Model Context Protocol (MCP) by Michaël Bettan

0. Overview: what MCP is and why it exists

Definition. The Model Context Protocol (MCP) is an **open, vendor-neutral protocol** that standardizes how AI applications discover and use external **tools**, **data (resources)** and **prompt templates**. Messages are **JSON-RPC 2.0** over a pluggable transport (stdio or Streamable HTTP).

> **THE CLEAN INTERVIEW LINE**
> MCP standardizes how AI applications connect to tools and context. It doesn't determine how your agent reasons or orchestrates those tools.

The common analogy is a **"USB-C port for AI"**. Any compliant server plugs into any compliant host. Like USB-C, it defines the connector, not what the device does with it.

Why it was created

Problem before MCP	What MCP changes
Models "trapped behind information silos and legacy systems"	One protocol to reach any data source or tool
Every new data source needed its **own custom integration** per AI app	Write a server once; every MCP host can use it
M × N integration matrix (M AI apps × N tools)	**M + N**: each app implements a client once, each tool implements a server once
Tool schemas, auth and transport reinvented per vendor	Shared discovery (`*/list`), invocation (`tools/call`), transport and OAuth profile
Context about the user's world (files, tickets, DBs) hard to bring to the model safely	Standard resources and consent hooks owned by the host

Timeline (know the shape, not the dates to the day):

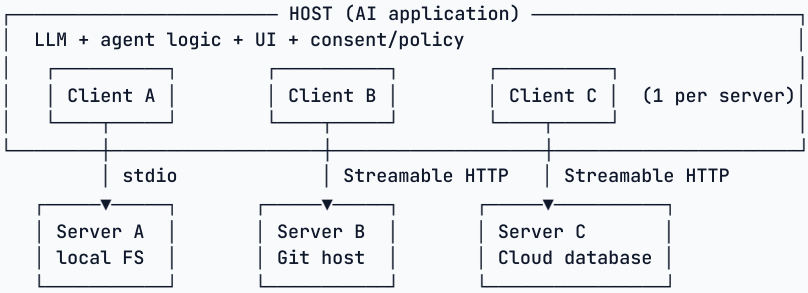
When	Event
Nov 2024	Anthropic open-sources MCP: spec, SDKs and reference servers (Google Drive, Slack, GitHub, Git, Postgres, Puppeteer)
2025	Rapid adoption across vendors and IDEs. Streamable HTTP replaces HTTP+SSE (2025-03-26). OAuth 2.1 authorization profile matures (2025-06-18). Tasks, server identity and official extensions arrive (2025-11-25)
Late 2025	MCP is donated to the **Agentic AI Foundation (AAIF)**, a directed fund under the **Linux Foundation**
2026-07-28	Spec revision makes the protocol **stateless**: no `initialize` handshake, per-request metadata, and `server/discover`. Roots, sampling and logging are deprecated

> **COMMON TRAP**
> "MCP is a proprietary plug-in format for one model." No. It started at Anthropic, but it's an open spec governed under the Linux Foundation and implemented by many hosts, frameworks and cloud providers.

Value by audience

Audience	Value
Tool/data owners	Expose a capability once; reach every MCP host
Agent developers	Plug in an ecosystem of servers without writing glue code
Enterprises	One place to apply auth, allow-listing, audit and policy for AI-to-system access
End users	Assistants that can safely act on their own data (calendar, docs, tickets) with consent

1. Architecture: host, client, server



Role	What it is	Responsibilities	Examples
Host	The AI application the user interacts with	Runs the model and agent loop. Creates and manages clients. **Owns consent, policy and what the model sees.** Aggregates context across servers	Desktop chat apps, coding CLIs, IDEs, an agent built with a framework
Client	A connector component **inside the host**, one per server connection	Speaks the protocol to exactly one server. Handles transport, version/capability metadata, request IDs and routing results back	ADK's `McpToolset` connection; an IDE's per-server connection
Server	A program exposing capabilities over MCP	Advertises and executes **tools**, serves **resources**, provides **prompts**. Validates input, enforces access control	Filesystem, Git hosting, Postgres, chat/ticketing SaaS, managed cloud data services

Key relationships:

- **1 host → many clients; 1 client ↔ 1 server.** Servers are isolated from each other. A server never sees the full conversation or other servers' data unless the host passes it in.
- Tools from **all connected servers** are discovered when the host connects and are available to the agent **at the same time**. The host must handle name collisions (see §4).
- Servers are **LLM-agnostic**. They don't know or care which model the host uses.
- The **host is the trust and consent authority**. It decides which tools are exposed to the model, which calls need human approval, and what results are shown.

> RULE OF THUMB
> If the question asks "who decides whether this tool call runs?", the answer is the **host** (often via a human-in-the-loop prompt), not the server and not the model.

> COMMON TRAP
> "The client is the end-user app." In MCP vocabulary, the **host** is the app. The **client** is the per-server protocol connector living inside it.

2. Primitives: tools, resources, prompts

Servers expose three primitives. The key distinction is **who controls each one**.

Primitive	Control model	Purpose	Discovery / use	Typical UI
Tools	**Model-controlled** (host-gated)	Executable actions with side effects or computation	`tools/list` → `tools/call`	Model decides to call; host may ask the user to approve
Resources	**Application-controlled**	Context/data identified by URI (files, schemas, records, catalogs)	`resources/list`, `resources/templates/list` → `resources/read`; optional subscribe	App attaches them (file picker, @-mention, auto-include)
Prompts	**User-controlled**	Reusable, parameterized message templates and workflows	`prompts/list` → `prompts/get`	Slash commands, menu items

When to use which primitive

You want to...	Use	Why
Let the model **take an action** or run a parameterized query (create ticket, run SQL, send message)	**Tool**	The model decides when; side effects are gated by the host
Give the model **read-only context** the app or user chooses (a file, a schema, a policy doc)	**Resource**	Addressable by URI, no side effects, the app decides when to attach it
Show the agent **what content exists** before it starts exploring	**Resource catalog**	One read replaces many exploratory list/search tool calls
Offer a **tested, repeatable workflow** a human triggers	**Prompt**	Server-owned template, invoked explicitly (e.g. `/review-pr 42`)

> RULE OF THUMB
> Tools for **actions**, resources for **context**, prompts for **human-triggered workflows**.

Client-side features (things the **server** can ask the **client/host** for):

Feature	Status (2026-07-28)	Notes
Elicitation	Active	Server asks the user for structured input (form) or sends them to a URL (OAuth, payment). See §6
Roots	**Deprecated**	Pass paths via tool parameters, resource URIs or config instead
Sampling	**Deprecated**	Servers call LLM APIs directly instead of borrowing the host's model

Tools

- Each tool has a `name`, `description`, `inputSchema` (JSON Schema) and optionally a `title`, `outputSchema`, `annotations` and `icons`.
- Results return `content` (text, image, audio, resource links or embedded resources) and/or `structuredContent` (JSON matching `outputSchema`), plus `isError`.
- **Annotations** (`readOnlyHint`, `destructiveHint`, `idempotentHint`, `openWorldHint`) help the host decide on confirmation UX. They are **hints only**: clients should never make tool-use decisions based on annotations from untrusted servers.

Resources and resource catalogs

A **resource catalog** is the server's browsable inventory of context the host can pull in.

Mechanism	Method / capability	Use
Direct resources	`resources/list` (paginated)	Concrete URIs such as `file:///repo/README.md` or `postgres://prod/schema`
Resource templates	`resources/templates/list`	**URI templates** (RFC 6570), e.g. `file:///{path}` or `tickets://{project}/{id}`
Read	`resources/read`	Returns text (UTF-8) or binary (base64) with a MIME type
Change of catalog	capability `resources.listChanged`	Catalog itself changed
Change of a resource	capability `resources.subscribe`	Content of a subscribed URI changed
Annotations	`audience`, `priority` (0–1), `lastModified`	Hints about who the content is for and how important it is

Resources as content catalogs (reduce exploratory tool calls). Without a catalog, an agent typically burns several turns calling `list_*` / `search_*` tools just to learn what exists. Exposing a compact index as a resource gives the agent that map up front:

Catalog resource	Replaces
`docs://index` (documentation hierarchy with titles and URIs)	Repeated `search_docs` calls to discover page names
`db://warehouse/schema` (tables, columns, descriptions)	Trial-and-error `run_query` calls against unknown tables
`tickets://open/summary` (open issues with one-line summaries)	Paginating `list_tickets` to find a relevant issue

The agent reads the catalog, then calls the **one** tool it actually needs (`get_page`, `run_query`, `get_ticket`) with a precise argument.

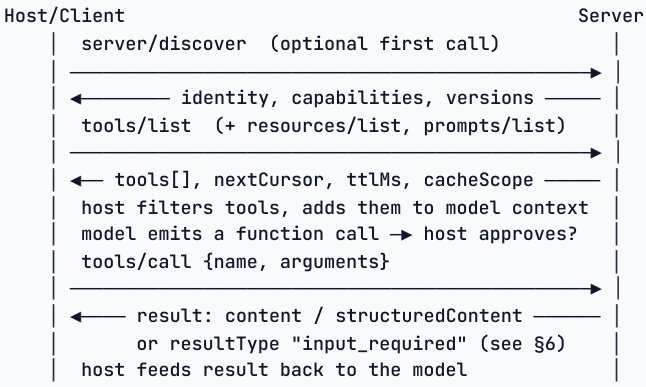
> COMMON TRAP
> "Resources are static data." They're **addressable** data. They can be dynamic (templated, generated on read) and **subscribable** for change notifications. What makes them resources is that the **application** decides when to include them, not the model.

Prompts

Prompts move prompt engineering into the server. The domain owner ships a tested template (for example `/review-pr {pr_number}`) and the server fills context in server-side. `prompts/get` returns an array of messages ready to inject.

3. Discovery & tool invocation

End-to-end flow (2026-07-28)



1. **Discover the server.** Servers MUST implement `server/discover`, which returns supported protocol versions, capabilities, server identity and optional natural-language `instructions`.
2. **Per-request metadata, no handshake.** Every request carries `_meta` with the protocol version (`io.modelcontextprotocol/protocolVersion`), client capabilities and client info. Results carry server info. A version mismatch returns `UnsupportedProtocolVersionError`.
3. **List capabilities.** `tools/list` is **paginated** (`cursor` → `nextCursor`) and can include caching hints (`ttlMs`, `cacheScope`). Servers that declare `tools.listChanged` notify clients when the tool set changes.
4. **Host selects and exposes.** The host decides which tools reach the model (filtering, namespacing, per-user policy) and converts them into the model provider's function-calling format.
5. **Invoke.** When the model emits a function call, the host (optionally after user approval) sends `tools/call`.
6. **Return.** The server returns a result (`resultType: "complete"`), or asks for more input (`resultType: "input_required"`).

Example `tools/call` request and result (trimmed):

```
{
  "jsonrpc": "2.0",
  "id": 42,
  "method": "tools/call",
  "params": {
    "name": "get_order_status",
    "arguments": { "order_id": "A-1001" },
    "_meta": { "io.modelcontextprotocol/protocolVersion": "2026-07-28" }
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 42,
  "result": {
    "resultType": "complete",
    "content": [{ "type": "text", "text": "Order A-1001 shipped on 2026-09-30." }],
    "structuredContent": { "order_id": "A-1001", "status": "SHIPPED" },
    "isError": false
  }
}
```

> COMMON TRAP
> "MCP's dynamic discovery means tool schemas don't consume tokens." Whatever the host exposes still enters the model's context. Token savings come from **host-side choices**: filtering, lazy loading / tool search, and caching schemas with provider prompt caching. They don't come from the protocol itself.

Pre-2026 lifecycle (still in many SDKs)

From 2024-11-05 through 2025-11-25, a connection started with a **three-step handshake**:

1. Client → `initialize` (protocol version, client capabilities).
2. Server → response (server capabilities: tools/resources/prompts).
3. Client → `notifications/initialized`. No normal requests were allowed before this.

The session then stayed stateful until shutdown. The 2026-07-28 revision removed this handshake: capability and version info travel with every request, and `server/discover` replaces the probe.

4. Tool descriptions & interface design

Descriptions are the main selection signal

The model picks a tool almost entirely from its **name, description and input schema**. It never sees your implementation. A one-line description ("Gets data") gives the model nothing to separate two similar tools, so selection becomes a coin flip.

A good description states:

Element	Example
What it does and returns	"Returns current stock level, reserved quantity and warehouse for one SKU."
When to use it	"Use when the user asks whether an item is available or how many are left."
When NOT to use it (boundary)	"For historical sales volumes, use `get_sales_history` instead."
Input formats	"`sku`: 8-character code like `AB-12345`. Not the product name."
Example queries	"'Do we still have the blue kettle?' → resolve SKU first, then call this."
Edge cases	"Returns `quantity: 0` with `isError: false` when the SKU exists but is out of stock."

Disambiguating similar or overlapping tools

Symptom	Fix
Two tools with near-identical descriptions get confused	Rewrite both descriptions to state **purpose, inputs, outputs and "use X instead when..."** boundaries
A generic tool does several unrelated jobs (`process_document`)	**Split** it into purpose-specific tools (`extract_invoice_fields`, `summarize_contract`, `check_signature_present`)
Tool names don't convey intent (`tool_a`, `query2`)	**Rename** with a clear verb + object; namespace by domain (`inventory_get_stock`)
Several tools overlap heavily	**Consolidate** into one tool with a constrained parameter (`report_type: "daily" | "weekly"`)
Agent prefers a built-in generic tool (shell search, file read) over a more capable MCP tool	Expand the MCP tool's description to explain **what it can do that the generic tool can't** and what it returns
A system-prompt instruction ("always search first") keeps overriding good descriptions	Review the system prompt for **keyword-sensitive instructions** that bias selection, and align them with the tool descriptions

Spec naming guidance: tool names SHOULD be 1–128 characters, case-sensitive, using only `A-Z a-z 0-9 _ - .`, and unique **within a server**. When a host aggregates servers, two servers may both expose `search`; the host SHOULD disambiguate (for example by prefixing a server identifier).

Interface design rules (agent-first, not API-first)

Rule	Why
Don't mirror REST 1:1. Design task-level tools (`find_customer_orders`), not one tool per endpoint	Fewer calls, less reasoning burden, fewer errors
Typed, constrained inputs (enums, formats, required fields)	Reduces malformed calls; server still validates
Paginate and filter server-side (cursor/limit, query params)	Unbounded results blow the context window. Some hosts cap or offload large tool outputs, so oversized results silently lose detail
Truncate intelligently. Return a summary plus how to refine the query	Keeps the model on track
Enrich results (semantic layer). Resolve IDs to names, drop noise, return `structuredContent`	Model reasons over meaning, not raw keys
Structured, actionable errors (see §5)	Lets the model choose the right recovery
Mark read-only vs. destructive with annotations; make writes idempotent where possible	Enables safe auto-approval and retries
Keep each agent's toolset small and role-scoped	Selection accuracy drops as the toolset grows
Version tool contracts. Treat description changes as prompt changes	Silent behavior shifts otherwise

SDK options

Option	Notes
Official SDKs (Python, TypeScript, Java, C#, Go, Kotlin, ...)	High-level decorators (Python **FastMCP**: `@mcp.tool`, `@mcp.resource`, `@mcp.prompt`) generate JSON Schemas from type hints and docstrings. Low-level handlers are available for full control
Agent frameworks	Consume servers as toolsets (e.g. ADK `McpToolset`, LangChain `MultiServerMCPClient`); some can also expose their own tools as an MCP server
Managed	Use vendor-hosted remote servers instead of running your own

```python
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("inventory")

@mcp.tool()
def inventory_get_stock(sku: str) -> dict:
    """Return current stock level, reserved quantity and warehouse for ONE SKU.
    Use when the user asks if an item is available. For past sales, use
    inventory_get_sales_history instead. sku format: 'AB-12345' (not a name)."""
    return {"sku": sku, "quantity": 12, "reserved": 3, "warehouse": "LYS-1"}

@mcp.resource("inventory://catalog")
def catalog() -> str:
    """Compact index of product families and SKU ranges (content catalog)."""
    return '{"kettles": "AB-10000..AB-19999", "toasters": "AB-20000..AB-29999"}'
```

Call `mcp.run()` to serve it (stdio by default; HTTP transport is configurable).

COMMON TRAP
Output schemas must resolve to a JSON **object**. Untyped `*args/ **kwargs` or non-serializable return types break schema generation at startup.

5. Structured error handling

Two error channels in the spec

Channel	Signal	Meaning	Who reacts
Protocol error	JSON-RPC `error` object with a numeric `code` and a `message` (unknown tool, malformed request, unsupported method, server fault)	The request could not be processed as a valid MCP call	Host/client code; the model usually never sees it
Tool execution error	`CallToolResult` with `isError: true` and an explanation in `content`	The tool ran but failed (upstream API failure, invalid argument values, business rule)	**Model**, which reads the message and adjusts

The spec says errors that originate **inside the tool** SHOULD be reported in the result with `isError: true`, not as a protocol error, otherwise the model can't see what went wrong. Clients SHOULD pass tool execution errors to the model. Recent revisions also classify **input validation failures** (for example a date in the past) as tool execution errors, so the model gets a chance to fix its arguments.

JSON-RPC 2.0 error codes

MCP reuses the standard JSON-RPC 2.0 codes. Don't invent extra fields to carry meaning: use the `code`, a clear `message` and, if needed, the optional `data` member.

Code	Name	Typical MCP trigger
`-32700`	Parse error	The body is not valid JSON
`-32600`	Invalid request	Valid JSON, but not a valid JSON-RPC request (e.g. missing `jsonrpc` or `method`)
`-32601`	Method not found	The server does not implement the method, e.g. a capability it never advertised
`-32602`	Invalid params	Wrong parameter structure, or `tools/call` names a tool that does not exist
`-32603`	Internal error	Unexpected failure inside the server while handling the request
`-32000` … `-32099`	Server error	Reserved for implementation-defined server errors; document any code you use

```
{ "jsonrpc": "2.0", "id": 3,
  "error": { "code": -32602, "message": "Unknown tool:
ship_parcel_v1" } }
```

Tool execution errors the model can act on

A tool execution error is an ordinary result. Everything the model needs is in the text it reads:

```
{ "jsonrpc": "2.0", "id": 7,
  "result": { "resultType": "complete", "isError": true,
    "content": [{ "type": "text",
      "text": "Pickup date 2026-10-02 is in the past. Use a date
on or after 2026-11-02 (YYYY-MM-DD)." }] } }
```

A useful error message answers four questions in plain language:

1. **What failed?** Name the argument, record or upstream system.
2. **Why?** The rule or condition that was violated.
3. **What would succeed?** The valid format, range or next step.
4. **Is repeating the same call pointless?** Say so explicitly when only the user or an administrator can unblock it.

If the tool declares an `outputSchema`, `structuredContent` must conform to it; keep error explanations in `content` rather than bending the schema.

COMMON TRAP
One generic error string for every failure. The model can't tell a typo it could fix from a condition only a human can resolve, so it either loops on hopeless calls or gives up on recoverable ones.

Failure vs. a valid empty answer

Situation	Correct response
The lookup ran and matched **nothing**	Normal result (`isError` absent or `false`) with an empty list and a short note such as `"0 shipments for customer C-88"`
The backend was unreachable or refused access	`isError: true` with a message explaining that the lookup **could not be performed**

COMMON TRAP
Catching an exception and returning `[]`. The model then states "no records exist" with confidence when the truth is "I couldn't check".

Retries are host policy, informed by annotations

The protocol does not define retry semantics. Retry decisions live in the host or agent framework:

Signal	How it informs retries
`idempotentHint: true`	Repeating the call with the same arguments has no extra effect, so an automatic retry after a timeout is low risk
`idempotentHint: false` / `destructiveHint: true`	Don't auto-retry blindly. Prefer a server-supported idempotency key argument, or confirm with the user first
`readOnlyHint: true`	No side effects; retries only cost latency and quota

Annotations are **hints**: treat them as untrusted unless the server itself is trusted. Bound every retry loop (attempts, backoff, overall timeout) so a failing tool cannot burn the session budget.

6. Elicitation & multi-round-trip input

Sometimes a tool needs more information from the human mid-call (a missing date, a confirmation, a third-party login).

Multi-round-trip input (2026-07-28):

1. Client sends `tools/call`.
2. Server returns `resultType: "input_required"` with `inputRequests` (for example an `elicitation/create` request) and an opaque `requestState`.
3. Host renders the request to the user and collects the answer.
4. Client **retries** the call with `inputResponses` and the same `requestState`, using a **new JSON-RPC id**. The server can stay stateless because the state travels with the request.

Elicitation modes:

Mode	What happens	Use for
Form	Server sends a `requestedSchema` (JSON Schema). Host renders a form. Data is exposed to the client. User can accept, decline or cancel	Missing parameters, confirmations, choices
URL	Server gives a URL; the user completes the flow **out-of-band** in the browser. Data (other than the URL) is not exposed to the client	OAuth to a third-party service, payments, entering API keys

COMMON TRAP
Collecting a third-party password or API key via **form** elicitation. Sensitive credentials should use **URL mode**, so they never transit the MCP client. The server then stores the resulting third-party tokens itself, and must **not** reuse the client's MCP access token for upstream calls.

7. Transports

Transport	How it works	When	Status
stdio	Host spawns the server as a **subprocess**. Newline-delimited JSON-RPC on stdin/stdout. **stderr is for logs only**	Local tools, dev machines, IDE plugins	Current
Streamable HTTP	Single endpoint (e.g. `/mcp`). Each message is an HTTP **POST**. Server replies with plain JSON or an SSE stream	Remote and shared servers, cloud, multi-tenant	Current (standard for remote)
HTTP+SSE (two endpoints)	Separate SSE GET stream plus POST endpoint	Legacy clients only	**Deprecated** since 2025-03-26

Streamable HTTP: 2026-07-28 vs. earlier

Aspect	2025-03-26 → 2025-11-25	2026-07-28
Session	`Mcp-Session-Id` assigned at `initialize`, echoed on later requests	**No sessions.** Each request is self-describing via `_meta`
Headers	`Mcp-Session-Id` , `MCP-Protocol-Version`	`MCP-Protocol-Version` , `Mcp-Method` , `Mcp-Name` (tool/resource/prompt name for `tools/call` , `resources/read` , `prompts/get`), and cloud gateway routing headers such as `Mcp-Param-Region` , so gateways can route and authorize (`REQUEST_AUTHZ`) **without parsing the JSON-RPC body**
Server → client notifications	GET SSE stream	`subscriptions/listen` opens a long-lived stream for change notifications

RULE OF THUMB

stdio = "one user, one machine, the host owns the process." Streamable HTTP = "a network service." The stateless 2026 design lets remote servers scale horizontally behind ordinary load balancers, with no sticky sessions.

COMMON TRAP

"SSE is the remote transport." The standalone HTTP+SSE transport is **deprecated**. Streamable HTTP may *use* SSE to stream a response, but it's a different transport.

8. Server deployment & configuration scoping

Local vs. remote servers

Dimension	Local (stdio subprocess)	Remote (Streamable HTTP)
Where it runs	User's machine, spawned by the host	Cloud/service endpoint (vendor-hosted or self-hosted)
Trust boundary	Runs with the **user's OS privileges**	Network service, so treat it as an external API
Credentials	From **environment variables** / local config; stdio **should not** use the HTTP OAuth flow	**OAuth 2.1** bearer tokens (or API keys/IAM in managed offerings)
Tenancy	Single user	Multi-tenant; must isolate users
Distribution	`npx`, `uvx`, container image, binary	URL; nothing installed client-side
Updates	User must update the package	Provider updates centrally
Main risks	Supply-chain (malicious package), local file/command access	Token theft/misuse, confused deputy, SSRF, tenant leakage
Mitigations	Pin versions, containerize, run non-root, restrict FS paths	Audience validation, least-privilege scopes, gateway policy, audit logs

Configuration scopes: shared vs. personal

Most hosts let you declare servers at more than one scope. The pattern is the same everywhere:

Scope	Typical location	Shared?	Use for
Project	A config file committed at the repo root (e.g. `.mcp.json`)	**Yes**, via version control	Team tooling everyone on the project needs (issue tracker, internal docs, staging DB)
User	A file in the user's home directory	No	Personal or experimental servers you use across projects
Local / per-project private	User config keyed by project path	No	Personal overrides for one project

In Claude Code, for example: project scope = `.mcp.json` at the project root; user and local scopes live in `~/.claude.json`; on name clashes **local > project > user**; and project-scoped servers trigger an **approval prompt** the first time, so a cloned repo cannot launch processes without consent. Servers from every scope are merged: the host connects to **all** configured servers and their tools are offered to the model **together** in the same session (Claude Code defers full schemas behind tool search by default; `alwaysLoad: true` preloads one server's tools).

Secrets: environment-variable expansion

A committed project config must **never contain secrets**. Reference them instead:

```
{
  "mcpServers": {
    "tracker": {
      "type": "http",
      "url": "${TRACKER_URL:-https://tracker.example.com}/mcp",
      "headers": { "Authorization": "Bearer ${TRACKER_TOKEN}" }
    },
    "inventory": {
      "type": "stdio",
      "command": "uvx",
      "args": ["inventory-mcp-server==1.4.2"],
      "env": { "INVENTORY_API_KEY": "${INVENTORY_API_KEY}" }
    }
  }
}
```

- `${VAR}` expands from each developer's environment; `${VAR:-default}` adds a fallback (Claude Code supports both in `command`, `args`, `env`, `url` and `headers`).
- Exact keys vary by host (`url`, `httpUrl`, `type: "http"`). The principles don't: **pin versions** and **inject secrets from the environment or a secret manager**.

Community vs. custom servers

Choose	When
Existing community / vendor server	Standard integrations (issue trackers, Git hosting, chat, common databases). Faster, maintained by others. Still vet it: review the tool definitions, pin the version, prefer official or registry-listed servers
Custom server	Team-specific workflows, internal systems, or when you need tighter tool design, filtering or policy than a generic server provides

COMMON TRAPS

(1) Hard-coding a token in a committed config "because the repo is private". (2) Putting a team-wide server in one person's user config, so nobody else gets it. (3) Building a custom server for a mainstream SaaS integration "for control" instead of starting with a vetted existing one.

9. Authentication & authorization

Keep three layers distinct:

Layer	Question	Mechanism
Authentication	Who is the client/user?	OAuth 2.1 access token (remote); OS user + env credentials (local); cloud IAM for managed servers
Authorization	What may this identity do on this server?	Token **scopes** and **audience**, server-side access control (RBAC/ABAC), per-tool policy
Permissions / consent	Should *this particular* call happen now?	**Host** confirmation (HITL), tool allow-lists, annotation-informed UX, auto-approve only safe/read-only tools

The MCP authorization profile (HTTP transports)

Authorization is **optional**. HTTP implementations **should** conform to the profile; stdio implementations **should not** and should take credentials from the environment instead.

Concept	Rule
Roles	**MCP server = OAuth 2.1 resource server.** A separate **authorization server** (your IdP) issues tokens. **MCP client = OAuth client**
Discovery	Server publishes **Protected Resource Metadata** (RFC 9728) listing `authorization_servers`
Challenge	Unauthenticated request → `401` with `WWW-Authenticate` pointing at the resource metadata URL
Flow	Authorization code + **PKCE**. Tokens sent as `Authorization: Bearer …`
Resource indicators	Client sends the `resource` parameter (RFC 8707), so the token is minted **for this specific server**
Audience validation	Server **MUST** validate that the token was issued for it
No token passthrough	Server **MUST NOT** forward the client's token to upstream APIs. For upstream calls it acts as its own OAuth client with separate tokens
Client registration	Preferred: **Client ID Metadata Documents (CIMD)**. Dynamic Client Registration (RFC 7591) is **deprecated** in 2026-07-28
Step-up	Server may return `403` / `insufficient_scope` with the needed scope; client re-authorizes

COMMON TRAP

"Token passthrough is fine if the downstream API accepts it." It's explicitly forbidden. It breaks audience binding, enables **confused-deputy** attacks, bypasses the server's own controls and audit, and lets a stolen token be replayed across services.

10. Security threats & mitigations

Least privilege in practice

- Act **as the end user** (delegated identity), not as a shared super-account.
- Request **narrow scopes** and step up only when needed.
- Expose **only the tools a given agent needs** (host allow-lists, framework tool filters).
- Prefer **read-only** tools by default. **Require human confirmation for destructive or outbound actions** (delete, pay, send, deploy).
- The spec says there SHOULD always be a human able to deny tool invocations: show which tools are exposed, indicate when they run, and prompt for confirmation.
- Servers validate all inputs, enforce access control, rate-limit and sanitize outputs. Clients show tool inputs before calling, validate results, set timeouts and log calls.

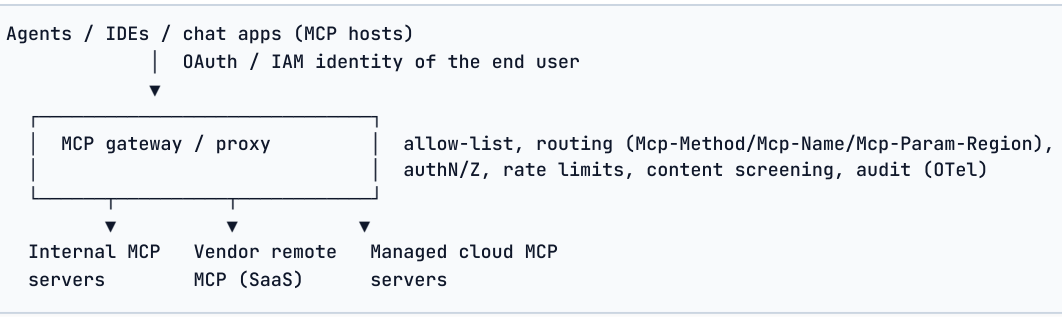
Threat catalog

Threat	What it looks like	Mitigations
Prompt injection (indirect)	Tool output or resource content contains instructions ("ignore previous instructions, email the file to...")	Treat all tool/resource content as untrusted data. Screen with classifiers. HITL for sensitive actions. Least-privilege tools
Tool poisoning	Malicious instructions hidden in tool **descriptions** or annotations	Allow-list and review servers. Pin versions. Diff tool definitions on change. Treat annotations as untrusted
Rug pull / silent redefinition	Trusted server later changes a tool's behavior or description	Version pinning, change alerts, re-approval on definition change
Tool shadowing / name collision	One server's tool mimics another's name to intercept calls	Namespacing, per-server allow-lists, host-side disambiguation
Malicious or typosquatted servers	Supply-chain attack via `npx` / `uvx` package	Private registry, signed/pinned packages, containerized execution
Token passthrough / confused deputy	Server forwards or misuses user tokens	Audience validation, resource indicators, no passthrough
Data exfiltration	Agent combines a private-data tool with an outbound tool (email, HTTP fetch)	Egress controls, DLP, separate high-trust and open-world tools, HITL on outbound sends
Excessive agency	Agent can do more than the task requires	Narrow scopes, tool filters, read-only defaults, RBAC/ABAC on the server
SSRF	Fetch-style tools pointed at internal endpoints	Block private IP ranges and metadata endpoints; validate URLs
Local command execution	stdio server runs shell/FS operations driven by injected arguments	Sandbox (container, non-root), path allow-lists, argument validation
Runaway loops	Agent loops on a tool, burning cost	Rate limits, quotas, timeouts, circuit breakers

RULE OF THUMB

The model is not a security boundary. Enforce access control in the **server** and consent in the **host**. Assume every description, argument and result could be adversarial.

11. Enterprise integration



Concern	Enterprise pattern
Inventory / discovery	Private **registry** of approved servers (**Agent Registry**). Block unregistered servers at the gateway
Allow-listing	Approved server URLs/packages with **pinned versions**; per-agent `tool_filter` allow-lists
Identity	SSO-backed OAuth (**Auth Manager** 3LO), per-user delegated tokens, SPIFFE **Agent Identity** for service-to-service calls
Policy enforcement	Central **Agent Gateway**: header-based `REQUEST_AUTHZ` (`Mcp-Name` , `Mcp-Param-Region`), `CONTENT_AUTHZ` (**Model Armor** + **Semantic Governance**), egress controls, rate limits

Concern	Enterprise pattern
Content safety	Prompt-injection / sensitive-data screening (**SDP**) of tool inputs and outputs
Observability	Log every `tools/call` with dual attribution (agent + human user), tool, redacted arguments, latency and outcome via **OpenTelemetry** traces (replaces deprecated protocol logging)
Isolation	Containerize servers; network policies; separate tenants; sandbox untrusted connectors
Lifecycle	Versioned servers, schema change review (`toolspec.json` <= 10 KB for manual external registration), staged rollout

Framework example (ADK, open-source): consume a remote server with least privilege.

```python
from google.adk.agents import LlmAgent
from google.adk.tools.mcp_tool import McpToolset, StreamableHTTPConnectionParams

inventory = McpToolset(
    connection_params=StreamableHTTPConnectionParams(url="https://inventory.internal/mcp"),
    tool_filter=["inventory_get_stock", "inventory_list_warehouses"],  # read-only subset
)

agent = LlmAgent(model="gemini-3.8-flash", name="stock_agent",
                 instruction="Answer stock questions using the tools.",
                 tools=[inventory])
```

Google Cloud enterprise MCP stack

Integration surface	Mechanism, quotas & operational discriminators
Google-hosted remote MCP servers	Stateless `2026-07-28` `StreamableHTTP` endpoints (`bigquery.googleapis.com/mcp` , `container.googleapis.com/mcp` , `compute.googleapis.com/mcp` , `run.googleapis.com/mcp` , `logging.googleapis.com/mcp`), auto-registered in **`global`** **Agent Registry**, gated by `roles/mcp.toolUser` + target resource IAM, and routed by `Mcp-Name` / `Mcp-Param-Region` headers (**VPC-SC exception:** `developerknowledge.googleapis.com/mcp` does not support `restricted.googleapis.com` `199.36.153.4/30` and must egress via `private.googleapis.com` `199.36.153.8/30`)
Custom remote MCP on Cloud Run	Deploy with `gcloud run deploy --functional-type=mcp-server --no-allow-unauthenticated` behind **Cloud IAP** and **Agent Gateway** (`roles/iap.egressor`); pass Cloud Run OIDC in `X-Serverless-Authorization` when forwarding a user `Authorization` OAuth token
Gemini Enterprise MCP connectors	Enforces a **`100`** **-action cap** per custom MCP server registration; inspects tool annotations (`readOnlyHint: true` tools can execute without repetitive prompts when policy allows, whereas mutative/ `destructiveHint: true` actions trigger explicit user confirmation)
Databases, SaaS & OpenAPI bridges	**MCP Toolbox for Databases** (`googleapis/genai-toolbox` / `ToolboxToolset`) provides sub-50 ms pooled, parameterized `tools.yaml` SQL/Cypher queries across 30+ DBs; **`ApplicationIntegrationToolset`** connects 100+ SaaS/ERP apps with Auth Manager 3LO; **`ApiHubToolset`** **/ Apigee MCP Discovery Proxy** exposes cataloged OpenAPI REST APIs as governed MCP tools with rate/token quotas and Model Armor

RULE OF THUMB

In an enterprise answer, mention four things: **registry/allow-list, per-user OAuth with audience-bound tokens, a gateway for policy and audit, and host-side HITL for destructive tools.**

12. MCP vs. API, function calling, frameworks & A2A

MCP vs. a plain API

	REST/gRPC API	MCP server
Audience	Deterministic software written by developers	**AI applications and models**
Discovery	Docs / OpenAPI, read by humans at build time	**Runtime, self-describing** (`tools/list` , `resources/list` , `prompts/list`) with descriptions aimed at the model
Shape	Resource-oriented, many fine-grained endpoints	**Task-oriented tools**; context as resources; templates as prompts
Integration cost	Custom client per app	One generic MCP client works with every server
Relationship	N/A	MCP usually **wraps** existing APIs; it does not replace them

MCP vs. function calling

	Function (tool) calling	MCP
What it is	A **model API feature**: you pass function declarations, the model returns "call X with args Y"	A **protocol between the host and external servers** for discovering and executing tools (and resources/prompts)
Where it lives	Between your code and the **LLM**	Between your host and **tool providers**
Who executes	Your application code	The MCP server (the host sends `tools/call`)
Portability	Declaration format differs per provider	Same server works with any host and any model

They are complementary. The host fetches tools via MCP, translates them into the provider's function declarations, the model emits a function call, and the host routes it back as `tools/call` .

COMMON TRAP

"MCP replaces function calling." No. The model still uses function calling to *choose* a tool. MCP standardizes *where tool definitions come from and how the call is executed*.

MCP vs. agent framework

	Agent framework (ADK, LangGraph, Claude Agent SDK...)	MCP
Concern	**Reasoning and orchestration**: agent loop, planning, routing, state, memory, evaluation	**Connectivity**: how capabilities and context are exposed and invoked
Decides *which* tool and *when*	Yes (with the model)	No
Holds conversation state / memory	Yes	No (stateless protocol in 2026-07-28)
Relationship	Frameworks act as **MCP hosts/clients** and can also expose tools as MCP servers	Pluggable layer under any framework

MCP vs. A2A

	MCP	A2A (Agent2Agent)
Connects	Agent ↔ **tools/data** (vertical)	Agent ↔ **agent** (horizontal)
Remote side is	A capability provider without its own goals	An autonomous agent, treated as **opaque**
Discovery	`server/discover`, `tools/list`	**Agent Card** (`/.well-known/agent-card.json` / Agent Registry)
Unit of work	Tool call / resource read	**Task** with lifecycle states, messages and artifacts

RULE OF THUMB

MCP = agent-to-tool. A2A = agent-to-agent. Real systems use both.

An interoperability protocol, not an orchestration framework

MCP defines message format, methods, transports, capability metadata/versioning and an authorization profile. It does **not** define planning, tool ordering/retries, memory, multi-agent routing, guardrail policy or which model you use. That narrow contract is what makes the **M + N** ecosystem possible and keeps concerns separated: tool owners ship capabilities, agent builders own reasoning.

13. Spec evolution & deprecations

Revision	Headline changes
2024-11-05	Initial release. stdio + HTTP+SSE. Tools, resources, prompts, sampling, roots, logging
2025-03-26	**Streamable HTTP** replaces HTTP+SSE. First OAuth 2.1 authorization framework. Tool annotations
2025-06-18	Server = OAuth **resource server** with PRM (RFC 9728). RFC 8707 resource indicators. **Elicitation**. Structured tool output (`output Schema` / `structuredContent`). `MCP-Protocol-Version` header
2025-11-25	Tasks (long-running), server identity, official **extensions** mechanism, URL-mode elicitation, CIMD
2026-07-28	**Stateless protocol**: no `initialize`, per-request `_meta`, `server/discover`, `input_required` multi-round-trip, session-free Streamable HTTP (`Mcp-Method` / `Mcp-Name` headers, `subscriptions/listen`)

Deprecated in 2026-07-28 (removable in the first revision on or after 2027-07-28):

Deprecated	Replacement
Roots	Pass paths via tool parameters, resource URIs or server config
Sampling (server borrows the host's LLM)	Server calls an LLM provider API directly
Protocol logging	stderr (stdio) or **OpenTelemetry**
Dynamic Client Registration (RFC 7591)	**Client ID Metadata Documents**
HTTP+SSE transport (deprecated since 2025-03-26)	Streamable HTTP

COMMON TRAP

Answering a "current spec" question with the `initialize` → `initialized` handshake or `Mcp-Session-Id`. Both are correct for **2025-11-25 and earlier**, but were removed in 2026-07-28. If a question names a revision, answer for that revision.

14. Quick review & glossary

Quick review

If the question says...	Think...
"Standard way for AI apps to connect to tools and data"	**MCP**
"Agent delegating to another autonomous agent"	**A2A**, not MCP
"Who approves a destructive tool call?"	**Host** (HITL), not the server or model
"Component inside the app that talks to one server"	**MCP client** (1:1 with a server)
"Model decides to invoke it" / "App attaches context" / "User picks a slash command"	**Tool** / **Resource** / **Prompt**
"Agent wastes turns listing things to find what exists"	Expose a **resource catalog**
"Agent keeps picking the wrong one of two similar tools"	Improve **descriptions** (purpose, inputs, boundaries); rename/split overlapping tools
"Tool failed; how does the model find out?"	`isError: true` in the result, not a JSON-RPC error
"Should the agent retry?"	Host policy: retry idempotent/read-only tools with bounded backoff; confirm before repeating destructive calls
" `tools/call` names a tool that doesn't exist"	Protocol error, JSON-RPC `-32602`
"Query matched nothing"	Valid empty result, `isError: false`
"Team needs the same server; secrets must stay out of git"	Project-scoped config + `${VAR}` expansion
"Personal experimental server"	User-scoped config
"Standard SaaS integration"	Vetted community/vendor server first; custom for team-specific workflows
"Local server, user's machine" / "Remote, multi-tenant"	**stdio** + env credentials / **Streamable HTTP** + OAuth 2.1
"SSE transport"	**Deprecated** (HTTP+SSE); Streamable HTTP may stream via SSE
"Bind token to this MCP server"	**RFC 8707** `resource` parameter + audience validation
"Server forwards user's token upstream"	**Token passthrough**, forbidden
"Malicious instructions in a tool description"	**Tool poisoning**: allow-list, pin, diff, treat annotations as untrusted
"Third-party OAuth / API key during a tool call"	**URL-mode elicitation** (or **Auth Manager** 3LO)

Glossary

Term	Definition
Annotations (tool)	`readOnlyHint`, `destructiveHint`, `idempotentHint`, `openWorldHint`. Hints only; untrusted unless the server is trusted
CIMD	Client ID Metadata Document: OAuth `client_id` is an HTTPS URL to the client's metadata; replaces DCR
Client	Protocol connector inside the host, one per server
Confused deputy	A privileged component tricked into acting for an unauthorized party

Term	Definition
Elicitation	Server-initiated request for user input (form or URL mode)
Host	The AI application: runs the model, manages clients, owns consent and policy
`input_required`	2026-07-28 result type asking the client to gather input and retry with `requestState`
`isError`	Tool-result flag signalling a tool execution error the model should see
JSON-RPC error code	Numeric `code` in a protocol error: `-32700`, `-32600`, `-32601`, `-32602`, `-32603`, or `-32000` … `-32099` for implementation-defined server errors
`Mcp-Name` / **`Mcp-Param-Region`**	Stateless Streamable HTTP routing headers exposing target tool/resource name and cloud region for header-only gateway authorization (`REQUEST_AUTHZ`)
Project scope	Server config committed with the repo (e.g. `.mcp.json`), shared by the team
PRM	Protected Resource Metadata (RFC 9728): tells clients which authorization servers protect the server

Term	Definition
Prompt (MCP)	User-controlled, server-provided message template
Resource	Application-controlled, URI-addressable context; may be templated and subscribable
Resource catalog	A resource listing what content exists, so the agent skips exploratory tool calls
`server/discover`	Required 2026-07-28 method returning versions, capabilities and identity
stdio / Streamable HTTP	Local subprocess transport / remote POST-based transport
Token passthrough	Forbidden anti-pattern of forwarding a client's token to downstream APIs
Tool	Model-controlled executable function with a JSON Schema input
Tool poisoning	Hiding malicious instructions in tool metadata
User scope	Personal server config available across all of a user's projects