Exploitez la puissance des serveurs MCP : Révolutionnez l'automatisation du réseau avec des solutions basées sur l'IA

Table des matières

Références

Introduction

Ce document décrit une architecture de référence complète pour construire des serveurs MCP (Model Context Protocol) prêts à l'emploi en utilisant les meilleures pratiques du secteur, démontrées par une mise en oeuvre réelle qui intègre Cisco Catalyst Center, ServiceNow et d'autres systèmes d'entreprise. Le MCP représente un changement de paradigme dans la façon dont les systèmes d'IA interagissent avec les services externes et les sources de données. Cependant, passer du prototype à la production nécessite la mise en oeuvre de modèles de niveau entreprise, notamment l'authentification, l'autorisation, la surveillance et l'évolutivité.

Informations générales

À mesure que les entreprises adoptent de plus en plus l'automatisation basée sur l'IA, le besoin de plates-formes d'intégration robustes, sécurisées et évolutives devient critique. Les intégrations point à point traditionnelles entraînent des frais de maintenance et des vulnérabilités en matière de sécurité. Le protocole MCP (Model Context Protocol) offre une approche standardisée des intégrations de systèmes d'intelligence artificielle, mais les déploiements en production nécessitent des fonctionnalités professionnelles qui vont au-delà des implémentations MCP de base.

Cet article explique comment créer une plate-forme de serveur MCP prête pour la production qui intègre :

1. Authentification d'entreprise : Intégration d'OpenID Connect (OIDC) à Cisco Duo
2. Autorisation à granularité fine : Policy-as-code avec Open Policy Agent (OPA)
3. Gestion sécurisée des secrets : HashiCorp Vault pour les informations d'identification et la configuration
4. Surveillance complète : Pile ELK pour l'observabilité et le dépannage
5. Orchestration du workflow : Temporal.io pour les processus longs et complexes
6. Intégration héritée : Proxy API REST pour les systèmes existants

Pourquoi cela est important

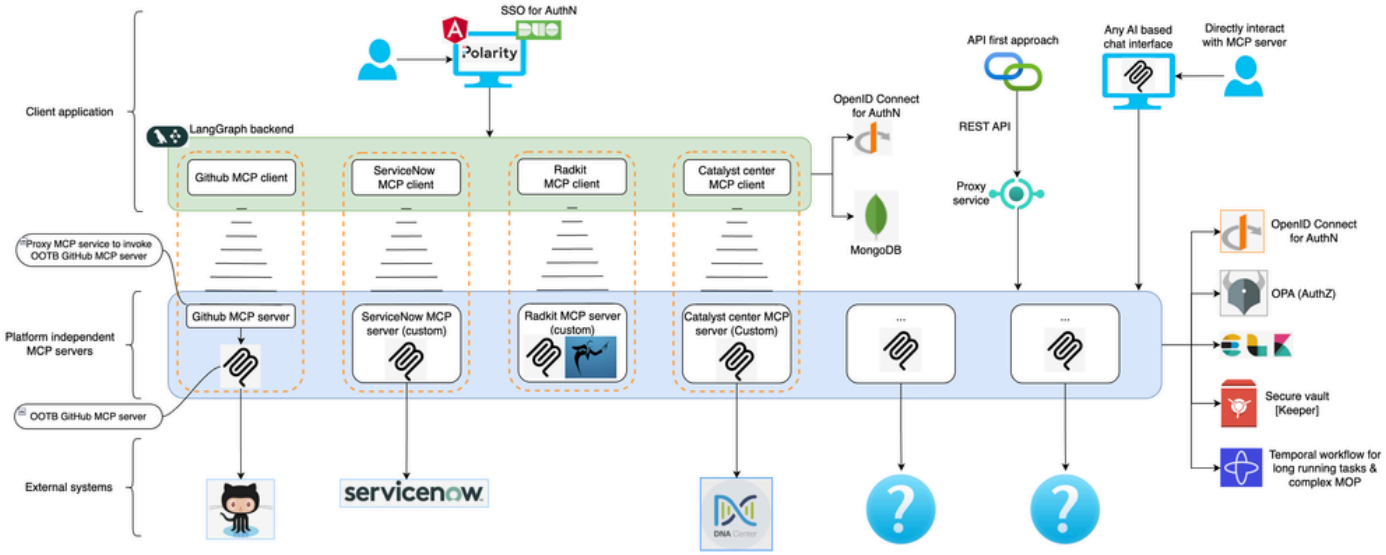
Les approches d'intégration traditionnelles présentent plusieurs limites :

1. Lacunes en matière de sécurité : Identifiants codés en dur et accès surprivilégié
2. Complexité opérationnelle : Difficulté à surveiller et à dépanner les systèmes distribués
3. Problèmes d'évolutivité : Les intégrations point à point ne s'adaptent pas aux besoins croissants
4. Frais généraux de maintenance : Chaque intégration nécessite une authentification personnalisée et une gestion des erreurs

L'approche MCP avec des modèles d'entreprise permet de relever ces défis tout en fournissant une base standardisée et réutilisable pour l'automatisation basée sur l'IA.

Présentation de l'architecture

L'architecture de référence met en oeuvre une approche en couches qui sépare les applications clientes de la plate-forme serveur MCP, permettant ainsi à plusieurs applications d'exploiter la même infrastructure MCP de niveau entreprise.



Architecture des composants

1. Couche application client

La couche client fournit des interfaces utilisateur et une logique d'orchestration :

- Frontend : Application angulaire utilisant l'interface utilisateur Cisco Polarity
- Principal : Système multi-agent LangGraph pour l'orchestration de workflow
- Authentification: Intégration d'OIDC aux fournisseurs d'identité d'entreprise

2. Couche plate-forme du serveur MCP

La couche plate-forme implémente des serveurs MCP d'entreprise avec des services partagés :

Serveurs MCP principaux :

- `mcp-catalyst-center` : Gestion des périphériques réseau Cisco
- `mcp-service-now` : Intégration ITSM et gestion des tickets
- `mcp-github` : Gestion du code source et du référentiel
- `mcp-radkit` : Analyse et surveillance du réseau
- `mcp-rest-api-proxy` : Intégration des systèmes existants

Services d'entreprise :

- Service d'authentification : Validation de jeton OIDC et gestion des utilisateurs
- Service d'autorisation : Application de la stratégie basée sur OPA

- Gestion des secrets : Stockage des informations d'identification et des configurations en chambre forte
- <Pile de surveillance : ELK pour la journalisation, les mesures et les alertes
- Moteur de workflow : Temporel pour l'orchestration des processus complexes

implémentation de sécurité d'entreprise

Authentification OpenID Connect

La plate-forme met en oeuvre l'authentification de niveau entreprise à l'aide d'OpenID Connect, offrant une intégration transparente avec les fournisseurs d'identité existants tout en prenant en charge l'authentification multifacteur via Cisco Duo.

Principaux avantages

- Authentification unique (SSO) : Les utilisateurs s'authentifient une seule fois sur tous les services MCP
- Multi-Factor Authentication : Cisco Duo intégré pour une sécurité renforcée
- Sécurité basée sur les jetons : Authentification sans état utilisant des jetons JWT
- Gestion centralisée : Provisionnement et déprovisionnement utilisateur via le fournisseur d'identité existant

Présentation de la mise en oeuvre

Fichier: `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")
```

Autorisation affinée avec Open Policy Agent

L'OPA fournit une autorisation flexible, basée sur des politiques en tant que code, qui permet un contrôle d'accès granulaire basé sur les attributs utilisateur, les types de ressources et les informations contextuelles.

Structure des politiques d'autorisation

Fichier: `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"]
}
```

Intégration OPA Python

<Fichier : `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
```

Gestion sécurisée des secrets avec HashiCorp Vault

HashiCorp Vault offre une gestion des secrets de niveau entreprise avec cryptage, contrôle d'accès et consignation des audits. La plate-forme MCP intègre Vault pour stocker et récupérer en toute sécurité des informations sensibles, notamment les informations d'identification API, les mots de passe de base de données et les données de configuration.

Fonctionnalités principales

- Cryptage au repos et en transit : Tous les secrets sont chiffrés à l'aide du chiffrement AES 256 bits
- Secrets dynamiques : Générer des informations d'identification limitées dans le temps pour les services externes
- Contrôle d'accès : Des politiques précises contrôlent qui peut accéder à quels secrets
- Journalisation d'audit : Piste d'audit complète de toutes les opérations d'accès secret
- Rotation secrète : Rotation automatisée des certificats et des certificats

Mise En Oeuvre

Fichier: 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}")
```

Structure du serveur MCP principal

Chaque serveur MCP présente un modèle cohérent avec l'intégration de la sécurité d'entreprise :

Fichier: `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}
```

Proxy API REST pour intégration existante

La plate-forme inclut un proxy API REST pour prendre en charge les clients non-MCP :

Fichier: `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}
```

surveillance et observabilité

Intégration de la pile ELK

La plate-forme implémente une journalisation complète à l'aide de la pile ELK :

Fichier: `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)
```

Indicateurs de surveillance clés

La plate-forme assure le suivi des indicateurs essentiels aux opérations de l'entreprise :

- Latence des demandes : Temps d'exécution par outil et centiles
- Mesures d'authentification : Taux de réussite/d'échec et temps de réponse
- Décisions d'autorisation : Fréquence et résultats des évaluations des politiques
- Accès secret : Modèles d'utilisation des informations d'identification et des opérations du coffre
- Taux d'erreurs : Seuils d'alerte et de suivi des erreurs au niveau du service
- Utilisation des ressources : Utilisation du processeur, de la mémoire et du réseau par service

Intégration de workflow temporelle

Pour les processus longs et complexes, la plate-forme exploite Temporal.io :

Fichier: `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"}
```

Déploiement et évolutivité

Orchestration du conteneur

La plate-forme utilise Docker Compose pour le développement. Kubernetes peut être utilisé pour la production :

Fichier: `docker-compose.yml` (extrait)

```
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]
```

Performances et sécurité

Meilleures pratiques de sécurité

1. Architecture de confiance zéro : Chaque demande est authentifiée et autorisée
2. Rotation secrète : Rotation secrète automatisée via Vault
3. Segmentation du réseau : Maillage de service avec mTLS
4. Journalisation d'audit : Piste d'audit complète dans ELK

Optimisation des performances

1. Regroupement de connexions : Réutilisation des connexions HTTP vers des API externes
2. Mise en cache : Cache Redis pour les données fréquemment consultées
3. Traitement asynchrone : E/S non bloquantes dans toute la pile
4. Équilibrage de charge : Répartir la charge sur plusieurs instances de serveur MCP

Mesures de surveillance

Les mesures clés suivies sont les suivantes :

- Latence des demandes par outil MCP
- Taux de réussite/échec d'authentification
- Décisions d'autorisation par ressource
- Fréquence de récupération des secrets
- Taux d'erreur par composant de service

Indicateurs de performances et résultats

Lors des tests de performances, la plate-forme a obtenu :

- Latence inférieure à 100 ms pour les décisions d'authentification
- Disponibilité de 99,9 % pour tous les services MCP
- Évolutivité linéaire jusqu'à 1 000 utilisateurs simultanés
- 90 % de réduction du temps de développement de l'intégration

Leçons apprises et meilleures pratiques

Facteurs clés de succès

1. Normalisation : Les bibliothèques courantes réduisent les doublons et les bogues
2. Observabilité : Une journalisation complète permet un dépannage rapide
3. Sécurité par conception : Authentification et autorisation dès le premier jour
4. Modularité : Les serveurs MCP indépendants permettent une évolutivité ciblée
5. Documentation: Une documentation API claire accélère l'adoption

Les pièges courants à éviter

1. Suringénierie : Commencez simplement et ajoutez de la complexité si nécessaire
2. Accouplement serré : Maintenir le couplage des serveurs MCP
3. Pensée après coup : Développez la sécurité depuis le début
4. Test insuffisant : Mettre en oeuvre des stratégies de test complètes
5. Mauvaise gestion des erreurs : Garantir une dégradation progressive

Améliorations futures

La feuille de route de la plate-forme inclut :

1. Points de vue axés sur l'IA : Détection d'anomalie basée sur ML
2. Prise en charge multicloud : Déploiement sur les fournisseurs cloud
3. Workflow Marketplace : Modèles de workflow réutilisables
4. Analyses avancées : Tableaux de bord et rapports en temps réel

Conclusion

La construction de serveurs MCP de production nécessite une prise en compte attentive des exigences de l'entreprise, notamment en matière de sécurité, d'évolutivité, de surveillance et de maintenance. Cette architecture de référence explique comment mettre en oeuvre ces fonctionnalités à l'aide d'outils et de modèles standard.

La conception modulaire permet aux entreprises d'adopter progressivement le protocole MCP tout en garantissant une sécurité et des opérations de niveau entreprise dès le premier jour. En tirant parti de technologies éprouvées comme OIDC, OPA, Vault et ELK, les équipes peuvent se concentrer sur la logique métier plutôt que sur les problèmes d'infrastructure.

À propos des auteurs

Cet article a été développé par l'équipe MCP Fusioners dans le cadre des initiatives internes de Cisco en matière d'innovation. Il présente des approches pratiques de l'intégration des systèmes d'intelligence artificielle d'entreprise.

Références

1. Spécification du protocole de contexte de modèle - https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. Documentation relative à Open Policy Agent - https://www.openpolicyagent.org/docs
4. Documentation HashiCorp Vault - https://www.vaultproject.io/docs
5. Documentation Temporal.io - https://docs.temporal.io/
6. Guide de la pile ELK - https://www.elastic.co/elastic-stack/

À propos de cette traduction
Cisco a traduit ce document en traduction automatisée vérifiée par une personne dans le cadre d'un service mondial permettant à nos utilisateurs d'obtenir le contenu d'assistance dans leur propre langue.

Il convient cependant de noter que même la meilleure traduction automatisée ne sera pas aussi précise que celle fournie par un traducteur professionnel.