# Aproveche el potencial de los servidores MCP: Revolucione la automatización de la red con soluciones impulsadas por IA

## Contenido

Referencias

# Introducción

En este documento se describe una arquitectura de referencia completa para crear servidores de protocolo de contexto de modelo (MCP) preparados para la producción utilizando las prácticas recomendadas del sector, demostradas a través de una implementación real que integra Cisco Catalyst Center, ServiceNow y otros sistemas empresariales. El MCP representa un cambio de paradigma en la forma en que los sistemas de IA interactúan con los servicios externos y las fuentes de datos. Sin embargo, para pasar del prototipo a la producción es necesario implementar patrones de nivel empresarial, como autenticación, autorización, supervisión y escalabilidad.

# Antecedentes

A medida que las organizaciones adoptan cada vez más la automatización impulsada por la IA, la necesidad de plataformas de integración robustas, seguras y escalables se hace crítica. Las integraciones punto a punto tradicionales crean sobrecargas de mantenimiento y vulnerabilidades de seguridad. El protocolo de contexto de modelo (MCP) ofrece un enfoque estandarizado para las integraciones de sistemas de IA, pero las implementaciones de producción requieren capacidades de nivel empresarial que vayan más allá de las implementaciones básicas de MCP.

En este artículo se muestra cómo crear una plataforma de servidor MCP preparada para la producción que incorpore:

1. Autenticación empresarial: Integración de OpenID Connect (IDC) con Cisco Duo
2. Autorización rigurosa: Política como código mediante Open Policy Agent (OPA)
3. Administración secreta segura: HashiCorp Vault para credenciales y configuración
4. Supervisión completa: Pila ELK para la observación y la resolución de problemas
5. Orquestación de flujo de trabajo: Temporal.io para procesos complejos de larga duración
6. Integración antigua: Proxies de API REST para sistemas existentes

## Por qué esto importa

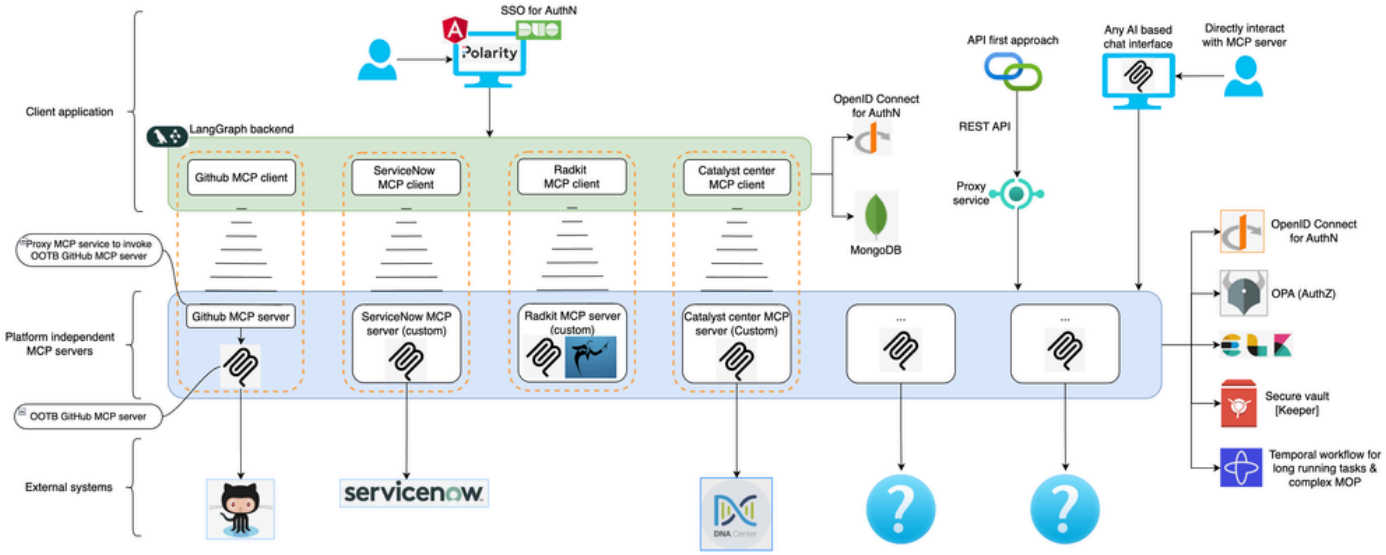
Los enfoques tradicionales de integración tienen varias limitaciones:

1. Lagunas de seguridad: Credenciales codificadas y acceso privilegiado
2. Complejidad operativa: Dificultad para supervisar y solucionar problemas de sistemas distribuidos
3. Problemas de escalabilidad: Las integraciones punto a punto no se amplían a medida que crecen los requisitos
4. Sobrecarga de Mantenimiento: Cada integración requiere autenticación personalizada y control de errores

El enfoque de MCP con patrones empresariales aborda estos retos al tiempo que proporciona una base estandarizada y reutilizable para la automatización impulsada por la IA.

# Descripción general de la arquitectura

La arquitectura de referencia implementa un enfoque por capas que separa las aplicaciones cliente de la plataforma de servidor MCP, lo que permite que varias aplicaciones aprovechen la misma infraestructura MCP de nivel empresarial.




## Arquitectura de componentes

1. Capa de aplicación cliente

La capa de cliente proporciona interfaces de usuario y lógica de orquestación:

- Frontend: Aplicación angular que utiliza el marco de interfaz de usuario de Cisco Polarity
- Motor: Sistema multiagente de LangGraph para la orquestación del flujo de trabajo
- Autenticación: Integración de OIDC con proveedores de identidad empresarial

2. Capa de plataforma de servidor MCP

La capa de plataforma implementa servidores MCP de nivel empresarial con servicios compartidos:

Servidores MCP de núcleo:

- `mcp-catalyst-center`: Gestión de dispositivos de red de Cisco
- `mcp-service-now`: Integración de ITSM y gestión de tickets
- `mcp-github`: Administración de repositorio y código fuente
- `mcp-radkit`: Supervisión y análisis de red
- `mcp-rest-api-proxy`: Integración de sistemas heredados

Servicios empresariales:

- Servicio de autenticación: Validación de tokens y gestión de usuarios de OIDC

- Servicio de autorización: Aplicación de políticas basada en OPA
- Administración secreta: Almacenamiento de configuración y credenciales basado en almacén
- <Pila de supervisión: ELK para registro, métricas y alertas
- Motor de flujo de trabajo: Temporal para la orquestación de procesos complejos

# Implementación de seguridad empresarial

## Autenticación de OpenID Connect

La plataforma implementa la autenticación de nivel empresarial mediante OpenID Connect, lo que proporciona una integración perfecta con los proveedores de identidad existentes, a la vez que admite la autenticación de varios factores a través de Cisco Duo.

### Ventajas clave

- Inicio de sesión único (SSO): Los usuarios se autentican una vez en todos los servicios MCP
- Autenticación de varios factores: Cisco Duo integrado para una mayor seguridad
- Seguridad basada en token: Autenticación sin estado mediante tokens JWT
- Gestión centralizada: Aprovisionamiento y desaprovisionamiento de usuarios a través del IdP existente

### Descripción general de implementación

Archivo: `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")
```

## Autorización detallada con Open Policy Agent

OPA proporciona una autorización flexible de política como código que permite un control de acceso detallado basado en atributos de usuario, tipos de recursos e información contextual.

### Estructura de directiva de autorización

Archivo: `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"]
}
```

## Integración de OPA de Python

<Archivo: 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
```

## Gestión segura de secretos con HashiCorp Vault

HashiCorp Vault proporciona gestión de secretos de clase empresarial con cifrado, control de acceso y registro de auditoría. La plataforma MCP integra Vault para almacenar y recuperar de forma segura información confidencial, como credenciales de API, contraseñas de bases de datos y datos de configuración.

Funciones esenciales

- Cifrado en reposo y en tránsito: Todos los secretos se cifran mediante la encriptación AES de 256 bits
- Secretos dinámicos: Generar credenciales con tiempo limitado para servicios externos
- Control de acceso: Las políticas específicas controlan quién puede acceder a qué secretos
- Registro de auditoría: Seguimiento de auditoría completo de todas las operaciones de acceso secreto
- Rotación secreta: Rotación automatizada de credenciales y certificados

Instrumentación

Archivo: 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}")
```

## Estructura del servidor MCP principal

Cada servidor MCP contiene un patrón coherente con la integración de seguridad empresarial:

Archivo: `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 para integración heredada

La plataforma incluye un proxy de API REST para admitir clientes que no sean MCP:

Archivo: `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}
```

# Monitoreo y Observabilidad

## Integración de la pila ELK

La plataforma implementa un registro completo utilizando la pila ELK:

Archivo: `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)
```

### Métricas de supervisión clave

La plataforma realiza un seguimiento de las métricas esenciales para las operaciones empresariales:

- Latencia de solicitud: Tiempo de ejecución por herramienta y percentiles
- Métricas de Autenticación: Tasas de éxito/fracaso y tiempos de respuesta
- Decisiones de autorización: Frecuencia y resultados de la evaluación de políticas
- Acceso secreto: Operaciones de almacén y patrones de uso de credenciales
- Tasas de error: Umbrales de alertas y seguimiento de errores de nivel de servicio
- Utilización de recursos: Uso de CPU, memoria y red por servicio

## Integración de flujo de trabajo temporal

Para procesos complejos y de larga duración, la plataforma aprovecha Temporal.io:

Archivo: `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"}
```

# Implementación y escalabilidad

## Orquestación de contenedores

La plataforma utiliza Docker Compose para el desarrollo. Kubernetes se puede utilizar para la producción:

Archivo: `docker-compose.yml` (extracto)

```
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]
```

# Consideraciones de rendimiento y seguridad

## Prácticas recomendadas de seguridad

1. Arquitectura sin confianza: Cada solicitud se autentica y se autoriza
2. Rotación secreta: Rotación secreta automatizada mediante Vault
3. Segmentación de red: Malla de servicio con mTLS
4. Registro de auditoría: Amplia pista de auditoría en ELK

## Optimizaciones de rendimiento

1. Agrupación de conexiones: Reutilización de conexiones HTTP a API externas
2. Almacenamiento en la memoria caché: Almacenamiento en caché basado en redis para datos a los que se accede con frecuencia
3. Procesamiento asíncrono: E/S sin bloqueos en toda la pila
4. Equilibrio de carga: Distribuir la carga entre varias instancias de servidor MCP

## Supervisión de métricas

Las métricas clave controladas incluyen:

- Latencia de solicitud por herramienta MCP
- Tasas de éxito/fracaso de la autenticación
- Decisiones de autorización por recurso
- Frecuencia de recuperación secreta
- Tasas de error por componente de servicio

# Métricas de rendimiento y resultados

Durante las pruebas de rendimiento, la plataforma logró:

- Latencia inferior a 100 ms para decisiones de autenticación
- 99,9% de tiempo de actividad en todos los servicios de MCP
- Escalabilidad lineal hasta 1000 usuarios simultáneos
- Reducción del 90% del tiempo de desarrollo de la integración

# Lecciones aprendidas y mejores prácticas

## Factores clave del éxito

1. Estandarización: Las bibliotecas comunes reducen la duplicación y los errores
2. Observabilidad: Registro completo que permite una rápida resolución de problemas
3. Seguridad por diseño: Autenticación y autorización desde el primer día
4. Modularidad: Los servidores MCP independientes permiten la ampliación dirigida
5. Documentación: La documentación API clara acelera la adopción

## Obstáculos habituales que se deben evitar

1. Ingeniería excesiva: Comience con una estrategia sencilla y añada complejidad según sea necesario
2. Acoplamiento estrecho: Mantener los servidores MCP sin conexión
3. Reflexión sobre seguridad: Aumente la seguridad desde el principio
4. Pruebas insuficientes: Implementar estrategias de prueba completas
5. Gestión de errores deficiente: Garantizar una degradación fluida

# Mejoras futuras

La guía básica de la plataforma incluye:

1. Perspectivas impulsadas por la IA: Detección de anomalías basada en ML
2. Compatibilidad con varias nubes: Implementación entre proveedores de nube
3. Mercado de flujo de trabajo: Plantillas de flujo de trabajo reutilizables
4. Análisis avanzado: Informes y paneles en tiempo real

# Conclusión

La creación de servidores MCP de nivel de producción requiere una cuidadosa consideración de los requisitos empresariales, incluidos los de seguridad, escalabilidad, supervisión y mantenimiento. Esta arquitectura de referencia demuestra cómo implementar estas capacidades mediante herramientas y patrones estándares del sector.

El diseño modular permite a las organizaciones adoptar MCP de forma gradual, a la vez que

garantiza la seguridad y las operaciones de nivel empresarial desde el primer día. Al aprovechar tecnologías de eficacia probada como OIDC, OPA, Vault y ELK, los equipos pueden centrarse en la lógica empresarial en lugar de en las preocupaciones relacionadas con la infraestructura.

## Acerca de los autores

Este artículo ha sido desarrollado por el equipo de Fusionadores de MCP como parte de las iniciativas de innovación interna de Cisco, y muestra enfoques prácticos para la integración de sistemas de IA empresarial.

## Referencias


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

Acerca de esta traducción
Cisco ha traducido este documento combinando la traducción automática y los recursos humanos a fin de ofrecer a nuestros usuarios en todo el mundo contenido en su propio idioma.

Tenga en cuenta que incluso la mejor traducción automática podría no ser tan precisa como la proporcionada por un traductor profesional.

Cisco Systems, Inc. no asume ninguna responsabilidad por la precisión de estas traducciones y recomienda remitirse siempre al documento original escrito en inglés (insertar vínculo URL).