
Model Context Protocol & Elasticsearch
By Moncef Abboud
cefboud.com

What is MCP?

…Like, Is It a Big Deal or Something?

(Spoiler: Yeah, kinda.)

✨ New: Remote MCP Servers

Awesome MCP Servers

A collection of servers for the Model Context Protocol.

Search MCP servers...

Featured | All | Official 🌟 | Search | Web Scraping | Communication | Productivity | Development | Database | Cloud Service | File System | Cloud Storage | Version Control | Other

Featured MCPs

View all

Bright Data sponsor

Discover, extract, and interact with the web - one interface powering automated access across the public internet.

View Details

Browserbase official

Automate browser interactions in the cloud (e.g. web navigation, data extraction, form filling, and more)

View Details

Chrome DevTools MCP official

chrome-devtools-mcp lets your coding agent (such as Gemini, Claude, Cursor or Copilot) control and inspect a live Chrome browser

View Details

Cloudflare official

Deploy, configure & interrogate your resources on the Cloudflare developer platform (e.g. Workers/KV/R2/D1)

View Details

Context 7 official

Up-to-date Docs For Any Cursor Prompt

View Details

DeepWiki by Devin official

Remote, no-auth MCP server providing AI-powered codebase context and answers

View Details

E2B official

Run code in secure sandboxes hosted by E2B

View Details

Exa official

Search Engine made for AIs by Exa

View Details

ChatGPT 5

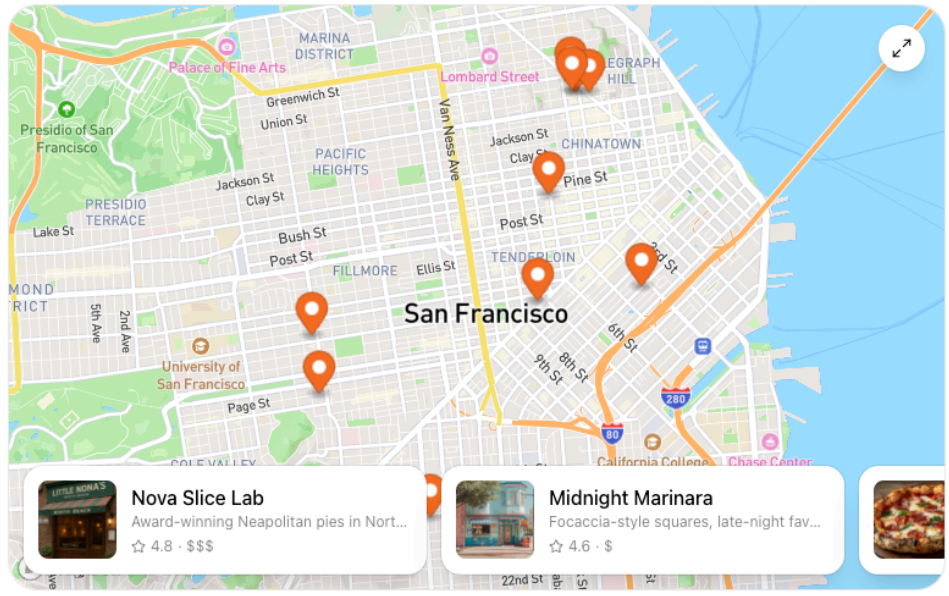
Memory Off

show me the pizza map

Looked for available tools

Served a fresh map

My Pizza Appp



Here's your **pizza map** 🍕 showing locations themed around **cheese** — enjoy exploring it!

DEVELOPER MODE

Ask anything

ChatGPT can make mistakes. Check important info.

Let’s Travel all the way back to 2022

LLMs are **token-only** systems

The Problem

- LLMs can only output tokens (text, vision)
- They can’t **execute actions**
- They can’t **query databases**
- They can’t **search your logs**

What if they could?

ReAct: Reasoning and Acting

Key Insight: Give LLMs a list of potential actions they can call

```
#### System Prompt ####

You run in a loop: **Thought → Action → PAUSE → Observation**, then give an **Answer**.
**Thought** explains your reasoning.
**Action** uses one of the tools below, then returns **PAUSE**.
**Observation** is the result of the action.

**Available actions:**

* `calculate:` Run a Python-style calculation (e.g., `calculate: 4 * 7 / 3.0`)
* `get_weather:` Get current weather for a location (e.g., `get_weather: Tokyo`)

Example:
Q: What's 5 times 3?
Thought: I should calculate it.
Action: calculate: 5 * 3
PAUSE
```

- https://til.simonwillison.net/llms/python-react-pattern

ReAct (2)

The Process:

1. LLM outputs action name + input
2. Runtime executes the action
3. Result returned as "observation"
4. LLM continues with new context

The Problem with Non-Standardized Tools

Each AI application has its **own format** for tool integration

```
# App A format
{"action": "search_logs", "params": {"query": "error", "time": "1h"}}

# App B format
<tool name="search_logs"><param name="query">error</param></tool>

# App C format
USE_TOOL(search_logs, query=error, timeframe=1h)
```

Result: Every tool needs custom integration for every AI app

The Schema Fragmentation Problem

Each LLM Provider Has Different Tool Formats

The Challenge:

Every tool developer must maintain **N different implementations** of the same tool

OpenAI Function Calling Schema

Tool Definition

```
{
  "type": "function",
  "function": {
    "name": "search_elasticsearch",
    "description": "Search Elasticsearch",
    "parameters": {
      "type": "object",
      "properties": {
        "index": {
          "type": "string",
          "description": "Index name"
        },
        "query": {
          "type": "string"
        }
      },
      "required": ["index", "query"]
    }
  }
}
```

Nested structure with "function" wrapper and "parameters" object

API Call

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
  model="gpt-4",
  messages=[
    {"role": "user",
     "content": "Search for errors"}
  ],
  tools=[tool_definition],
  tool_choice="auto"
)
```

Anthropic (Claude) Tool Schema

Tool Definition

```json
{
  "name": "search_elasticsearch",
  "description": "Search Elasticsearch",
  "input_schema": {
    "type": "object",
    "properties": {
      "index": {
        "type": "string",
        "description": "Index name"
      },
      "query": {
        "type": "string"
      }
    },
    "required": ["index", "query"]
  }
}
```

Flatter structure with "input_schema" instead of "parameters"

API Call

```python
import anthropic

client = anthropic.Anthropic()

response = client.messages.create(
  model="claude-3-5-sonnet-20241022",
  max_tokens=1024,
  messages=[
    {"role": "user",
     "content": "Search for errors"}
  ],
  tools=[tool_definition]
)
```

Google (Gemini) Function Schema

Tool Definition

```
{
  "name": "search_elasticsearch",
  "description": "Search Elasticsearch",
  "parameters": {
    "type": "object",
    "properties": {
      "index": {
        "type": "string",
        "description": "Index name"
      },
      "query": {
        "type": "string"
      }
    },
    "required": ["index", "query"]
  }
}
```

API Call

```
import google.generativeai as genai

model = genai.GenerativeModel(
  'gemini-1.5-pro',
  tools=[tool_definition]
)

response = model.generate_content(
  "Search for errors"
)
```

Similar to OpenAI but without the "function" wrapper, uses "parameters" directly

Schema Differences Summary

Provider	Wrapper	Args Key	Other Differences
OpenAI	`function` object	`parameters`	Nested structure
Claude	None	`input_schema`	Flat structure
Gemini	None	`parameters`	Similar to OpenAI

The Problem

```python
# Without a standard, you need:

def tool_for_openai():
    return {"type": "function", ...}

def tool_for_claude():
    return {"name": ..., "input_schema": ...}

def tool_for_gemini():
    return {"name": ..., "parameters": ...}
```

3× the code, 3× the bugs, 3× the maintenance

Enter MCP

Model Context Protocol

An **agreed-upon standard** for AI applications and tools to communicate

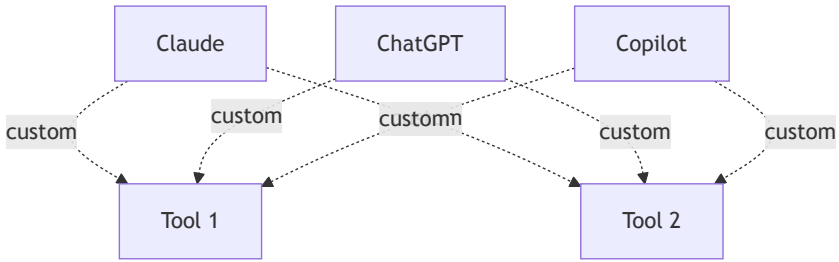
Think of it like:

- **HTML + HTTP** for the web
- **JDBC/ODBC** for databases

One standard. Universal compatibility.

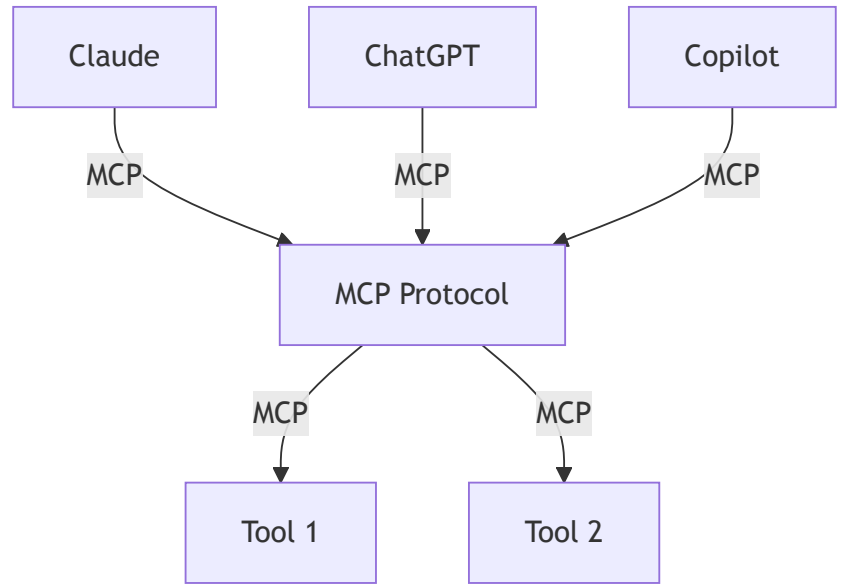
MCP = Interoperability

Without MCP



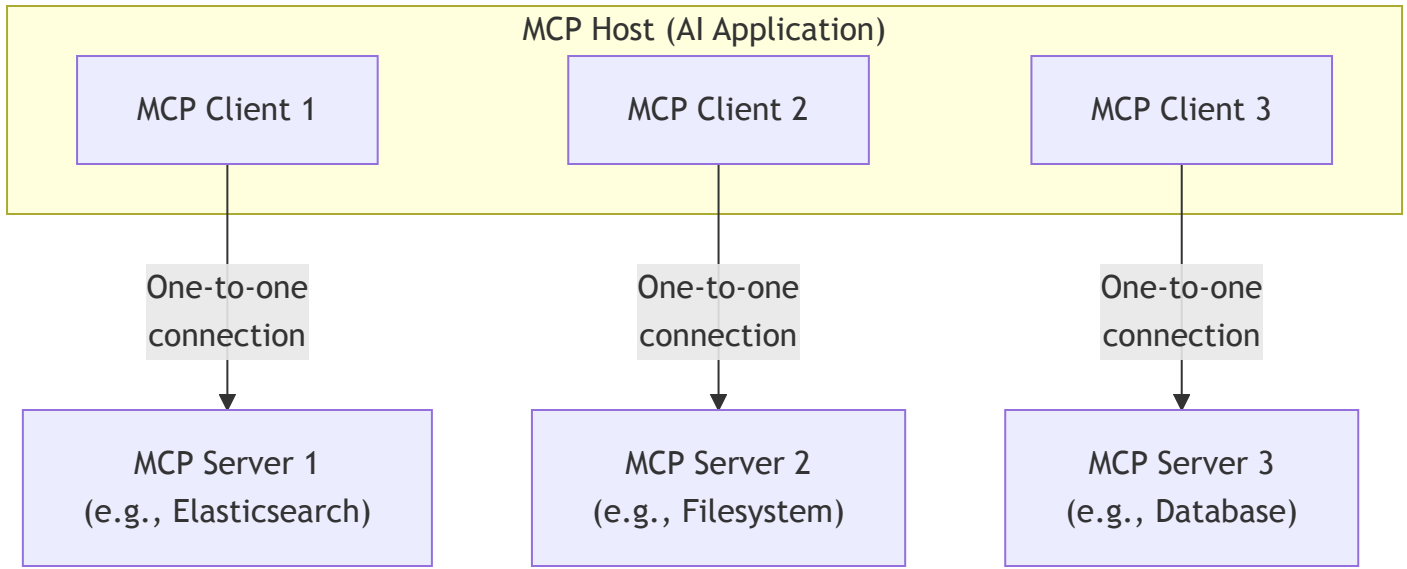
N×M integrations needed

With MCP



Plug and play compatibility

The Three Participants

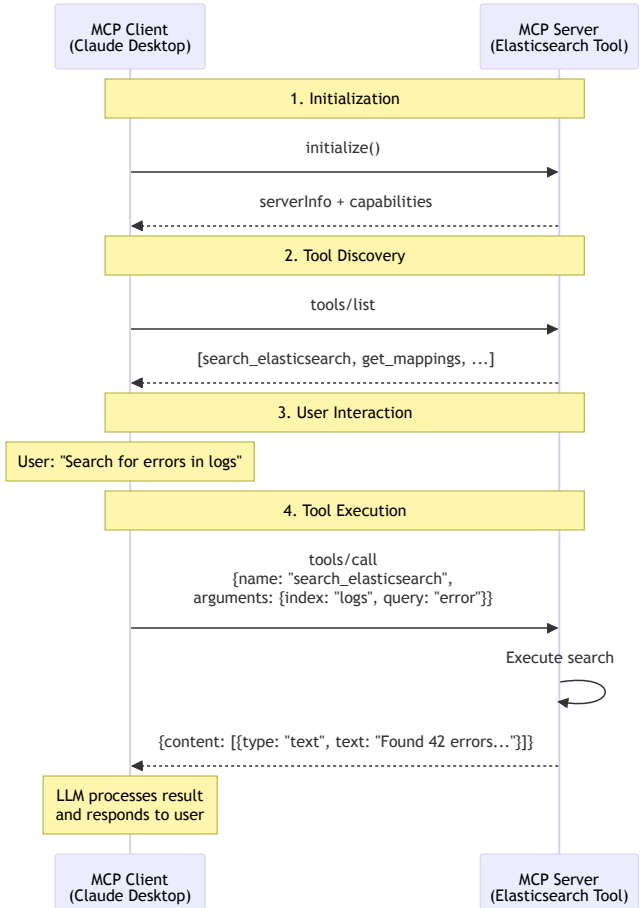


🔷 MCP Host: Claude Desktop, VS Code, your future AI coworker

🟢 MCP Client: 1 per MCP server, Bridges host ↔ server.

🟠 MCP Server: **Elasticsearch**, Figma, Github, any service with an MCP API

MCP Protocol Flow



MCP's Two Layers

📦 Data Layer (Inner)

What messages are exchanged

Messages like:

- `tools/list` - Get available tools
- `tools/call` - Execute a tool
- `resources/list` - Get resources
- Session initialization

🚚 Transport Layer (Outer)

How messages are transmitted and exchanged.

JSON-RPC

JSON Remote Procedure Call

```
// Client Request
{
  "jsonrpc": "2.0",
  "method": "subtract",
  "params": { "minuend": 42, "subtrahend": 23 },
  "id": 1
}

// Server Response
{
  "jsonrpc": "2.0",
  "result": 19,
  "id": 1
}
```

- Method name + parameters
- Requests have IDs (mapped to responses)
- Multiple requests can be in flight
- **Notifications** have no ID (no response expected)

JSON-RPC Notifications

Fire-and-forget messages with no ID

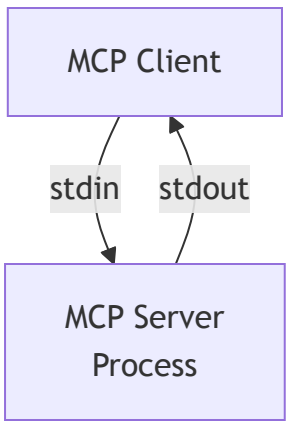
```
{
  "jsonrpc": "2.0",
  "method": "notifications/tools/list_changed"
}
```

- Initialization complete
- Tools list updated
- Status changes
- Log messages

Transport Layer

💻 Standard I/O

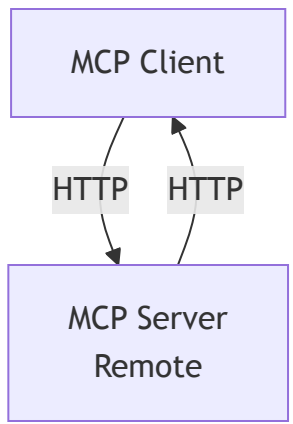
For local servers



- Same machine
- Fast & secure
- No network needed
- Process communication

🌐 Streamable HTTP

For remote servers



- Over the network
- HTTP POST for client-to-server messages
- SSE for streaming and server notifications

MCP Primitives

The building blocks of functionality

Server Primitives

What MCP Servers offer to Clients

🔧 Tools

Actions/Functions Server operations with names, descriptions, parameters, and results.

📚 Resources

Context Data to enrich LLM understanding Files, databases, or documents that enrich LLM context.

📝 Prompts

Packaged Workflows and best practices Packaged workflows for guided task execution with built-in expertise.

Client Primitives

What MCP Clients offer to Servers

🤖 Sampling

LLM calls are made by the client on demand; server requests and user approves each interaction for control and transparency.

💬 Elicitation

Server engages the user to fill in gaps—confirm actions, provide missing data, or set preferences before proceeding.

Security

Critical for remote servers and your ES cluster!

MCP emphasizes security with multiple authentication schemes:

```
Authorization: Bearer <token>
Authorization: ApiKey <elasticsearch-api-key>
Authorization: Basic <base64-credentials>
```

For Elasticsearch MCP Server:

- `ES_API_KEY` - Recommended approach
- `ES_USERNAME` + `ES_PASSWORD` - Basic auth

OAuth2.1

Latest MCP authorization relies builds on top of OAuth2

- Authorization Server Metadata (RFC8414)
- Dynamic Client Registration Protocol (RFC7591)
- Protected Resource Metadata (RFC9728)
- Resource Indicator (RFC8707)

Elasticsearch MCP in Action

Live demo time!

Introducing Elastic's MCP Server

Official implementation from Elastic

```
# Docker image
docker.elastic.co/mcp/elasticsearch

# Supports MCP protocols
- stdio (local)
- SSE (deprecated)
- streamable-HTTP (recommended)

# Elasticsearch versions
- 8.x ✅ Officially supported
- 9.x ✅ Officially supported
- 7.x ⚠️ May work, no guarantees
```

Status: ⚠️ EXPERIMENTAL (but fully functional!)

Setup: Docker Command

Running the Elasticsearch MCP Server

```
# For stdio mode (local)
docker run -i --rm \
  -e ES_URL="https://my-cluster.es.io:9200" \
  -e ES_API_KEY="your-api-key-here" \
  docker.elastic.co/mcp/elasticsearch \
  stdio

# For HTTP mode (remote)
docker run --rm \
  -e ES_URL="https://my-cluster.es.io:9200" \
  -e ES_API_KEY="your-api-key-here" \
  -p 8080:8080 \
  docker.elastic.co/mcp/elasticsearch \
  http
```

Environment Variables:

- `ES_URL` - Your cluster URL
- `ES_API_KEY` - API key (recommended) or…
- `ES_USERNAME` + `ES_PASSWORD` - Basic auth

Claude Desktop Configuration

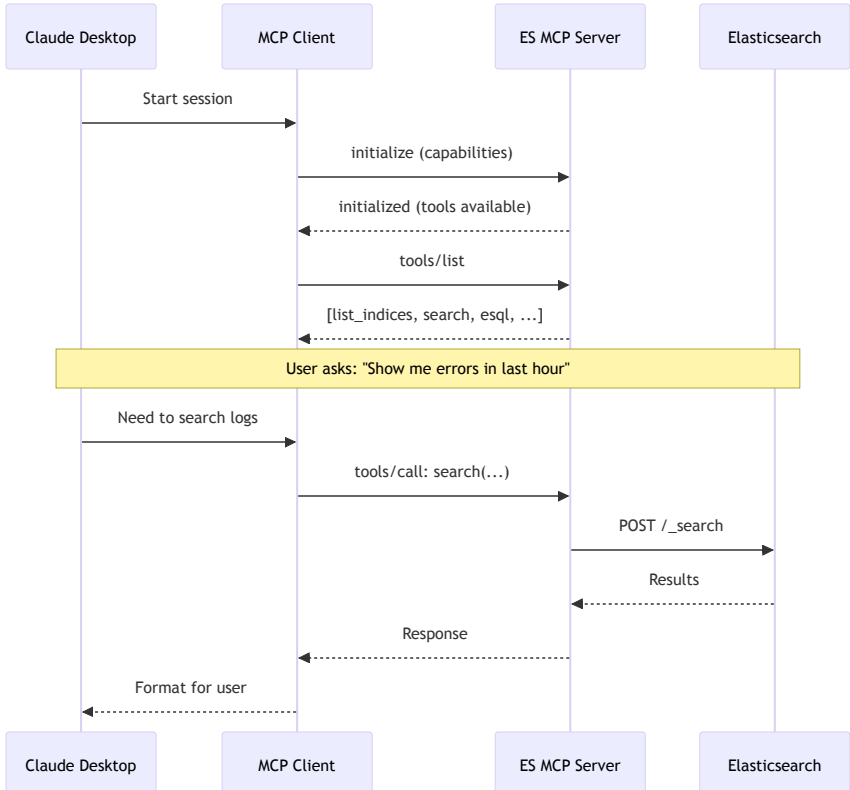
Connecting Claude to Elasticsearch

```
{
  "mcpServers": {
    "elasticsearch": {
      "command": "docker",
      "args": [
        "run", "-i", "--rm",
        "-e", "ES_URL",
        "-e", "ES_API_KEY",
        "docker.elastic.co/mcp/elasticsearch",
        "stdio"
      ],
      "env": {
        "ES_URL": "https://my-cluster.es.io:9200",
        "ES_API_KEY": "VnVhQ2ZHY0JDZGJrU ... "
      }
    }
  }
}
```

Location: `~/Library/Application Support/Claude/claude_desktop_config.json`

Restart Claude Desktop and you're ready! 🚀

The Complete Exchange Flow



Real Example: Initialization

Step 1: Client initializes session

```
{
  "jsonrpc": "2.0",
  "method": "initialize",
  "id": 1,
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "roots": { "listChanged": true },
      "sampling": {}
    },
    "clientInfo": {
      "name": "Claude Desktop",
      "version": "0.7.4"
    }
  }
}
```

Server responds:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "tools": { "listChanged": true },
      "resources": {}
    },
    "serverInfo": {
      "name": "elasticsearch-mcp-server",
      "version": "0.4.0"
    }
  }
}
```

Real Example: Getting Tools

Step 2: Client requests available tools

```
{
  "jsonrpc": "2.0",
  "method": "tools/list",
  "id": 2
}
```

Real Example: Getting Tools (2)

Server responds with tool definitions:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "tools": [
      {
        "name": "list_indices",
        "description": "List all available Elasticsearch indices",
        "inputSchema": { "type": "object", "properties": {} }
      },
      {
        "name": "search",
        "description": "Perform an Elasticsearch search",
        "inputSchema": {
          "type": "object",
          "properties": {
            "index": { "type": "string" },
            "body": { "type": "object" }
          },
          "required": ["index", "body"]
        }
      }
      // ... more tools
```

Real Example: User Query

User asks: "Show me error logs from the last hour"

Step 3: LLM decides to list indices first

```
{
  "jsonrpc": "2.0",
  "method": "tools/call",
  "id": 3,
  "params": {
    "name": "list_indices"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "content": [
      { "index": "app-logs", "status": "open","docs.count": 12},
      { "index": "products", "status": "open", "docs.count": 13}
    ]
```

Key Takeaways

1. **MCP is a protocol** - like HTTP/HTML for AI tools
2. **Two layers**: Data (JSON-RPC) + Transport (stdio/HTTP)
3. **Three participants**: Host (Claude) → Client → Server (Elasticsearch)
4. **Primitives**: Tools, Resources, Prompts / Sampling, Elicitation,
5. **Elasticsearch MCP is neat** -Talk to your ES using natural language. Unlock agentic workflows.
6. **Powerful capabilities** - search, ES|QL, mappings.

Get in touch: cefboud.com

–

Slides: github.com/CefBoud/slides