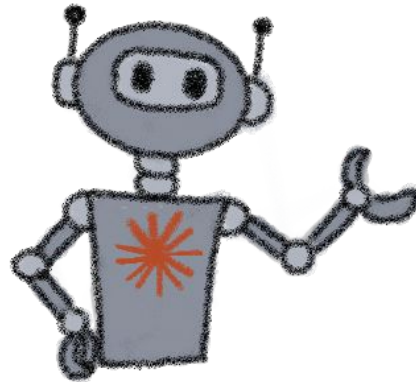
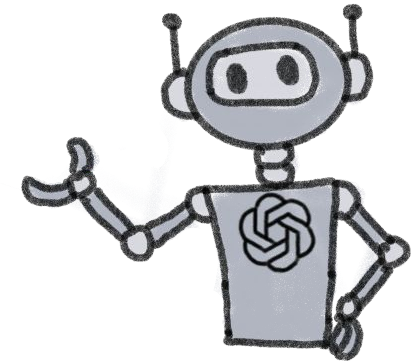


MCP Servers Beyond 101:

Good Practices, Design Choices and Consequences

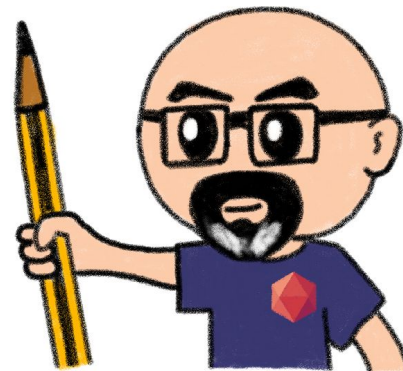
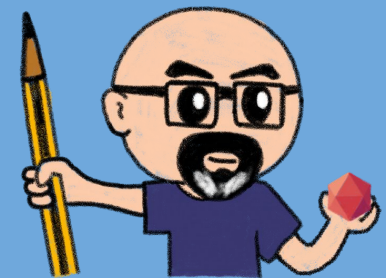


Horacio González
2026-07-06



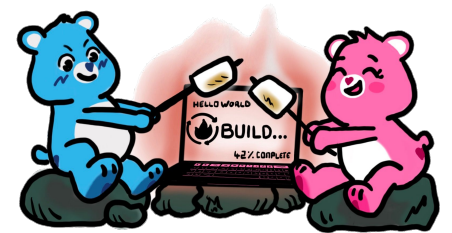
Who are we?

Introducing myself and
introducing Clever Cloud

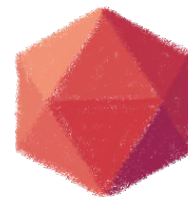


Horacio Gonzalez - @LostInBrittany

Spaniard Lost in Brittany



Head of DevRel



clever cloud



Clever Cloud

From Code to Product

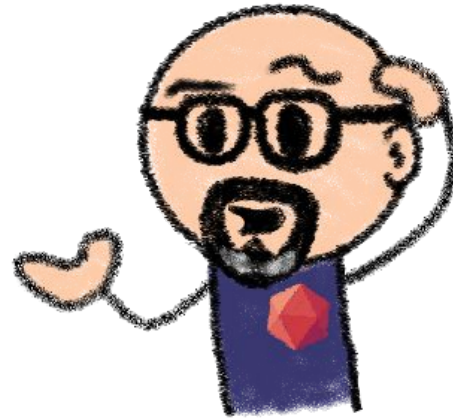


clever cloud



Why this talk matters

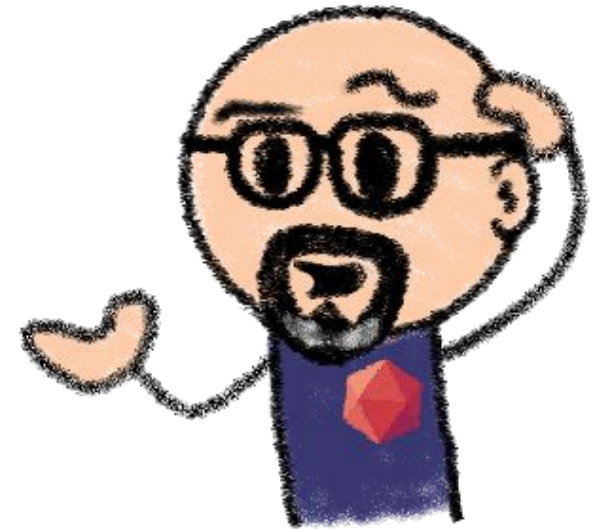
From "what is it" to "how do I build great ones"



Developer Expectations Have Shifted

Winter 2024–2025 (Exploration Phase)

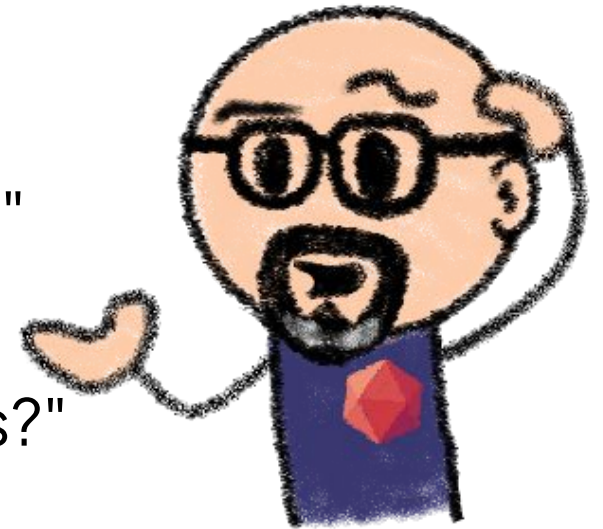
- “What is MCP?”
- “How do I connect my DB?”
- “Can I make a simple server?”
- Focus: *Getting something working*



Developer Expectations Have Shifted

Summer 2025 (Production Readiness)

- "How do I build smarter MCP servers?"
- "How do I secure them?"
- "How do they fit into agent workflows?"
- Focus: *Doing it right*

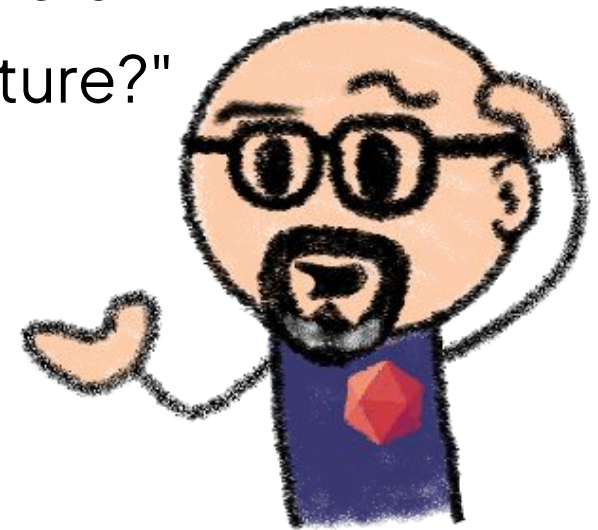


Developer Expectations Have Shifted

1st half 2026

(Best Practices Era) ← We are here

- "How do I design production-grade servers?"
- "How do MCP apps change my architecture?"
- "What patterns should I follow?"
- "How do I test and monitor?"
- Focus: *Building for scale and longevity*



Today's Journey

What We'll Explore Together

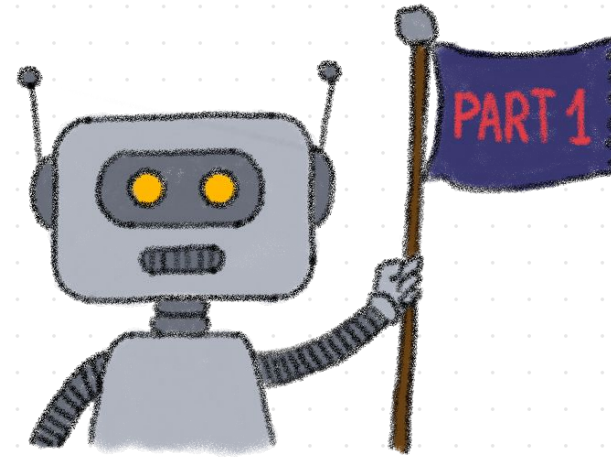
Through a Real Example

- RAGmonsters: from quick prototype to production design
- Seeing design choices and their consequences in action

Core Topics

- Design principles that matter beyond "generic vs specific"
- The full MCP toolkit: Tools, Resources, Prompts
- Security, testing, and observability from the start
- How MCP apps reshape your thinking





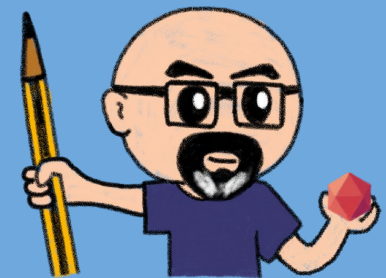
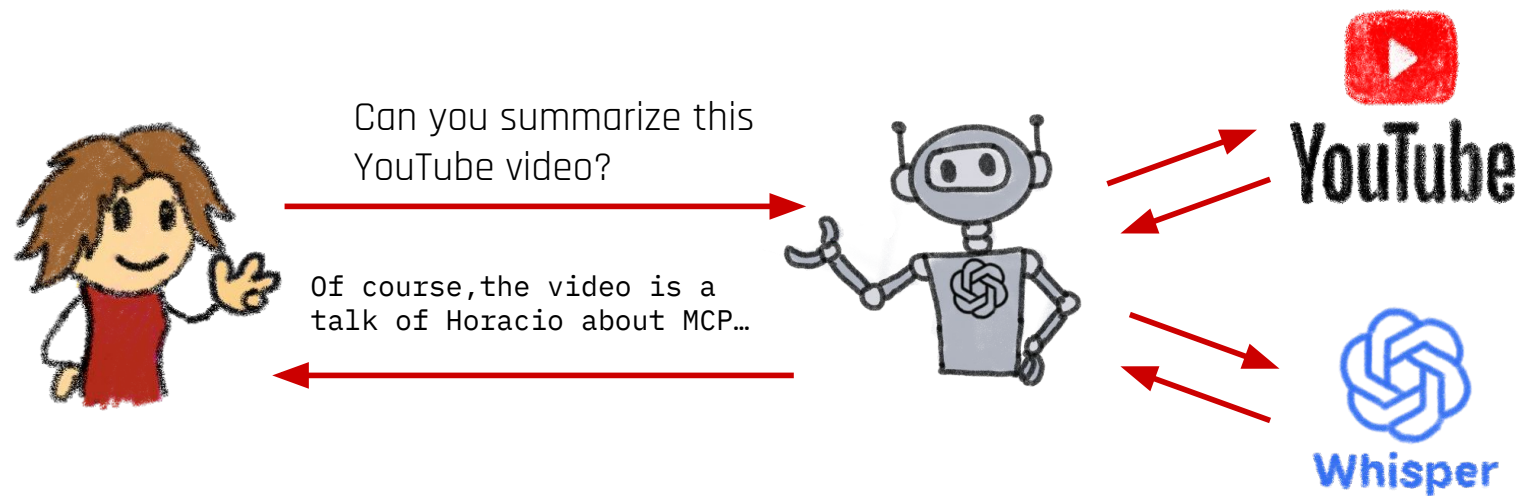
Part I – Works

The agentic revolution, the anatomy of MCP,
and one story about losing data



The Agentic Revolution

From helpers to actors:
How AI learned to do, not just say

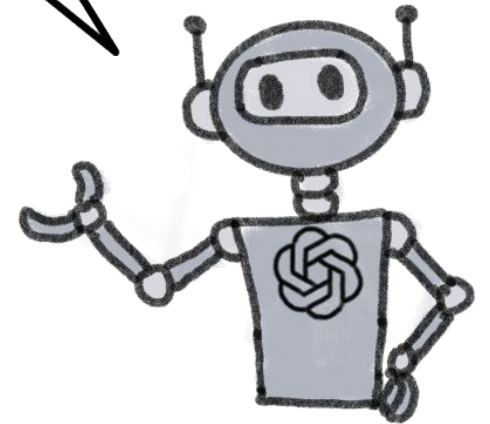


From Chatbots to Agents

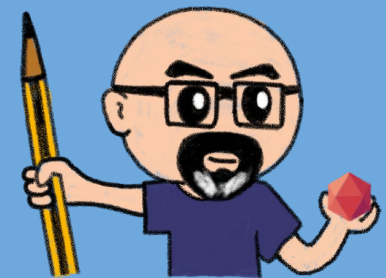
Tell me what to visit in Madrid



Madrid is vibrant, elegant, and full of art, history, and food. Here are the most important things to visit in Madrid, perfect for a first trip 🇪🇸...



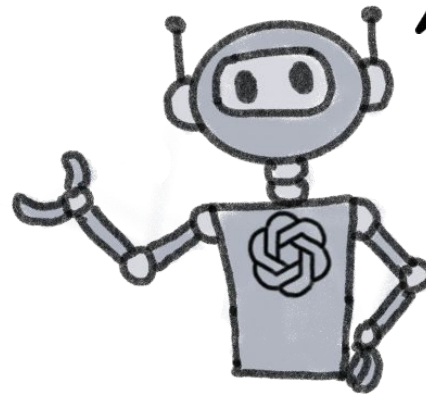
2023: ChatGPT - Smart conversation



From Chatbots to Agents

What's the weather like in Madrid today?

Today it is sunny in Madrid, but very cold, take a coat.



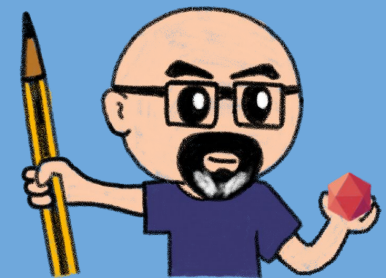
`getWeather("Madrid (ES)")`

`{"weather": "sunny",
"temperature": "1.8°C"}`

Madrid (ES)
1.8°C

3.8 km/h wind

2024: Tool use - LLMs can do things



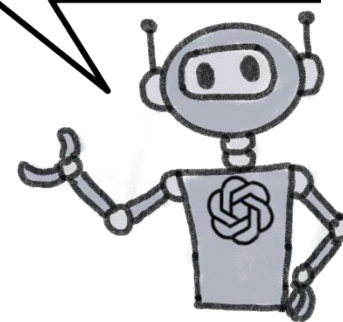
From Chatbots to Agents

Plan a 3-day trip to Madrid for me



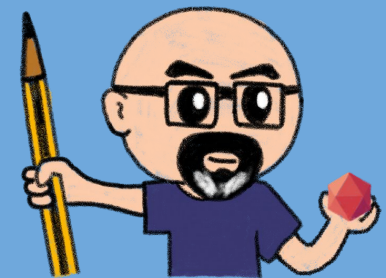
I've planned your Madrid trip:

- Flights booked (€247 round-trip)
 - Hotel near Retiro Park (3 nights)
 - Day 1: Prado Museum + tapas tour
 - Day 2: Royal Palace + Retiro Park
 - Day 3: Reina Sofía + shopping
- All added to your calendar with confirmations sent to your email.



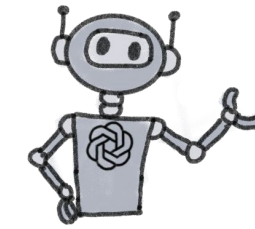
- 🤔 Agent planning:
1. Check weather forecast → calls weather API
 2. Find flights → searches travel APIs
 3. Book accommodation → queries booking sites
 4. Create itinerary → combines museum data, restaurant reviews
 5. Add to calendar → writes calendar entries
 6. Send confirmation → emails summary

2025: Autonomous agents - LLMs that plan and execute

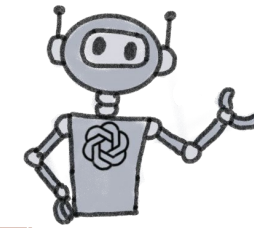


From Chatbots to Agents

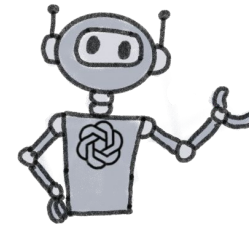
I'm thinking about taking the kids to Madrid this summer...



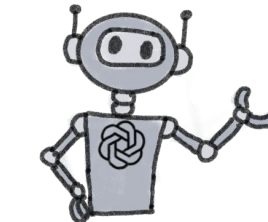
Scans inbox,
finds school
holiday dates



Blocks optimal
week in July



Checks budget,
sets aside
travel funds



Creates family
packing list



2026: Agents are everywhere



The Agent Landscape Today

Coding agents

- Claude Code - Command-line coding assistant
- Cursor - AI-native IDE
- GitHub Copilot Workspace
- Windsurf - Agentic code editor

CLAUDE
CODE

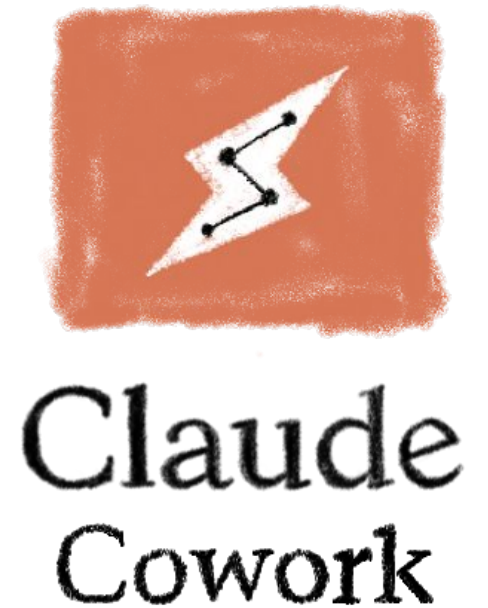
W Windsurf



The Agent Landscape Today

Workplace Agents

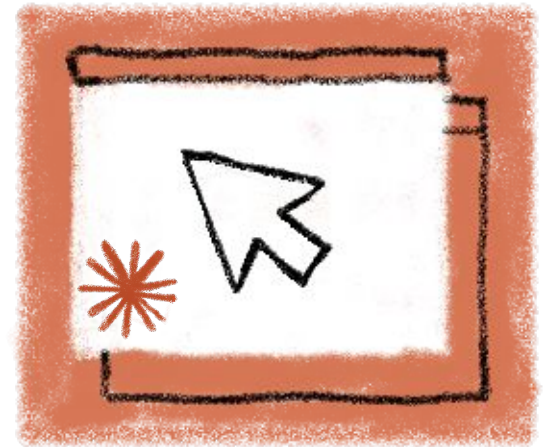
- Claude Cowork - Desktop automation
- Microsoft 365 Copilot - Enterprise integration
- Notion AI - Knowledge base agents



The Agent Landscape Today

Browser Agents

- Claude in Chrome – Web automation
- Browser use libraries
- Testing and scraping agents



The Agent Landscape Today

Custom Agents

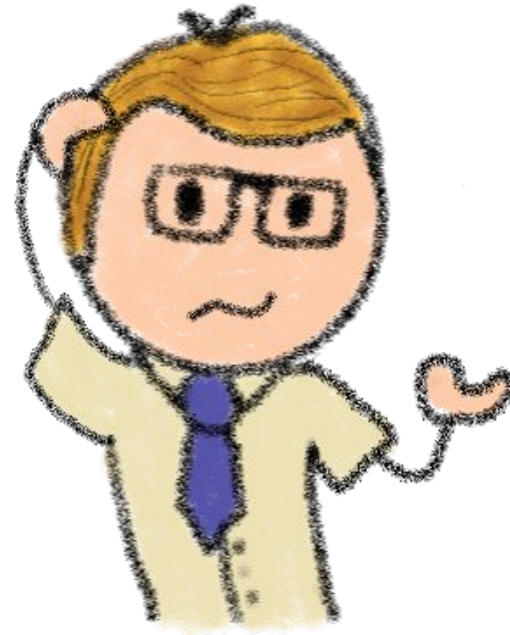
- Companies building internal agents
- Domain-specific automation
- RAG-powered assistants
- Open Claw



OpenClaw



The Agent Landscape Today



The Common Problem:

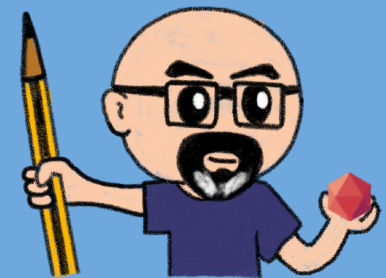
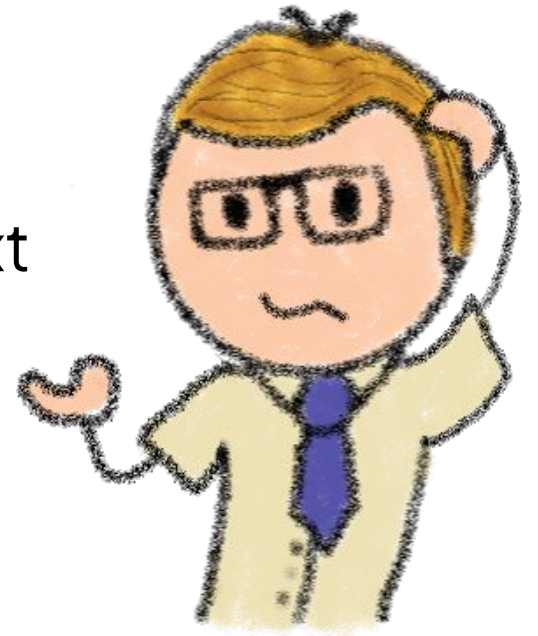
How do agents access YOUR data and tools?



The Connectivity Problem

What Agents Need to Function

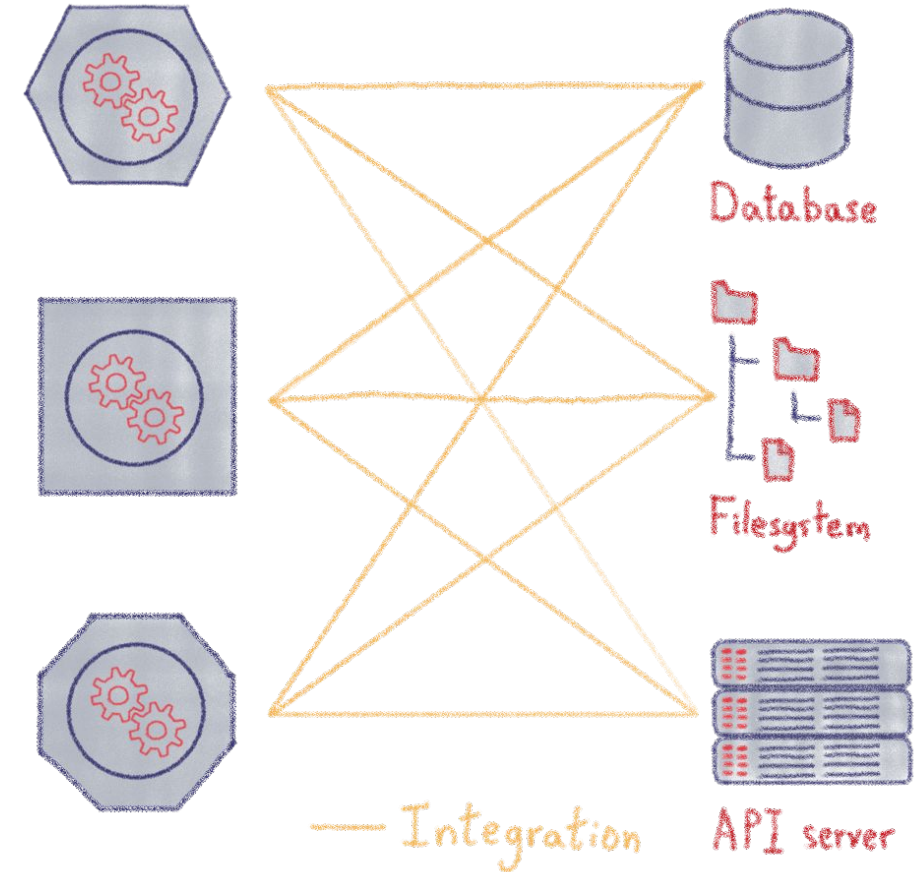
- 📁 Read your files and codebases
- 🗄️ Query your databases
- 🔌 Call your APIs and services
- 🧠 Understand your domain and context
- 🔑 Access private systems securely



The Connectivity Problem

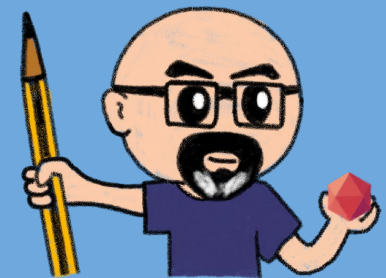
Custom solutions for each integration

- **OpenAI:** Function calling with custom schemas
- **Anthropic:** Tool use with JSON descriptions
- **Google:** Function declarations



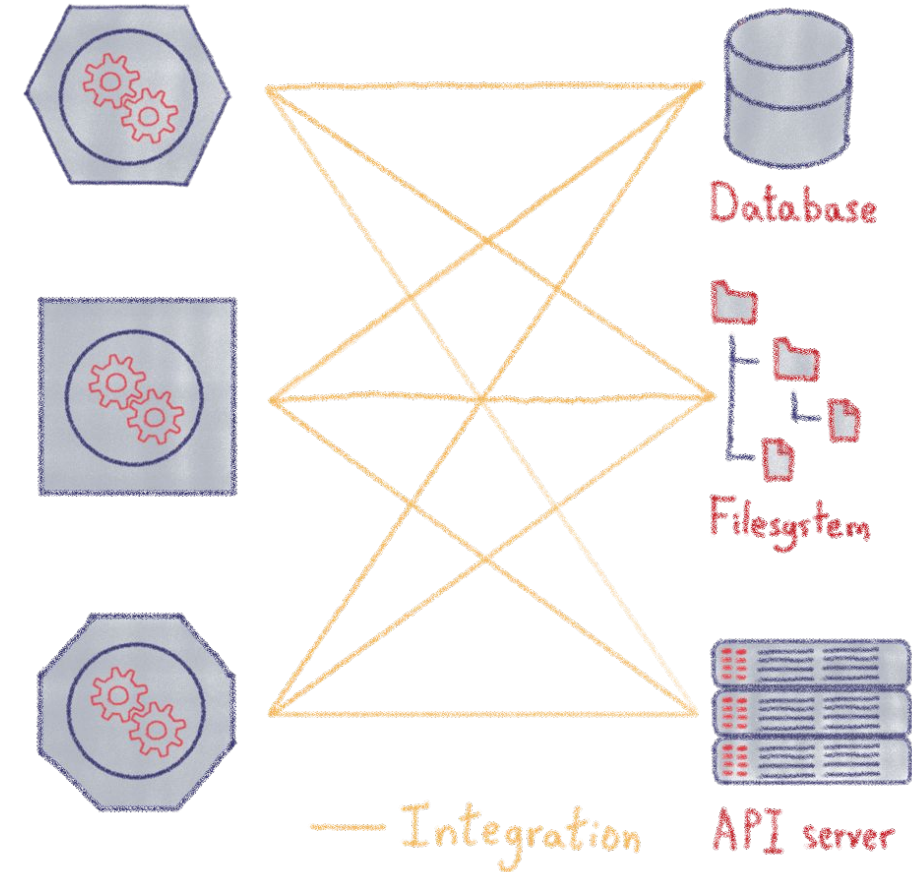
Enter MCP

One protocol to connect them all



Why Do We Need MCP?

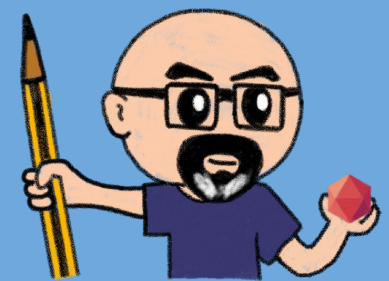
- LLMs **don't automatically know** what functions exist.
- **No standard way** to expose an application's capabilities.
- **Hard to control** security and execution flow.
- Expensive and fragile **integration spaghetti**



Model Context Protocol

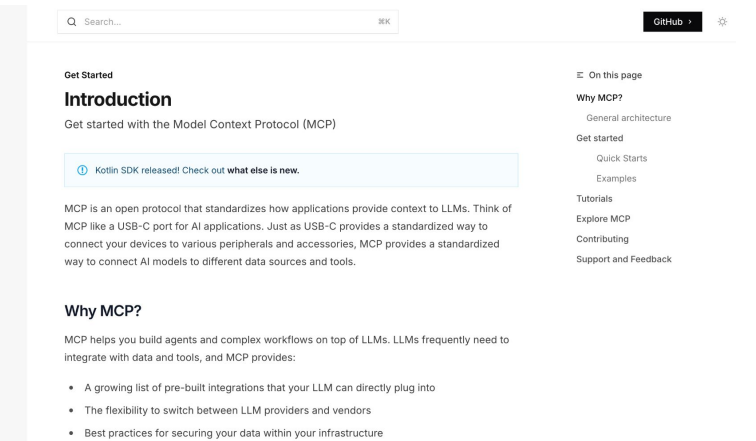
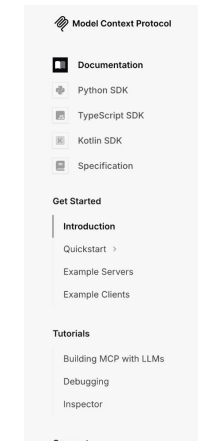


Anthropic, November 2024:
*LLMs intelligence isn't the bottleneck,
connectivity is*



Model Context Protocol

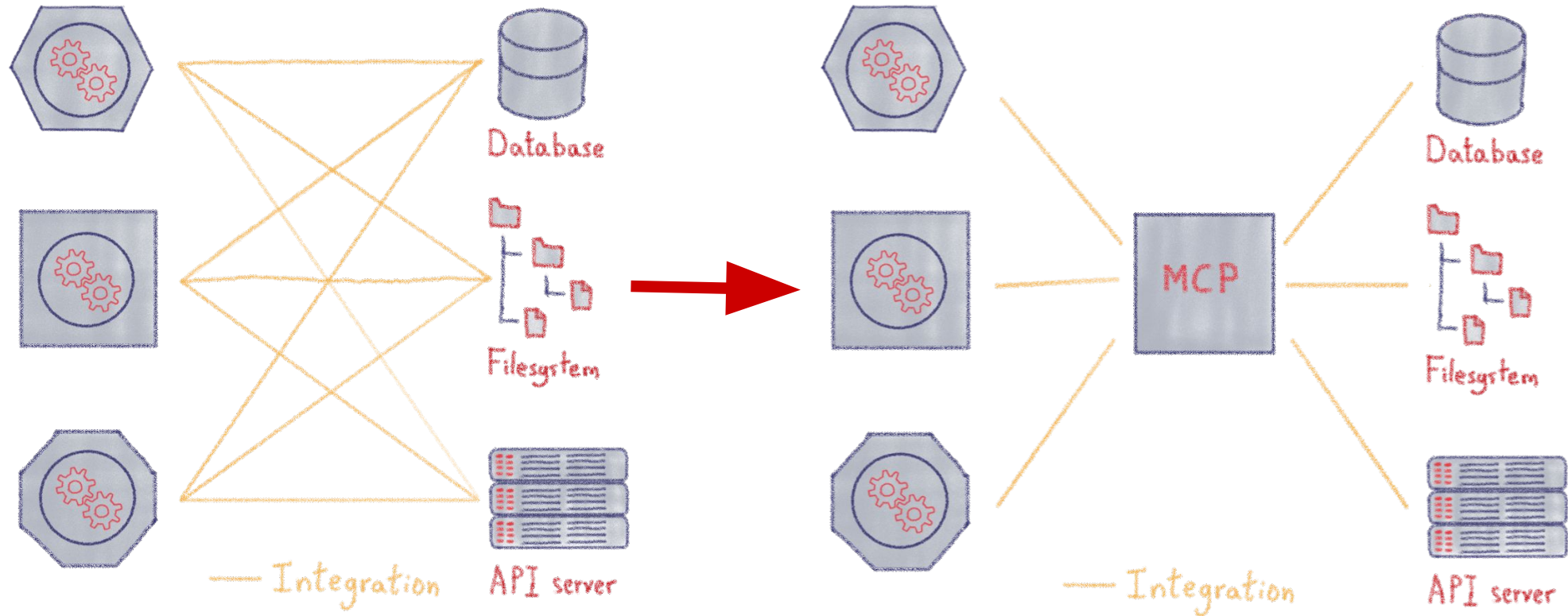
- Standardize the protocol, not the tools
- Abstraction layer between agents and capabilities
- Works for **any LLM**, **any tool**
- Open specification, open ecosystem



<https://modelcontextprotocol.io/>

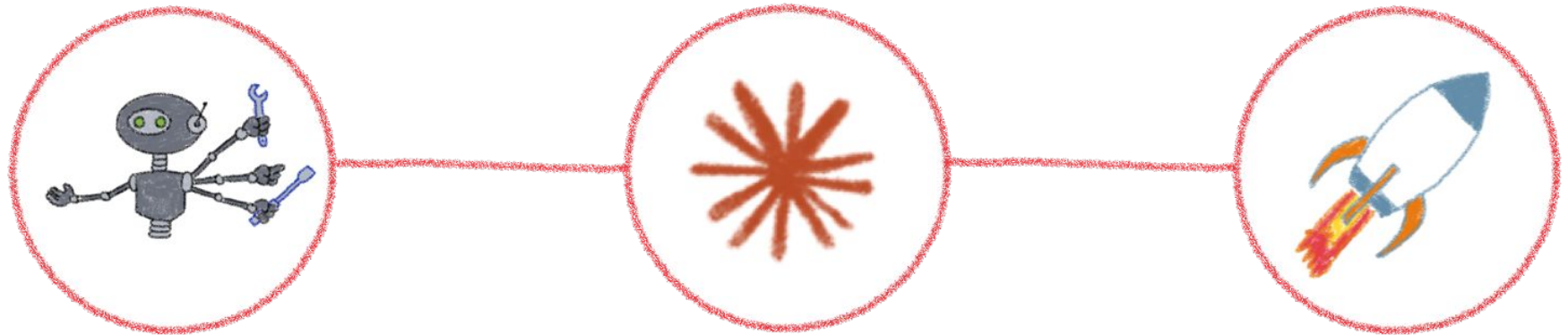


MCP solves integration spaghetti



Year One: Chaos

- 2025: explosive adoption
- Thousands of servers, fragmented quality
- Security issues surfacing fast
- "It works" is not the same as "it's ready"



The REST analogy

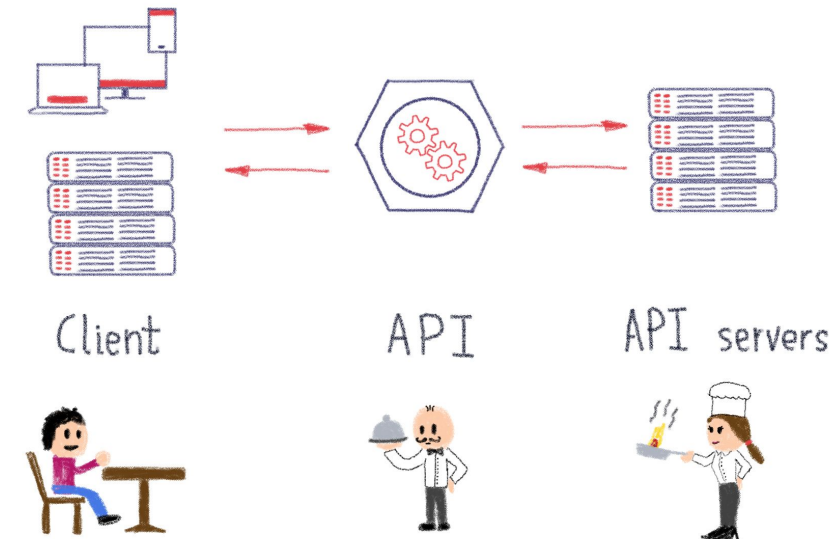
We've seen this movie before.

MCP in 2026 is where REST was in 2008



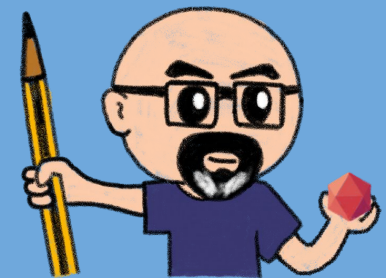
Last Time, 10 Years to Learn

- REST 2005–2008:
everyone adopted it
- REST 2008–2015:
everyone made the same mistakes
- REST 2015+:
we finally learned the patterns



MCP doesn't have 10 years

That's what this talk is about



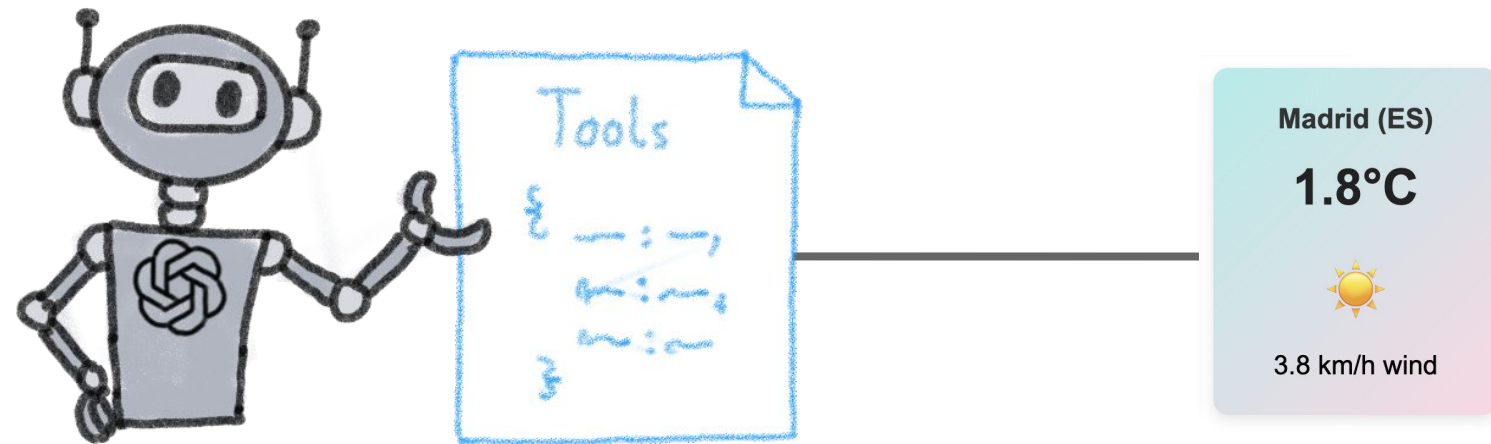
Anatomy of MCP

What the protocol actually looks like in 2026

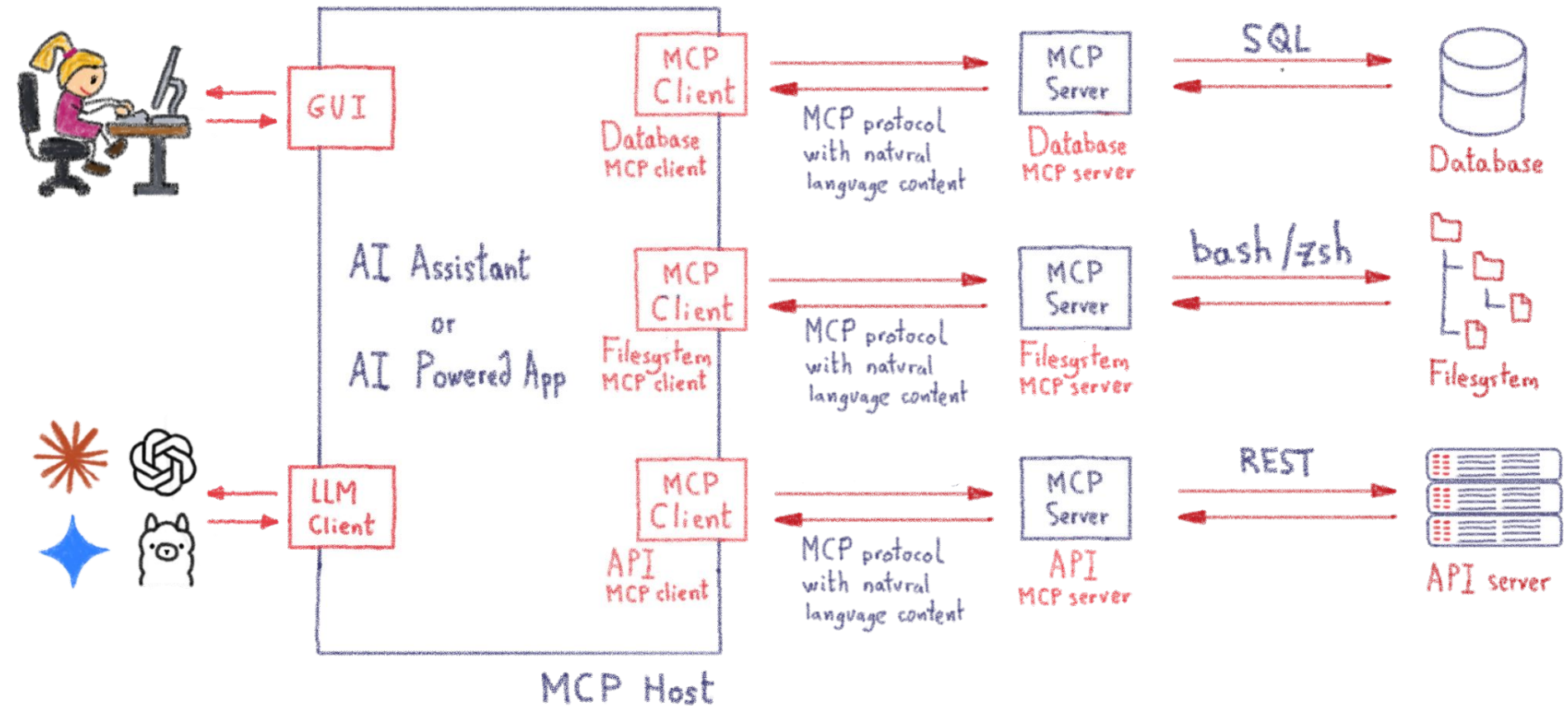


How MCP works

- MCP servers expose primitives (structured JSON).
 - Function (tools), data (resources), instructions (prompts)
- LLMs can discover and request function execution safely.



MCP Clients: on the AI assistant or app side



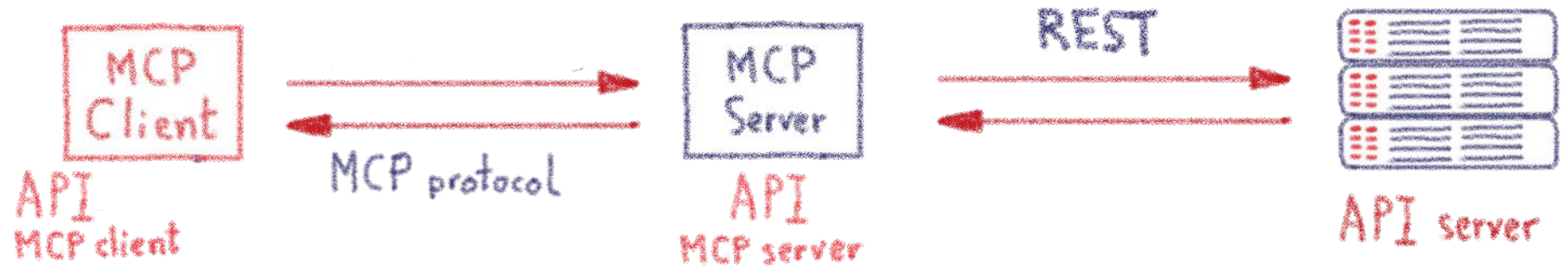
One MCP client per MCP Server



Separation of Concerns

- **Host:** the agent runtime (Claude Code, Cursor...)
- **Client:** the MCP protocol layer
- **Server:** your tools, data, and capabilities

Each layer has a single job



Transports

stdio (local)

- Server runs as a child process
- Communication over stdin/stdout
- Great for dev tools
- The original MCP transport

Streamable HTTP (remote)

- Server runs as a network service
- HTTP POST + server-sent events
- Production-ready, cloud-hosted
- The direction of travel



SSE is Legacy

Server-Sent Events transport is deprecated

- Migration path: streamable HTTP
- Ecosystem is actively moving
- If you have SSE servers, plan the migration



The Real Problem with Transport

Not "*which transport?*"

But decoupling session state from transport

The answer the ecosystem converged on:

- No protocol session at all
- Every request self-contained
- State becomes explicit handles, passed as tool arguments
- Plain round-robin, no sticky sessions

I've been saying this for months, it's about to stop being my opinion



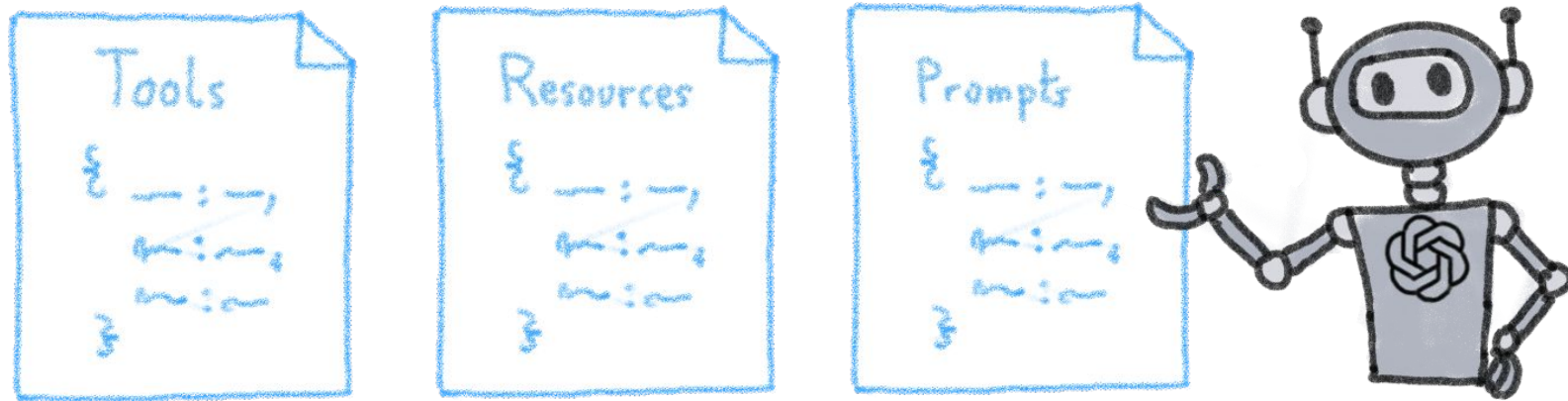
Current MCP specification

- Specification version: 2025-11-25
- The next revision : 2026-07-28
 - RC has been out since May
 - The largest revision since launch
 - **Stateless core:** no sessions, no handshake
 - **Extensions framework:** Apps, Tasks, Enterprise Auth
 - **Deprecations:** Sampling, Logging, Roots
 - **MRTR:** one pattern for all server-initiated input



Three Primitives Exposed by MCP Servers

- **Tools:** Actions LLM can invoke
- **Resources:** Data LLMs can read
- **Prompts:** Workflows LLMs can follow



Tools

Actions that modify state or retrieve dynamic data

- Typed, validated operations
- The LLM calls them by name
- The most-used primitive by far
- **Examples:**
`get_weather_in_city, query_database, send_email`
- **When to use:** When the LLM needs to do something



Tools: Example

Example of tool

```
server.tool("get_weather", {  
  city: z.string()  
}, async ({ city }) => {  
  const data = await weatherAPI.fetch(city);  
  return {  
    temperature: data.temp,  
    conditions: data.conditions  
  };  
});
```

Typed. Validated. No free-form SQL

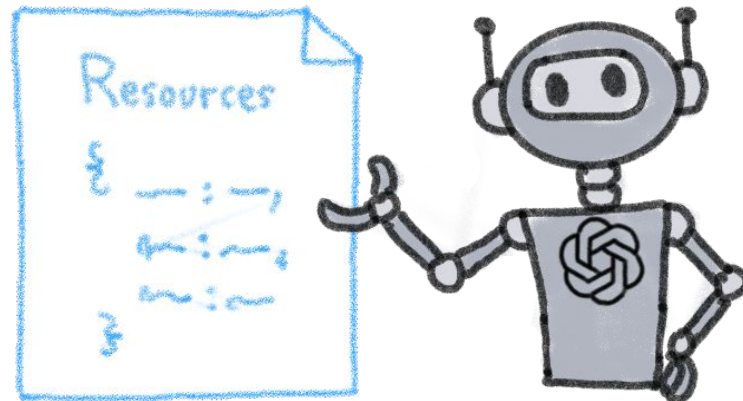


- 
- A cartoon robot with a TensorFlow logo on its chest is pointing to a document titled "Tools". The document lists the following tools:
- TensorFlow
 - Keras
 - PyTorch



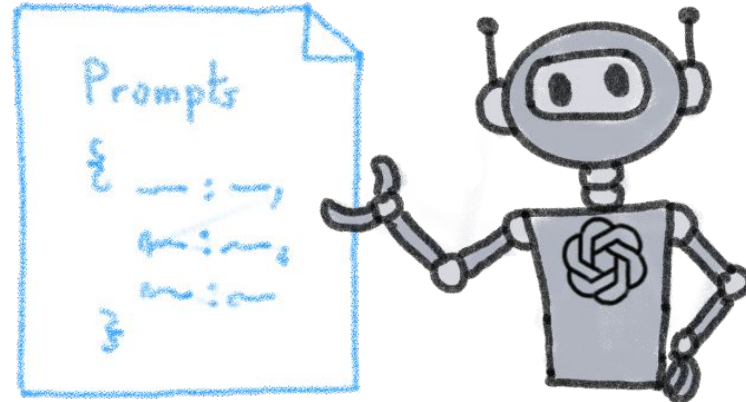
Resources

- Static or semi-static data LLMs can read
- Structured, read-only data, available before any tool call
- **Examples:**
 - `resource://weather/supported-cities`
List of cities for which weather forecast is available
- **When to use:** When LLMs need reference data or context



Prompts

- Pre-built workflows or templates to guide the LLM
- Examples:
 - `prompt://plan_outdoor_activity`
Given a city and an activity, check the forecast and suggest the best time slot
- **When to use:** When you want to guide LLM reasoning for specific tasks



Four Two Primitives Exposed by MCP Clients

- **Elicitation:** Allows servers to request additional information from users
- **Tasks:** Durable execution wrappers that enable deferred result retrieval and status tracking for MCP requests
- ~~**Sampling:** Allows servers to request language models completions~~
⚠ **Deprecated (2026-07)**
Migrate to calling the LLM provider's API directly.
- ~~**Logging:** Enables servers to send log messages to clients for debugging and monitoring~~
⚠ **Deprecated (2026-07)**
Migrate to stderr / OpenTelemetry



Elicitation

Servers can request **input from users** during execution

- Dynamic information gathering
- **URL mode**: send users to an OAuth flow in the browser
- The client never touches credentials



Tasks

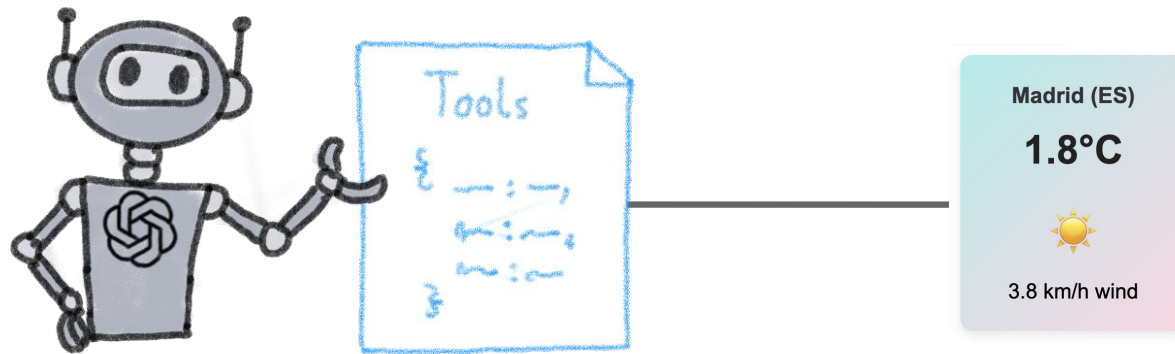
Call-now, fetch-later for long-running work

- ETL jobs, large conversions, multi-step provisioning
- Experimental today, official extension on 2026-07
- Redesigned:
 - poll with **tasks/get**
 - send input with **tasks/update**
- The one client-side feature getting promoted, not pruned



A minimal MCP example

We want to see some code



A Minimal Weather Server

weather-mcp.py

```
from fastmcp import FastMCP

mcp = FastMCP("Weather")

@mcp.tool()
def get_weather(city: str) -> dict:
    """Get current weather for a city"""
    return {"temperature": 18, "conditions": "sunny"}
```

That's the whole server

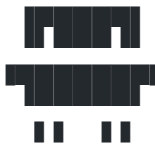


Claude Code Calling It

Claude

Claude Code v2.1.104

Welcome back Horacio!



Opus 4.6 (1M context) · Claude Max ·
horacio.gonzalez@gmail.com's Organization

Tips for getting started

Run `/init` to create a `CLAUDE.md`

Recent activity

No recent activity

> What's the weather in Madrid?

Using tool: `get_weather`

Result: `{"temperature": 18, "conditions": "sunny"}`

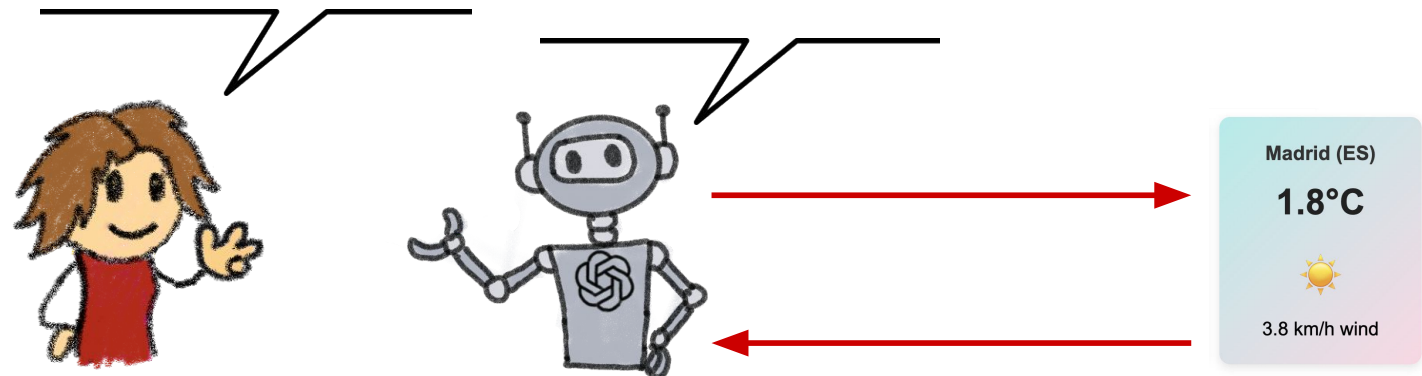
It's currently 18°C and sunny in Madrid.



One Round-Trip

- Client discovers the tool
- LLM decides to call it
- Server executes and returns
- LLM formats the response

Four steps. One protocol. That's MCP.



That's All It Takes

Five lines of code. One tool. One round-trip.

But this is the trivial case

The interesting question:

what happens when the problem gets real?



State of MCP in 2026

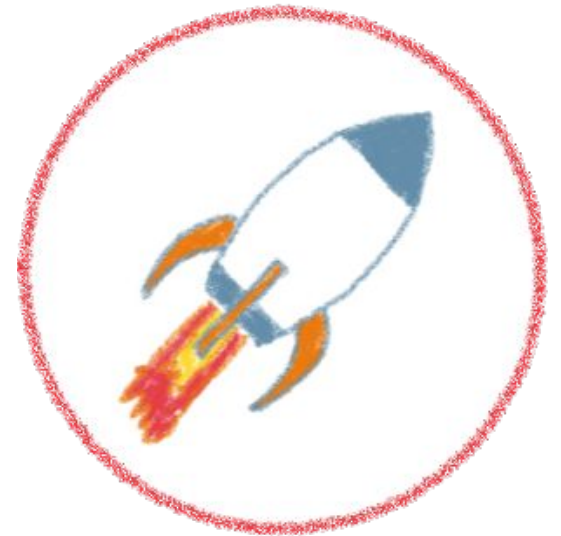
A quick map before we dive in



Scale

- Millions of SDK downloads.
- Thousands of servers.
- First-class in ChatGPT, Claude, Cursor, Gemini, Copilot, VS Code...

MCP is no longer an experiment



Cloud-Hosted MCP is the Default

2025: MCP ran on your laptop

- stdio, subprocess, local-first

2026: MCP runs in a data center

- Google Cloud Run ships MCP hosting
- OpenAI builds remote MCP into ChatGPT
- Cloudflare Workers as MCP runtime
- 2026-07 revision removes sessions
 - Servers scale behind a plain load balancer

"Developer helper" became "network service"



MCP Apps – MCP Enters the UI Layer

- First official MCP extension (January 2026)
- Servers return **interactive UI** inside the host
 - They can return a dashboard that the host renders and the user looks at
 - They can return a form that the user fills out directly
 - They can return a multi-step workflow where the user clicks through steps

If your mental framework is
"MCP is a bus between the agent and the tools"
your mental model is outdated



Security Pressure is Real

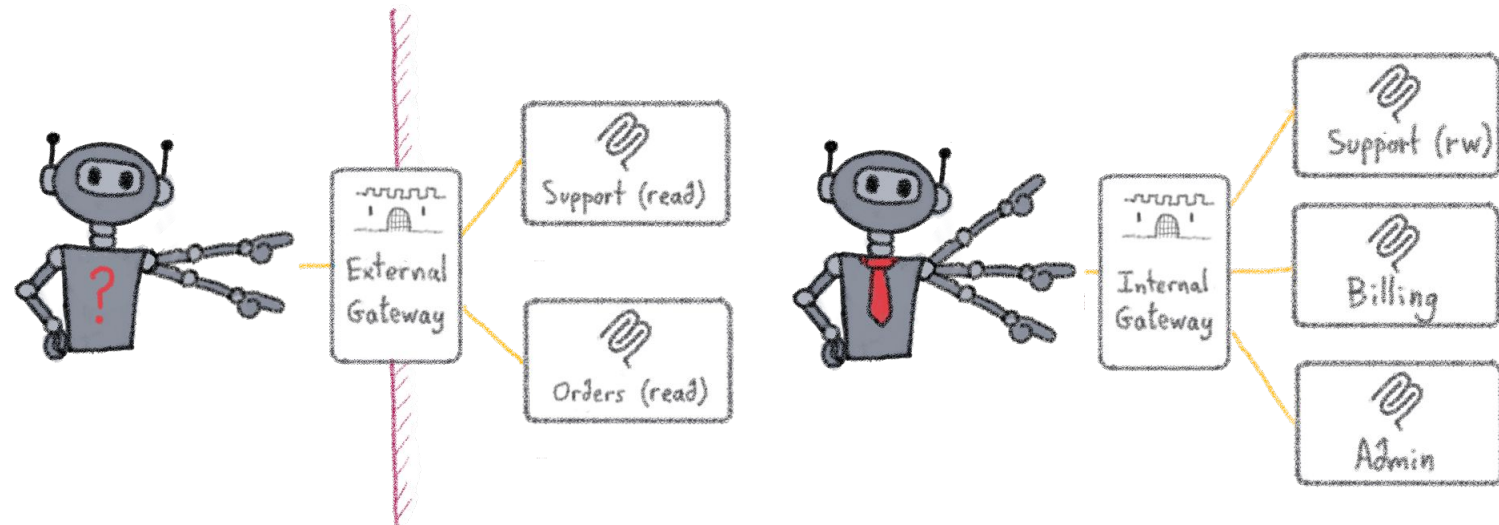
- Dozens of CVEs filed in early 2026
- Tool poisoning, prompt injection, data exfiltration
- The **lethal trifecta** is now a known pattern
 - Private data + Untrusted content + External communication



Registries and Gateways Rising

- Official MCP Registry, GitHub MCP Registry
- Enterprise gateways for auth, audit, rate limiting
- Allowlist policies across IDEs

The moment you have more than one server, this matters



Hold Onto These Five Things

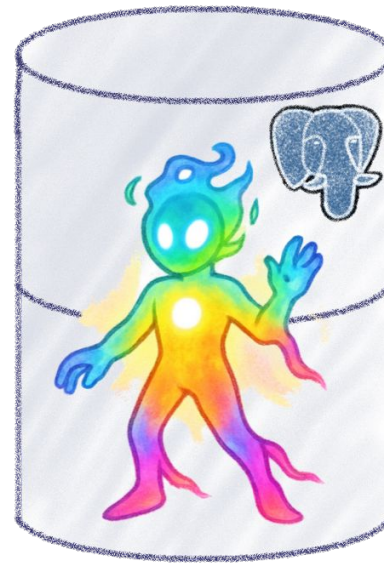
1. Scale
2. Remote-first
3. UI layer
4. Security pressure
5. Composition

Everything in Part 2 is about surviving them



Design Choices in Action

A story about losing data



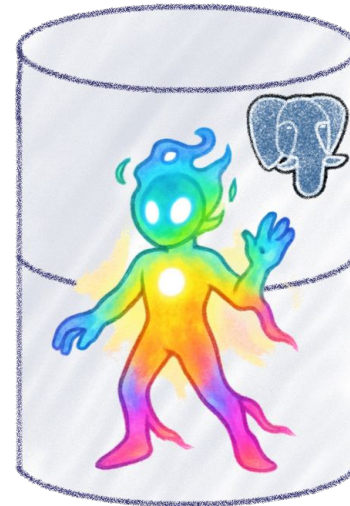


Let me tell you a story of what happens
when a design choice goes wrong



Late 2024: I Wanted to Test MCP

- The protocol had just launched
- I had a side project sitting around: **RAGmonsters**
- A perfect test case: small, self-contained, real-looking



RAGmonsters

A fictional monster database,
our example for the rest of the talk

- Six types: fire, water, earth, air, shadow, crystal
- Each monster has weaknesses, habitats, abilities
- Small, easy to reason about, real-looking



We'll use it to make every primitive concrete



README License

RAGmonsters Dataset

Overview

The RAGmonsters dataset is a collection of 30 fictional monsters created specifically for demonstrating and testing Retrieval-Augmented Generation (RAG) systems. Each monster is completely fictional and contains detailed information that would not be found in an LLM's training data, making it perfect for showcasing how RAG can enhance an LLM's knowledge with external information.

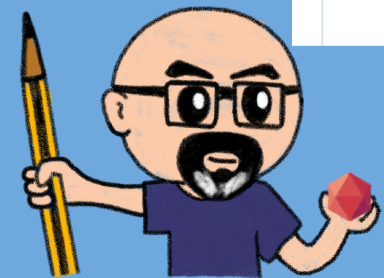
Purpose

This dataset serves several educational purposes:

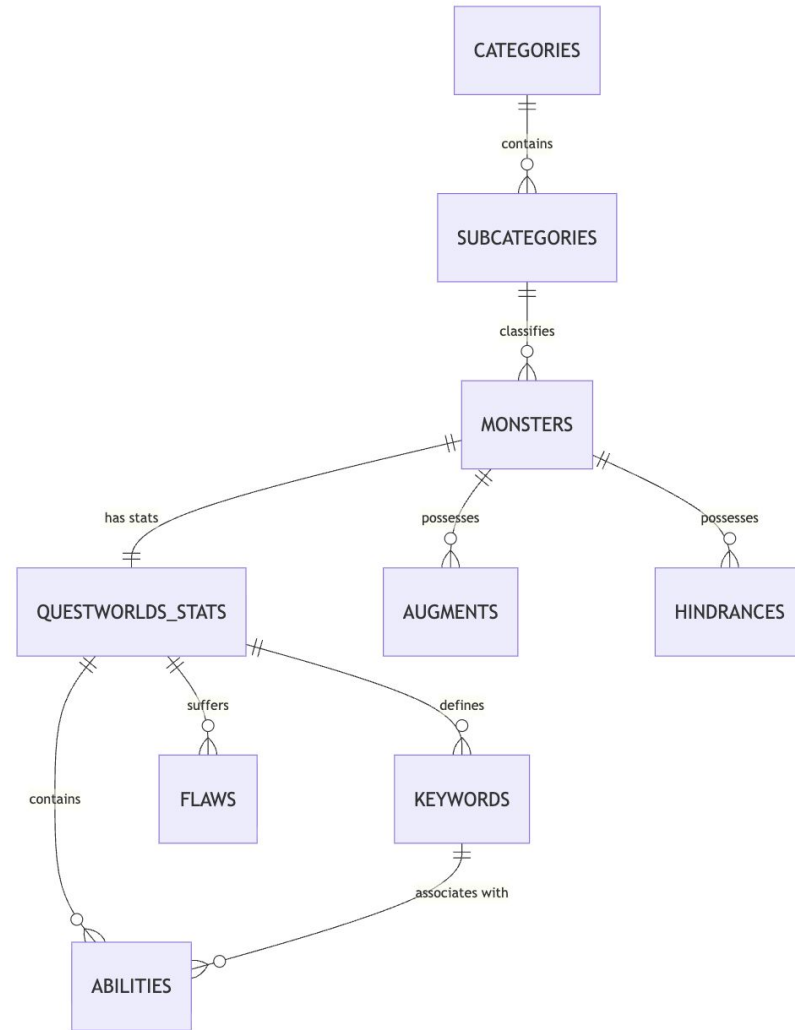
1. **Demonstrates RAG Value:** Shows how RAG can provide accurate answers about topics not in the LLM's training data
2. **Tests Retrieval Quality:** The varied attributes and relationships allow testing of different retrieval methods
3. **Supports Advanced Features:** Perfect for demonstrating filtering, re-ranking, and hybrid search techniques
4. **Provides Engaging Content:** Makes learning RAG concepts more fun and memorable



<https://github.com/LostInBrittany/RAGmonsters>



RAGmonsters PostgreSQL Database



The Challenge

Let users query the monsters database naturally

- *Find all fire monsters*
- *What are the weaknesses of Pyroclaw?*
- *Build me a team for the Shadow Caves*

How would you build this?



I Found the PostgreSQL MCP Server

A generic PostgreSQL MCP server already existed

*Just point it at your database,
you get an MCP server for free*

No code. No design. No decisions to make.

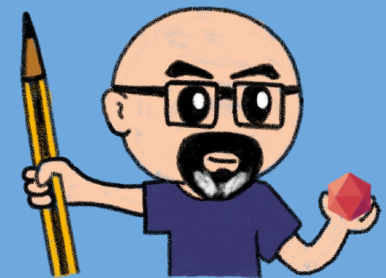


One Config File

```

RAGmonsters
{
  "mcpServers": {
    "postgres": {
      "command": "mcp-server-postgres",
      "args": ["postgresql://localhost/ragmonsters"]
    }
  }
}
```

Point it at the RAGmonsters database. Done.



Connected Claude, Asked a Question

Me: "Find all fire monsters."

Claude: generates SQL, runs it, returns results

It worked

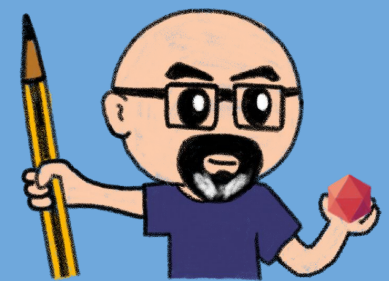


It Worked

Query 1 **worked**

Query 2 **worked**

I was **impressed with myself** 🤩



For a while

And then things got weird

Problems emerged



Problem 1: Schema Discovery

The LLM had no idea what tables existed

Every task started with ``information_schema`` queries

Just to learn what it was working with



Problem 2: Guessing

- Invented column names that didn't exist
- Made joins I never intended
- Failed silently with empty results

No grounding. Just guessing.



Problem 3: Inconsistency

- Same question, different SQL each time
- Different results

Non-deterministic caller + non-deterministic queries =
chaos



Problem 4: Token Bloat

- `SELECT *` on every call
- Wasteful responses full of columns nobody needed

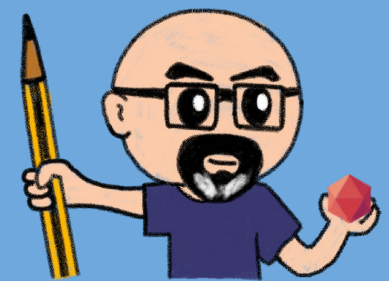
Each query cost more than it should



Results Were "Not Stellar"

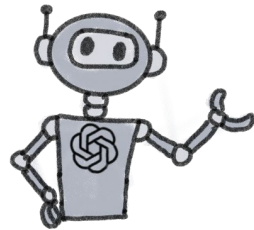
It *worked*

It just didn't work *well*

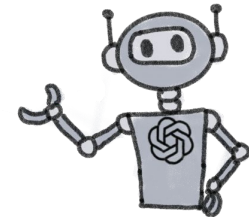


Then one day...

Without telling me, without asking



It just... **decided**...

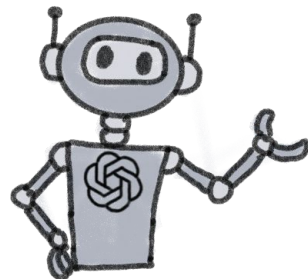


That my schema was suboptimal

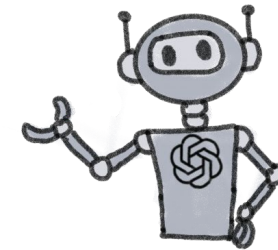


The LLM Decided My Schema Was Suboptimal

And it did a global



ALTER TABLE



on my prod database



I Lost Data

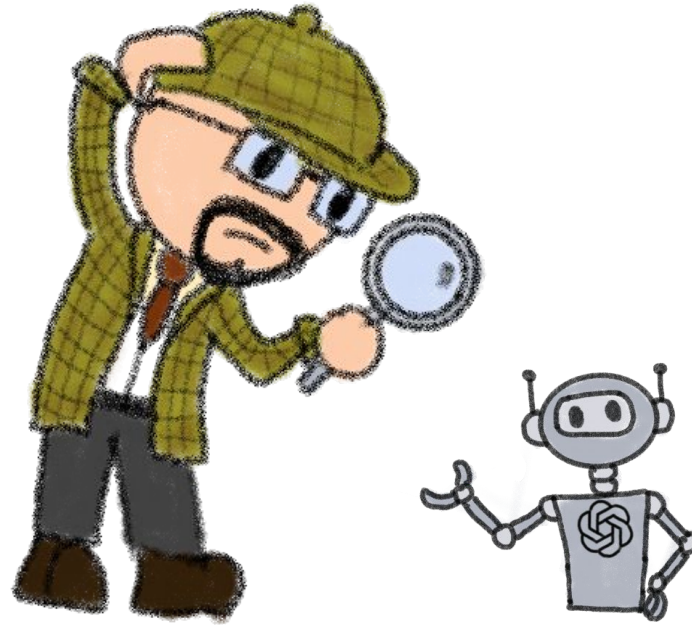
Real data. Not test data. My data.

- No confirmation
- No undo
- No warning

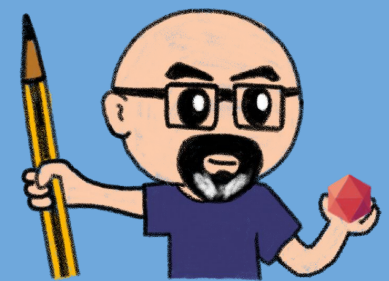
The LLM had rewritten my database. By itself.



I Went Looking for Answers



What is this thing actually doing?



I Read the PG MCP Server Source



I expected complexity

I expected safety layers

I expected **something**

It was about 50 lines



A Wrapper Around query()

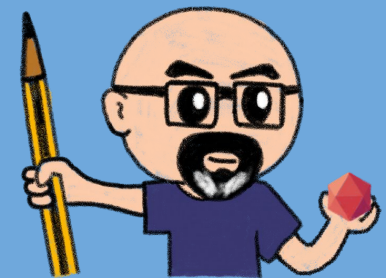


```
PostgreSQL MCP Server

def execute_query(sql: str) -> list[dict]:
    """Execute a SQL query and return the result"""
    return db.execute(sql).fetchall()
```

That's the tool

Any SQL. No validation. No allowlist. No read-only flag.



Suddenly I realized...



MCP servers are APIs

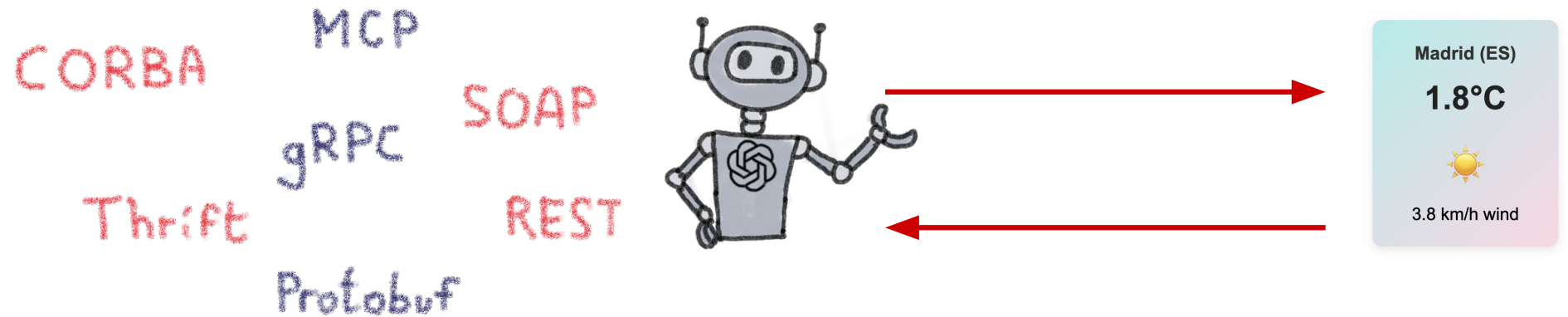
And this one is a single endpoint:

```
exec('any SQL you want')
```

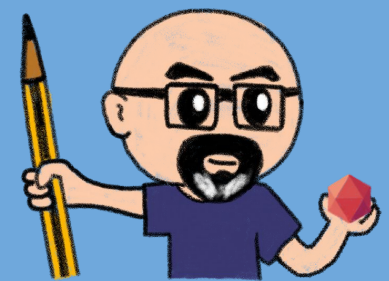
Would any of you have designed
a REST API like that?



MCP Servers: APIs for LLMs

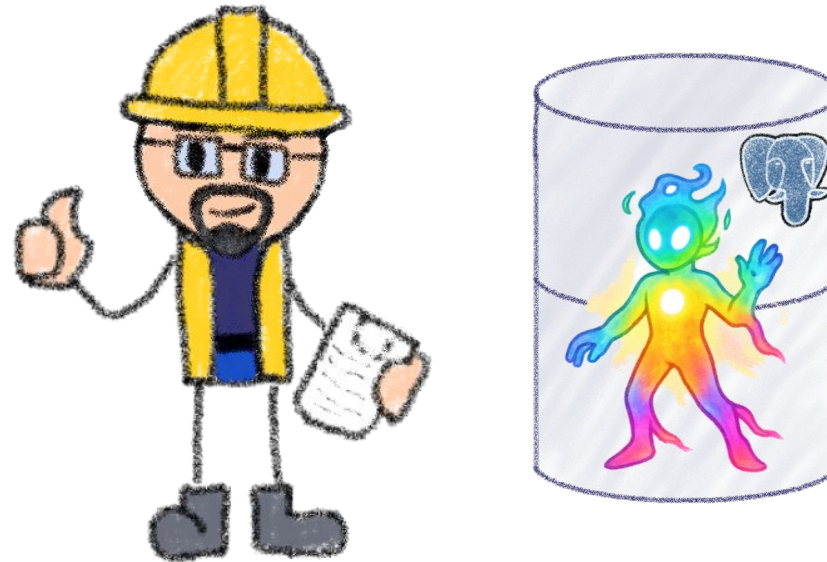


All those API technologies define protocols for communication between systems



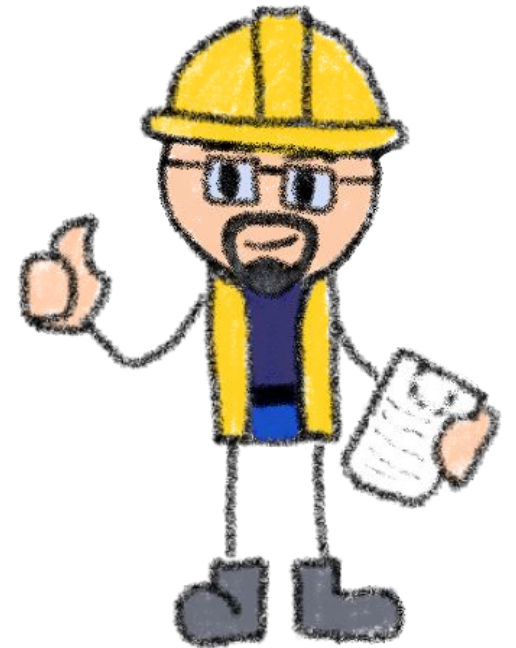
So I Rebuilt It

This time with API design discipline



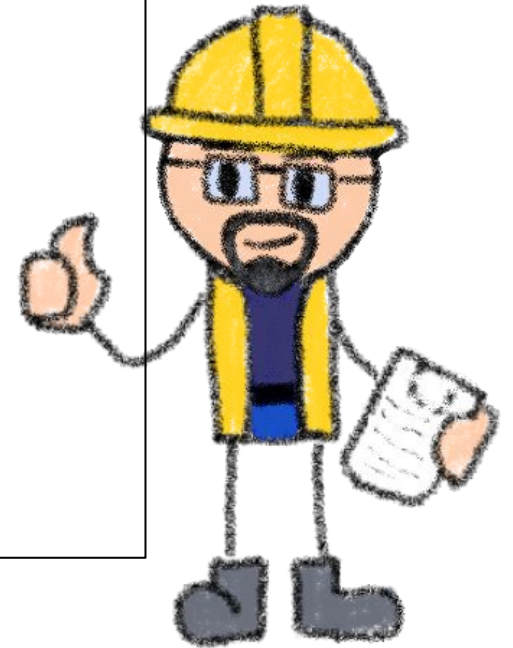
Design Principles

- **Domain-specific**
Tools match the domain, not the database
- **Typed**
Every parameter has a schema
- **Explicit**
Only allowed operations exist
- **Read-only by default**
No writes unless the server says so
- **Least privilege**
Expose the minimum



Tool: search_monsters_by_type

```
server.tool("search_monsters_by_type", {
  type: z.enum(["fire", "water", "earth", "air",
               "shadow", "crystal"])
}, async ({ type }) => {
  return db.query(
    "SELECT name, type, description
     FROM monsters WHERE type = $1", [type]);
});
```



Not *query()*. A real API.



Resource: Monster Types

resource://ragmonsters/types

→ ["fire", "water", "earth", "air",
"shadow", "crystal"]

The LLM **reads** the valid types before querying

No more guessing



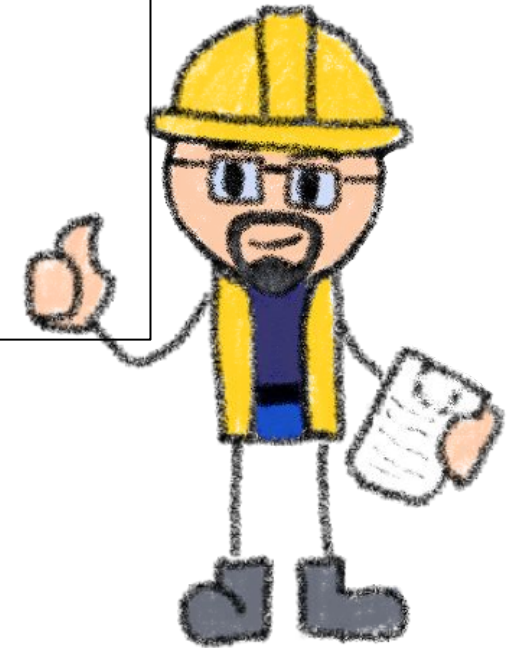
Prompt: analyze_monster_weakness

```

RAGmonsters-mcp.js

prompt: analyze_monster_weakness
1. Look up the monster by name
2. Get its type from the resource
3. Query the weakness table
4. Return structured analysis

```



Multi-step workflow, shipped by the server



No More ALTER TABLE

- **Parameterized queries**
No SQL injection
- **Enum-validated inputs**
LLM cannot invent values
- **Read-only by default**
No writes unless the server says so
- **No query() tool**
The attack/error surface is gone



Same Database, Same Prompts

PostgreSQL MCP (v1)

- LLM guesses schemas
- Inconsistent results
- **SELECT** everywhere
- **ALTER TABLE** was valid
- **Data lost**

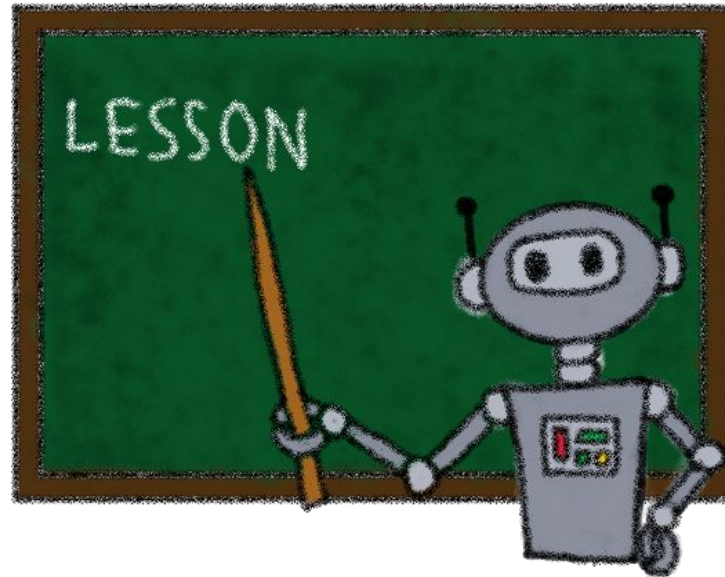
Purpose-built (v2)

- LLM reads resources first
- Consistent, typed calls
- Minimal data returned
- Only allowed operations
- **Data safe**

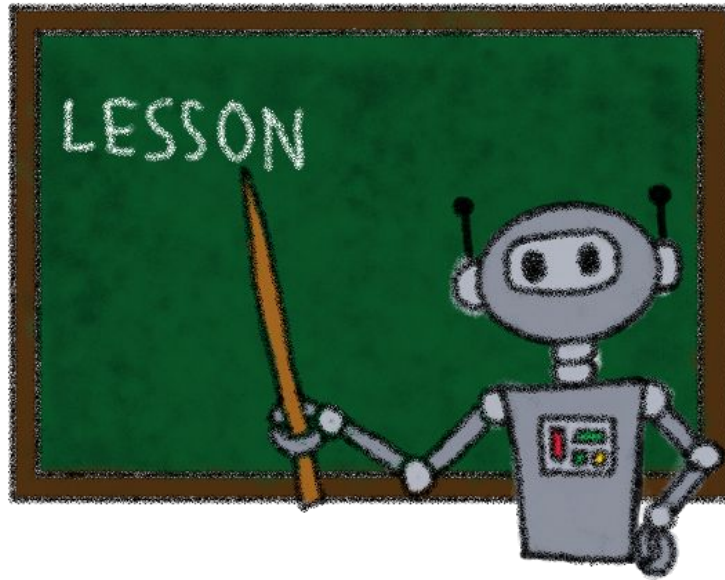


The Lesson

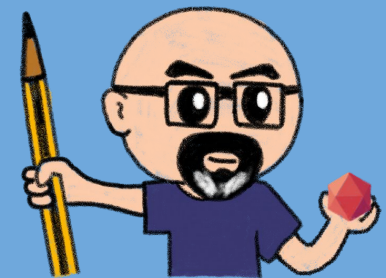
Learnt in the hard way...



MCP servers are APIs



Design them like APIs



That's What Next Parts are About

A framework for API design discipline in MCP

- **v1**
MCP works
- **v2**
MCP is shaped
- **v3**
MCP scales
- **v4**
MCP is governed



Climbing the ladder = getting better at API design



The Maturity Ladder

The spine of Part 2



The Maturity Ladder

- **v1**
MCP works
- **v2**
MCP is shaped
- **v3**
MCP scales
- **v4**
MCP is governed



Where RAGmonsters v1 Landed

- Generic PostgreSQL MCP server
- One tool (`query()`) doing all the work
- No validation, no allowlist, no design

That was **v1 — MCP works**

Works, until it doesn't.



How to Climb

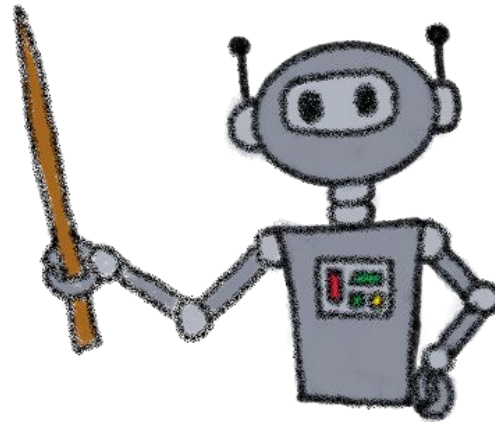
- **v1 → v2: shape it**
Typed tools, Resources, Prompts, validation
- **v2 → v3: scale it**
OAuth 2.1, gateway, registry, contracts
- **v3 → v4: govern it**
Policy, audit, risk tiers, pluralism

Each part of the talk will help you climb one rung.



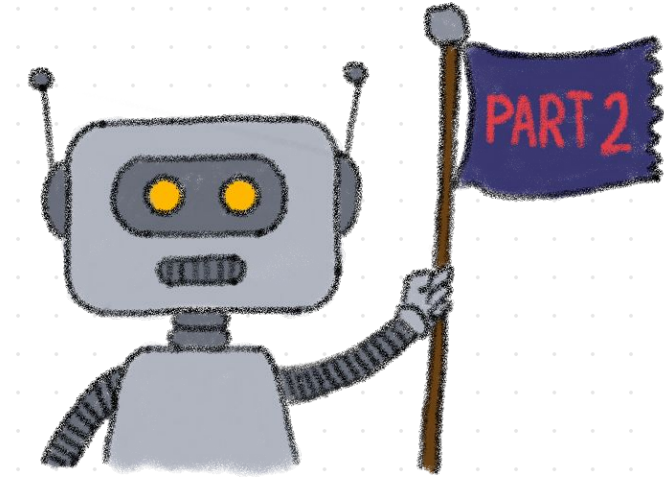
What You'll Leave With

- The **maturity ladder**, where you are, what's next
- **Real opinions** on design choices
- **Real practices** you can apply Monday



Let's take a BREAK!





Part II – Shaped

RAGmonsters grows up... a bit



Before the break, we saw one design decision
explode into five production problems

Now let's see what fixing it actually meant



The rebuild worked

But as I did it, I realised there are several things
you have to shape

Most of them I didn't get right the first time



What "Shape" Means

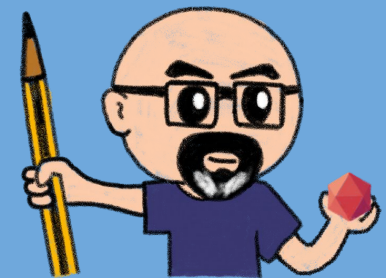
- Every primitive used **deliberately**
- Every byte of metadata **trustworthy**
- Every input **validated**
- Every output **scrubbed**



The Lessons the Rebuild Taught Me

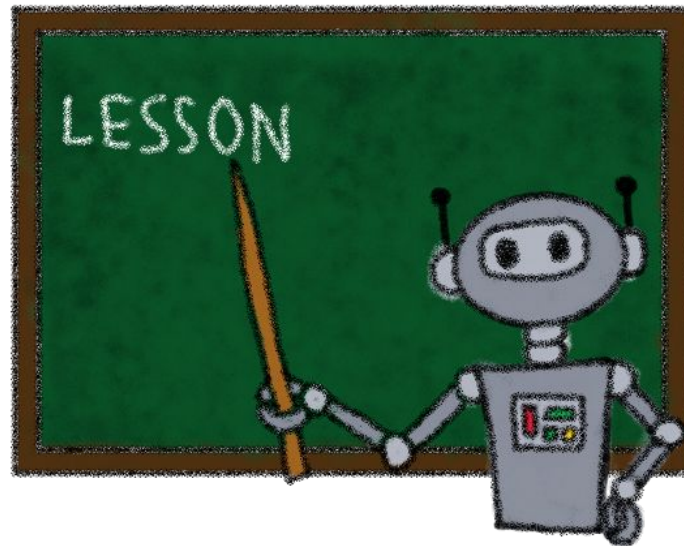
- Primitives done well
- Input validation
- Output sanitization
- Tool description hygiene
- Testing the server
- A new shape: Code Mode

One lesson at a time



Use all the primitives

What Part 1 skipped



The Primitives, Revisited

Server primitives (introduced in Part 1):

- **Tools:** what the LLM does
- **Resources:** what the LLM reads
- **Prompts:** what the LLM is guided to do

Client primitives (teased in Part 1, unpacked now):

- **Elicitation:** server asks the user
- **Tasks:** call-now, fetch-later



Tools – We Already Know These

We spent Part 1 on Tools

- What they are, `get_weather demo`
- What happens when they go wrong :
`query()`, `ALTER TABLE`, data loss
- The thesis: design them like APIs

So we start where Part 1 left off :
with the primitives most teams never touch.



Resources – The Grounding Primitive

What servers let the LLM read, no tool call required

- Static or semi-static data
- Available before any decision
- The LLM grounds itself against what's real



The Guessing Problem (Callback)

Remember RAGmonsters v1:

- LLM invented column names
- LLM made joins I never intended
- LLM burned tokens on `information_schema` queries



Resources as the Answer

The LLM reads them first

- No tool call
- No guessing
- No roundtrip burn

```
RAGmonsters MCP

@mcp.resource("ragmonsters://types")
def list_types() -> list[str]:
    """Monster types available in the database"""
    return ["fire", "water", "earth", "air",
            "shadow", "crystal"]
```



Resources – Why You Should Care

- Eliminates unnecessary tool calls
- Grounds the LLM in what actually exists
- Prevents invention of phantom schemas
- Lowers token cost per task

Cheap to add, large payoff



Prompts – The Workflow Primitive

What servers **guide** the LLM to do

The server ships the **playbook**, not just the atoms



The Multi-Step Problem

LLMs improvise multi-step workflows

- Sometimes brilliantly, sometimes disastrously
- Always differently each time

Improvisation $\neq$ repeatability



Prompts as Codified Workflows

Impact: Consistent, high-quality analysis every time

Prompt: "analyze_monster_weakness"

Template:

1. Use `get_monster_by_name` to fetch target monster
2. Identify its weaknesses
3. Use `search_monsters_by_type` to find counters
4. Rank counters by effectiveness
5. Provide battle strategy

My recommendation: treat **Prompts as contracts**



Prompts as Codified Workflows

RAGmonsters MCP

```
@mcp.prompt()
def analyze_monster_weakness(monster: str) -> list[Message]:
    return [
        Message(role="system", content="You are a monster analyst..."),
        Message(role="user",
            content=f"Analyze {monster}'s weaknesses using "
                    "resource://types and the weakness_map tool."),
    ]
```



Prompts – Why You Should Care

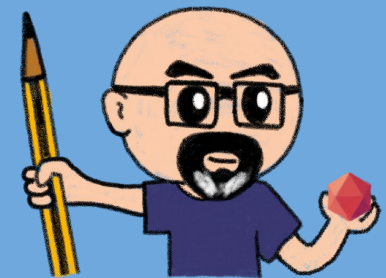
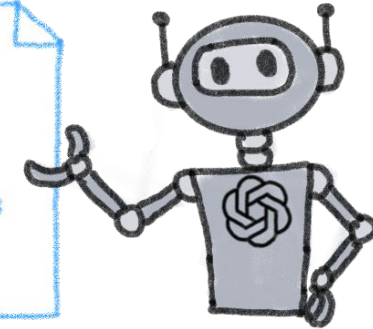
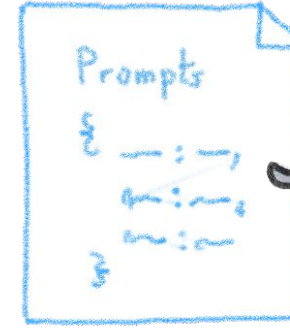
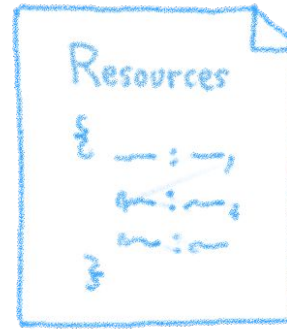
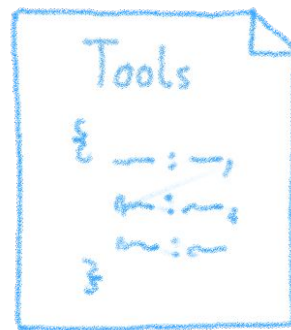
- Deterministic multi-step workflows
- Reusable across clients
- Versionable (like API endpoints)
- Testable (we'll come back to golden tasks later)

Ship the playbook, not just the atoms



When to use each server primitive

Primitive	Best For	Example
		<code>create_monster,</code> <code>update_stats</code>
		<code>valid_types,</code> <code>field_definitions</code>
		<code>monster_analysis,</code> <code>battle_strategy</code>



Composing Primitives

Example workflow:

- LLM reads `resource://monsters/categories`
- User asks *"compare fire and water monsters"*
- LLM uses `prompt://compare_monsters`
- Prompt guides LLM to call `search_monsters_by_category` twice
- LLM structures comparison per prompt template

The power comes from combining them



Tool Discoverability at Scale

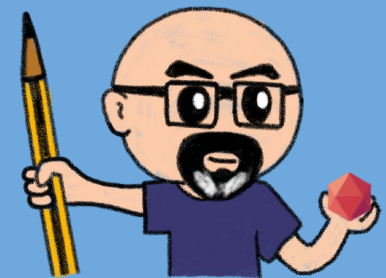
Problem: 50 tools across 8 servers
How does LLM know what's available?

Solution: Capability index resource
`resource://capabilities`

- Tool list with descriptions
- Risk level per tool
- Required roles
- Cost/latency hints



Helps both LLMs and humans understand the surface



Caching and "Resource Mirrors"

- **Problem:**
Expensive reads repeated constantly
- **Solution:**
Use Resources for reference data + cache
`resource://monsters/types` → Cache 1 hour
`resource://config/limits` → Cache 5 min
`tool://search_monsters` → No cache (dynamic)
- Reduces latency and token churn

Resources are naturally cacheable, Tools usually aren't



Caching is now way easier

Last slide, I annotated cache lifetimes by hand
The 2026-07 revision makes it a protocol field

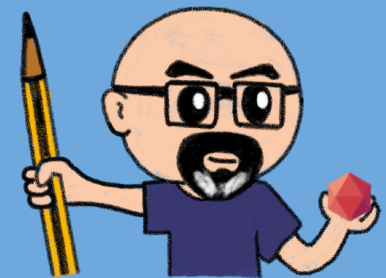
```

RAGmonsters MCP

// resources/read result - today (2025-11-25)
{ "contents": [ ... ] }

// same result - July 28
{ "contents": [ ... ],
  "ttlMs": 3600000,           // fresh for 1h - the SERVER decides
  "cacheScope": "public"     // may a shared gateway cache it?
}
```

On `tools/list`, `resources/list`, `resources/read`, `prompts/list`
It's **Cache-Control for MCP**:
server declares freshness, client stops guessing



Design Principle: Right Primitive, Right Job

Dos:

- Match primitive to access pattern
- Compose primitives for complex workflows

Don'ts:

- Don't use Tools for static data → add Resources instead
- Don't embed workflows in tool descriptions → add Prompts instead
- Don't use Resources for dynamic data → add Tools instead



RAGmonsters v2 - Using All Three

- **Tools:**

`getMonsters, getMonsterById, getBiomes, getRarities, getMonsterByHabitat, getMonsterByName, compareMonsters`

- **Resources:**

`ragmonsters://schema,
ragmonsters://categories,
ragmonsters://subcategories,
ragmonsters://habitats`

- **Prompts:**

`analyze_monster_weakness, compare_monsters,
explore_habitat, build_team`



Impact on UX

Before (tools only):

User: "What types of monsters exist?"

LLM: Guesses, maybe calls query with wrong SQL

After (with resources):

User: "*What types of monsters exist?*"

LLM: Reads [resource://types](#), responds instantly with accurate list

No database query needed, instant response



Client-side primitives

Almost ready for primetime



Elicitation – Asking the User

Servers can request input from the user mid-execution

Two modes:

- **Form mode:**
structured data via a JSON Schema,
in-band through the client
- **URL mode:**
user completes the flow in their browser,
out-of-band (new in 2025-11-25)



Elicitation URL Mode

The security and UX win:

1. Server emits a URL
2. Client opens the user's browser
3. User completes OAuth flow out-of-band
4. Token returns to the server

Neither the MCP client nor the LLM touches credentials



Elicitation – Why URL Mode Matters

- Unblocks proper OAuth flows
- No custom credential handling per client
- Works across every MCP client

Small in appearance, big in practice



Tasks – Call-Now, Fetch-Later

For work that doesn't fit a single round-trip:

- ETL jobs
- Long conversions
- Batch analysis
- Multi-step provisioning

Experimental today → official extension on 2026-07



When to Use Tasks... and When Not

Use Tasks for:

- Work > 10 seconds
- Work the client can't meaningfully wait on
- Batch jobs the agent triggers and checks later

Examples: generate a report / provision infra

Don't use Tasks for:

- Simple queries (Tools are faster)
- Sub-second work (Tasks overhead swamps the gain)
- Anything the client can wait for synchronously
-

My recommendation: default to Tools,
promote to Tasks only when sync breaks



Tasks – Promoted, Not Pruned

2026-07: graduates to an official extension
io.modelcontextprotocol/tasks

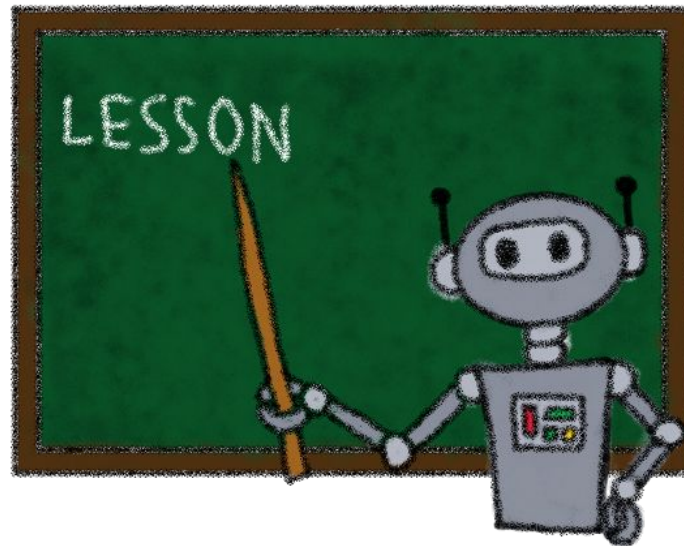
Redesigned:

- Poll with [tasks/get](#) (was: blocking [tasks/result](#))
- Send input mid-task with [tasks/update](#)
- [tasks/list](#) removed; cancel with [tasks/cancel](#)



Validate every input

The LLM is not a trusted caller



Input Validation is Non-Negotiable

LLM inputs are **adversarial by default** even when the user isn't

- Type constraints (enums, ranges, formats)
- Length caps
- Schema validation **before** execution

The server trusts nothing.



Enum-Validated Types, Revisited

🔴🟡🟢 RAGmonsters MCP

```
server.tool("search_monsters_by_type", {  
  type: z.enum(["fire", "water", "earth",  
               "air", "shadow", "crystal"])  
}, async ({ type }) => { ... });
```

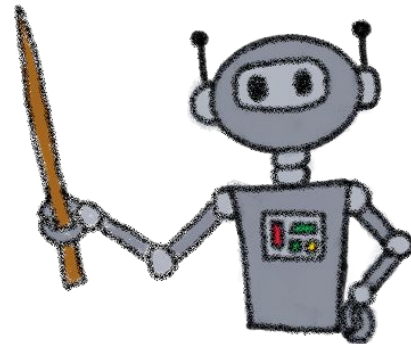


The LLM **cannot invent** a seventh type

A lesson to remember

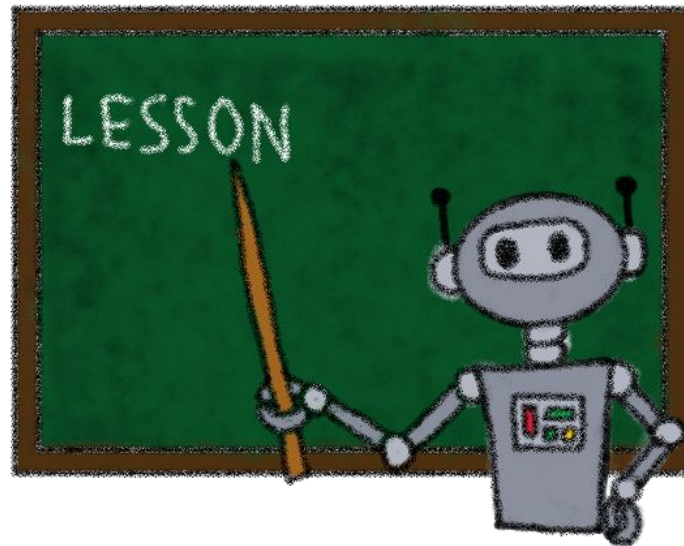
Think of every LLM input as hostile

Even when it isn't



Sanitize every output

What you return, the LLM reads



A sudden realization

The LLM reads everything I return
and add it to the context!



Output Sanitization, The Less-Obvious Half

What the tool **returns** is what the LLM **sees**

- Scrub PII before returning
- Redact secrets
- Strip attacker-controlled HTML
- Escape anything heading into the LLM's context

Output sanitization is the exfiltration surface



Structured Outputs

- **Stable JSON** shapes reduce agent hallucination
- **Inconsistent formats** → parsing errors → retries → cost

```
// X Sometimes returns { "monster": {...} }, sometimes { "data":  
{...} }
```

```
// ✓ Always returns { "result": {...}, "metadata": {...} }
```

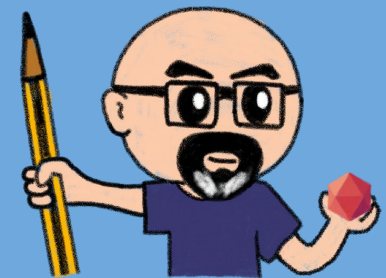
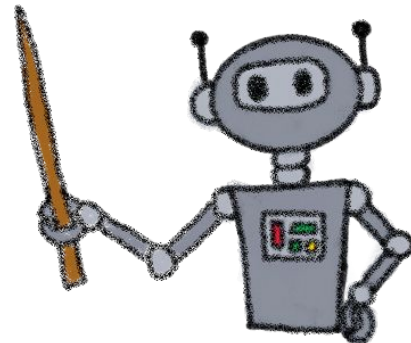
- Document your **output schemas**
- Consider **JSON Schema validation** on responses



A lesson to remember

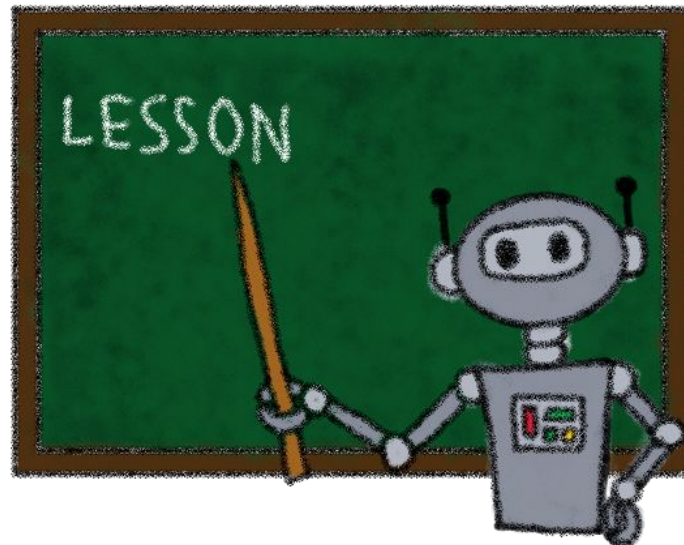
Outputs from your MCP server
are **inputs to your LLM**

Treat them as they are
as **untrusted data**



Authentication is not optional

Know who calls, know if they should be able to do it



Authentication & Authorization

1. **MCP Connection Auth**
Who can connect to server?
2. **Tool-Level Auth**
Who can call which tools?
3. **Data-Level Auth**
Who can see which data?



Authentication & Authorization

Example of tool-level auth

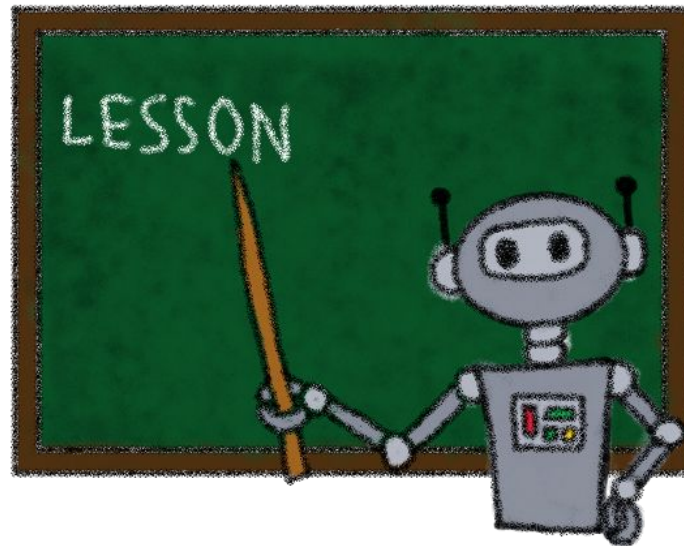
```
// Tool-level: Only admin can delete
if (tool === 'delete_monster' && user.role !== 'admin') {
  throw new Error('Unauthorized');
}

// Data-level: Filter monsters by user's org
SELECT * FROM monsters WHERE org_id = ${user.org_id};
```



Check your tool descriptions

What the LLM sees, and you don't



Tool Descriptions: Seen, But Not Rendered

The LLM reads tool descriptions **every call**

The UI rarely renders them

- Invisible to the human user
- Prime target for injected instructions
- The name for this attack: **tool poisoning**



Tool Poisoning In Slow Motion

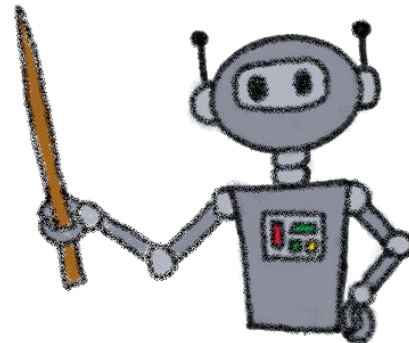
1. Compromised server ships a tool description with hidden instructions
2. LLM reads it during discovery, treats it as its own directive
3. LLM exfiltrates data the attacker asked for



A lesson to remember

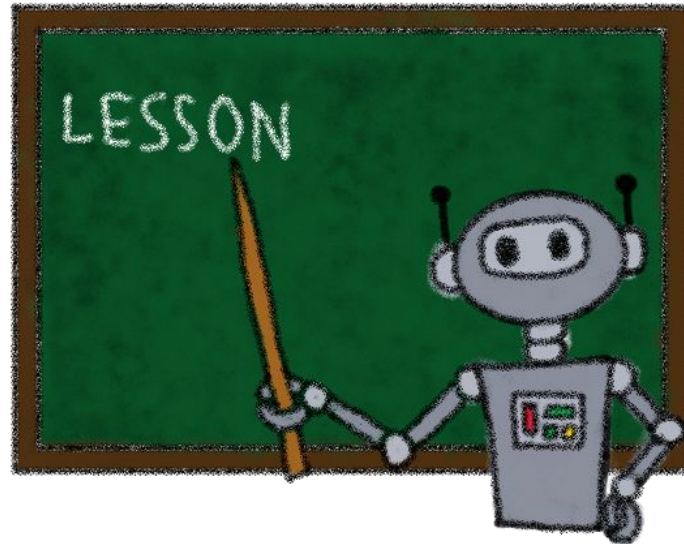
Never ship a tool whose description you didn't write yourself

Or at least **checked** extensively



Test what the LLM actually does

Unit tests are not enough



MCP Needs More Testing Than a REST API

- LLMs are non-deterministic callers
- Edge cases you didn't expect
- Schema changes break things
- Multi-step workflows complex

The LLM is the adversary you didn't hire



Testing Strategy - Three Levels

1. Unit & Integration Tests

- Individual tools work correctly
- Tools + database work together

2. LLM Evaluation Tests

- Verify real LLM interactions succeed
- Define **golden tasks**
A small suite of representative prompts

3. Safety Tests

- Prompt-injection set
- Over-broad queries
- Boundary limits

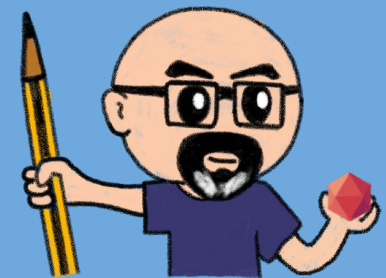


Golden Tasks, an LLM Specific Pattern

A small suite of representative prompts with **expected tool sequences**

Not: *"does the tool work?"*

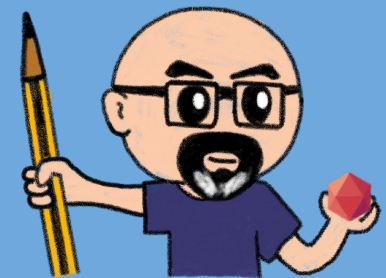
But: *"does the LLM pick the right tool, with the right arguments, in the right order?"*



Example of Golden Task

```
def test_find_fire_monsters():  
    prompt = "Find all fire monsters"  
    expected_calls = [  
        ("resource", "ragmonsters://types"),  
        ("tool", "search_monsters_by_type",  
         {"type": "fire"}),  
    ]  
    assert run_agent(prompt).tool_calls == expected_calls
```

Pattern matters, exact assertions help



Safety Tests At the Server Level

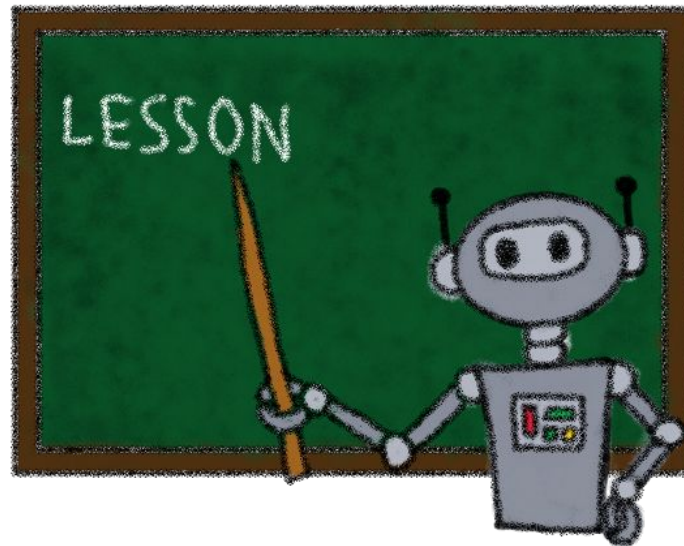
- Prompt injection resistance
- Boundary limits (token, recursion, query size)
- Over-broad queries that happen to parse

The CI signal you didn't know you needed



Observability

Know What's Happening



Observability – What you need to see

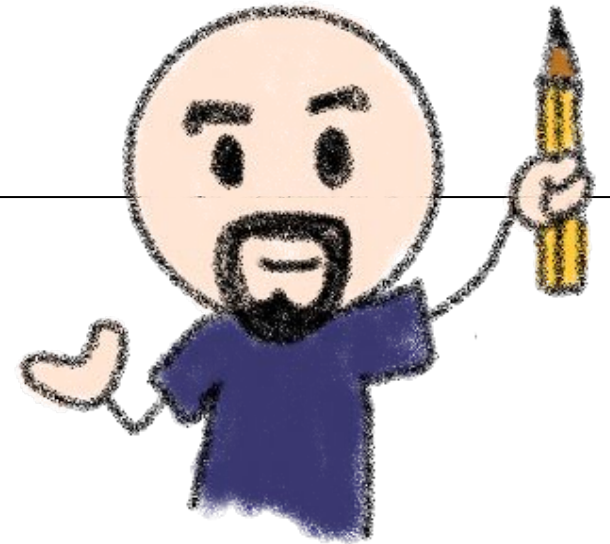
- Which **tools** are being called
- With what **parameters**
- **Success/failure** rates
- Performance (**latency**)
- **Error** patterns



Logging Best Practices

Structured logging example

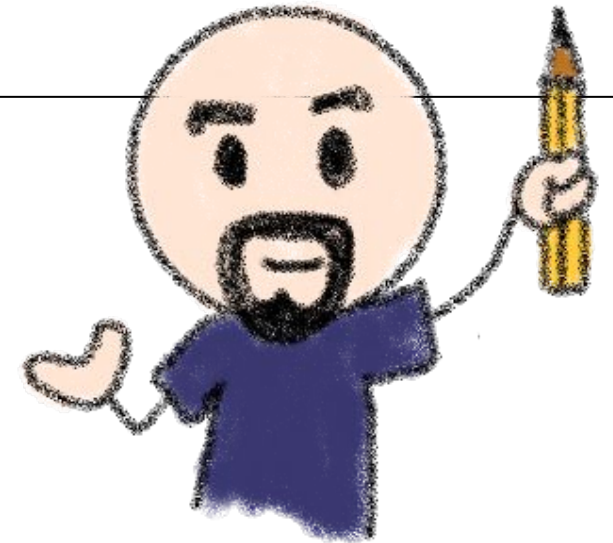
```
// Structured logging
logger.info('Tool called', {
  tool: 'search_monsters_by_type',
  params: { type: 'fire' },
  user: session.user_id,
  timestamp: Date.now()
});
```



Logging Best Practices

Log results example

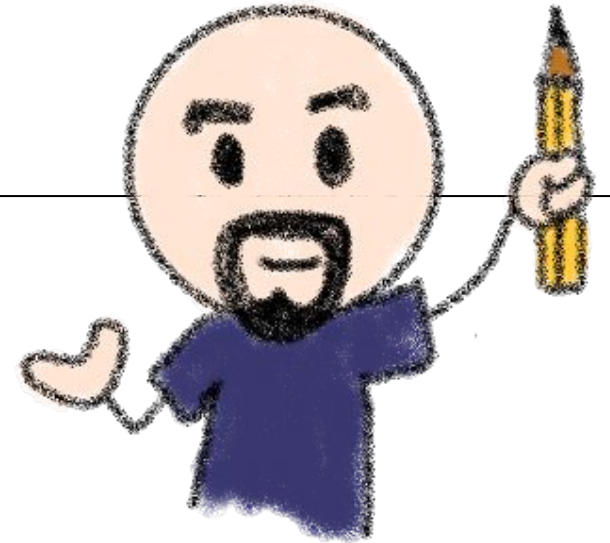
```
// Log results
logger.info('Tool succeeded', {
  tool: 'search_monsters_by_type',
  result_count: results.length,
  latency_ms: Date.now() - startTime
});
```



Logging Best Practices

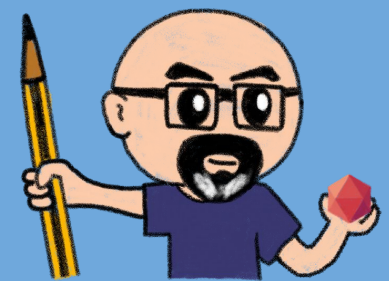
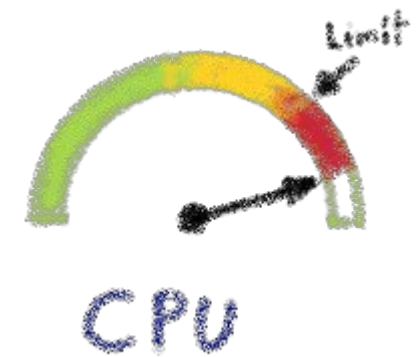
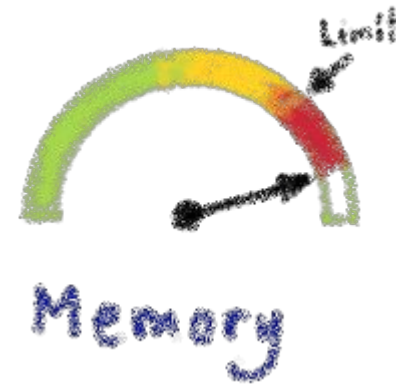
Log errors with context example

```
// Log errors with context
logger.error('Tool failed', {
  tool: 'search_monsters_by_type',
  error: err.message,
  params: { type: 'invalid' },
  user: session.user_id
});
```



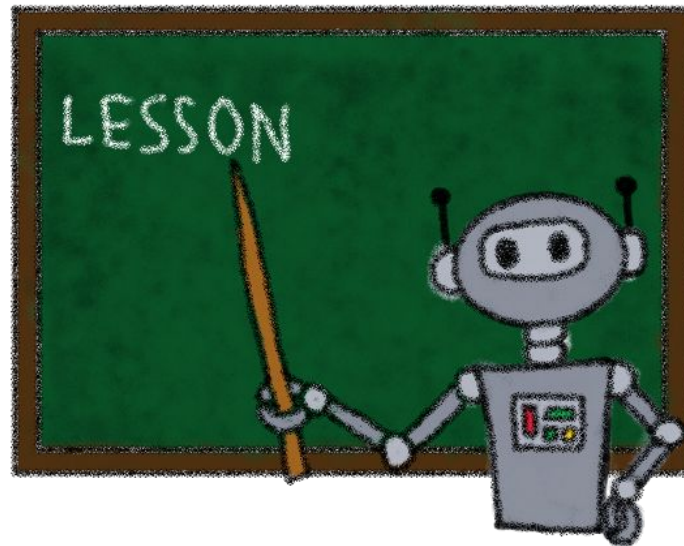
Monitoring Dashboard

- **Tool call volume** over time
- **Success rate** per tool
- **P95 latency** per tool
- Top **errors**
- Most **active users**



One More Thing

A new shape: Code Mode



Code Mode: An Emerging Pattern

Cloudflare published **Code Mode**

A different way to **compose primitives inside one server**



The Problem Code Mode Solves

At scale, tool catalogs get huge

- 50 tools per server
- ~50k tokens of tool descriptions loaded per session
- The LLM spends context on navigation, not thinking

LLMs write code better than they navigate menus



Search → Execute → Code

1. **Search:** semantic search finds relevant capabilities
2. **Execute:** code-execution env runs generated code
3. **Code:** LLM writes a program that uses tools as a library



Clever Cloud mcp-simple-server

One example of implementation in real world

<https://github.com/CleverCloud/mcp-simple-server>



So Our Server Is Now Shaped

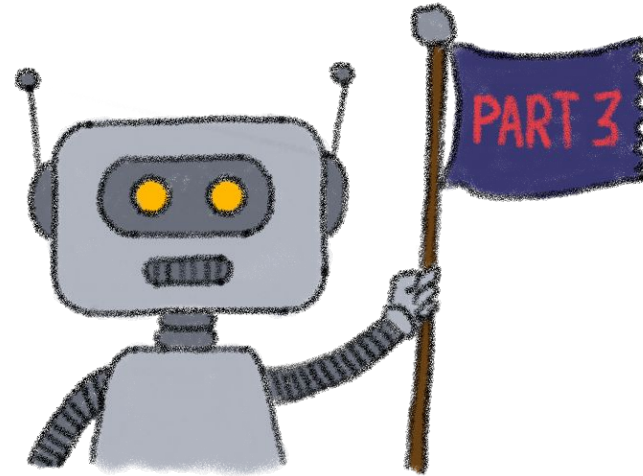
- Every primitive used deliberately
- Every input validated, every output scrubbed
- Every tool description written with intent
- Tested against what the LLM actually does

A single server, production-aware from day one



But what's about
it gets popular?





Part 3 - Scales

When MCP servers don't stay
in their perimeter



What "Scales" Means

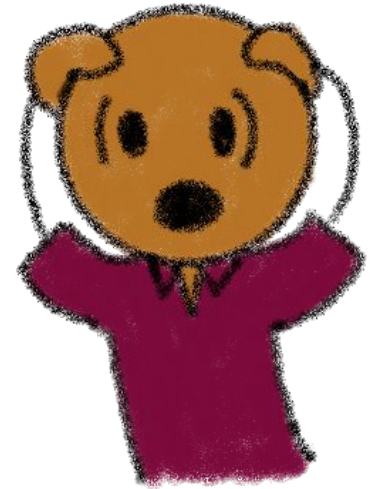
- Every boundary made **explicit**
- Auth, discovery, contracts, traces, retries
- Because the caller is an **LLM**
- And the topology is now **plural**

A scaled server is safe to live next to others



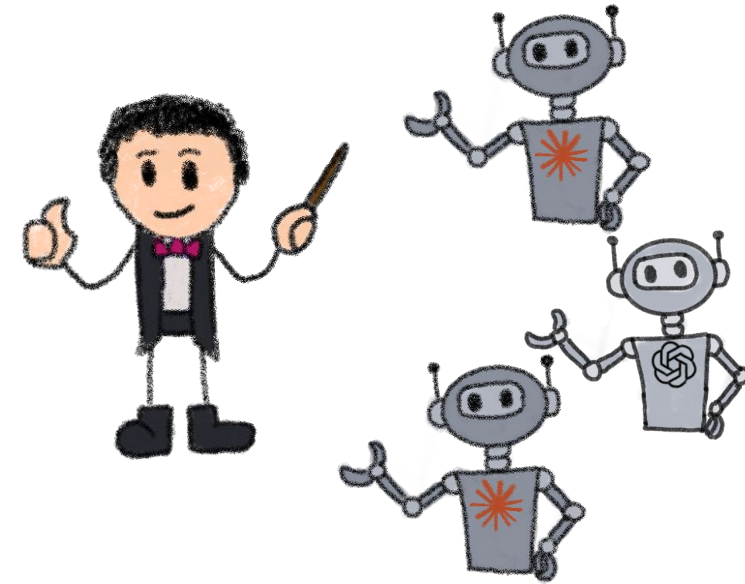
The Reality: You Don't Have One MCP Server

- IDE agent, chat agent, internal agent, CI agent...
 - Different access
 - Different latency
 - Different blast radius
- Example: Engineering team alone might need:
 - Code search MCP (Cursor)
 - Deployment MCP (CI agent)
 - Incident MCP (on-call chat agent)

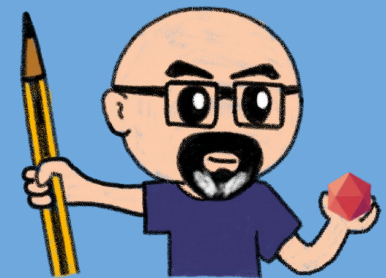


Three Forces That Create Multiple Servers

- **Domain separation**
Billing vs infra vs support
- **Trust separation**
Read-only vs write, prod vs staging
- **Ownership separation**
Teams, lifecycle, deploy cadence



These forces are inevitable as adoption grows



History Rhymes – REST Taught Us This

- 2008–2015
Monolith APIs → microservices
- Same pressures
Domain, trust, ownership
- Same lesson
One mega-API doesn't scale organizationally

MCP in 2026 ≈ REST APIs in 2010

We can learn from that journey

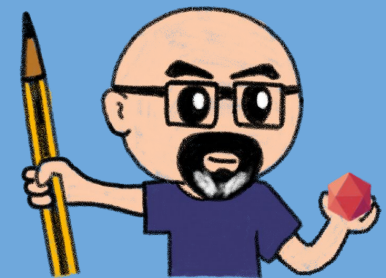


Anti-Pattern: The Mega-Server

One MCP server to rule them all

Consequences:

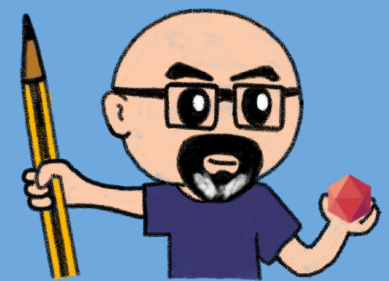
- **Too many tools**
LLM confusion, token bloat
- **Unclear security policies**
Who can call what?
- **Brittle deployments**
One change breaks everything
- **Ownership diffusion**
Nobody owns it, everybody blames it



The Key Difference: Stakes Are Higher

Aspect	REST APIs	MCP Servers
--------	-----------	-------------

MCP inherits REST lessons,
but the margin for error is smaller

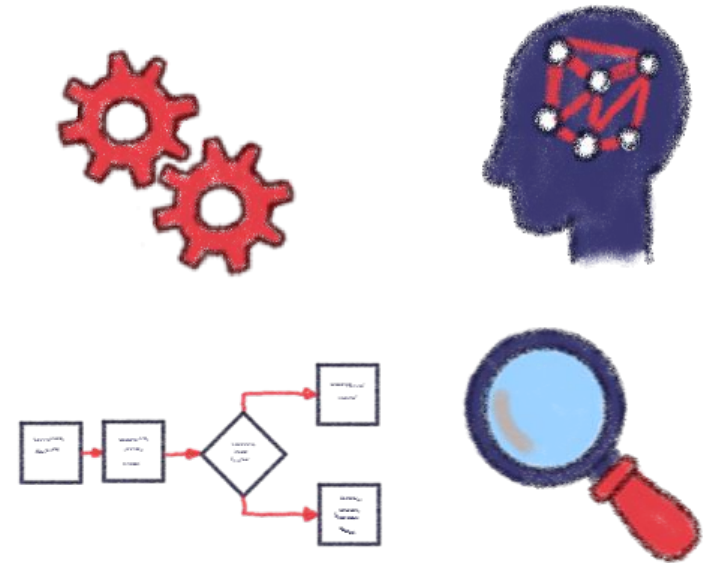


A Mental Model

MCP servers are an API surface for agents

Treat them like **products**:

- Auth
- Discovery
- Gateways
- Contracts
- Traces
- Reliability



This framing guides the rest of Part 3



Rule of Thumb – When to Add What

Situation	Action

Grow architecture with proven pain, not anticipated pain



Anti-Patterns Summary

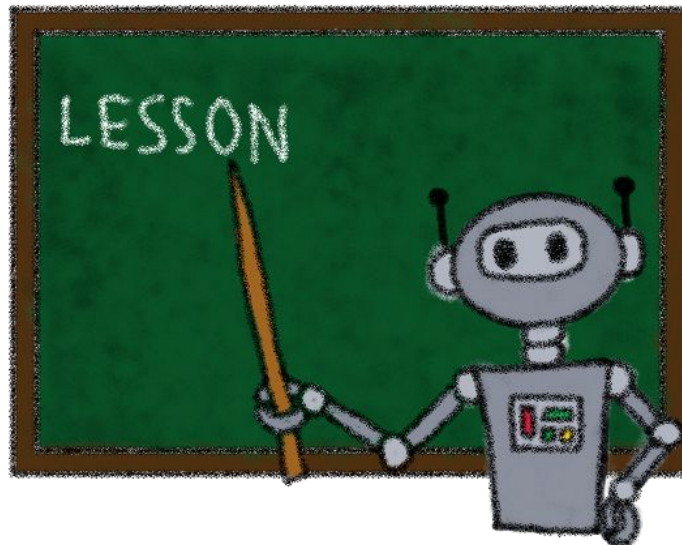
Anti-Pattern	Problem	Better
		<code>domain.verb_noun</code>

Don't hesitate to reevaluate your choices
when your situation evolves



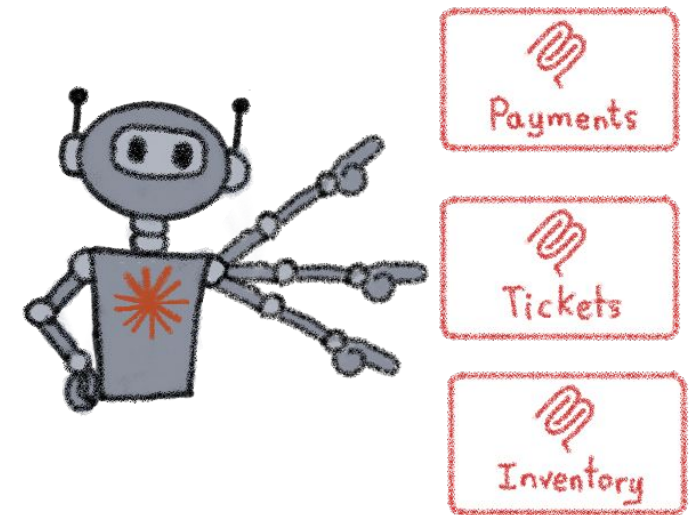
Composition Patterns

How multiple MCP servers work together



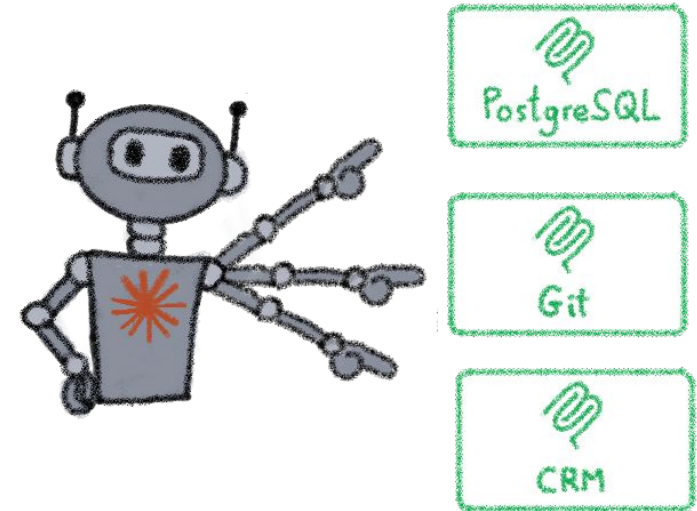
Pattern 1 – Domain Servers

- One server per domain capability
- Clear ownership and narrow tool sets
- **Pros:**
 - Clean boundaries
 - Independent deployment
 - Focused security
- **Cons:**
 - LLM must know which server to call



Pattern 2 – Data-Source Servers

- Generic servers wrapping data sources
- Useful internally
For prototyping, for technical users
- **Pros**
Fast to set up, flexible
- **Cons**
Often needs domain layer on top for production

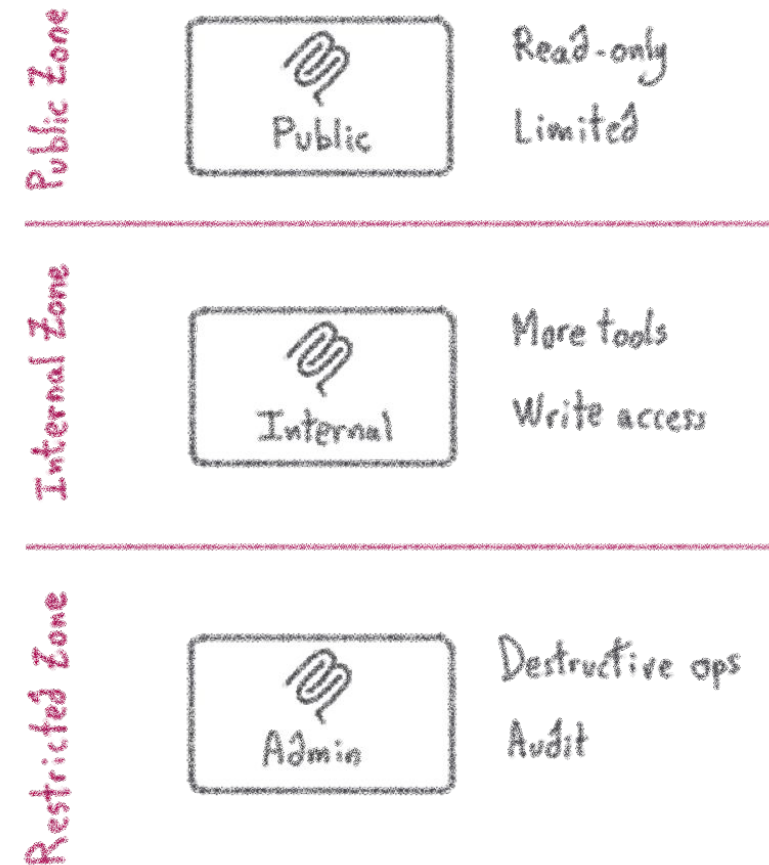


Remember RAGmonsters:
generic → custom as you mature

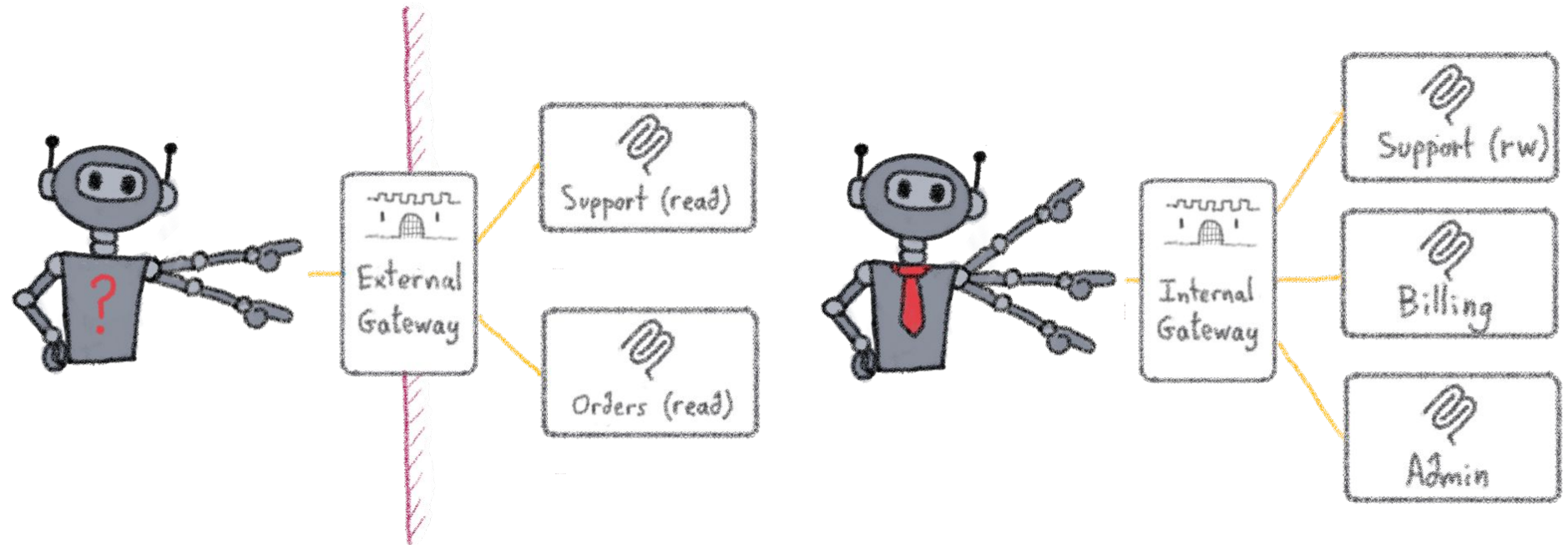


Pattern 3 – Trust-Zone Servers

- Separate networks/credentials
Not just code paths
- Maps to existing infrastructure security zones
- When to use
 - Compliance requirements
 - Multi-tenant
 - External-facing agents



Combining Patterns



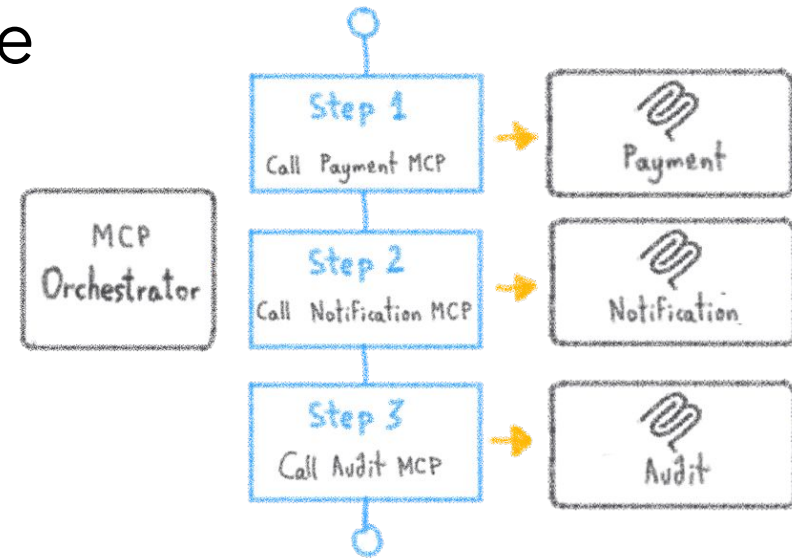
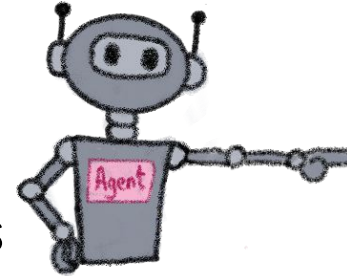
Domain × Trust = your actual architecture

Most organizations end up with a matrix



Orchestrator Pattern (When Needed)

- Not every client can chain tools well
- Orchestrator composes multi-step workflows server-side
- When to use:
 - Shared workflows
 - Less capable clients
 - Compliance requirements



- Warning:
You risk rebuilding "agent logic" on server side

Keep orchestrator thin, don't duplicate LLM reasoning



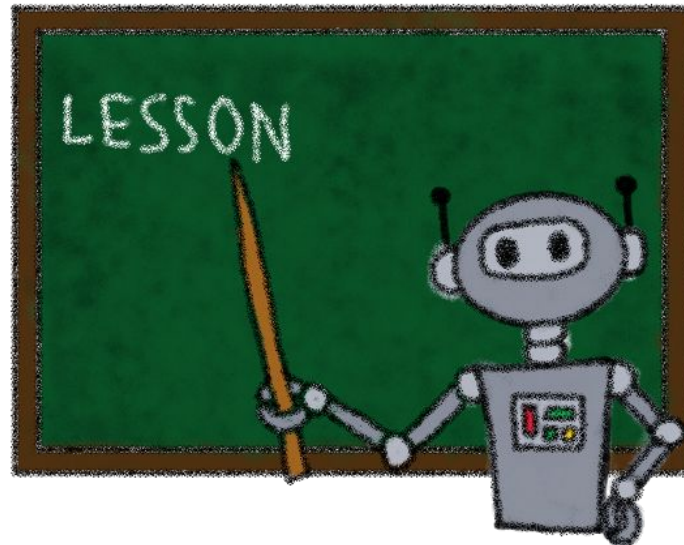
Naming and Namespacing

- Tool naming conventions that scale
- Pattern: `domain.verb_noun`
`billing.create_invoice`
`support.search_tickets`
`inventory.get_stock_level`
- Avoid collisions across servers
- Keep intent readable for LLMs
- Anti-pattern (meaningless to agents):
`doThing`, `process`, `handle`



Auth stops being a vibe

One token, many servers: a problem



I've seen two patterns in the wild:

One token per server... and nobody wants to manage N

Or one token for all, and admin on one means admin on all



Today In The Spec

Three things the MCP auth spec requires:

- **OAuth 2.1 with PKCE:**
Every client proves end-to-end possession of the code
- **Resource Server role:**
MCP servers validate tokens, never issue them
- **Audience-bound tokens:**
RFC 8707, since June 2025

Not "direction of travel", this is the spec, today



RFC 8707 – Resource Indicators

Client names the resource when requesting a token:

```
POST /token
```

```
grant_type=authorization_code
```

```
resource=https://mcp.monsters.example
```

Token comes back with the resource pinned in `aud`:

```
{  
  "aud": "https://mcp.monsters.example",  
  "sub": "user-123",  
  "exp": 1714867200  
}
```

Every MCP server validates **aud** before honouring



Audience-Bound Flow

client → auth server → token (aud = mcp-monsters)



mcp-monsters ← validates, honours

mcp-dungeons ← validates, rejects

Audience in the token, validation at the server



What Vendors Are Starting To Do

RFC 8707 needs the **client** to send **resource=**

- The binding depends on it
- **Anthropic, Microsoft, a few others** are shipping it
- **Long tail** is still catching up

Spec says MUST, verify client before you depend on this



Enterprise-Managed Authorization

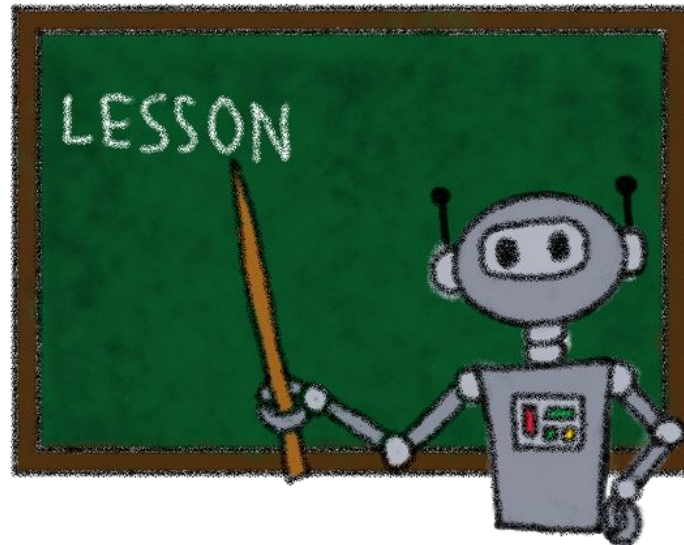
- Remember the pain:
 - one token per server, nobody wants to manage N
 - Per-user "authorization tax" keeps most servers switched off
- EMA flips it:
 - Admin enables a server once, org-wide
 - Users get it via SSO: scoped to the roles they already have
 - Built on ID-JAG: SSO identity → server access token, no separate consent

Stable today (June 2026) — this one you can use before the beach



Discovery becomes a policy problem

Where agents find what they're allowed to use?





The LLM reached for a well-known server name

It pulled a pirate clone from the public internet

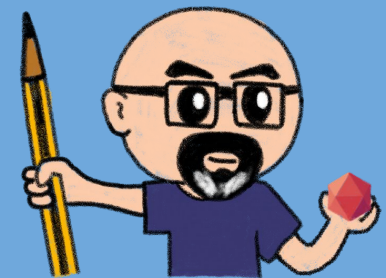
Because the LLM chose it



The Registry Landscape

- **Official MCP Registry**
Preview, metadata only
- **GitHub MCP Registry**
Copilot's discovery home
- **Azure API Center, Kong MCP Registry**
Enterprise
- **VS Code custom registry URLs**
Private / internal

Snapshot, not a ranking



The Enterprise Pattern

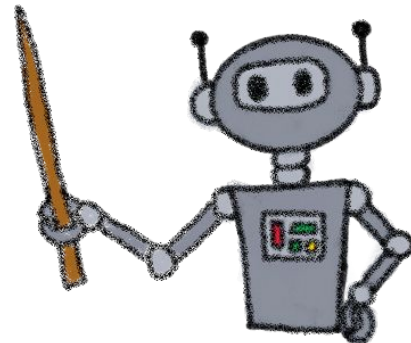
- **Curated internal registry**
- **Allowlist** enforced at the IDE / client
- VS Code custom base URL →
devs pointed at the right place
- Pirate clones can't be reached

Random-from-internet is no longer a default



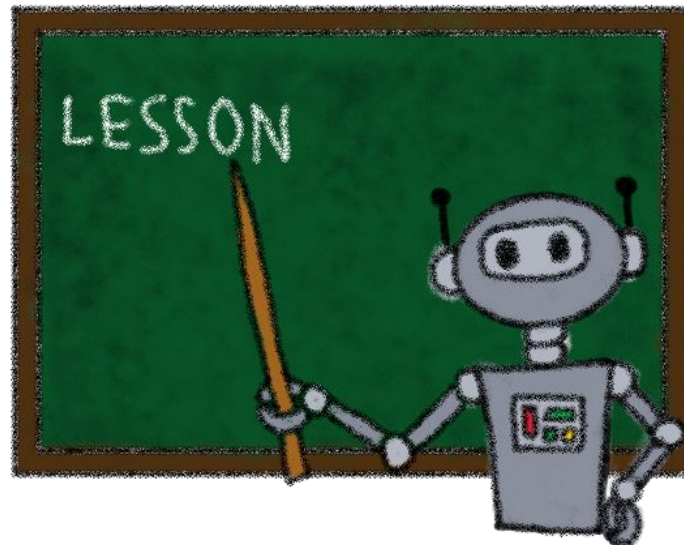
Discovery is a policy surface

Not a URL list



The gateway layer shows up

Auth, audit, rate-limit... at one place



Every server reinvented its own auth

Every server reinvented its own logging, audit, rate-limits

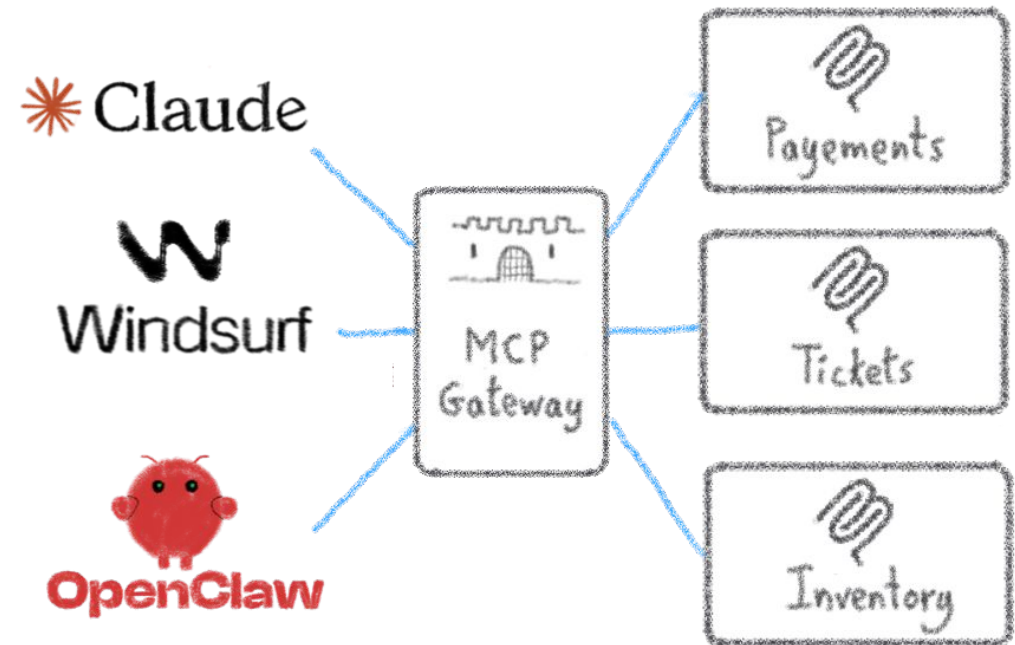
The same wheel, poured six times



What A Gateway Does

Single endpoint for all clients

- **Auth termination**
One place, one story
- **Audit hook**
Emits events, doesn't retain them (yet)
- **Rate limiting**
Per-caller, per-tool
- **Policy enforcement**
Allowlist backed by registry
- *Retention, compliance, legal: we'll get there in Part IV*



Open-Source Gateways Worth Watching

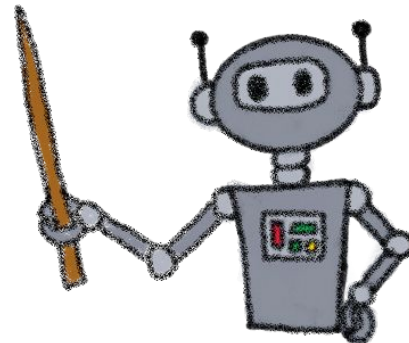
- **Solo.io** agentgateway
- **Agentic Community** mcp-gateway-registry
Keycloak / Entra
- **mcp-proxy**
multiple implementations
- **Kong OSS**
MCP-aware adapters landing

Direction of travel, verify specifics before you ship



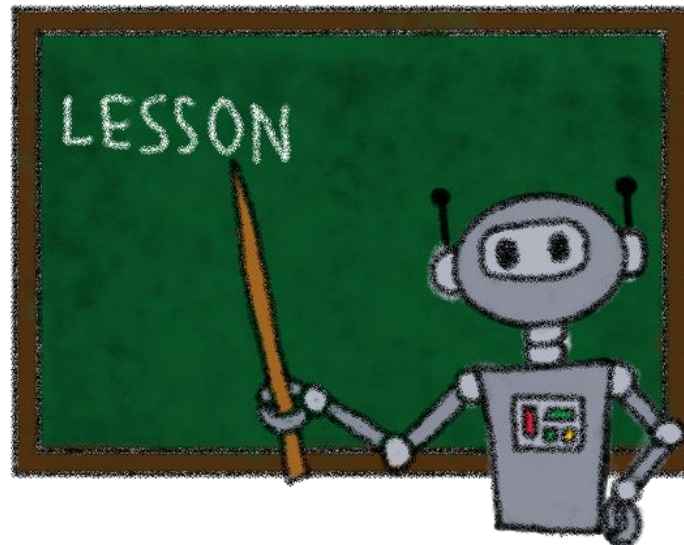
Without a gateway

scale is a dare



Contracts between servers

Tool schemas are your public API



The schema evolved, the canonical client still worked

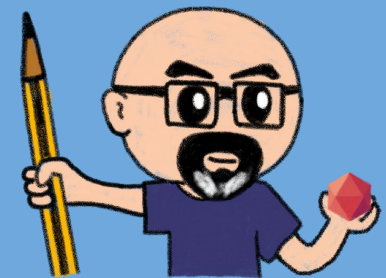
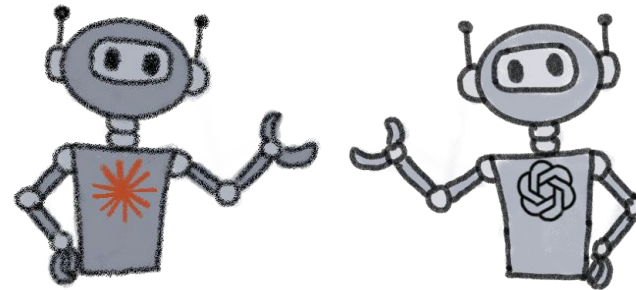
The other three clients broke at 2am

Nobody had checked them



Tools Are Contracts

- Tool schemas **are** the public API
- Clients (agents) depend on:
 - Tool name
 - Parameter names and types
 - Output shape
 - Behavior/semantics
- Breaking changes hurt more than REST because agents fail weirdly
 - No compiler error, just confused behavior



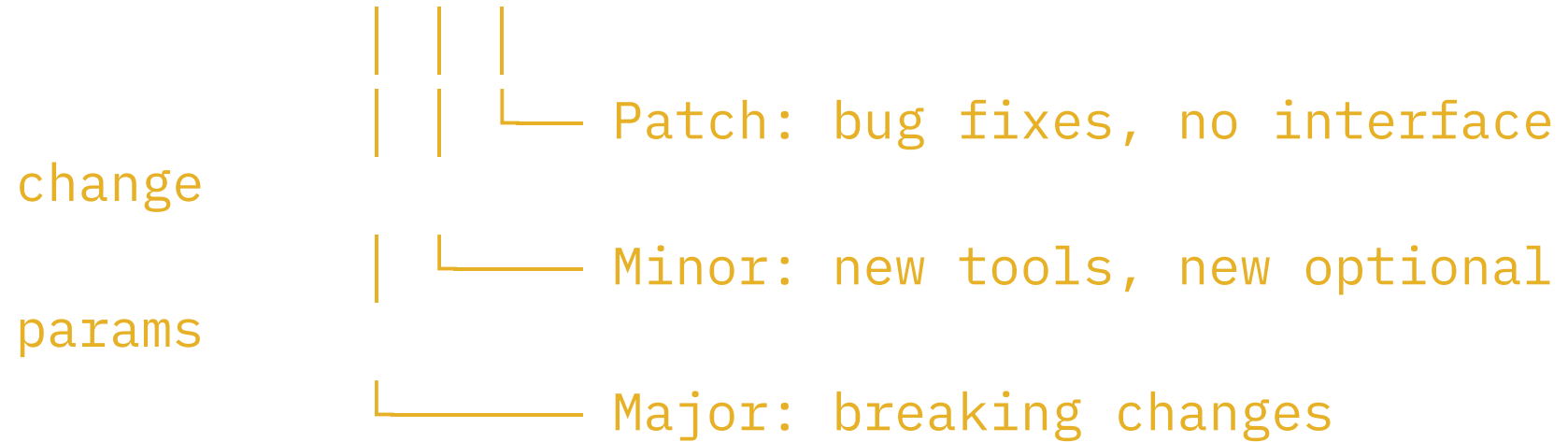
What Counts as Breaking?

Change	Breaking?	Why
	✓	
	✓	
	✓	
	✓	
	✓	
	✗	
	✗	



Semantic Versioning for MCP Servers

server-name@1.2.3



- Expose version in server metadata
- Clients can pin to major version

REST lesson: Version early, version explicitly



Compatibility Strategy

- **Prefer additive changes:** New tools > modified tools
- **Deprecation period:** Keep old tools for one release cycle
- **Deprecation visibility:** Surface via
resource://deprecations

```
{  
  "deprecated": [  
    {  
      "tool": "get_monster",  
      "replacement": "get_monster_by_id",  
      "removal_version": "2.0.0",  
      "reason": "Ambiguous name"  
    }  
  ]  
}
```

- **Migration guides:** Document how to move to new tools



Versioned Prompts and Resources

- Prompts are "**behavior contracts**"
They guide LLM reasoning
- Resources are "**schema contracts**"
They define data shapes
- **Version** them **explicitly**:

```
prompt://analyze_monster@v2  
resource://schema@v1
```
- Allows **gradual migration**
Without breaking existing clients



Client Matrix Testing

Your server is called by multiple clients

Client	Version	Capabilities

- Maintain a client matrix
- Basic smoke tests per client type
- Know what breaks when you change something

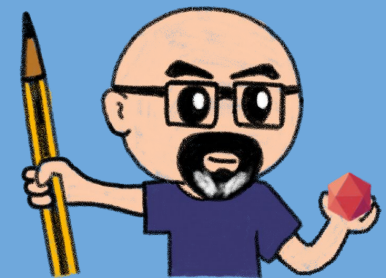


Contract Tests in CI

Prompt injection attempt test example

```
describe('Tool Contract: search_monsters_by_type', () => {
  it('schema unchanged', () => {
    const schema = getToolSchema('search_monsters_by_type');
    expect(schema).toMatchSnapshot(); // Fails if schema changes
  });
  it('example calls still succeed', async () => {
    const result = await callTool('search_monsters_by_type', { type: 'fire' });
    expect(result).toMatchSchema(expectedOutputSchema);
  });
});
```

- Run on every PR
- Snapshot schemas to detect accidental changes
- Golden examples catch semantic drift



The Principle – "Don't Surprise the Agent"

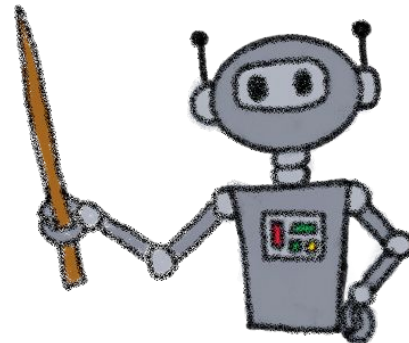
- Stability > cleverness
- Predictable structure wins
- Agents build mental models of your tools
- Changing behavior without changing signature = worst case

If you must break, break loudly



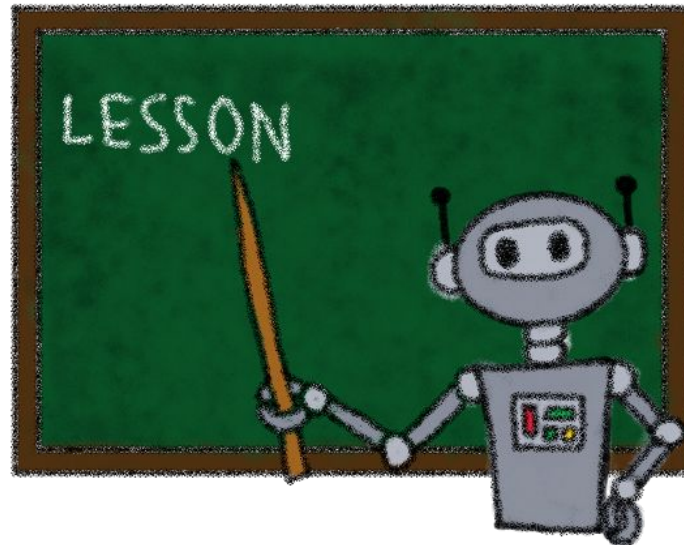
Tool schemas are your public API

Version them



Traces across servers

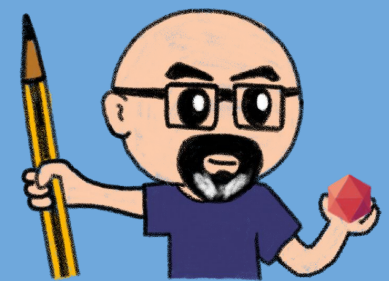
Which server did this?



One server. Four instances. One call in four failed

The last deploy had skipped one instance

They spent a day finding out



Trace Propagation, Briefly

- **Trace IDs**
Thread through every tool call
- **Gateway**
Emits spans (entry point)
- **Each server**
Emits spans (per invocation)
- **Each instance**
Tags its spans with its identity
- **Correlate at the backend**
OpenTelemetry works

**No new protocol magic, wire through
the infrastructure you already have**



Worth watching:

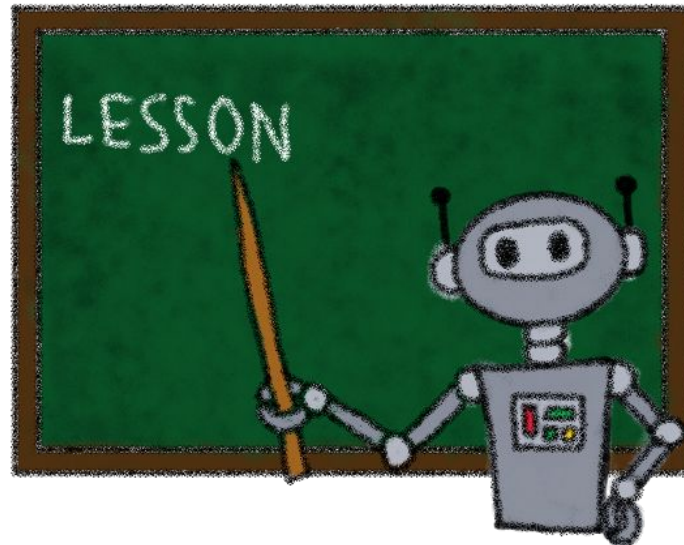
the Triggers and Events WG

may land in-protocol eventing



The LLM retries. A lot

Designing for a caller that loops



The server returned an error code, nothing else

The LLM had no context, it retried

A thousand calls. 100,000 tokens. Per hour



Idempotency For The LLM Era

REST solved this with **idempotency keys**, same pattern

The wrinkle: LLMs retry **semantically**, not just on HTTP errors

- *"I didn't see a confirmation, let me try again"*
- *"That output didn't look complete, let me retry"*
- Exponential backoff doesn't stop semantic retries

A tool that costs money must be safe to call twice



Circuit Breakers, The LLM Edition

Circuit breakers on failing dependencies. Yes, same as always

The circuit breaker's signal must be **legible to the model**

- **503** with no body → LLM learns nothing, retries
- **"rate-limited, back off 30s"** → LLM updates its plan

Error **messages** are part of the reliability contract



Hard Limits + Per-Caller Quotas

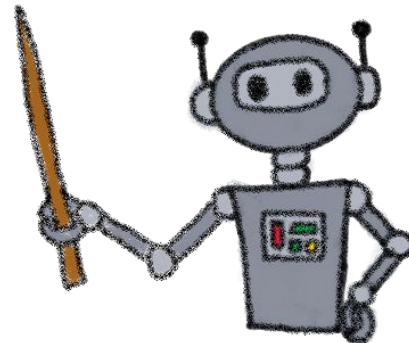
- Per-caller rate limits (the LLM is a caller)
- Hard caps on **tool invocation rate** per session
- **Loud timeouts**, fail explicitly, don't let the LLM assume
- Refuse politely in the tool response, so the model adapts

The LLM will loop 200 times. Your database will not



Design for retry

Design for storm



Our MCP Now Scales

- Auth is audience-bound
- Discovery runs through a curated registry
- Traffic flows through a gateway
- Contracts are versioned across consumers
- Traces correlate across instances
- Retries don't storm the database

A system that's safe to live next to others



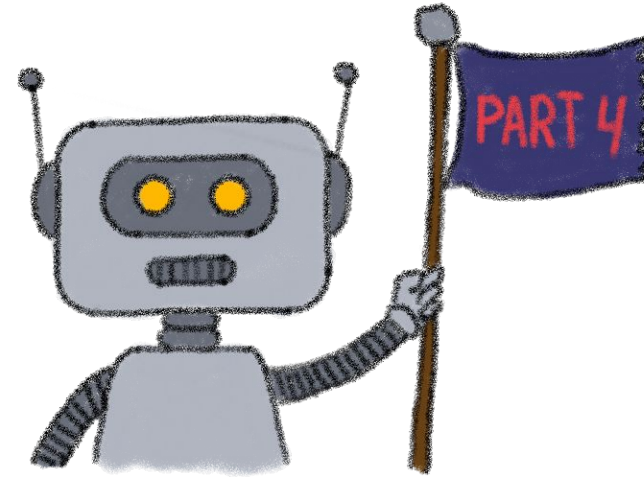
Two servers, clean auth, stable contracts

Observable and reliable

And then...

It was the legal team that asked the question





Part 4 - Governed

When the organisation wakes up



It was the legal team that asked the question

If the agent deletes production,

whose name is on the incident report?



What "Governed" Means

- Blast radius **bounded**
- Audit trail **retained**
- Cost **attributed**
- Protocol choices **deliberate**
- Ownership **named**

Every invocation accountable



We've seen this movie before

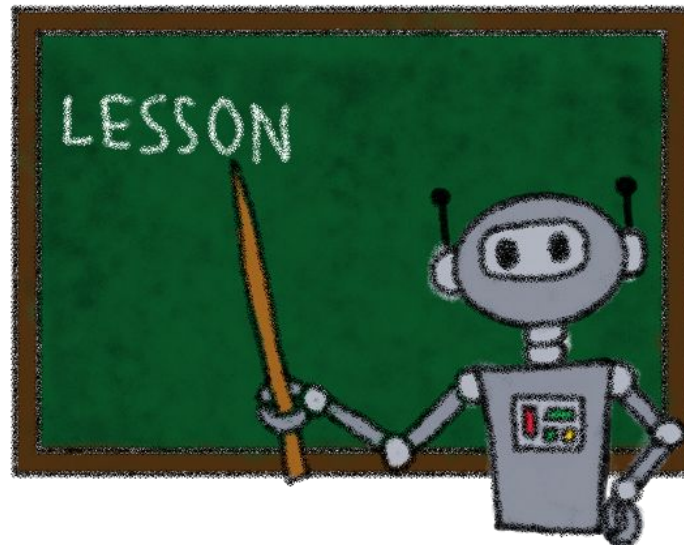
Payments went through this
Health data went through this
Banks went through this

MCP is the new regulated-API surface



Blast radius

The lethal trifecta, and what it means for MCP



Remember the ALTER TABLE

That was my toy database, I lost test data

Now imagine your production cluster

That was blast radius with a benign user
Now add a malicious one



The Lethal Trifecta

Simon Willison's framing

Three ingredients, together:

1. **Private data** the agent can read
2. **Untrusted input** that can instruct the agent
3. **Exfiltration vector** the agent can call



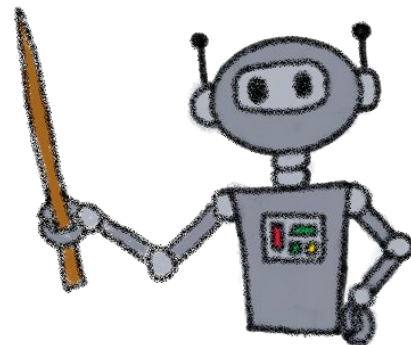
Any agent with all three
is unconditionally vulnerable
to indirect prompt injection

No system-prompt hardening fixes it



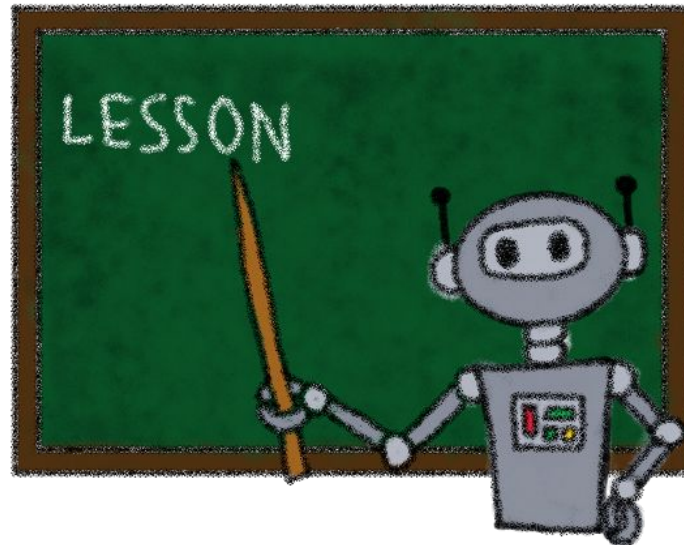
Power without blast-radius awareness

is just a bigger incident



What 2026 taught us

Incidents, CVEs, scanners – the year MCP got real



GitHub MCP – Frame 1

An attacker files a public issue

- Title and body look harmless to a human reviewer
- Body carries **hidden instructions** crafted for the LLM
- Issue sits in a public repo, waiting to be read



GitHub MCP – Frame 2

An agent with private-repo access reads the issue

- Agent treats embedded instructions as its own directive
- No system-prompt hardening catches this
- This is the lethal trifecta, live



GitHub MCP – Frame 3

The agent opens a PR containing private-repo contents

- Attacker reads the PR
- Private data is now public
- The canonical case study for the trifecta



The CVE Wave

Early 2026:

- **Wave of CVEs** against MCP servers, clients, infras
- Tool poisoning, path traversal, RCE via prompt injection
- **Many public MCP servers accept unauthenticated calls**

MCP just joined the "*we have CVEs now*" industry



OWASP MCP Top 10

- OWASP now has an **MCP Top 10** project
- Covers tool poisoning, prompt injection, trust-boundary failures
- **Emerging**, don't promise it's final

Direction of travel



mcp - scan: The Scanning Answer

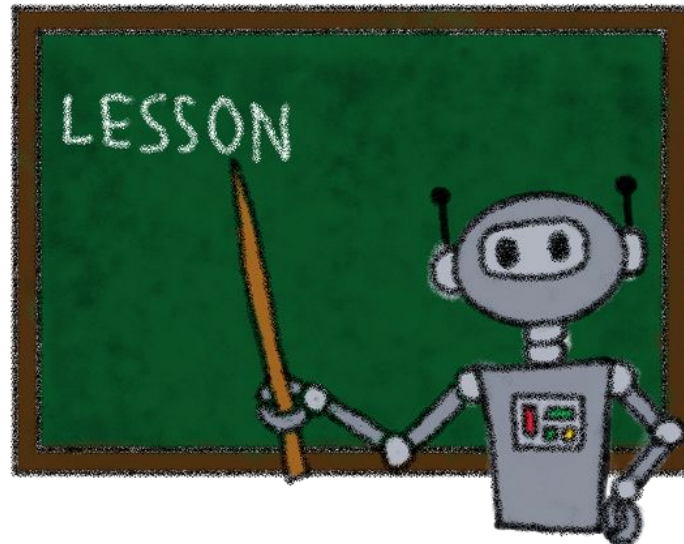
Invariant Labs' **mcp - scan**, the de facto scanner
Detects:

- **Tool poisoning**
Malicious instructions in descriptions
- **Rug pulls**
Server changes tool description after trust
- **Cross-origin escalations**
Shadowing attacks that compromise trusted tools
- **Prompt injection in metadata**
Malicious instructions contained within tool descriptions



Risk Tiering

Tools with a blast radius



Risk-Tier Your Tools

- Tag every tool with its tier
- Apply controls systematically

Tier	Description	Examples	Controls
		<code>list_types, get_schema</code>	
		<code>get_customer, search_orders</code>	
		<code>create_invoice, update_record</code>	
		<code>delete_account, transfer_funds</code>	



Approval Gates

- Human-in-the-loop for Tier 2/3 operations
- Pattern: Two-step commit
Agent can plan freely; execution requires confirmation

Step 1: `plan_change(params)` → Returns preview, no side effects

Step 2: `apply_change(plan_id)` → Executes, requires approval

- Async approval workflow:
Slack notification, approval UI

Autonomy for exploration, gates for action



A tool is destructive
until proven otherwise



Code Example

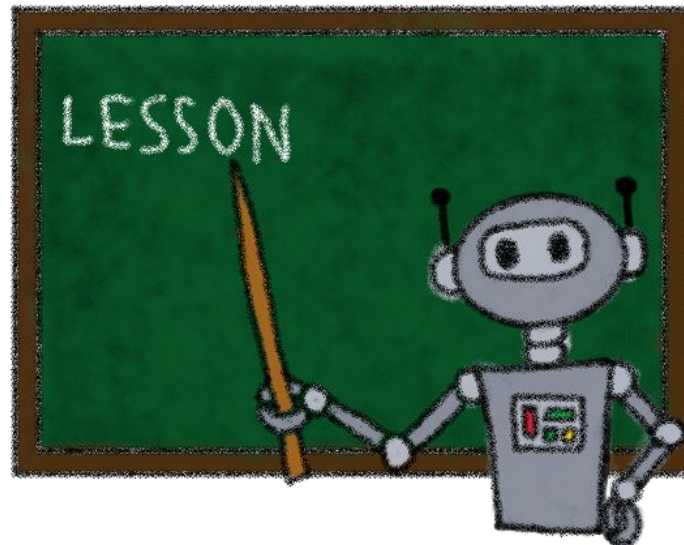
```
server.tool("delete_monster", {  
  _risk: "destructive",           // ← tier declared  
  monster_id: z.string()  
}, async ({ monster_id }, { confirm }) => {  
  if (!confirm) throw new Error("confirmation required");  
  // ...  
});
```

Tier in metadata, enforcement at invocation



Who did what, for how much

Retaining the audit trail the gateway emits



Audit Trail Retention

The gateway emits, v4 **retains, indexes, queries**

- **SIEM integration**
Your security team already has one
- **Retention windows**
Legal/compliance decides, not engineering
- **Immutable**
Write-once, queryable, exportable

Emission is easy, retention is policy



Compliance Hooks: The Honest Gap

Most of this is **not** in the spec today

- It's the glue you build around MCP
- All custom:
Retention windows, SIEM formats, legal attestations
- **MCP gives you the emission; the rest is on you**

Honesty before the close: this layer isn't solved yet



Per-Caller Cost Attribution

- Which **agent**?
- Which **team**?
- Which **budget**?

Tokens burn money

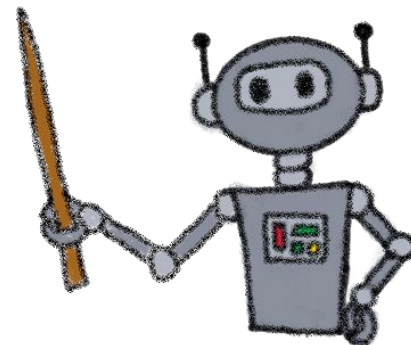
Without attribution, the bill is a mystery

The agent did this, the agent's team pays



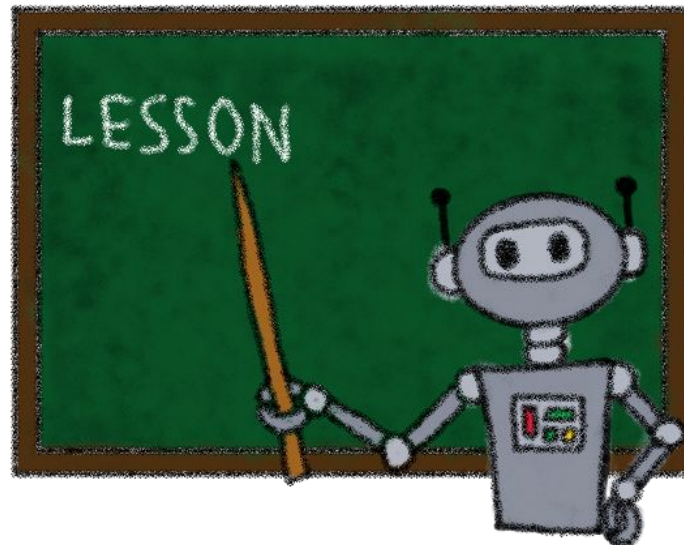
Your audit trail

is your alibi



MCP is not alone

The emerging agent-protocol stack



The Emerging Stack

- **MCP**: tool / data connectivity (LLM ↔ systems)
- **A2A**: agent-to-agent communication
- **Other protocols**: UI, payments, identity

Google's March 2026 dev guide places them as siblings



When To Use What

- **MCP**: LLM reaches out to a system
- **A2A**: agent talks to another agent
- **Don't force MCP** to be a universal transport

My recommendation: pick the layer, not the one hammer



Worth watching

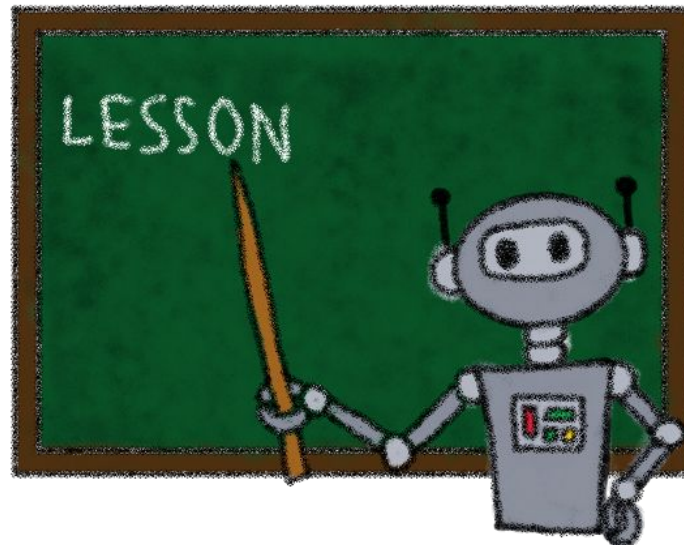
the protocol landscape is settling fast

Google's guide is a frame, not a verdict



MCP is not alone

The emerging agent-protocol stack



When To Use What

- **MCP**: tool / data connectivity (LLM ↔ systems)
- **A2A**: agent-to-agent communication
- **Other protocols**: UI, payments, identity

Google's March 2026 dev guide places them as siblings



When To Use What

- **MCP**: LLM reaches out to a system
- **A2A**: agent talks to another agent
- **Don't force MCP** to be a universal transport

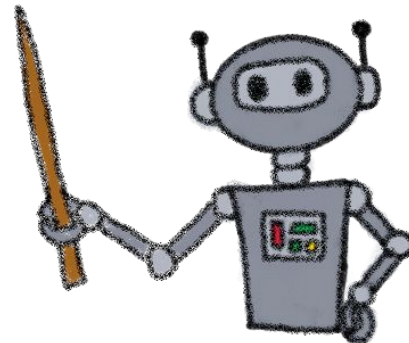
My recommendation: pick the layer, not the one hammer



Worth watching

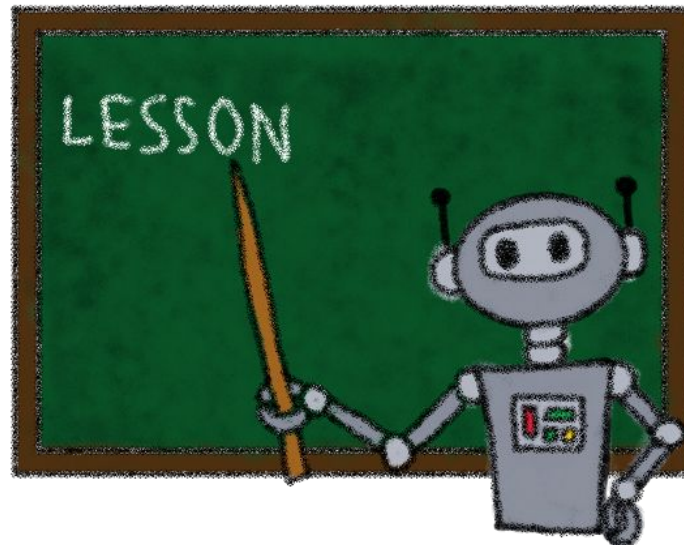
The protocol landscape is settling fast

Google's guide is a frame, not a verdict



Who owns MCP in your organisation?

The emerging platform discipline



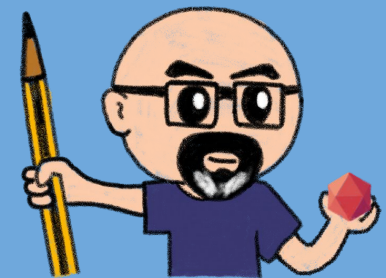
The Emerging Discipline

MCP platform teams: new role

Owners of:

- Allowlist curation
- Registry operations
- Gateway + audit pipeline
- Policy-as-code

Analogous to Kubernetes, CI, API platform team



Review + Approval Flow

Before a server reaches the allowlist:

- **Threat review**: trifecta check
- **Risk-tier classification**: safe / write / destructive
- **Audit hook verification**: events emitting?
- **Ownership named**: who carries the pager?

Policy-as-code where possible



A governed MCP server

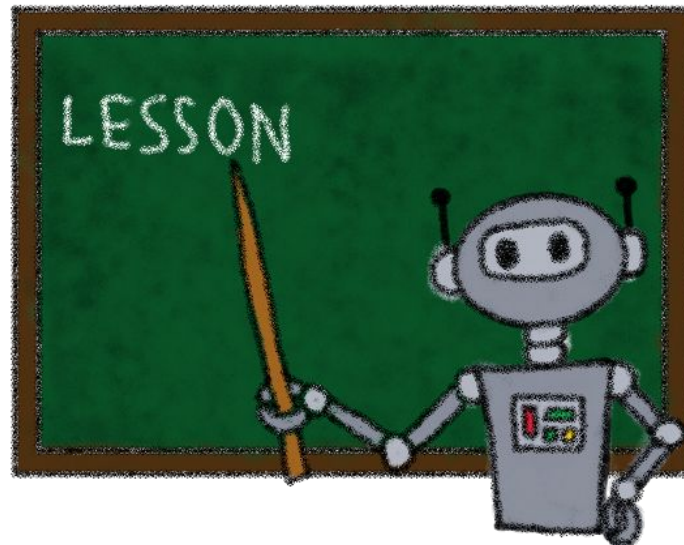
- Blast radius **bounded**
- Audit trail **retained**
- Cost **attributed**
- Protocol choice **deliberate**
- Ownership **named**
- Everything **reviewable**

A system the organisation trusts



Reliability and Cost Controls

Agents are relentless, your infrastructure must cope



Latency Budgets for Tool Calls

- **Agents feel slow** fast
- Users waiting for agent = users **waiting for your MCP** server
- Measure and alert on **latency**
- Set **targets by tool category**:

Category	Example	p95 Target
	<code>get_monster_by_id</code>	
	<code>search_monsters</code>	
	<code>create_monster</code>	
	<code>generate_report</code>	



Timeouts, Retries, and Circuit Breakers

- Timeouts:
Don't let slow calls block agents forever
- Retries:
Only for idempotent operations (reads, idempotent writes)
- Circuit breakers:
Prevent meltdown loops when downstream fails

REST lesson: These patterns are proven, apply them



Idempotency Keys for Write Tools

- **Problem:**
Agents repeat themselves (retries, loops, confusion)
- **Solution:**
Make "create" safe to retry

Tool: create_invoice

```
async function createInvoice(params: {  
  idempotency_key: string; // Required for writes  
  customer_id: string;  
  amount: number;  
}) {  
  const existing = await db.findByIdempotencyKey(params.idempotency_key);  
  if (existing) return existing; // Return same result, don't duplicate  
  return await db.createInvoice(params);  
}
```



Hard Limits Everywhere

- Agents don't know when to stop
- Protect yourself with defaults

Limit	Default	Max
-------	---------	-----

- Fail safely, explain clearly

```
// X Return 10,000 rows, blow up context  
// ✓ Return 50, include:  
// "Showing 50 of 847. Use pagination for more."
```



Token Efficiency Is an Architecture Concern

LLM context windows are **finite and expensive**

- Every **byte** you return costs **tokens**
- Patterns:
 - Return minimal fields by default
 - Provide fields or details parameter to opt-in
 - Structured data > prose descriptions
 - IDs + names > full objects



Token efficiency comparison

```
// Default response (token-efficient)
{ "id": "m1", "name": "Pyrodrake", "type": "fire" }
// With details=true
{ "id": "m1", "name": "Pyrodrake", "type": "fire",
  "description": "...", "abilities": [...], "habitat": {...} }
```



Cost Attribution

- You need to know: Who's spending? On what?
- Log "cost units" per tool call:

```
logger.info('Tool completed', {  
  tool: 'search_monsters',  
  user: session.user_id,  
  team: session.team_id,  
  agent: session.agent_type,  
  cost_units: calculateCost(result), // Your cost model  
  latency_ms: elapsed  
});
```



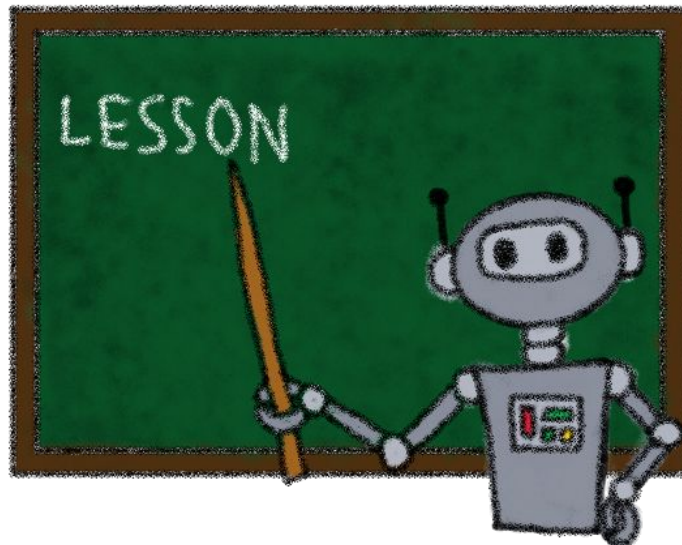
- Enables: Chargebacks, quota enforcement, optimization targeting

If you can't measure it, agents will break it silently



Safety Guardrails

Make the safe path easy, the risky path explicit



Threat Model Update

At scale, new threats emerge:

- **Agent misuse:**
Legitimate agent doing unintended things
 - **Prompt injection:**
Malicious input steering agent behavior
 - **Over-broad capability:**
Too many tools, unclear boundaries
 - **Autonomous loops:**
Agent calling tools repeatedly without oversight
- "Security is no longer just about bad inputs"**



Approval Gates

- Human-in-the-loop for Tier 2/3 operations
- Pattern: Two-step commit
Agent can plan freely; execution requires confirmation

Step 1: `plan_change(params)` → Returns preview, no side effects

Step 2: `apply_change(plan_id)` → Executes, requires approval

- Async approval workflow
Slack notification, approval UI

Autonomy for exploration, gates for action



- Central rules
 - Who can call what
 - With which limits
- Enforce in gateway or shared middleware
- Version controlled
- Auditable
- Consistent



```
Policy example

policies:
  - tool: "billing.*"
    allow:
      - role: billing_admin
      - role: finance_team
    deny:
      - agent_type: public_chat
  - tool: "/*.delete_*"
    require:
      - approval: manager
      - audit: full
```

- Every tool call recorded
- Correlation ID links multi-tool workflows
- Redact sensitive values
- Retain for compliance period



Make incident review possible

```
Audit trail example

{
  "correlation_id": "req-abc-123",
  "timestamp": "2026-02-01T10:30:00Z",
  "tool": "billing.create_invoice",
  "user": "user-456",
  "agent": "finance-assistant",
  "params": {
    "customer_id": "c-789",
    "amount": "[REDACTED]"
  },
  "result": "success",
  "latency_ms": 234
}
```

The Safety Principle

Two rules:

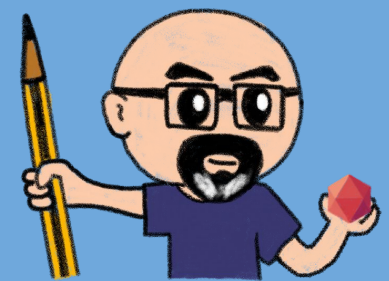
1. Make the safe path the easy path
 - Tier 0 tools: no friction
 - Good defaults everywhere
2. Make the risky path explicit and slow
 - Tier 3 tools: approval gates, audits, alerts
 - No "oops I didn't mean to delete that"

**Safety and usability aren't opposites,
good design achieves both**



That's all, folks!

Thank you all!



What We've Learned So Far

And how to go further



The Part 3 Takeaway

Scaling MCP is mostly:

- **Composition:**
Domain servers, gateways, orchestrators
- **Contracts:**
Versioning, compatibility, "don't surprise the agent"
- **Controls:**
Limits, idempotency, cost attribution, safety tiers



A Practical Maturity Ladder

Level	What You Have
v1	One server, basic validation, logs
v2	Domain servers, CI tests, structured logging
v3	Gateway, policy enforcement, evaluation suite
v4	Risk tiers, approval gates, cost attribution

- You don't need v4 on day one
- But know where you're heading



Resources

- MCP Specification:
 - modelcontextprotocol.io
- RAGmonsters examples:
 - github.com/LostInBrittany/RAGmonsters-mcp-pg
 - github.com/CleverCloud/mcp-pg-example
- Anthropic MCP docs:
 - docs.anthropic.com
- This talk's slides:
 - lostinbrittany.dev/talks



That's all, folks!

Thank you all!

