利用MCP服务器的强大功能：利用人工智能驱动的解决方案彻底改变网络自动化

目录

简介

本文档介绍一个综合参考架构，用于使用行业最佳实践构建生产就绪型模型上下文协议(MCP)服务器。该架构通过集成Cisco Catalyst Center、ServiceNow和其他企业系统的实际实施进行了演示。MCP代表了AI系统与外部服务和数据源交互的模式转变。但是，从原型过渡到生产需要实施企业级模式，包括身份验证、授权、监控和可扩展性。

背景信息

随着组织越来越多地采用AI驱动的自动化，对稳健、安全且可扩展的集成平台的需求变得至关重要。传统的点对点集成会产生维护开销和安全漏洞。模型上下文协议(MCP)为AI系统集成提供了标准化方法，但生产部署需要超越基本MCP实施的企业级功能。

本文演示如何构建生产就绪型MCP服务器平台，该平台将：

1. 企业身份验证:OpenID Connect(OIDC)与思科双核集成
2. 精细授权:使用开放策略代理(OPA)的策略为代码
3. 安全密钥管理:用于凭证和配置的HashiCorp Vault
4. 全面监控:ELK堆栈，用于观察和故障排除
5. 工作流程协调：适用于复杂、长期运行的进程的temporal.io
6. 现有集成:用于现有系统的REST API代理

这一点为何重要

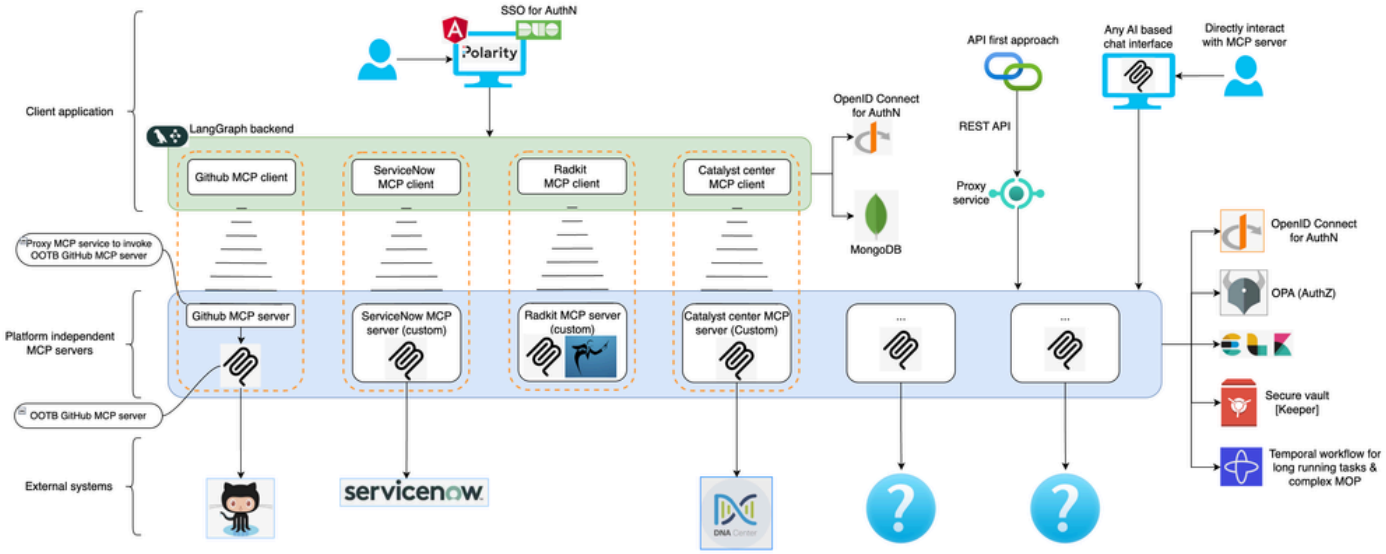
传统集成方法存在以下几个局限性：

1. 安全差距:硬编码凭证和超权限访问
2. 操作复杂性：难以对分布式系统进行监控和故障排除
3. 可扩展性问题:点对点集成无法随着需求的增长而扩展
4. 维护开销:每次集成都需要自定义身份验证和错误处理

采用企业模式的MCP方法可解决这些挑战，同时为AI驱动的自动化提供可重复使用的标准化基础。

体系结构概述

参考架构实施分层方法，将客户端应用与MCP服务器平台分开，使多个应用能够利用相同的企业级MCP基础设施。



组件架构

1.客户端应用层

客户端层提供用户界面和协调逻辑：

- 前端:使用Cisco Polarity UI框架的角度应用
- 后端:工作流协调的LangGraph多Agent系统
- 身份验证:OIDC与企业身份提供商集成

2. MCP服务器平台层

平台层实施具有共享服务的企业级MCP服务器：

核心MCP服务器：

- `mcp-catalyst-center`:思科网络设备管理
- `mcp-service-now`:ITSM集成和票证管理
- `mcp-github`:源代码和存储库管理
- `mcp-radkit`:网络分析和监控
- `mcp-rest-api-proxy`:传统系统集成

企业服务:

- 身份验证服务:OIDC令牌验证和用户管理
- 授权服务:基于OPA的策略实施
- 密钥管理:基于保险存储的凭证和配置存储
- <Monitoring Stack:ELK用于记录、度量和警报
- 工作流引擎:复杂流程协调的时间性

企业安全实施

OpenID连接身份验证

该平台使用OpenID Connect实施企业级身份验证，提供与现有身份提供商的无缝集成，同时支持通过Cisco Duo进行多重身份验证。

主要好处

- 单点登录(SSO):用户跨所有MCP服务进行一次身份验证
- 多重身份验证:集成的Cisco Duo，增强安全性
- 基于令牌的安全性:使用JWT令牌的无状态身份验证
- 集中管理:通过现有IdP进行用户调配和取消调配

实施概述

文件:`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")
```

使用开放策略代理的精细授权

OPA提供灵活的策略即代码授权，可根据用户属性、资源类型和情景信息实现细粒度访问控制。

授权策略结构

文件: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集成

<文件：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
```

使用HashiCorp Vault进行安全密钥管理

HashiCorp Vault通过加密、访问控制和审计日志记录提供企业级密钥管理。MCP平台集成了Vault，可以安全地存储和检索敏感信息，包括API凭证、数据库密码和配置数据。

主要特点

- 静态和传输中的加密:所有秘密都使用AES 256位加密进行加密
- 动态机密:为外部服务生成受时间限制的凭据
- 访问控制:精细策略控制谁可以访问哪些机密
- 审核日志记录：完成所有秘密访问操作的审计追踪
- 密钥轮替:自动轮换凭证和证书

实现

文件:`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}")
```

核心MCP服务器结构

每个MCP服务器都包含与企业安全集成一致的方式：

文件: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}
```

用于传统集成的REST API代理

该平台包括支持非MCP客户端的REST API代理：

文件: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}
```

监控和可观察性

ELK堆栈集成

该平台使用ELK堆栈实施全面的日志记录：

文件: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)
```

关键监控指标

该平台跟踪企业运营的基本指标：

- 请求延迟:每工具执行时间和百分比
- 身份验证指标：成功/失败率和响应时间
- 授权决策:政策评价频率和成果
- 秘密访问:保管库操作和凭据使用模式
- 错误率:服务级别错误跟踪和警报阈值
- 资源利用率:每项服务的CPU、内存和网络使用率

临时工作流集成

对于复杂、长期运行的流程，该平台利用Temporal.io:

文件: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"}
```

部署和可扩展性

容器协调

该平台使用Docker Compose进行开发。Kubernetes可用于生产：

文件:`docker-compose.yml`(摘录)

```
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]
```

性能和安全注意事项

安全最佳实践

1. 零信任架构:每个请求都经过身份验证和授权
2. 密钥轮替:通过Vault进行自动密钥轮替
3. 网络分段:使用mTLS的服务网格
4. 审核日志记录：ELK中的全面审计跟踪

性能优化

1. 连接池:重用与外部API的HTTP连接
2. 缓存:基于Redis的频繁访问数据缓存
3. 异步处理:整个堆叠中的无阻塞I/O
4. 负载均衡：跨多个MCP服务器实例分配负载

监控指标

跟踪的关键指标包括：

- 每个MCP工具的请求延迟
- 身份验证成功/失败率
- 每个资源的授权决策
- 秘密检索频率
- 按服务组件划分的错误率

绩效指标和结果

在性能测试过程中，该平台实现了：

- 用于身份验证决策的100毫秒以下延迟
- 所有MCP服务的正常运行时间均为99.9%
- 最多支持1000个并发用户的线性可扩展性
- 集成开发时间减少90%

经验教训和最佳做法

关键成功因素

1. 标准化:通用库可减少重复和错误
2. 可观性：全面的日志记录可实现快速故障排除
3. 设计安全性:从第一天开始进行身份验证和授权
4. 模块化:独立的MCP服务器实现目标扩展
5. 文档:清晰API文档可加快采用速度

要避免的常见陷阱

1. 过度工程:开始简单，然后根据需要增加复杂性
2. 紧耦合:保持MCP服务器松散耦合
3. 事后考虑安全：从头开始构建安全性
4. 测试不充分:实施全面的测试策略
5. 错误处理不佳:确保平稳降级

未来的增强功能

平台规划图包括：

1. AI驱动的见解:基于ML异常检测
2. 多云支持:跨云提供商的部署
3. 工作流市场:可重复使用的工作流程模板
4. 高级分析:实时控制面板和报告

结论

构建生产级MCP服务器需要仔细考虑企业要求，包括安全性、可扩展性、监控性和可维护性。此参考架构演示如何使用行业标准工具和模式实施这些功能。

模块化设计使组织能够逐步采用MCP，同时从第一天起就确保企业级安全性和运营。通过利用OIDC、OPA、Vault和ELK等成熟的技术，团队可以专注于业务逻辑而非基础设施问题。

关于作者

本文由MCP Fusioners团队开发，是思科内部创新计划的一部分，展示了企业AI系统集成的实用方法。

参考

1. 模型上下文协议规范 — https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. 打开策略代理文档 — https://www.openpolicyagent.org/docs
4. HashiCorp Vault文档 — https://www.vaultproject.io/docs
5. Temporal.io文档 — https://docs.temporal.io/
6. ELK堆栈指南 — https://www.elastic.co/elastic-stack/

关于此翻译
思科采用人工翻译与机器翻译相结合的方式将此文档翻译成不同语言，希望全球的用户都能通过各自的语言得到支持性的内容。

请注意：即使是最好的机器翻译，其准确度也不及专业翻译人员的水平。

Cisco Systems, Inc. 对于翻译的准确性不承担任何责任，并建议您总是参考英文原始文档（已提供链接）。