 구축하려면 보안, 확장성, 모니터링 및 유지 보수를 비롯한 엔터프라이즈 요구 사항을 신중하게 고려해야 합니다. 이 참조 아키텍처는 업계 표준 도구 및 패턴을 사용하여 이러한 기능을 구현하는 방법을 보여 줍니다.

조직은 모듈형 설계를 통해 MCP를 점진적으로 채택하는 동시에 처음부터 엔터프라이즈급 보안 및 운영을 보장할 수 있습니다. OIDC, OPA, Vault, ELK와 같은 검증된 기술을 활용함으로써 팀은 인프라 문제 보다는 비즈니스 논리에 집중할 수 있습니다.

작성자 정보

이 문서는 MCP Fusioners 팀에서 Cisco의 내부 혁신 이니셔티브의 일환으로 엔터프라이즈 AI 시스템 통합에 대한 실용적인 접근 방식을 보여 주기 위해 개발했습니다.

참조

1. 모델 컨텍스트 프로토콜 사양 - https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. 정책 에이전트 설명서 열기 - https://www.openpolicyagent.org/docs
4. HashiCorp 자격 증명 모음 문서 - https://www.vaultproject.io/docs
5. Temporal.io 문서 - https://docs.temporal.io/
6. ELK 스택 가이드 - https://www.elastic.co/elastic-stack/

이 번역에 관하여

Cisco는 전 세계 사용자에게 다양한 언어로 지원 콘텐츠를 제공하기 위해 기계 번역 기술과 수작업 번역을 병행하여 이 문서를 번역했습니다. 아무리 품질이 높은 기계 번역이라도 전문 번역가의 번역 결과물만큼 정확하지는 않습니다. Cisco Systems, Inc.는 이 같은 번역에 대해 어떠한 책임도 지지 않으며 항상 원본 영문 문서(링크 제공됨)를 참조할 것을 권장합니다.