# **MCP\_Guarda – Architecture Graph (Neo4j), Cypher, Build Plan, and Pitch Prep**

# **The problem**

# MCP is a USB-C–style port for AI tools, and it can directly touch files, networks, prod systems, etc. That’s powerful—and dangerous if not fenced. [Anthropic](https://docs.anthropic.com/en/docs/mcp?utm_source=chatgpt.com)

# Official MCP docs explicitly say the protocol can’t enforce security by itself; implementors must add consent, access control, and data protections. [Model Context Protocol](https://modelcontextprotocol.io/specification/2025-03-26?utm_source=chatgpt.com)

# Security notes warn that local MCP servers run **with your user permissions** (so they can do anything you can do), and Anthropic expects *you* to audit/permit servers. [Model Context Protocol](https://modelcontextprotocol.io/quickstart/user?utm_source=chatgpt.com)[Anthropic](https://docs.anthropic.com/en/docs/claude-code/security?utm_source=chatgpt.com)

# Emerging guidance and vendor posts echo the same themes: over-broad scopes, server compromise, prompt-injection, and the need for fine-grained, externalized authorization.

# **policy proxy + consent gateway**

A tiny, local, drop-in **JSON-RPC proxy** that sits between your MCP client (e.g., Claude Desktop) and any MCP server. It enforces **deny-by-default** policies, shows **human-readable approvals** for risky actions, and sandboxes server side-effects.

**Why this is simple & effective**

* No new crypto. We use **WebAuthn/OS biometrics** for user-presence approvals.
* No protocol changes. It’s a **shim** that speaks MCP on both sides and applies rules in the middle.
* Works today with any server the client can launch (you point the client at mcp-guard which then launches the real server).

## **Core features (MVP)**

1. **Scope & method allowlists**
   * Per-server, per-tool, per-method policy (e.g., files.read, files.write, net.fetch, shell.exec).
   * Defaults: read = allow with logging; write/exec/network = **Ask** (user approval), with optional “remember for 10 min”.
   * Mirrors MCP best-practice advice to add consent & access control beyond the spec. [Model Context Protocol+1](https://modelcontextprotocol.io/specification/draft/basic/security_best_practices?utm_source=chatgpt.com)
2. **Intent binding in plain English**
   * Every approval dialog summarizes the *exact* action (path diff for file writes, URL+method for network calls, command/args for exec).  
     The proxy only permits the request that matches the presented summary; anything else requires a fresh approval.
3. **Filesystem & network sandboxing**
   * Path allowlists (e.g., only ~/Projects/foo), and automatic *write-quarantine* (writes go to a staging dir until approved).
   * Network egress allowlists (e.g., only api.github.com, company-internal), block everything else by default.
   * This aligns with “least privilege” and sandboxing recommendations you see in MCP guidance. [Model Context Protocol](https://modelcontextprotocol.io/specification/draft/basic/security_best_practices?utm_source=chatgpt.com)
4. **Externalized authorization policies**
   * Simple YAML today; pluggable policy engines later (OPA/Cerbos) for enterprises, matching current proposals for fine-grained auth. [cerbos.dev](https://www.cerbos.dev/blog/dynamic-authorization-for-ai-agents-guide-to-fine-grained-permissions-mcp-servers?utm_source=chatgpt.com)
5. **Secrets brokering (no raw tokens to servers)**
   * The proxy injects short-lived, scoped **ephemeral tokens** per request and rotates them—servers never see long-lived creds.
6. **Audit trail**
   * Structured logs for each tool call, decision (Allow/Ask/Deny), and diff for file writes—vital for MCP since traditional tools miss this. [wiz.io](https://www.wiz.io/academy/model-context-protocol-security?utm_source=chatgpt.com)

## 

## 

## 

## **Final Architecture (components & responsibilities)**

**Actors**

* **User** — human operator.
* **MCP Client** — e.g., Claude Desktop or any MCP‑capable agent.
* **MCP Servers** — local/remote tools (files, shell, HTTP, git, cloud APIs, etc.).

**Core components (MCP\_Guarda)**

1. **Proxy Bridge** (JSON‑RPC 2.0): speaks MCP on both sides; launches/attaches to MCP servers; routes requests/responses.
2. **Policy Engine**: evaluates requests using
   * *capability class* (files.read/write, net.fetch, shell.exec, etc.),
   * *context* (resource path/host, calling tool, client origin, tenant),
   * *mode* (allow/ask/deny),
   * *time/rate* rules.
3. **Consent Service**: presents human‑readable **intent summaries** and requires **user‑presence** (WebAuthn/OS biometric) for *ask* rules; caches short approvals (e.g., 10 min window) keyed to exact intent.
4. **Filesystem Guard**: path allowlists, write‑quarantine (staging dir), safe diff display for writes; prevents path traversal.
5. **Network Guard**: DNS/host allowlists, method filters, URL canonicalization; blocks egress by default.
6. **Secrets Broker**: injects **ephemeral, scoped tokens** into requests; never hands long‑lived secrets to MCP servers.
7. **Token Service**: issues short‑lived signed tokens for downstream calls and for **client → proxy** sessions.
8. **Audit Logger**: structured, append‑only log of all tool calls, policy decisions, approvals, and side‑effects (with hashes/diffs).
9. **Sandbox Manager** (optional in PoC): containerizes untrusted servers; bind‑mounts only allowed paths; egress controls.
10. **Config & Policy Store**: YAML/JSON files for PoC; pluggable OPA/Cerbos later.

**Key relationships**

* MCP Client → **Proxy Bridge** → MCP Server(s).
* Proxy consults **Policy Engine**; if *ask*, it calls **Consent Service**.
* **Filesystem/Network Guards** enforce on the outbound side before executing.
* **Secrets Broker** provides scoped creds when policy allows.
* **Audit Logger** records every request, decision, and resulting side‑effect.