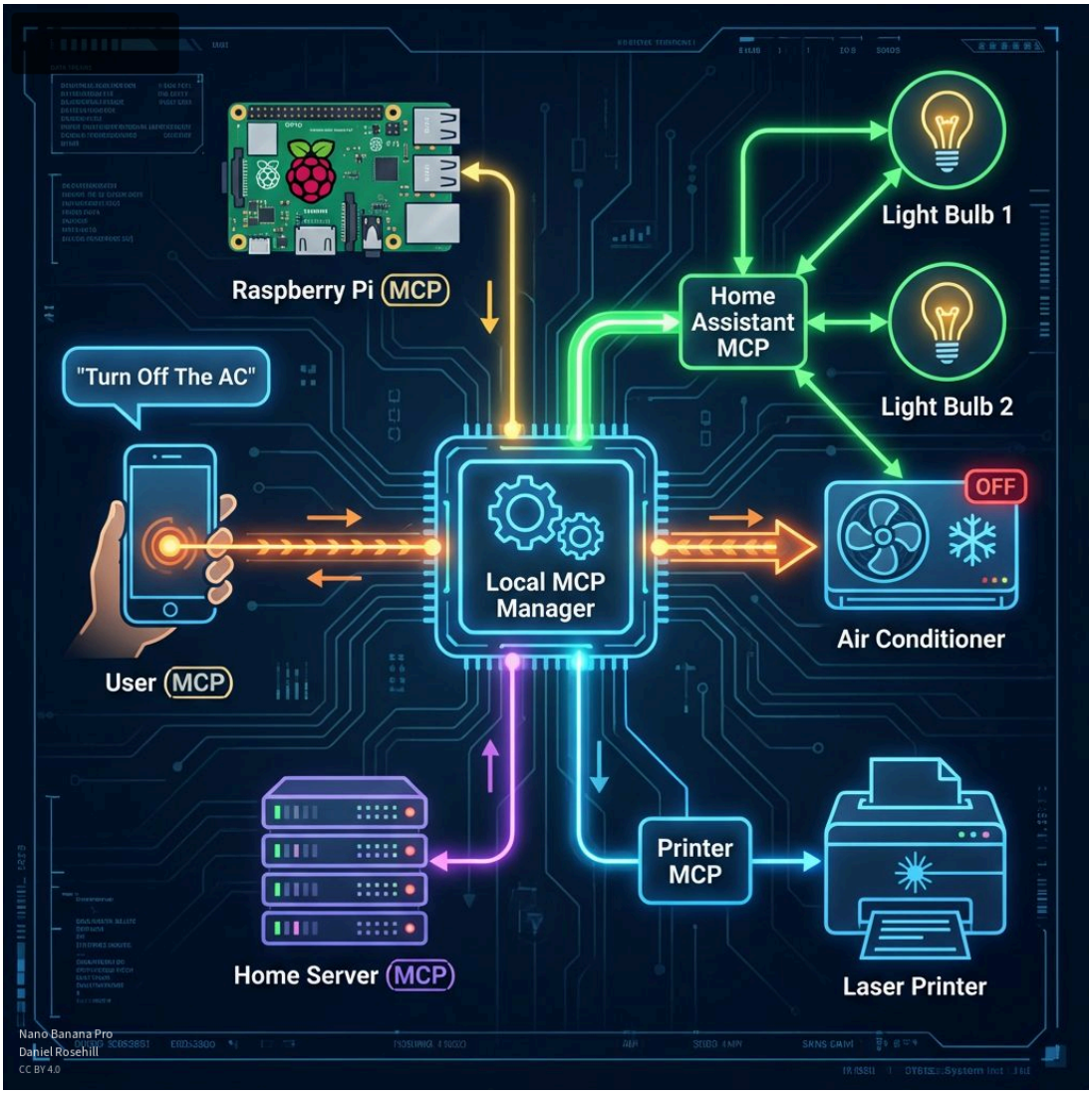
MCP Network Manager Pattern

Local Resource Aggregation Through MCP



[Daniel Rosehill]

DSRHoldings • AI and Cloud Automation

25 December 2025

At a Glance

Pattern

Chained MCP servers enable lightweight, stable administration of resource-constrained devices across local networks—without the overhead of persistent SSH connections or remote agent execution.

Key Advantages

- 5MB idle memory vs 50-100MB for SSH sessions
- Stateless HTTP requests eliminate connection overhead
- Gateway authentication simplifies multi-agent workflows

A pattern for managing networked devices through lightweight MCP servers. Originally developed for homelab administration, but applicable to any local network environment—home, small business, or enterprise.

This document covers three things:

1. **Why MCP beats the alternatives** — Three approaches that look similar on paper but produce very different results in practice
2. **Aggregation** — Why deploying MCP servers one-by-one doesn't scale, and how to bundle them through a central gateway
3. **Remote access** — Taking the pattern from local-only to anywhere

Contents

Part 1: Three Approaches, Very Different Results

AI agents like Claude Code are excellent system administrators, but resource-constrained devices (Raspberry Pi, SBCs, embedded systems) struggle with traditional remote access methods. SSH sessions drop. Running agents directly on the device exhausts memory. Nothing works reliably.

Here are three ways to approach this problem. They look similar. They're not.

Approach 1: SSH from Local Agent

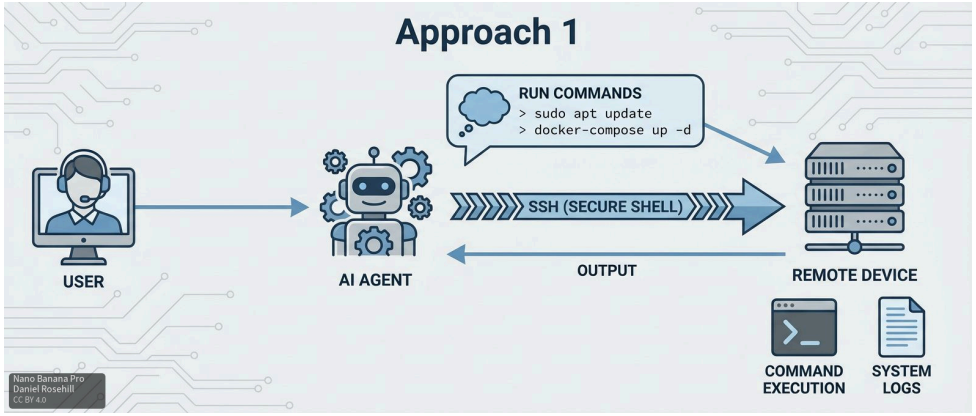


Figure 1: SSH from local agent — *Diagram: Nano Banana Pro*

The agent runs locally and dispatches SSH commands to the remote device.

- SSH maintains persistent connections with overhead
- Works, but exhibits latency and connection instability
- Familiar pattern, inconsistent results

Approach 2: Run the Agent on the Remote Device

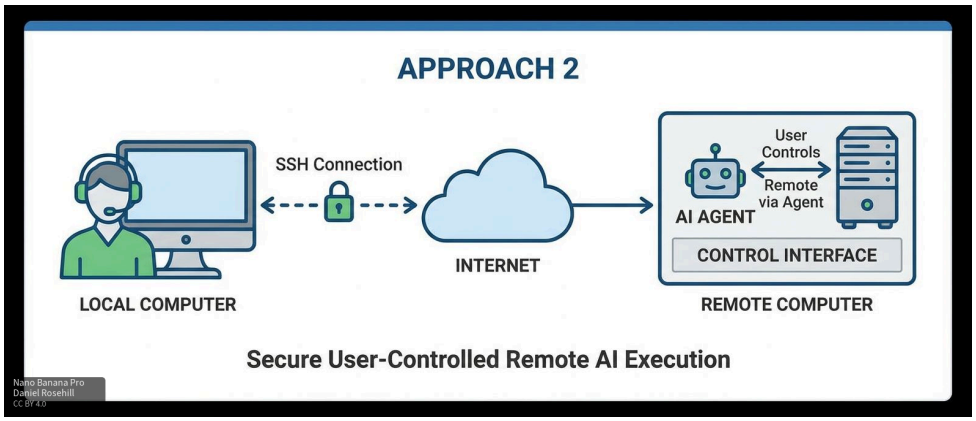


Figure 2: Remote agent execution — *Diagram: Nano Banana Pro*

SSH into the device and run the AI agent there.

- Agent memory + SSH session overhead combined
- Rapidly exhausts resources on constrained hardware

- Results in OOM kills and shell failures

Approach 3: MCP Server on Remote Device

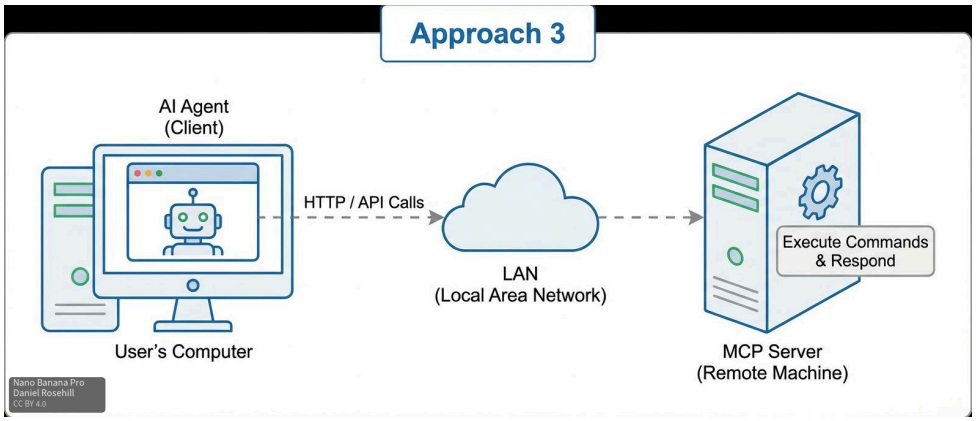


Figure 3: MCP server pattern — *Diagram: Nano Banana Pro*

The agent stays local (with full resources). A lightweight MCP server on the remote device handles requests.

- Agent runs locally with full system resources
- MCP server idles at 5MB RAM
- Stateless HTTP eliminates connection overhead
- Device administration through structured tool calls

This is the pattern. It can run standalone for a single device, or scale into the hub-and-spoke architecture below.

Why the Difference Matters

Aspect	SSH Session	MCP HTTP Server
Connection Model	Persistent, stateful	Stateless requests
Terminal Allocation	Requires PTY	No PTY required
Process Model	Shell process maintained	Per-request processing
Idle Memory	50-100MB	5MB
Constrained Hardware	Prone to failure	Graceful handling

The key differentiator is state management. MCP servers drop to near-idle between requests. No agent processes run on the low-resource device—it just listens until called.

Part 2: Aggregation

Deploying individual MCP servers works, but managing them one-by-one doesn't scale. If you have a Pi, a server, Home Assistant, a NAS, and a firewall—that's five separate MCP connections to configure and maintain.

The solution: an MCP aggregator that bundles all your network resources behind a single endpoint.

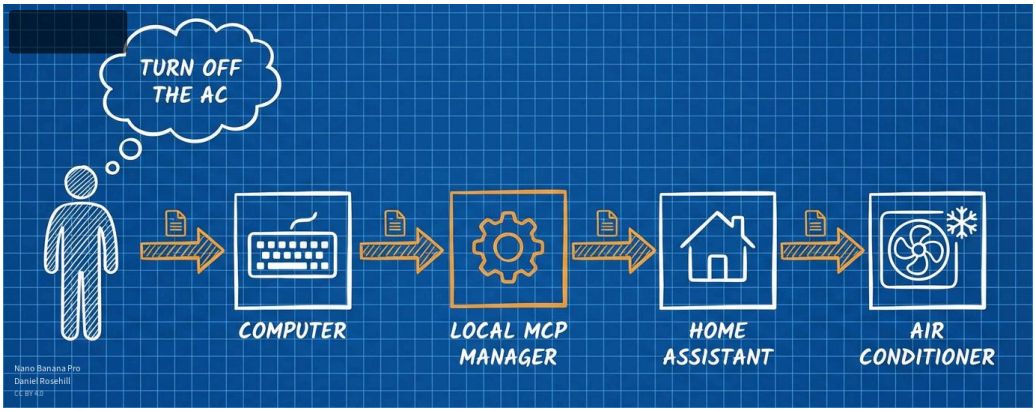


Figure 4: Aggregated MCP architecture — *Diagram: Nano Banana Pro*

How It Works

Your AI client connects to one aggregator. The aggregator connects to all your edge MCP servers:

```
Claude Code  →  MCP Aggregator  →  Pi MCP
                                →  Server MCP
                                →  Home Assistant MCP
                                →  NAS MCP
```

Benefits:

- **Single connection point** for all network resources
- **Centralized authentication** — manage credentials in one place
- **Cross-device workflows** — "check the Pi's temperature, then turn on the fan via Home Assistant"
- **Tool namespacing** — `pi_admin()`, `ha_control()`, `nas_backup()` all available through one interface

What to Aggregate

Some applications already have MCP servers (Home Assistant has one built in). Others need a simple custom server deployed. The aggregator unifies both:

Type	Examples	MCP Status
Native MCP	Home Assistant	Built-in
OS-level	Raspberry Pi, Ubuntu servers	Deploy minimal server
Appliances	OPNsense, Synology NAS	Deploy or build custom
Applications	CUPS printing, Lyrion audio	Build custom

Part 3: Remote Access

So far, everything runs on the local network. But what if you want to manage your home network from anywhere?

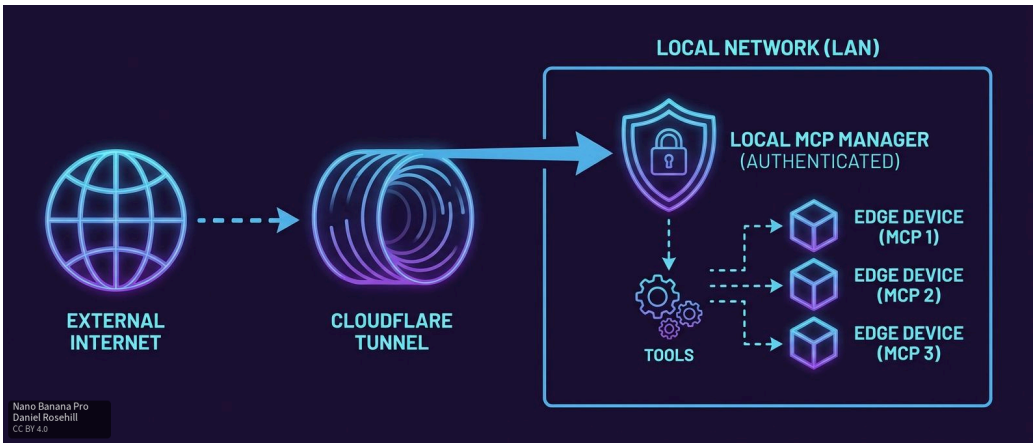


Figure 5: Remote access via tunnel — *Diagram: Nano Banana Pro*

The Pattern

Expose your LAN aggregator through a secure tunnel (Cloudflare Tunnel or Tailscale). Your remote client connects to the tunnel endpoint, which routes to the aggregator, which routes to your devices.

```
Remote Client  →  Cloudflare Tunnel  →  LAN Aggregator  →  Edge MCPs
    (anywhere)        (secure)           (always-on)        (local)
```

Design Principle

The aggregator should run on always-on hardware—not your desktop workstation. A Raspberry Pi works fine for this role. It needs to be available whenever you want remote access.

Chaining Aggregators

The LAN aggregator is itself an MCP server. This means you can chain it with other aggregators like MetaMCP, mixing local network resources with cloud-based MCP endpoints:

```
Claude Code  →  MetaMCP (cloud)  →  LAN Manager MCP (your network)
                                 →  GitHub MCP
                                 →  Notion MCP
                                 →  Other cloud MCPs
```

One client, unified access to everything.

Example Deployment: Home/SMB Network

This deployment demonstrates how a local MCP aggregator can unify control over diverse network resources—combining OS-level controllers, application-specific MCPs, and generic endpoints into a single interface.

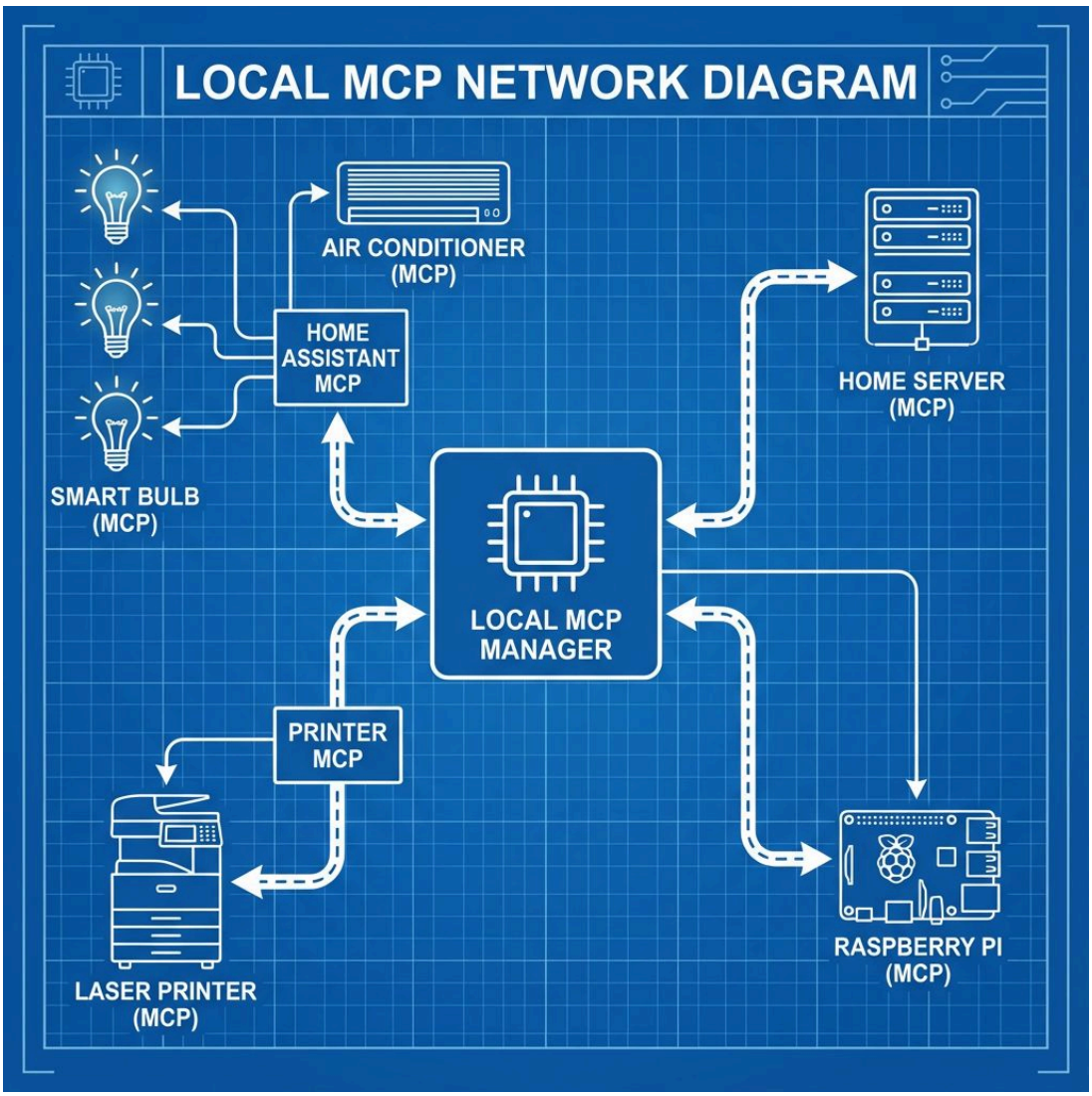


Figure 6: Local MCP network diagram — *Diagram: Nano Banana Pro*

Networked Endpoints

The local MCP manager aggregates the following resources:

Generic / OS Controllers:

- Local VM / home server
- Proxmox (host level controller)
- Raspberry Pi
- Nano Pi x3 (multiroom speakers, Lyrion)

OS-Specific MCPs:

- OPNsense for firewall management

- HAOS (Home Assistant)
- Synology MCP (NAS operations)

Application-Specific MCPs:
- Network CUPS MCP for home printing

MCP Capability Matrix

MCP	Status	Example Tools
Home Assistant	Ready (Built-in)	`HassTurnOn`, `HassLightSet`, `GetLiveContext`
OPNsense	Needs Creation	`create_firewall_rule`, `list_interfaces`
Synology NAS	Needs Creation	`list_shares`, `create_snapshot`
Proxmox VE	Needs Creation	`list_vms`, `start_container`
CUPS (Printing)	Needs Creation	`list_printers`, `print_document`
Raspberry Pi	Needs Creation	`run_command`, `read_file`, `gpio_control`

Implementation

Keep It Simple

- **Aggregator**: Python with FastAPI or Starlette
- **Edge servers**: Minimal Python (100 lines)
- **Tunnel**: Cloudflare Tunnel or Tailscale

Deployment Steps

1. Inventory your LAN devices (Pi, servers, NAS, Home Assistant, etc.)
2. Deploy minimal MCP servers on each target
3. Set up the aggregator on an always-on host
4. Configure remote access through your tunnel of choice
5. Add authentication to the aggregator endpoint

Reference Projects

Project	Description	Use Case
mcp-proxy	Lightweight MCP proxy	stdio-to-SSE conversion
MetaMCP	Multi-server aggregation	Mixed local/remote MCP
FastMCP	Pythonic MCP framework	Rapid aggregator development

Appendix: Minimal MCP Server

A reference implementation for edge devices. Runs at 5MB idle, 100 lines of Python.

```python
#!/usr/bin/env python3
"""Minimal MCP-compatible HTTP server for resource-constrained devices."""
from http.server import HTTPServer, BaseHTTPRequestHandler
import subprocess, json, os

class Handler(BaseHTTPRequestHandler):
    def do_POST(self):
        length = int(self.headers.get('Content-Length', 0))
        data = json.loads(self.rfile.read(length)) if length else {}

        if self.path == '/cmd':
            result = subprocess.run(data.get('cmd', 'echo ok'),
                shell=True, capture_output=True, text=True, timeout=300)
            response = {'stdout': result.stdout, 'stderr': result.stderr,
                        'code': result.returncode}
        elif self.path == '/status':
            # Memory, CPU, disk status
            ...

        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(response).encode())

if __name__ == '__main__':
    HTTPServer(('0.0.0.0', 8222), Handler).serve_forever()
```

Credits

Author: Daniel Rosehill • danielrosehill.com

Business: DSRHoldings • dsrholdings.cloud

Diagrams: Created with Nano Banana Pro

License: CC BY 4.0