06 — MCP

🕒 state of the art 2026-07 · revised 2026-07-31 · 📖 ~17 min read

Learning objectives

By the end of this chapter, you should be able to:

1. **Explain** why MCP (Model Context Protocol) became the lingua franca of agent integration — the open-standard argument against the M×N cost of point-to-point integrations;
2. **Compare** the protocol's transports (stdio, Streamable HTTP, deprecated SSE (Server-Sent Events)) and decide which to use for local versus remote servers;
3. **Evaluate** the protocol surface (tools, resources, prompts, roots, sampling, elicitation) and what a mature client needs to support;
4. **Recognize** the MCP server as an attack surface — a tool's description is untrusted input — and name the containment defenses;
5. **Implement** the MCP client adapter (stdio) behind a port in harness-zero (step 7).

The problem

No harness can embed tools for every system in the world — databases, issue trackers, browsers, internal APIs. Without a standard, each harness would write N integrations and each tool would be rewritten for M harnesses: the classic **M×N** problem. MCP solves it via the **open standard** route: a server exposes *tools*, *resources* and *prompts* over a common protocol (JSON-RPC (Remote Procedure Call) 2.0), and any client harness consumes them without knowing who implemented them. Each side writes once — M+N instead of M×N.

In little more than two years this became industry consensus — in the studied cohort, **10 of the 11 harnesses in the cohort** are full MCP clients, all built on the protocol's official SDKs. The decisions that still differentiate the implementations:

- **Transports**: stdio (local process), Streamable HTTP (remote), and legacy SSE.
- **Authentication**: OAuth for remote servers — with which flows and providers?
- **Resilience**: reconnection, unavailable servers, dynamic changes to the tool list.
- **Surface**: only *tools*, or also *resources*, *prompts*, *roots*, *sampling*, *elicitation*?
- **Role**: is the harness client-only, or also a **server** — consumable by other agents?
- **Security**: an MCP server is third-party code injecting text into the model's context. Who treats that as an attack surface?

Scientific foundations

MCP was born as an **industry specification**, not from a paper — and the academic literature that caught up with it concentrates, revealingly, on **security**. The design decision this whole literature supports is a single, decisive one: **a tool's description (and its return value) from an MCP server is untrusted input**, and must be treated with the same skepticism as any external content.

- **Indirect prompt injection** — Greshake et al., arXiv 2302.12173 (AISec '23), the paper that defined the threat: LLM (Large Language Model)-integrated applications blur the boundary between *data* and *instructions*, so any retrieved content is a potential instruction channel. Translated to the MCP client: a tool's description field and the text the server returns are **data**, never trusted instructions.
- **The MCP systematization** — Hou et al., "MCP: Landscape, Security Threats, and Future Research Directions", arXiv 2503.23278 (also in ACM TOSEM): the canonical SoK. It decomposes the server lifecycle (creation → deployment → operation → maintenance) and shows that the **same server is attackable in different phases** — spoofing at installation, *tool poisoning* at runtime. Decision: the harness needs **per-phase** trust boundaries, not a single gate.
- **Tool description as a vector, measured** — MCPTox, arXiv 2508.14925, the first *tool poisoning* benchmark over 45 real servers / 353 tools: success rates of up to ~73%, and — the uncomfortable finding — **more capable models were more susceptible**, with safety alignment offering minimal protection before execution. Hard decision: you cannot trust the model to self-filter; the tool's metadata has to be blocked **before** it enters the context window.
- **The base rate is empirical, not hypothetical** — "MCP at First Glance", arXiv 2506.13538 audited 1,899 open-source servers: **7.2% with general vulnerabilities and 5.5% with MCP-specific *tool poisoning***, in classes that only partially overlap with traditional appsec. Decision: assume a non-trivial base rate of poisoned servers in the real world; MCP-aware scanning, not just SAST. (See also MCP Safety Audit, arXiv 2504.03767, which shows code execution and credential theft exploits via *legitimately registered* tools.)
- **Choose the protocol by trust context** — the interoperability survey, arXiv 2505.02279 compares MCP, ACP, A2A and ANP: MCP assumes a **relatively trusted** client-server boundary; exposing MCP tools across organizational boundaries does not inherit A2A/ANP's identity guarantees and requires additional authn/authz (ties into ch. 17).

> Editorial note (living book): this was the book's most rarefied bibliography dimension — recorded as an "academic gap". Between rounds it matured from gap to **consolidated security literature** (an SoK, benchmarks, empirical audits). The migration is noted in `bibliografia.md`.

(Full bibliography and pointers: `livro/bibliografia.md`.)

Industry sources

- **MCP architecture** (official spec): defines what the harness needs to map — on the server side, *tools/resources/prompts*; on the client side, *roots/sampling/elicitation*. The design decision: separating what the server *offers* from what the client *grants* — `sampling` and `roots` exist so the server can

request an inference or a file scope **without ever having direct access** to the model or the filesystem, keeping the host as the single point of trust.

- **Transports** (spec): two transports over JSON-RPC — **stdio** (local, no network overhead, the default for local servers) and **Streamable HTTP** (remote). The old **HTTP+SSE was deprecated in the 2025-03-26 revision** and survives only for backwards compatibility — the modern client tries `POST InitializeRequest` first and only falls back to SSE on a 4xx.
- **Introducing the Model Context Protocol** (Anthropic, Nov 25, 2024): the announcement that opened the standard, with the **"USB-C for AI"** analogy (one connector, many peripherals) — which is exactly the harness's M×N argument. *(anthropic.com returns 403 through the proxy; date and framing confirmed via VentureBeat.)*
- **Adoption as the turning point**: OpenAI adopted MCP in Mar 2025 (Agents SDK (Software Development Kit), TechCrunch); Google/Gemini followed (The New Stack); Microsoft brought MCP to GA in Copilot Studio and to Windows. Key decision: when the second-largest lab adopts its competitor's protocol, MCP stops being a vendor bet and becomes **neutral infrastructure** — designing for MCP reduces lock-in risk.
- **Authorization (OAuth 2.1)**: the spec treats every remote server as an **OAuth 2.1 Resource Server** — validating tokens issued by an external Authorization Server (RFC 9728 + 8414 + 7591). Descope's practical guide translates it into a decision: separating *who serves the tool* from *who issues identity* enables corporate SSO and per-resource scoped tokens, instead of credentials embedded in the server.
- **Security in practice** — Invariant Labs' Tool Poisoning Attack (Apr 1, 2025) coined the term: malicious instructions hidden in a tool's *description* that the user never reads but the model obeys. Trail of Bits showed "line jumping": the mere **registration** of a server is already attack surface, before any invocation — the trust gate has to be at *connect*, not at *call*. And the lethal trifecta (Simon Willison): private data + untrusted content + external communication — MCP makes it far too easy to glue together tools that, combined, close all three corners (read email + open a public PR = exfiltration). The design rule is to prevent the three from coexisting in the same loop.
- **Governance (MCP became a boundary, not a prosthesis)**: the official registry (preview, Sep 2025) is a *community-owned* API layer — the harness discovers servers via a standardized API, not hardcoded lists. And in Dec 2025 Anthropic donated MCP to the Agentic AI Foundation, under the Linux Foundation — alongside **goose and AGENTS.md** as founding projects. The protocol now evolves by consensus of a steering group, not by one vendor's roadmap.
- **The 2026-07-28 Specification** (official MCP blog): the announcement of the protocol's biggest revision — stateless core, MRTR, extensions, caching and a deprecation policy (see §6 of the state of the art).
- **See also**: the living collection Awesome Harness Engineering — Skills & MCP gathers more consultable resources for this dimension (patterns, articles, and implementations), curated by problem.

The state of the art

1. The discipline's clearest standardization

Eleven harnesses in the cohort, several languages (TypeScript, Rust, Python), the same protocol, the official SDKs. It is the most limpid case of convergence the book has recorded: where tool design, loop and compaction

diverge, MCP unified. The single exception in the cohort is **Aider** (MCP score 0) — and it is a *philosophical* exception, not a lag: the *context-first* school bets on curated context and edit formats, and forgoes MCP on purpose.

2. Transports converged — stdio local, Streamable HTTP remote

The default has stabilized: **stdio** for local servers (the harness launches the process), **Streamable HTTP** for remote ones. **SSE** became legacy — present only as a compatibility fallback (opencode does *automatic HTTP→SSE fallback*). More rigorous harnesses pin the **protocol revision** (IronClaw explicitly speaks `2025-06-18`), a sign that the protocol has versions and the client needs to negotiate them.

3. The turn: the harness became an MCP server too

This is the chapter's strongest *dated update* — and a **prediction by the book itself that expired**. In the early rounds we noted that "none of the harnesses acts as an MCP *server* in the core; harness-as-a-service shows up via A2A/ACP". Round 2 refuted that: **Codex, Hermes, OpenClaw, OpenHands and n8n expose themselves as MCP servers**. The harness stopped being merely a consumer of tools and became a **piece consumable by other agents** — Codex exposes itself to IDEs and other hosts; OpenClaw serves its channel conversations to Claude Code/Codex; n8n publishes its workflow graph as an MCP endpoint. With that, the protocol surface widens beyond *tools*: **sampling** (the server requests completions from the client — Hermes) and **elicitation** (the server requests structured input — Codex) enter the state of the art. Harness-as-a-service, which used to exist only via A2A/ACP, now has a native MCP route.

4. Authentication: OAuth 2.1 is the floor; enterprise raises the bar

For remote servers, the OAuth flow with PKCE + local callback + token storage became the minimum (opencode, gemini-cli, Codex, Hermes, OpenClaw). The competitive differential sits above it: **gemini-cli** adds **Google auth** providers and **service account impersonation** (MCP designed for corporate GCP); **OpenClaw** stores tokens in SQLite and supports **mTLS**. Enterprise authentication is today's MCP feature frontier.

5. Security: the MCP server is third-party code — and the cohort has started treating it that way

If an MCP server injects text into the context and can see tool arguments, it is attack surface — and the literature (above) shows the threat is measurable and common. The defenses observed in the cohort, in layers (all connect to ch. 07):

- **Test the vector**: gemini-cli includes a **prompt injection via MCP** eval — the only one that treats the server as a *tested* attacker.
- **Filter the environment**: OpenClaw, when launching a stdio server, **blocks dangerous environment variables** (`NODE_OPTIONS`, `LD_*`, `DYLD_*`) that would allow loading code into the process.
- **Mediate credentials**: IronClaw adapts MCP tools into *capabilities* **without granting ambient authority** — FS, secrets and network remain mediated, and **the server never sees the secret** (the credential is injected at the edge). OpenHands does **secret redaction and restoration** in configuration round-trips.

The bar rose from "connect a server" to "connect a **contained** server" — exactly what the *tool poisoning* and *line jumping* papers call for: a gate at connect time, sanitization between servers, and no trust in the model's self-filtering.

6. The stateless turn — the 2026-07-28 spec

Three days before this revision, the protocol went through its **biggest change since launch** (official announcement; changelog). The core became **stateless**: the `initialize`/`notifications/initialized` handshake and the `Mcp-Session-Id` header are gone — each request travels independently, with protocol, identity and capabilities in `_meta` (plus an optional `server/discover` for discovery). The motivation is infrastructural: MCP servers can now scale behind an ordinary round-robin load balancer, without *sticky sessions*. Server-initiated requests (`elicitation/create`, `sampling/createMessage`, `roots/list`) give way to **MRTR (Multi Round-Trip Requests)**: the server responds `resultType: "input_required"` and the client retries with `inputResponses` — bidirectionality becomes an explicit round trip. Rounding out the package: a **formal extension framework** (Tasks becomes `io.modelcontextprotocol/tasks`; MCP Apps and Enterprise Managed Authorization as extensions), **caching as a contract** (`ttlMs`/`cacheScope` in listing responses — see ch. 04), header-based routing (`Mcp-Method`/`Mcp-Name`) and the **first formal deprecation policy** (a minimum 12-month window) — under which **Sampling, Roots, Logging, the legacy HTTP+SSE transport and DCR (Dynamic Client Registration, replaced by CIMD — Client ID Metadata Documents)** are deprecated. Editorial reading: what sections 1–5 describe is still the protocol *installed* in the cohort (the 12-month window exists for exactly that), but the direction has changed — and harness adoption of 2026-07-28 is the number one item to measure in the next benchmark round.

> **EXECUTIVE SUMMARY**
>
> The protocol turned a page on 2026-07-28: a **stateless** core (no handshake, no `Mcp-Session-Id`), MRTR in place of server-initiated sampling/elicitation, formal extensions, caching as a contract (`ttlMs`) and the first deprecation policy (12 months) — under which Sampling, Roots, Logging and the HTTP+SSE transport fall. What the cohort *runs today* is still the protocol of sections 1–5 (the window exists for that); what you *write today* should already target 2026-07-28. **What to steal:** treat the tool description as untrusted input (the literature measures ~73% *tool poisoning* success); in new code, prefer stateless Streamable HTTP (the SSE fallback is now a deprecated transport); if you expose an MCP server, filter the subprocess environment and never let the server see secrets; if your audience is enterprise, OAuth 2.1 with impersonation is the floor — and plan the DCR→CIMD migration within the window.

Hands-on — harness-zero, step 7

Step 7 (`harness-zero/etapas/07-mcp/`) gives harness-zero an **MCP client adapter (stdio)** behind a port. Faithful to hexagonal architecture *by refactoring*: step 2's `ToolPort` already defines what a tool is; now an adapter discovers tools from an external MCP server (via `stdio`, launching the process) and presents them to

the loop as native tools — the model cannot tell the difference. You connect the included example server (`servidor_mcp_exemplo.py`) — and, as an extension, any real filesystem MCP server. Period note: the step's `ClienteMCP` implements the 2025-06 protocol's `initialize` handshake — which the 2026-07-28 spec **removed** (stateless core); it keeps working within the 12-month deprecation window, and the difference between the two generations is, in itself, a lesson, lists its tools, and calls them through the same path as local tools. Completeness exercise: the client handles the *happy path*; you add **graceful degradation** (a server that dies does not take down the session) and an **env filter** on the stdio subprocess — the state of the art's minimum defense.

Check your understanding

1. Why does MCP reduce integration cost from M×N to M+N, and what does that have to do with "open standard"? (Each harness and each tool writes once; the common protocol decouples the two sides.)
2. You are about to connect a third-party MCP server exposing a `search_tickets` tool. Give two reasons to distrust it and two concrete defenses. (Tool description = *tool poisoning*; return value = indirect injection; and *registration* itself is a vector — *line jumping*. Defenses: filtered env on the stdio subprocess; mediated credential — the server never sees the secret; injection eval; gate at connect; ch. 07 containment.)
3. A harness that is both MCP **client AND server** gains what that a client-only one does not — and which protocol primitives does that activate? (It becomes a piece consumable by other agents; it activates *sampling* and *elicitation*, the server asking the client.)

Appendix A — How each repository handles MCP

> Per-harness evidence, with paths — online supplement, expanded each round.

opencode (round 1) — the most complete protocol implementation

`packages/opencode/src/mcp/` (~1,000 lines in `index.ts`, plus `catalog.ts`, `oauth-provider.ts`, `auth.ts`). Three transports — `StdioClientTransport`, `StreamableHTTPClientTransport` and `SSEClientTransport` with **automatic HTTP→SSE fallback**. Full OAuth: authorization with a local callback server, PKCE, dedicated `opencode mcp auth` command. Covers the wide surface: `ToolListChanged` notifications, logging, roots, prompts, resources and resource templates. Server instructions go into the system prompt (`system.ts:mcp()`) — the server can teach the model how to use it (which is also the injection vector).

gemini-cli (round 1) — enterprise-grade OAuth

`packages/core/src/tools/mcp-client.ts` + `mcp-client-manager.ts`, the same three transports by config. The differential in `packages/core/src/mcp/`: beyond standard OAuth, **Google auth** providers

and **service account impersonation** — MCP for corporate GCP. Tools become `DiscoveredMCPTool` with per-server namespacing; MCP prompts exposed; management via `/mcp` and `~/.gemini/settings.json`. Notable: the eval suite includes a **prompt injection via MCP** test (ch. 11) — the only one to treat an MCP server as a tested attack surface.

OpenHarness (round 1) — pragmatic client

`src/openharness/mcp/` (`McpClientManager`) on the `mcp>=1.0.0` SDK: **stdio** and **Streamable HTTP** transports (no SSE), with connection status, auto-reconnect and **graceful degradation** when a server dies (`call_tool`/`read_resource` do not take down the session). Resources exposed as its own tools (`list_mcp_resources`, `read_mcp_resource`); `mcp_auth` for authentication. Config via `oh mcp` and `--mcp-config`.

Goose (round 2) ⭐ MCP-native — the protocol as backbone

The extreme case: **every tool is MCP**. The `goose-mcp` built-ins (memory, computercontroller, tutorial…) are real `rmcp::ServerHandler` servers served **in-process over `DuplexStream`** (virtual stdio) and can run standalone (`goose mcp <server>`). Even developer/shell/edit are "platform extensions" speaking `McpClientTrait`. A single abstraction for the entire tool surface — the protocol is not an integration, it is the architecture. (Goose is also one of the founding projects of the Agentic AI Foundation.)

Codex CLI (round 2) — client and server, four transports

`rmcp-client/` + `mcp-server/` (Codex exposes itself as an MCP server). **Four transports** (stdio, streamable HTTP, in-process, process-executor); **full OAuth** with refresh transactions and store locking; **elicitation**; server prewarm/refresh; per-MCP-tool approval templates. Integrates MCP into containment (per-tool approval).

Hermes (round 2) — client and server, with *sampling*

Client with stdio/StreamableHTTP/SSE, OAuth, per-server timeouts, **sampling** (the server can request completions from the client) and per-server opt-in parallelism; `mcp_serve.py` exposes Hermes to other MCP hosts.

OpenClaw (round 2) — client and server, with env filtering

`openclaw mcp serve` exposes channel conversations via stdio to Codex/Claude Code. Client: `mcp.servers` registry with stdio/SSE/streamable-http, **OAuth PKCE in SQLite**, **mTLS**, tool filters, probe/doctor — and an **env security filter** on stdio (blocks `NODE_OPTIONS`, `LD_*`, `DYLD_*`). MCP Apps support with an isolated-origin sandbox.

OpenHands (round 2) — bidirectional, with secret redaction

Client (per-agent MCP config with secret redaction/restoration in GET/PUT round-trips) and **server** (the app-server is a FastMCP exposing PR tools — `create_pr`/`create_mr` — to the sandboxes, plus an **MCP proxy**

for Tavily providing search without exposing the API key). Agent profiles reference server subsets.

IronClaw (round 2) — capability-mediated MCP

`ironclaw_mcp` adapts MCP tools into **capabilities without granting ambient authority**: FS, secrets and network remain mediated; **Streamable HTTP** (protocol `2025-06-18`); **mediated credential injection** (the server never sees the secret); resources accounted for by the governor. Ch. 07's containment model applied to MCP.

ohmo (round 2) — inherited

Complete via `McpClientManager` (inherited from the base); server counts in state and a summary exposed to the gateway. Gap: no MCP config of its own (`~/.ohmo/mcp.json` does not exist) and no per-channel/per-sender MCP isolation.

n8n (round 2) — bidirectional in the workflow engine

MCP Client Tool (SSE + Streamable HTTP, Bearer/OAuth2, tool filtering, per-execution session cache) and **MCP Server Trigger** (`McpTrigger` + `McpServer.ts`) — exposes the connected n8n tools as an MCP endpoint to external clients. Official SDK. The "inverted harness" also speaks MCP in both roles.

Aider (round 2) — absent by philosophy

MCP score **0**. The *context-first* school in its pure state: the 3s sit where the philosophy bets (context, edit formats, git, evals), and the MCP gap is a choice, not a lag.

Frameworks (frameworks round)

Agents SDK (OpenAI): support for MCP servers as a tool source; LangGraph/langchain: MCP adapters for tools; CrewAI: MCP integration via toolkit; software-agent-sdk: MCP-style annotations in the tool contract. MCP is an assumed integration point at the framework layer too — reinforcing the standardization thesis.