06 — MCP

🕒 estado da arte 2026-07 · revisão 2026-07-31 · 📖 ~17 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Explicar** por que o MCP (Model Context Protocol) virou a língua franca da integração de agentes — o argumento do padrão aberto contra o custo M×N de integrações ponto a ponto;
2. **Comparar** os transportes do protocolo (stdio, Streamable HTTP, SSE (Server-Sent Events) depreciado) e decidir qual usar para servidor local × remoto;
3. **Avaliar** a superfície do protocolo (tools, resources, prompts, roots, sampling, elicitation) e o que um cliente maduro precisa suportar;
4. **Reconhecer** o servidor MCP como superfície de ataque — a descrição de uma tool é input não confiável — e nomear as defesas de contenção;
5. **Implementar** o adapter MCP client (stdio) atrás de uma porta no harness-zero (etapa 7).

O problema

Nenhum harness consegue embutir tools para todos os sistemas do mundo — bancos de dados, rastreadores de issues, navegadores, APIs internas. Sem um padrão, cada harness escreveria N integrações e cada ferramenta seria reescrita para M harnesses: o clássico problema **M×N**. O MCP resolve pela via do **padrão aberto**: um servidor expõe *tools*, *resources* e *prompts* num protocolo comum (JSON-RPC (Remote Procedure Call) 2.0), e qualquer harness cliente os consome sem saber quem os implementou. Cada lado escreve uma vez — M+N em vez de M×N.

Em pouco mais de dois anos isso virou consenso de indústria — na coorte estudada, **10 dos 11 harnesses da coorte** são clientes MCP completos, todos sobre os SDKs oficiais do protocolo. As decisões que ainda diferenciam as implementações:

- **Transportes**: stdio (processo local), Streamable HTTP (remoto) e o SSE legado.
- **Autenticação**: OAuth para servidores remotos — com que fluxos e provedores?
- **Resiliência**: reconexão, servidores indisponíveis, mudança dinâmica da lista de tools.
- **Superfície**: só *tools*, ou também *resources*, *prompts*, *roots*, *sampling*, *elicitation*?
- **Papel**: o harness é só cliente, ou também **servidor** — consumível por outros agentes?
- **Segurança**: um servidor MCP é código de terceiros injetando texto no contexto do modelo. Quem trata isso como superfície de ataque?

Fundamentos científicos

O MCP nasceu como **especificação de indústria**, não de um paper — e a literatura acadêmica que o alcançou concentra-se, de forma reveladora, em **segurança**. A decisão de projeto que toda essa literatura sustenta é uma só, e decisiva: **a descrição de uma tool (e o retorno) de um servidor MCP é input não confiável**, e deve ser tratada com o mesmo ceticismo de qualquer conteúdo externo.

- **Injeção indireta de prompt** — Greshake et al., arXiv 2302.12173 (AISec '23), o paper que definiu a ameaça: aplicações integradas a LLM (Large Language Model) borram a fronteira entre *dados* e *instruções*, então qualquer conteúdo recuperado é um canal de instrução em potencial. Traduzido para o cliente MCP: o campo de descrição de uma tool e o texto que o servidor devolve são **dados**, nunca instruções confiáveis.
- **A sistematização do MCP** — Hou et al., "MCP: Landscape, Security Threats, and Future Research Directions", arXiv 2503.23278 (também em ACM TOSEM): o SoK canônico. Decompõe o ciclo de vida do servidor (criação → deploy → operação → manutenção) e mostra que o **mesmo servidor é atacável em fases diferentes** — spoofing na instalação, *tool poisoning* em runtime. Decisão: o harness precisa de fronteiras de confiança **por fase**, não um único gate.
- **Descrição de tool como vetor, medido** — MCPTox, arXiv 2508.14925, primeiro benchmark de *tool poisoning* sobre 45 servidores reais / 353 tools: taxa de sucesso de até ~73%, e — o achado desconfortável — **modelos mais capazes foram mais suscetíveis**, com o alinhamento de segurança oferecendo proteção mínima antes da execução. Decisão dura: não dá para confiar no modelo para se autofiltrar; a metadata da tool tem que ser barrada **antes** de entrar na janela de contexto.
- **A base é empírica, não hipotética** — "MCP at First Glance", arXiv 2506.13538 auditou 1.899 servidores open-source: **7,2% com vulnerabilidades gerais e 5,5% com *tool poisoning*** específico de MCP, em classes que só parcialmente coincidem com appsec tradicional. Decisão: assuma uma taxa-base não-trivial de servidores envenenados no mundo real; scanning ciente-de-MCP, não só SAST. (Ver também MCP Safety Audit, arXiv 2504.03767, que mostra exploits de execução de código e roubo de credencial por tools *legitimamente registradas*.)
- **Escolha o protocolo pelo contexto de confiança** — survey de interoperabilidade, arXiv 2505.02279 compara MCP, ACP, A2A e ANP: o MCP assume uma fronteira cliente-servidor **relativamente confiável**; expor tools MCP através de fronteiras organizacionais não herda as garantias de identidade de A2A/ANP e exige authn/authz adicional (liga ao cap. 17).

> Registro editorial (livro vivo): esta era a dimensão de bibliografia mais rarefeita do livro — registrada como "lacuna acadêmica". Entre as rodadas ela amadureceu de lacuna para **literatura de segurança consolidada** (um SoK, benchmarks, auditorias empíricas). A migração está anotada em `bibliografia.md`.

(Bibliografia completa e ponteiros: `livro/bibliografia.md`.)

Fontes da indústria

- **Arquitetura do MCP** (spec oficial): define o que o harness precisa mapear — no servidor, *tools/resources/prompts*; no cliente, *roots/sampling/elicitation*. A decisão de design: separar o que o servidor *oferece* do que o cliente *concede* — `sampling` e `roots` existem para o servidor pedir uma inferência ou um escopo de arquivos **sem nunca ter acesso direto** ao modelo ou ao filesystem, mantendo o host como único ponto de confiança.
- **Transportes** (spec): dois transportes sobre JSON-RPC — **stdio** (local, sem overhead de rede, o default para servidores locais) e **Streamable HTTP** (remoto). O antigo **HTTP+SSE foi depreciado na revisão 2025-03-26** e só sobrevive por retrocompatibilidade — o cliente moderno tenta `POST InitializeRequest` primeiro e só cai para SSE em 4xx.
- **Introducing the Model Context Protocol** (Anthropic, 25/nov/2024): o anúncio que abriu o padrão, com a analogia do **"USB-C para IA"** (um conector, muitos periféricos) — que é exatamente o argumento M×N do harness. *(anthropic.com retorna 403 pelo proxy; data e framing confirmados por VentureBeat.)*
- **Adoção como ponto de virada**: a OpenAI adotou o MCP em mar/2025 (Agents SDK (Software Development Kit), TechCrunch); o Google/Gemini seguiu (The New Stack); a Microsoft levou o MCP a GA no Copilot Studio e ao Windows. Decisão-chave: quando o segundo maior laboratório adota o protocolo do concorrente, MCP deixa de ser aposta de vendor e vira **infraestrutura neutra** — projetar para MCP reduz o risco de lock-in.
- **Autorização (OAuth 2.1)**: a spec trata todo servidor remoto como **OAuth 2.1 Resource Server** — valida tokens emitidos por um Authorization Server externo (RFC 9728 + 8414 + 7591). O guia prático da Descope traduz em decisão: separar *quem serve a tool* de *quem emite identidade* permite SSO corporativo e tokens com escopo por recurso, em vez de credenciais embutidas no servidor.
- **Segurança na prática** — o Tool Poisoning Attack da Invariant Labs (1/abr/2025) cunhou o termo: instruções maliciosas escondidas na *descrição* de uma tool que o usuário nunca lê mas o modelo obedece. O Trail of Bits mostrou o "line jumping": o simples **registro** de um servidor já é superfície de ataque, antes de qualquer invocação — o gate de confiança tem que ser no *conectar*, não no *chamar*. E the lethal trifecta (Simon Willison): dados privados + conteúdo não confiável + comunicação externa — o MCP torna fácil demais colar tools que, juntas, fecham as três pontas (ler e-mail + abrir PR público = exfiltração). A regra de design é impedir que as três coexistam no mesmo loop.
- **Governança (o MCP virou fronteira, não prótese)**: o registry oficial (preview, set/2025) é uma camada de API *community-owned* — o harness descobre servidores via API padronizada, não listas hardcoded. E em dez/2025 a Anthropic doou o MCP para a Agentic AI Foundation, sob a Linux Foundation — ao lado de **goose e AGENTS.md** como projetos fundadores. O protocolo agora evolui por consenso de um steering group, não pela roadmap de um vendor.
- **The 2026-07-28 Specification** (blog oficial do MCP): o anúncio da maior revisão do protocolo — núcleo stateless, MRTR, extensões, cache e política de depreciação (ver §6 do estado da arte).
- **Consulte também**: a coleção viva Awesome Harness Engineering — Skills & MCP reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. A padronização mais clara da disciplina

Onze harnesses da coorte, várias linguagens (TypeScript, Rust, Python), o mesmo protocolo, os SDKs oficiais. É o caso mais límpido de convergência que o livro registrou: onde design de tools, loop e compactação divergem, o MCP unificou. A única exceção na coorte é o **Aider** (MCP nota 0) — e é uma exceção *filosófica*, não um atraso: a escola *context-first* aposta em contexto curado e formatos de edição, e abre mão de MCP de propósito.

2. Transportes convergiram — stdio local, Streamable HTTP remoto

O default estabilizou: **stdio** para servidores locais (o harness lança o processo), **Streamable HTTP** para remotos. O **SSE** virou legado — presente só como fallback de compatibilidade (o opencode faz *fallback automático HTTP→SSE*). Harnesses mais rigorosos fixam a **revisão do protocolo** (o IronClaw fala explicitamente `2025-06-18`), sinal de que o protocolo tem versões e o cliente precisa negociá-las.

3. A virada: o harness virou também servidor MCP

Este é o *update datado* mais forte do capítulo — e uma **previsão do próprio livro que expirou**. Nas primeiras rodadas, anotamos que "nenhum dos harnesses atua como *servidor* MCP no core; o harness-como-serviço aparece por A2A/ACP". A rodada 2 refutou isso: **Codex, Hermes, OpenClaw, OpenHands e n8n expõem-se como servidores MCP**. O harness deixou de ser só um consumidor de tools e passou a ser uma **peça consumível por outros agentes** — o Codex se expõe a IDEs e outros hosts; o OpenClaw serve suas conversas de canal ao Claude Code/Codex; o n8n publica seu grafo de workflow como endpoint MCP. Com isso, a superfície do protocolo se alarga para além de *tools*: **sampling** (o servidor pede completions ao cliente — Hermes) e **elicitation** (o servidor pede input estruturado — Codex) entram no estado da arte. O harness-como-serviço, que antes só existia por A2A/ACP, agora tem uma via MCP nativa.

4. Autenticação: OAuth 2.1 é o piso; o enterprise sobe a régua

Para servidores remotos, o fluxo OAuth com PKCE + callback local + storage de tokens virou o mínimo (opencode, gemini-cli, Codex, Hermes, OpenClaw). O diferencial competitivo está acima: o **gemini-cli** adiciona provedores **Google auth** e **impersonation de service account** (MCP pensado para GCP corporativo); o **OpenClaw** guarda tokens em SQLite e suporta **mTLS**. Autenticação empresarial é hoje a fronteira de features de MCP.

5. Segurança: o servidor MCP é código de terceiros — e a coorte começou a tratá-lo assim

Se um servidor MCP injeta texto no contexto e pode ver argumentos de tools, ele é superfície de ataque — e a literatura (acima) mostra que a ameaça é medível e comum. As defesas observadas na coorte, em camadas (todas conectam ao cap. 07):

- **Testar o vetor**: o gemini-cli inclui um eval de **prompt injection via MCP** — o único que trata o servidor como atacante *testado*.
- **Filtrar o ambiente**: o OpenClaw, ao lançar um servidor stdio, **bloqueia variáveis de ambiente perigosas** (`NODE_OPTIONS`, `LD_*`, `DYLD_*`) que permitiriam carregar código no processo.

- **Mediar credenciais**: o IronClaw adapta tools MCP a *capabilities* **sem conceder autoridade ambiente** — FS, segredos e rede continuam mediados, e **o servidor nunca vê o secret** (a credencial é injetada pela borda). O OpenHands faz **redação e restauração de segredos** em round-trips de configuração.

A régua subiu de "conectar um servidor" para "conectar um servidor **contido**" — exatamente o que os papers de *tool poisoning* e *line jumping* pedem: gate no momento de conectar, sanitização entre servidores, e nenhuma confiança na autofiltragem do modelo.

6. A guinada stateless — a spec 2026-07-28

Três dias antes desta revisão, o protocolo passou pela sua **maior mudança desde o lançamento** (anúncio oficial; changelog). O núcleo virou **stateless**: caem o handshake `initialize`/`notifications/initialized` e o header `Mcp-Session-Id` — cada requisição viaja independente, com protocolo, identidade e capacidades em `_meta` (e um `server/discover` opcional para descoberta). A motivação é infraestrutural: servidores MCP passam a escalar com um load balancer round-robin comum, sem *sticky sessions*. As requisições iniciadas pelo servidor (`elicitation/create`, `sampling/createMessage`, `roots/list`) dão lugar ao **MRTR (Multi Round-Trip Requests)**: o servidor responde `resultType: "input_required"` e o cliente retenta com `inputResponses` — a bidirecionalidade vira ida-e-volta explícita. Completam o pacote: **framework formal de extensões** (Tasks vira `io.modelcontextprotocol/tasks`; MCP Apps e Enterprise Managed Authorization como extensões), **cache como contrato** (`ttlMs`/`cacheScope` nas respostas de listagem — ver cap. 04), roteamento por headers (`Mcp-Method`/`Mcp-Name`) e a **primeira política formal de depreciação** (janela mínima de 12 meses) — sob a qual **Sampling, Roots, Logging, o transporte legacy HTTP+SSE e o DCR (Dynamic Client Registration, substituído pelo CIMD — Client ID Metadata Documents)** ficam depreciados. Leitura editorial: o que as seções 1–5 descrevem continua sendo o protocolo *instalado* na coorte (a janela de 12 meses existe para isso), mas a direção mudou — e a adoção da 2026-07-28 pelos harnesses é o item nº 1 a medir na próxima rodada do benchmark.

> **LEITURA EXECUTIVA**
>
> O protocolo virou de página em 2026-07-28: núcleo **stateless** (sem handshake, sem `Mcp-Session-Id`), MRTR no lugar de sampling/elicitation iniciados pelo servidor, extensões formais, cache como contrato (`ttlMs`) e a primeira política de depreciação (12 meses) — sob a qual caem Sampling, Roots, Logging e o transporte HTTP+SSE. O que a coorte *roda hoje* ainda é o protocolo das seções 1–5 (a janela existe para isso); o que se *escreve hoje* já deve mirar a 2026-07-28. **O que roubar:** trate a descrição de tool como input não confiável (a literatura mede ~73% de sucesso de *tool poisoning*); em código novo, prefira Streamable HTTP stateless (o fallback SSE agora é transporte depreciado); se for expor um servidor MCP, filtre o ambiente do subprocesso e nunca deixe o servidor ver segredos; se seu público é enterprise, OAuth 2.1 com impersonation é o piso — e planeje a migração DCR → CIMD dentro da janela.

Mão na massa — harness-zero, etapa 7

A etapa 7 (`harness-zero/etapas/07-mcp/`) dá ao harness-zero um **adapter MCP client (stdio)** atrás de uma porta. Fiel à arquitetura hexagonal *por refatoração*: a `ToolPort` da etapa 2 já define o que é uma tool; agora um adapter descobre tools de um servidor MCP externo (via `stdio`, lançando o processo) e as apresenta ao loop como tools nativas — o modelo não distingue. Você conecta o servidor de exemplo incluído (`servidor_mcp_exemplo.py`) — e, como extensão, qualquer servidor MCP real de filesystem. Nota de época: o `ClienteMCP` da etapa implementa o handshake `initialize` do protocolo 2025-06 — que a spec 2026-07-28 **removeu** (núcleo stateless); ele segue funcionando na janela de depreciação de 12 meses, e a diferença entre as duas gerações é, em si, uma aula, lista suas tools, e as chama pelo mesmo caminho das tools locais. Exercício de completude: o cliente trata o *happy path*; você adiciona a **degradação graciosa** (um servidor que cai não derruba a sessão) e um **filtro de env** no subprocesso stdio — a defesa mínima do estado da arte.

Verificação

1. Por que o MCP reduz o custo de integração de M×N para M+N, e o que isso tem a ver com "padrão aberto"? (Cada harness e cada ferramenta escrevem uma vez; o protocolo comum desacopla os dois lados.)
2. Você vai conectar um servidor MCP de terceiros que expõe uma tool `search_tickets`. Cite dois motivos para desconfiar dele e duas defesas concretas. (Descrição de tool = *tool poisoning*; retorno = injeção indireta; e o *registro* já é vetor — *line jumping*. Defesas: env filtrado no subprocesso stdio; credencial mediada — servidor não vê o secret; eval de injeção; gate no conectar; contenção do cap. 07.)
3. Um harness que é **cliente E servidor** MCP ganha o quê que um cliente-só não tem — e que primitivas do protocolo isso ativa? (Vira peça consumível por outros agentes; ativa *sampling* e *elicitation*, o servidor pedindo ao cliente.)

Apêndice A — Como cada repositório trata o MCP

> Evidência por harness, com paths — complementação online, expandida a cada rodada.

opencode (rodada 1) — a implementação mais completa de protocolo

`packages/opencode/src/mcp/` (~1.000 linhas em `index.ts`, + `catalog.ts`, `oauth-provider.ts`, `auth.ts`). Três transportes — `StdioClientTransport`, `StreamableHTTPClientTransport` e `SSEClientTransport` com **fallback automático HTTP→SSE**. OAuth completo: autorização com callback server local, PKCE, comando dedicado `opencode mcp auth`. Cobre a superfície larga: notificações `ToolListChanged`, logging, roots, prompts, resources e resource templates. Instruções do servidor entram no system prompt (`system.ts:mcp()`) — o servidor pode ensinar o modelo a usá-lo (o que é também o vetor de injeção).

gemini-cli (rodada 1) — OAuth de nível corporativo

`packages/core/src/tools/mcp-client.ts` + `mcp-client-manager.ts`, os mesmos três transportes por config. O diferencial em `packages/core/src/mcp/`: além do OAuth padrão, provedores **Google auth** e **impersonation de service account** — MCP para GCP corporativo. Tools viram `DiscoveredMCPTool` com namespacing por servidor; prompts MCP expostos; gestão via `/mcp` e `~/.gemini/settings.json`. Notável: a suíte de evals inclui teste de **prompt injection via MCP** (cap. 11) — o único a tratar servidor MCP como superfície de ataque testada.

OpenHarness (rodada 1) — cliente pragmático

`src/openharness/mcp/` (`McpClientManager`) sobre o SDK `mcp>=1.0.0`: transportes **stdio** e **Streamable HTTP** (sem SSE), com status de conexão, auto-reconnect e **degradação graciosa** quando um servidor cai (`call_tool`/`read_resource` não derrubam a sessão). Resources expostos como tools próprias (`list_mcp_resources`, `read_mcp_resource`); `mcp_auth` para autenticação. Config via `oh mcp` e `--mcp-config`.

Goose (rodada 2) ⭐ MCP-nativo — o protocolo como espinha dorsal

O caso extremo: **toda tool é MCP**. Os built-ins de `goose-mcp` (memory, computercontroller, tutorial…) são servidores `rmcp::ServerHandler` reais servidos **in-process sobre `DuplexStream`** (stdio virtual) e podem rodar standalone (`goose mcp <server>`). Até developer/shell/edit são "platform extensions" falando `McpClientTrait`. Uma única abstração para toda a superfície de ferramentas — o protocolo não é integração, é a arquitetura. (O Goose é, também, um dos projetos fundadores da Agentic AI Foundation.)

Codex CLI (rodada 2) — cliente e servidor, quatro transportes

`rmcp-client/` + `mcp-server/` (o Codex se expõe como servidor MCP). **Quatro transportes** (stdio, streamable HTTP, in-process, process-executor); **OAuth completo** com refresh transactions e store locking; **elicitation**; prewarm/refresh de servidores; templates de aprovação por tool MCP. Integra o MCP à contenção (aprovação por tool).

Hermes (rodada 2) — cliente e servidor, com *sampling*

Cliente com stdio/StreamableHTTP/SSE, OAuth, timeouts por servidor, **sampling** (o servidor pode requisitar completions ao cliente) e paralelismo opt-in por servidor; `mcp_serve.py` expõe o Hermes a outros hosts MCP.

OpenClaw (rodada 2) — cliente e servidor, com filtro de env

`openclaw mcp serve` expõe conversas dos canais via stdio a Codex/Claude Code. Cliente: registry `mcp.servers` com stdio/SSE/streamable-http, **OAuth PKCE em SQLite**, **mTLS**, filtros de tools, probe/doctor — e **filtro de segurança de env** em stdio (bloqueia `NODE_OPTIONS`, `LD_*`, `DYLD_*`). Suporte a MCP Apps com sandbox de origem isolada.

OpenHands (rodada 2) — bidirecional, com redação de segredos

Client (config MCP por agente com redação/restauração de segredos em round-trips GET/PUT) e **server** (o app-server é um FastMCP expondo tools de PR — `create_pr`/`create_mr` — aos sandboxes, mais um **proxy MCP para Tavily** que dá busca sem expor a API key). Perfis de agente referenciam subconjuntos de servidores.

IronClaw (rodada 2) — MCP mediado por *capability*

`ironclaw_mcp` adapta tools MCP a **capabilities sem conceder autoridade ambiente**: FS, segredos e rede continuam mediados; **Streamable HTTP** (protocolo `2025-06-18`); **injeção de credencial mediada** (o servidor nunca vê o secret); recursos contabilizados pelo governor. O modelo de contenção do cap. 07 aplicado ao MCP.

ohmo (rodada 2) — herdado

Completo via `McpClientManager` (herdado da base); contagem de servidores no estado e resumo exposto ao gateway. Lacuna: sem config MCP própria (`~/.ohmo/mcp.json` não existe) e sem isolamento de MCP por canal/remetente.

n8n (rodada 2) — bidirecional no motor de workflow

MCP Client Tool (SSE + Streamable HTTP, Bearer/OAuth2, filtro de tools, cache de sessão por execução) e **MCP Server Trigger** (`McpTrigger` + `McpServer.ts`) — expõe as tools n8n conectadas como endpoint MCP a clientes externos. SDK oficial. O "harness invertido" também fala MCP nos dois papéis.

Aider (rodada 2) — ausente por filosofia

MCP nota **0**. A escola *context-first* em estado puro: as notas 3 estão onde a filosofia aposta (contexto, formatos de edição, git, evals), e a lacuna de MCP é escolha, não atraso.

Frameworks (rodada frameworks)

Agents SDK (OpenAI): suporte a servidores MCP como fonte de tools; LangGraph/langchain: adaptadores MCP para tools; CrewAI: integração MCP via toolkit; software-agent-sdk: anotações MCP-style no contrato de tools. O MCP é ponto de integração assumido também na camada de frameworks — reforço da tese de padronização.