MCPサーバのパワーを活用：AIを活用したソリューションでネットワーク自動化を革新

内容

はじめに

このドキュメントでは、業界のベストプラクティスを使用して実稼働対応Model Context Protocol(MCP)サーバを構築するための包括的なリファレンスアーキテクチャについて説明します。Cisco Catalyst Center、ServiceNow、およびその他のエンタープライズシステムを統合した実際の実装を通じて実証されています。 MCPは、AIシステムが外部のサービスやデータソースと相互作用する方法のパラダイムシフトを表します。ただし、プロトタイプから実稼働への移行には、認証、許可、監視、拡張性を含むエンタープライズグレードのパターンを実装する必要があります。

背景説明

AIを活用した自動化を採用する企業が増えるにつれ、堅牢でセキュアかつスケーラブルな統合プラットフォームの必要性が高まっています。従来のポイントツーポイント統合では、メンテナンスのオーバーヘッドとセキュリティの脆弱性が生じます。Model Context Protocol(MCP)は、AIシステム統合に対する標準化されたアプローチを提供しますが、実稼働環境では、基本的なMCP実装を超えるエンタープライズグレードの機能が必要です。

この記事では、次のものを組み込んだ実稼働準備のMCPサーバプラットフォームを構築する方法について説明します。

1. エンタープライズ認証:OpenID Connect(OIDC)とCisco Duoの統合
2. きめ細かい許可：オープンポリシーエージェント(OPA)を使用したコード単位のポリシー
3. Secure Secret Management：クレデンシャルと設定用のHashiCorp Vault
4. 包括的なモニタリング：監視とトラブルシューティングのためのELKスタック
5. ワークフローオーケストレーション：複雑で長時間実行されるプロセス向けのTemporal.io
6. レガシー統合：既存システムのREST APIプロキシ

なぜ、これが重要なのか

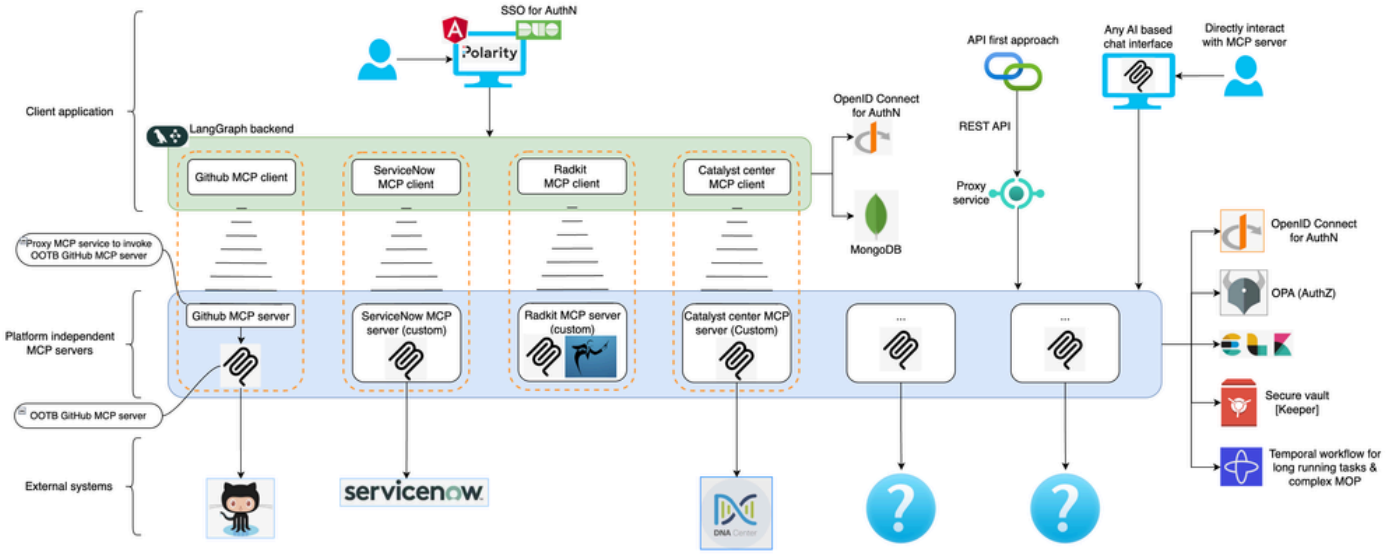
従来の統合アプローチには、次のような制約があります。

1. セキュリティギャップ：ハードコードされたクレデンシャルと過剰な特権アクセス
2. 運用の複雑さ：分散システムの監視とトラブルシューティングが困難
3. 拡張性の問題：ポイントツーポイント統合は、増大する要件に合わせて拡張できません
4. メンテナンスのオーバーヘッド：各統合には、カスタム認証とエラー処理が必要です

エンタープライズパターンを備えたMCPアプローチは、AI主導の自動化のための標準化された再利用可能な基盤を提供しながら、これらの課題に対処します。

アーキテクチャ概要

リファレンスアーキテクチャでは、クライアントアプリケーションをMCPサーバプラットフォームから分離する階層型アプローチを実装し、複数のアプリケーションが同じエンタープライズグレードのMCPインフラストラクチャを利用できるようにします。



コンポーネントアーキテクチャ

1. クライアントアプリケーション層

クライアント層は、ユーザインターフェイスとオーケストレーションロジックを提供します。

- フロントエンド:Cisco Polarity UIフレームワークを使用した角度のあるアプリケーション
- バックエンド：ワークフローオーケストレーション用のLangGraphマルチエージェントシステム
- 認証：企業アイデンティティプロバイダーとのOIDC統合

2. MCPサーバプラットフォーム層

プラットフォーム層は、共有サービスを備えたエンタープライズグレードのMCPサーバを実装します。

コアMCPサーバ：

- `mcp-catalyst-center`:Ciscoネットワークデバイス管理
- `mcp-service-now`:ITSM統合およびチケット管理
- `mcp-github`：ソースコードおよびリポジトリ管理
- `mcp-radkit`：ネットワーク分析とモニタリング
- `mcp-rest-api-proxy`：レガシーシステム統合

Enterprise Services:

- 認証サービス:OIDCトークン検証およびユーザー管理
- 認証サービス:OPAベースのポリシー適用
- シークレット管理：資格情報と構成の保管をボールトベースで行います。
- <モニタリングスタック：ロギング、メトリック、アラートのELK
- ワークフローエンジン：複雑なプロセスのオーケストレーションのための一時的な機能

企業セキュリティの実装

OpenID接続認証

このプラットフォームは、OpenID Connectを使用してエンタープライズグレードの認証を実装し、Cisco Duoを通じて多要素認証をサポートしながら、既存のアイデンティティプロバイダーとのシームレスな統合を提供します。

主な利点

- シングルサインオン(SSO)：ユーザはすべてのMCPサービスで1回認証されます
- Multi-Factor Authentication:Cisco Duoの統合によりセキュリティを強化
- トークンベースのセキュリティ:JWTトークンを使用したステートレス認証
- 一元管理：既存のIdPを使用したユーザプロビジョニングおよびプロビジョニング解除

実装の概要

ファイル：`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")
```

Open Policy Agentによる詳細な許可

OPAは、柔軟なポリシーをコードとして許可し、ユーザ属性、リソースタイプ、コンテキスト情報に基づいて、きめ細かなアクセス制御を可能にします。

認可ポリシー構造

ファイル：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の統合

<ファイル：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
```

HashiCorp Vaultを使用したセキュアシークレット管理

HashiCorp Vaultは、暗号化、アクセス制御、監査ロギングを備えたエンタープライズグレードのシークレット管理を提供します。MCPプラットフォームは、Vaultを統合して、APIクレデンシャル、データベースパスワード、設定データなどの機密情報を安全に保存および取得します。

主な特長

- 保管中および転送中の暗号化：すべてのシークレットはAES 256ビット暗号化を使用して暗号化されます
- ダイナミックシークレット：外部サービスの期限付き資格情報を生成します
- アクセス制御：きめ細かなポリシーにより、誰がどのシークレットにアクセスできるかを制御
- 監査ロギング：すべてのシークレットアクセス操作の完全な監査証跡
- シークレットローテーション：資格情報と証明書の自動ローテーション

実装

ファイル：`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}")
```

コアMCPサーバ構造

各MCPサーバには、エンタープライズセキュリティ統合の一貫したパターンが含まれています。

ファイル：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}
```

レガシー統合用のREST APIプロキシ

このプラットフォームには、非MCPクライアントをサポートするREST APIプロキシが含まれています。

ファイル：`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}
```

監視と監視

ELKスタックの統合

プラットフォームは、ELKスタックを使用して包括的なロギングを実装します。

ファイル：`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)
```

主要なモニタリングメトリック

このプラットフォームは、企業の運用に不可欠なメトリックを追跡します。

- 要求遅延：ツールごとの実行時間とパーセンタイル
- 認証メトリック：成功/失敗率および応答時間
- 許可の決定：ポリシー評価の頻度と結果
- シークレットアクセス：ボールト操作とクレデンシャルの使用パターン
- エラー率：サービスレベルのエラー追跡および警告しきい値
- リソース使用率：サービスあたりのCPU、メモリ、およびネットワーク使用率

一時的なワークフローの統合

複雑で長時間実行されるプロセスでは、プラットフォームはTemporal.ioを活用します。

ファイル：`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"}
```

導入と拡張性

コンテナオーケストレーション

このプラットフォームでは、開発にDocker Composeを使用します。Kubernetesは実稼働に使用できます。

ファイル：`docker-compose.yml`（抜粋）

```
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]
```

パフォーマンスとセキュリティの考慮事項

セキュリティのベストプラクティス

1. ゼロトラストアーキテクチャ：すべての要求が認証および許可されます
2. シークレット回転:Vaultを使用したシークレット回転の自動化
3. ネットワークのセグメント化:mTLSによるサービスメッシュ
4. 監査ロギング:ELKの包括的な監査証跡

パフォーマンスの最適化

1. 接続プーリング：外部APIへのHTTP接続を再利用
2. キャッシング：頻繁にアクセスするデータのRedisベースのキャッシング
3. 非同期処理：スタック全体でノンブロッキングI/Oを実現
4. ロードバランシング：複数のMCPサーバインスタンス間で負荷を分散

メトリックの監視

主なメトリックは次のとおりです。

- MCPツールごとの要求遅延
- 認証の成功/失敗率
- リソースごとの承認の決定
- 秘密の取得頻度
- サービスコンポーネント別のエラー率

パフォーマンスメトリックと結果

パフォーマンステスト中、プラットフォームは以下を達成しました。

- 認証の決定に100ミリ秒未満の遅延
- すべてのMCPサービスで99.9 %のアップタイム
- 最大1000人の同時ユーザに対応する直線的な拡張性
- 統合開発時間を90 %短縮

得られた教訓とベストプラクティス

主な成功要因

1. 標準化：共通ライブラリにより重複やバグが減少
2. 可観測性：包括的なロギングによる迅速なトラブルシューティング
3. 設計によるセキュリティ：初日から認証と許可
4. モジュール性：独立したMCPサーバにより、対象を絞った拡張が可能
5. Documentation：明確なAPIドキュメントにより導入を加速

避けるべき一般的な落とし穴

1. オーバーエンジニアリング：シンプルな構成から始め、必要に応じて複雑さを増やす
2. 緊密な連携:MCPサーバの疎結合を維持
3. セキュリティの後付け：最初からセキュリティを構築する
4. 不十分なテスト：包括的なテスト戦略の実施
5. 不適切なエラー処理：グレースフルなデグレードを保証

今後の機能拡張

プラットフォームロードマップには次のものが含まれます。

1. AI-Driven Insights:MLベースの異常検出
2. マルチクラウドサポート：クラウドプロバイダー間の導入
3. Workflow Marketplace：再利用可能なワークフローテンプレート
4. 高度な分析：リアルタイムダッシュボードとレポート

結論

実稼働グレードのMCPサーバを構築するには、セキュリティ、スケーラビリティ、モニタリング、メンテナンス性などのエンタープライズ要件を慎重に検討する必要があります。このリファレンスアーキテクチャでは、業界標準のツールとパターンを使用してこれらの機能を実装する方法を示します。

モジュラ設計により、組織はMCPを徐々に導入しながら、エンタープライズクラスのセキュリティと運用を初日から保証できます。OIDC、OPA、Vault、ELKなどの実績のあるテクノロジーを活用することで、チームはインフラストラクチャの問題ではなく、ビジネスロジックに集中できます。

著者について

この記事は、シスコの社内イノベーションイニシアチブの一環としてMCP Fusionersチームによって開発され、エンタープライズAIシステム統合に対する実用的なアプローチを実証しています。

参考資料

1. Model Context Protocol Specification:https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0:https://openid.net/specs/openid-connect-core-1_0.html
3. Open Policy Agentのドキュメント – https://www.openpolicyagent.org/docs
4. HashiCorp Vaultドキュメント – https://www.vaultproject.io/docs
5. Temporal.ioドキュメント – https://docs.temporal.io/
6. ELKスタックガイド – https://www.elastic.co/elastic-stack/

翻訳について
シスコは世界中のユーザにそれぞれの言語でサポート コンテンツを提供するために、機械と人による翻訳を組み合わせて、本ドキュメントを翻訳しています。ただし、最高度の機械翻訳であっても、専門家による翻訳のような正確性は確保されません。シスコは、これら翻訳の正確性について法的責任を負いません。原典である英語版（リンクからアクセス可能）もあわせて参照することを推奨します。