Sfruttare la potenza dei server MCP: Rivoluziona l'automazione della rete con le soluzioni basate sull'intelligenza artificiale

Sommario

Riferimenti

Introduzione

Questo documento descrive un'architettura di riferimento completa per la creazione di server MCP (Model Context Protocol) pronti per la produzione che utilizzano le best practice del settore, dimostrata tramite un'implementazione reale che integra Cisco Catalyst Center, ServiceNow e altri sistemi aziendali. MCP rappresenta un cambiamento di paradigma nel modo in cui i sistemi AI interagiscono con servizi esterni e origini dati. Tuttavia, il passaggio dal prototipo alla produzione richiede l'implementazione di modelli di livello enterprise, tra cui autenticazione, autorizzazione, monitoraggio e scalabilità.

Premesse

Con l'adozione sempre più diffusa dell'automazione basata sull'intelligenza artificiale, la necessità di piattaforme di integrazione robuste, sicure e scalabili diventa sempre più critica. Le tradizionali integrazioni point-to-point creano un sovraccarico di manutenzione e vulnerabilità della sicurezza. Il protocollo MCP (Model Context Protocol) offre un approccio standardizzato alle integrazioni dei sistemi AI, ma le implementazioni di produzione richiedono funzionalità di livello enterprise che vanno oltre le implementazioni MCP di base.

In questo articolo viene illustrato come creare una piattaforma server MCP pronta per la produzione che includa:

1. Autenticazione Enterprise: Integrazione di OpenID Connect (OIDC) con Cisco Duo
2. Autorizzazione specifica: Policy-as-code con Open Policy Agent (OPA)
3. Gestione protetta da segreto: HashiCorp Vault per credenziali e configurazione
4. Monitoraggio completo: Stack ELK per l'osservabilità e la risoluzione dei problemi
5. Orchestrazione flusso di lavoro: Temporal.io per processi complessi e di lunga durata
6. Integrazione legacy: Proxy API REST per sistemi esistenti

Perché è importante

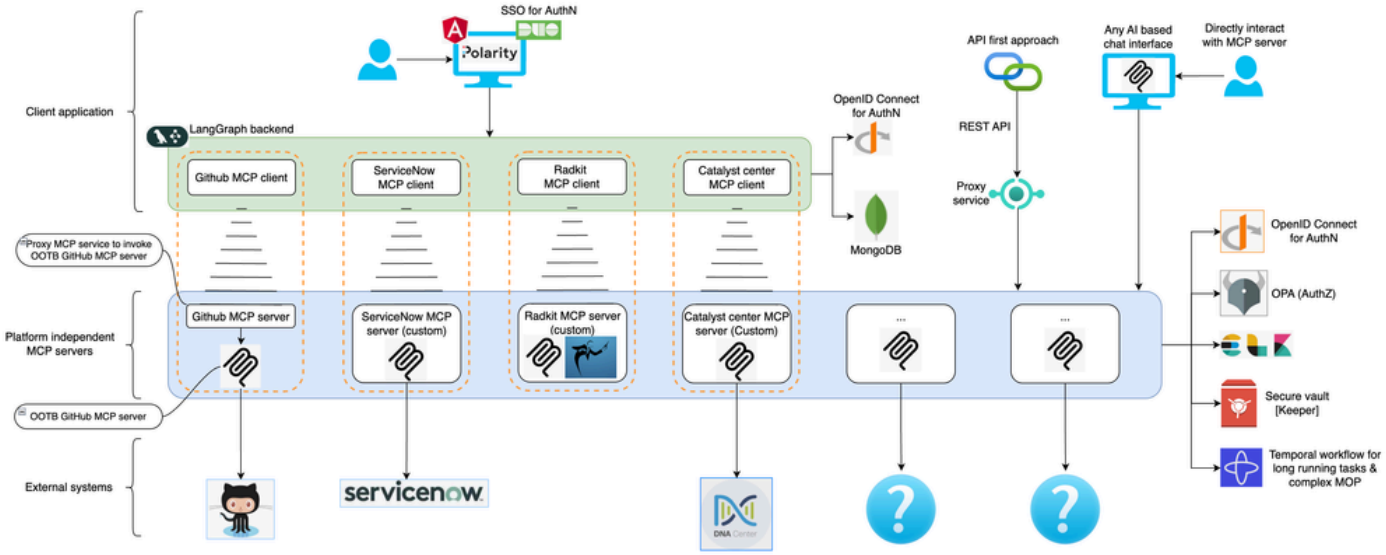
Gli approcci di integrazione tradizionali presentano diverse limitazioni:

1. Lacune nella sicurezza: Credenziali hardcoded e accesso con privilegi eccessivi
2. Complessità operativa: Difficoltà di monitoraggio e risoluzione dei problemi dei sistemi distribuiti
3. Problemi di scalabilità: Le integrazioni point-to-point non sono scalabili a causa delle crescenti esigenze
4. Costi generali di manutenzione: Ogni integrazione richiede autenticazione personalizzata e gestione degli errori

L'approccio MCP con modelli aziendali risolve queste problematiche fornendo al contempo una base standardizzata e riutilizzabile per l'automazione basata sull'intelligenza artificiale.

Panoramica dell'architettura

L'architettura di riferimento implementa un approccio su più livelli che separa le applicazioni client dalla piattaforma server MCP, consentendo a più applicazioni di utilizzare la stessa infrastruttura MCP di classe enterprise.



Architettura dei componenti

1. Livello applicazione client

Il livello client fornisce le interfacce utente e la logica di orchestrazione:

- Frontend: Applicazione angolare con framework interfaccia utente Cisco Polarity
- Back-end: Sistema LangGraph multi-agent per l'orchestrazione del flusso di lavoro
- Autenticazione: Integrazione OIDC con provider di identità aziendali

2. Livello piattaforma server MCP

Il livello di piattaforma implementa server MCP di classe enterprise con servizi condivisi:

Server MCP di base:

- `mcp-catalyst-center`: Cisco network device management
- `mcp-service-now`: Integrazione ITSM e gestione ticket
- `mcp-github`: Gestione del codice sorgente e del repository
- `mcp-radkit`: Analisi e monitoraggio della rete
- `mcp-rest-api-proxy`: Integrazione di sistemi legacy

Servizi aziendali:

- Servizio di autenticazione: Convalida e gestione utenti token OIDC
- Servizio di autorizzazione: Applicazione delle policy basata su OPA

- Gestione dei segreti: Storage delle credenziali e della configurazione basato su Vault
- <Stack di monitoraggio: ELK per registrazione, metriche e avvisi
- Motore flusso di lavoro: Temporale per l'orchestrazione di processi complessi

Implementazione della sicurezza aziendale

Autenticazione connessione OpenID

La piattaforma implementa l'autenticazione di livello enterprise utilizzando OpenID Connect, fornendo un'integrazione perfetta con i provider di identità esistenti e supportando l'autenticazione a più fattori tramite Cisco Duo.

Vantaggi principali

- Single Sign-On (SSO): Gli utenti eseguono l'autenticazione una sola volta su tutti i servizi MCP
- Multi-Factor Authentication: Cisco Duo integrato per una maggiore sicurezza
- Sicurezza basata su token: Autenticazione senza stato tramite token JWT
- Gestione centralizzata: Provisioning e deprovisioning degli utenti tramite IdP esistente

Panoramica sull'implementazione

File: `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")
```

Autorizzazione dettagliata con Open Policy Agent

OPA fornisce un'autorizzazione flessibile basata su regole come codice che consente un controllo dettagliato degli accessi in base agli attributi utente, ai tipi di risorse e alle informazioni contestuali.

Struttura dei criteri di autorizzazione

File: `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"]
```

```
}
```

Integrazione con Python OPA

<File: `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
```

Gestione protetta dei segreti con HashiCorp Vault

HashiCorp Vault fornisce una gestione segreta di livello enterprise con crittografia, controllo degli accessi e registrazione di verifica. La piattaforma MCP integra Vault per archiviare e recuperare in modo sicuro le informazioni riservate, incluse le credenziali API, le password del database e i dati di configurazione.

Caratteristiche principali

- Crittografia su disco e in transito: Tutti i segreti vengono crittografati utilizzando la crittografia AES a 256 bit
- Segreti dinamici: Genera credenziali con limiti di tempo per i servizi esterni
- Controllo di accesso: Le policy specifiche controllano gli utenti che possono accedere a determinati segreti
- Registrazione di controllo: Audit trail completo di tutte le operazioni di accesso segreto
- Rotazione segreto: Rotazione automatica di credenziali e certificati

Implementazione

File: `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}")
```

Struttura di base del server MCP

Ogni server MCP contiene uno schema coerente con l'integrazione della sicurezza aziendale:

File: `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 per integrazione legacy

La piattaforma include un proxy API REST per il supporto di client non MCP:

File: `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}
```

Monitoraggio e osservabilità

Integrazione stack ELK

La piattaforma implementa una registrazione completa utilizzando lo stack ELK:

File: `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)
```

Metriche di controllo chiave

La piattaforma tiene traccia delle metriche essenziali per le operazioni aziendali:

- Latenza richiesta: Tempo di esecuzione per strumento e percentili
- Metriche di autenticazione: Percentuali di operazioni riuscite/non riuscite e tempi di risposta
- Decisioni di autorizzazione: Frequenza e risultati della valutazione delle politiche
- Accesso segreto: Operazioni di vaulting e modelli di utilizzo delle credenziali
- Percentuali errori: Soglie di avviso e rilevamento degli errori a livello di servizio
- Utilizzo risorse: Utilizzo di CPU, memoria e rete per servizio

Integrazione flusso di lavoro temporale

Per processi complessi e di lunga durata, la piattaforma utilizza Temporal.io:

File: `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"}
```

Installazione e scalabilità

Orchestrazione contenitore

La piattaforma utilizza Docker Compose per lo sviluppo. Kubernetes può essere utilizzato per la produzione di:

File: `docker-compose.yml` (estratto)

```
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]
```

Considerazioni su prestazioni e sicurezza

Procedure ottimali per la sicurezza

1. Architettura senza trust: Ogni richiesta è autenticata e autorizzata
2. Rotazione segreto: Rotazione segreta automatizzata tramite vaulting
3. Segmentazione della rete: Service mesh con mTLS
4. Registrazione di controllo: Audit trail completo in ELK

Ottimizzazioni delle prestazioni

1. Pool di connessioni: Riutilizza connessioni HTTP a API esterne
2. Memorizzazione nella cache: Cache basata su Redis per i dati ad accesso frequente
3. Elaborazione asincrona: I/O non bloccante nell'intero stack
4. Bilanciamento del carico: Distribuire il carico tra più istanze del server MCP

Monitoraggio delle metriche

Le metriche chiave rilevate includono:

- Latenza richiesta per strumento MCP
- Percentuali di operazioni riuscite/non riuscite di autenticazione
- Decisioni di autorizzazione per risorsa
- Frequenza di recupero dei segreti
- Frequenze di errore per componente del servizio

Metriche delle prestazioni e risultati

Durante il test delle prestazioni, la piattaforma ha ottenuto:

- Latenza inferiore a 100 ms per le decisioni di autenticazione
- 99,9% di tempo di attività in tutti i servizi MCP
- Scalabilità lineare fino a 1.000 utenti simultanei
- 90% di riduzione dei tempi di sviluppo dell'integrazione

Lezioni apprese e procedure ottimali

Principali fattori di successo

1. Standardizzazione: Le librerie comuni riducono la duplicazione e i bug
2. Osservabilità: Registrazione completa per una rapida risoluzione dei problemi
3. Sicurezza fin dalla progettazione: Autenticazione e autorizzazione a partire dal primo giorno
4. Modularità: I server MCP indipendenti consentono una scalabilità mirata
5. Documentazione: Una documentazione chiara sulle API accelera l'adozione

Problemi comuni da evitare

1. Progettazione eccessiva: Iniziare con semplicità e aggiungere complessità in base alle esigenze
2. Accoppiamento stretto: Mantieni i server MCP in un collegamento libero
3. Security Dopodiché: Creazione della protezione in dall'inizio
4. Test insufficiente: Implementazione di strategie di testing complete
5. Gestione errori inadeguata: Garantire una degradazione graduale

Miglioramenti futuri

La roadmap della piattaforma include:

1. Informazioni approfondite guidate dall'intelligenza artificiale: Rilevamento di anomalie basato su ML
2. Supporto multi-cloud: Distribuzione tra provider di cloud
3. Marketplace flusso di lavoro: Modelli di workflow riutilizzabili
4. Analisi avanzata: Dashboard e report in tempo reale

Conclusioni

La creazione di server MCP di livello produttivo richiede un'attenta considerazione dei requisiti aziendali, tra cui sicurezza, scalabilità, monitoraggio e manutenzione. Questa architettura di riferimento illustra come implementare queste funzionalità utilizzando modelli e strumenti standard.

La struttura modulare consente alle organizzazioni di adottare gradualmente il protocollo MCP, garantendo al contempo la sicurezza e le attività aziendali sin dal primo giorno. Sfruttando tecnologie comprovate come OIDC, OPA, Vault ed ELK, i team possono concentrarsi sulla logica aziendale piuttosto che sulle problematiche dell'infrastruttura.

Informazioni sugli autori

Questo articolo è stato sviluppato dal team MCP Fusioners come parte delle iniziative di innovazione interne di Cisco, dimostrando approcci pratici all'integrazione dei sistemi di IA

aziendali.

Riferimenti

1. Specifica protocollo contesto modello - https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. Documentazione di Open Policy Agent - https://www.openpolicyagent.org/docs
4. Documentazione sugli archivi HashiCorp - https://www.vaultproject.io/docs
5. Documentazione di Temporal.io - https://docs.temporal.io/
6. Guida allo stack ELK - https://www.elastic.co/elastic-stack/

Informazioni su questa traduzione

Cisco ha tradotto questo documento utilizzando una combinazione di tecnologie automatiche e umane per offrire ai nostri utenti in tutto il mondo contenuti di supporto nella propria lingua. Si noti che anche la migliore traduzione automatica non sarà mai accurata come quella fornita da un traduttore professionista. Cisco Systems, Inc. non si assume alcuna responsabilità per l’accuratezza di queste traduzioni e consiglia di consultare sempre il documento originale in inglese (disponibile al link fornito).