Aproveite o poder dos servidores MCP: Revolucione a automação da rede com soluções baseadas em IA

Contents

Referências

Introdução

Este documento descreve uma arquitetura de referência abrangente para criar servidores de Protocolo de Contexto de Modelo (MCP - Model Context Protocol) prontos para produção usando as melhores práticas do setor, demonstradas por meio de uma implementação real que integra o Cisco Catalyst Center, o ServiceNow e outros sistemas corporativos. O MCP representa uma mudança de paradigma em como os sistemas de IA interagem com serviços externos e origens de dados. No entanto, passar do protótipo para a produção exige a implementação de padrões de nível empresarial, incluindo autenticação, autorização, monitoramento e escalabilidade.

Informações de Apoio

À medida que as empresas adotam cada vez mais a automação acionada por IA, a necessidade de plataformas de integração robustas, seguras e escaláveis se torna essencial. As integrações ponto a ponto tradicionais criam sobrecarga de manutenção e vulnerabilidades de segurança. O protocolo de contexto de modelo (MCP) oferece uma abordagem padronizada para integrações do sistema de IA, mas as implantações de produção exigem recursos de nível empresarial que vão além das implementações básicas do MCP.

Este artigo demonstra como criar uma plataforma de servidor MCP pronta para produção que incorpora:

1. Autenticação empresarial: Integração do OpenID Connect (OIDC) com o Cisco Duo
2. Autorização Refinada: Política como código usando o Open Policy Agent (OPA)
3. Gerenciamento de Segredo Seguro: HashiCorp Vault para credenciais e configuração
4. Monitoramento abrangente: Pilha ELK para observabilidade e solução de problemas
5. Orquestração do fluxo de trabalho: Temporal.io para processos complexos e de longa duração
6. Integração legada: Proxies de API REST para sistemas existentes

Por que isso é importante

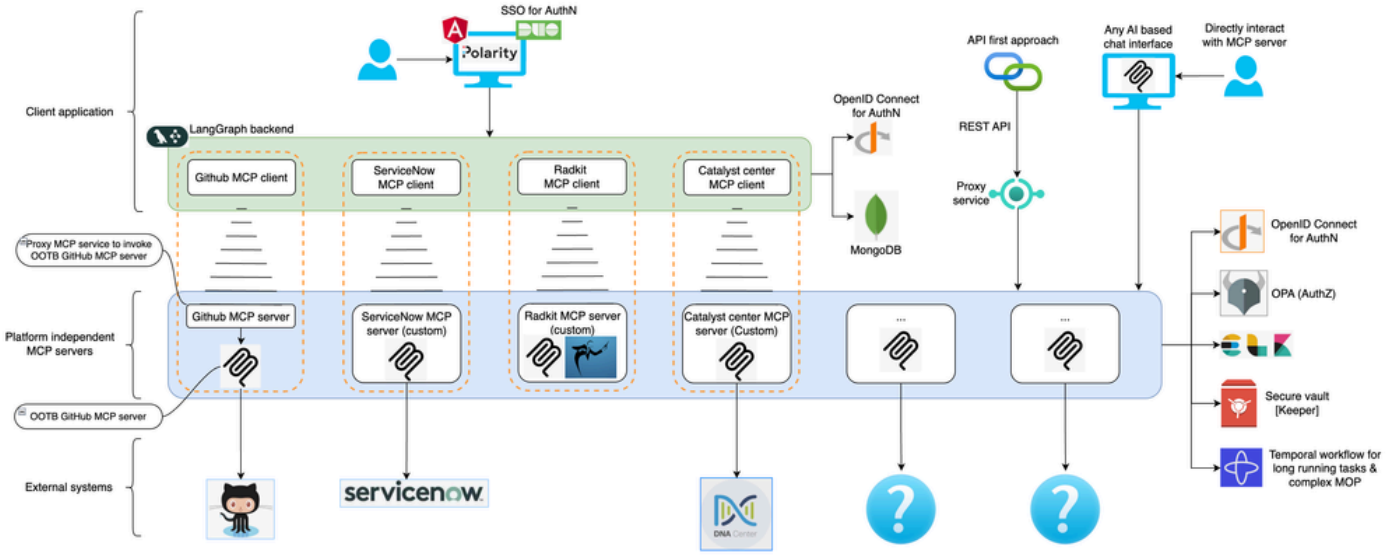
As abordagens tradicionais de integração sofrem de várias limitações:

1. Lacunas de segurança: Credenciais codificadas e acesso privilegiado excessivo
2. Complexidade operacional: Dificuldade de monitorar e solucionar problemas de sistemas distribuídos
3. Problemas de escalabilidade: As integrações ponto-a-ponto não são escaláveis de acordo com os requisitos cada vez maiores
4. Sobrecarga de manutenção: Cada integração requer autenticação personalizada e tratamento de erros

A abordagem do MCP com padrões empresariais aborda esses desafios enquanto fornece uma base padronizada e reutilizável para a automação orientada pela IA.

Visão geral da arquitetura

A arquitetura de referência implementa uma abordagem em camadas que separa os aplicativos clientes da plataforma do servidor MCP, permitindo que vários aplicativos aproveitem a mesma infraestrutura MCP de nível empresarial.



Arquitetura do componente

1. Camada de Aplicação do Cliente

A camada de cliente fornece interfaces de usuário e lógica de orquestração:

- Interface: Aplicação angular usando a estrutura de interface de usuário de polaridade Cisco
- Infraestrutura: Sistema multiagente LangGraph para orquestração de fluxo de trabalho
- Autenticação: Integração do OIDC com provedores de identidade empresarial

2. Camada de Plataforma de Servidor MCP

A camada de plataforma implementa servidores MCP de nível empresarial com serviços compartilhados:

Servidores MCP de núcleo:

- `mcp-catalyst-center`: Gerenciamento de dispositivos de rede da Cisco
- `mcp-service-now`: Integração de ITSM e gerenciamento de tíquetes
- `mcp-github`: Gerenciamento de código-fonte e repositório
- `mcp-radkit`: Análise e monitoramento de rede
- `mcp-rest-api-proxy`: Integração do sistema antigo

Serviços corporativos:

- Serviço de autenticação: Validação de token OIDC e gerenciamento de usuário

- Serviço de Autorização: Aplicação de política baseada em OPA
- Gerenciamento de Segredos: Armazenamento de configuração e credencial baseado em cofre
- <Pilha de monitoramento: ELK para registro, métricas e alertas
- Mecanismo de Fluxo de Trabalho: Temporal para orquestração de processos complexos

Implementação de segurança corporativa

Autenticação do OpenID Connect

A plataforma implementa autenticação de nível empresarial usando o OpenID Connect, proporcionando integração perfeita com os provedores de identidade existentes, ao mesmo tempo em que suporta autenticação de vários fatores através do Cisco Duo.

Principais benefícios

- Logon Único (SSO): Os usuários autenticam uma vez em todos os serviços MCP
- Autenticação Multifator: Cisco Duo integrado para maior segurança
- Segurança baseada em token: Autenticação stateless usando tokens JWT
- Gerenciamento centralizado: Provisionamento e desprovisionamento de usuários por meio do IdP existente

Visão geral da implementação

Arquivo: `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")
```

Autorização refinada com Open Policy Agent

O OPA oferece autorização flexível e por política como código, permitindo um controle de acesso refinado com base nos atributos do usuário, tipos de recursos e informações contextuais.

Estrutura da Política de Autorização

Arquivo: `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"]
}
```

Integração OPA Python

<Arquivo: `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
```

Gerenciamento Seguro de Segredos com HashiCorp Vault

O HashiCorp Vault fornece gerenciamento de segredo de nível empresarial com criptografia, controle de acesso e registro de auditoria. A plataforma MCP integra o Vault para armazenar e recuperar com segurança informações confidenciais, incluindo credenciais de API, senhas de banco de dados e dados de configuração.

Recursos Principais

- Criptografia em repouso e em trânsito: Todos os segredos são criptografados usando a criptografia AES de 256 bits
- Segredos dinâmicos: Gerar credenciais com tempo limitado para serviços externos
- Controle de acesso: Políticas refinadas controlam quem pode acessar quais segredos
- Log de auditoria: Concluir a trilha de auditoria de todas as operações de acesso secreto
- Rotação secreta: Rotação automatizada de credenciais e certificados

Implementação

Arquivo: 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}")
```

Estrutura do servidor Core MCP

Cada servidor MCP contém um padrão consistente com integração de segurança empresarial:

Arquivo: `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 de API REST para integração herdada

A plataforma inclui um proxy de API REST para oferecer suporte a clientes não MCP:

Arquivo: `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}
```

Monitoramento e Observabilidade

Integração de pilha ELK

A plataforma implementa um registro abrangente usando a pilha ELK:

Arquivo: `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)
```

Principais métricas de monitoramento

A plataforma rastreia métricas essenciais para operações corporativas:

- Latência da Solicitação: Percentis e tempo de execução por ferramenta
- Métricas de autenticação: Taxas de sucesso/falha e tempos de resposta
- Decisões de autorização: Frequência e resultados da avaliação das políticas
- Acesso secreto: Operações de cofre e padrões de uso de credenciais
- Taxas de erro: Rastreamento de erros de nível de serviço e limites de alerta
- Utilização de recursos: Uso de CPU, memória e rede por serviço

Integração de Fluxo de Trabalho Temporal

Para processos complexos e de longa execução, a plataforma utiliza o Temporal.io:

Arquivo: `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"}
```

Implantação e escalabilidade

Orquestração de contêineres

A plataforma usa o Docker Compose para desenvolvimento. Kubernetes pode ser usado para produção:

Arquivo: `docker-compose.yml` (trecho)

```
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]
```

Considerações sobre desempenho e segurança

Práticas recomendadas de segurança

1. Arquitetura Zero-Trust: Todas as solicitações são autenticadas e autorizadas
2. Rotação secreta: Rotação secreta automatizada através do Vault
3. Segmentação de rede: Malha de serviço com mTLS
4. Log de auditoria: Trilha de auditoria completa em ELK

Otimizações de desempenho

1. Pooling de Conexão: Reutilizar conexões HTTP para APIs externas
2. Armazenamento em cache: Cache baseado em Redis para dados acessados com frequência
3. Processamento assíncrono: E/S sem bloqueio em toda a pilha
4. Balanceamento de carga: Distribuir a carga em várias instâncias do servidor MCP

Monitorando Métricas

As principais métricas rastreadas incluem:

- Latência de solicitação por ferramenta MCP
- Taxas de êxito/falha de autenticação
- Decisões de autorização por recurso
- Frequência de recuperação de segredo
- Taxas de erro por componente de serviço

Métricas e resultados de desempenho

Durante o teste de desempenho, a plataforma obteve:

- Latência inferior a 100 ms para decisões de autenticação
- 99,9% de tempo de atividade em todos os serviços MCP
- Escalabilidade linear para até 1000 usuários simultâneos
- Redução de 90% no tempo de desenvolvimento da integração

Lições aprendidas e práticas recomendadas

Principais fatores de sucesso

1. Padronização: Bibliotecas comuns reduzem a duplicação e os bugs
2. Observabilidade: O registro abrangente permite a rápida solução de problemas
3. Segurança por projeto: Autenticação e autorização desde o primeiro dia
4. Modularidade: Os servidores MCP independentes permitem o dimensionamento direcionado
5. Documentação: A documentação clara da API acelera a adoção

Armadilhas comuns para evitar

1. Engenharia excessiva: Comece de forma simples e adicione complexidade conforme necessário
2. Acoplamento Estreito: Manter os servidores MCP acoplados livremente
3. Reflexão sobre segurança: Incorpore segurança desde o início
4. Teste insuficiente: Implementar estratégias de teste abrangentes
5. Tratamento insatisfatório de erros: Garantir uma degradação suave

Aprimoramentos futuros

O roteiro da plataforma inclui:

1. Ideias orientadas por IA: Detecção de anomalias com base em ML
2. Suporte a várias nuvens: Implantação em provedores de nuvem
3. Mercado de Fluxo de Trabalho: Modelos de fluxo de trabalho reutilizáveis
4. Análise avançada: Painéis e relatórios em tempo real

Conclusão

A criação de servidores MCP de nível de produção requer uma consideração cuidadosa dos requisitos corporativos, incluindo segurança, escalabilidade, monitoramento e capacidade de manutenção. Essa arquitetura de referência demonstra como implementar esses recursos usando ferramentas e padrões padrão do setor.

O design modular permite que as organizações adotem o MCP gradualmente, ao mesmo tempo em que garante a segurança e as operações de nível empresarial desde o primeiro dia. Aproveitando tecnologias comprovadas como OIDC, OPA, Vault e ELK, as equipes podem se concentrar na lógica empresarial, e não nas questões de infraestrutura.

Sobre os autores

Este artigo foi desenvolvido pela equipe MCP Fusioners como parte das iniciativas internas de inovação da Cisco, demonstrando abordagens práticas para a integração do sistema de IA empresarial.

Referências

1. Especificação do protocolo de contexto de modelo - https://modelcontextprotocol.io/
2. OpenID Connect Core 1.0 - https://openid.net/specs/openid-connect-core-1_0.html
3. Documentação do Open Policy Agent - https://www.openpolicyagent.org/docs
4. Documentação do HashiCorp Vault - https://www.vaultproject.io/docs
5. Documentação do Temporal.io - https://docs.temporal.io/
6. Guia de pilha ELK - https://www.elastic.co/elastic-stack/

Sobre esta tradução
A Cisco traduziu este documento com a ajuda de tecnologias de tradução automática e humana para oferecer conteúdo de suporte aos seus usuários no seu próprio idioma, independentemente da localização.

Observe que mesmo a melhor tradução automática não será tão precisa quanto as realizadas por um tradutor profissional.

A Cisco Systems, Inc. não se responsabiliza pela precisão destas traduções e recomenda que o documento original em inglês (link fornecido) seja sempre consultado.