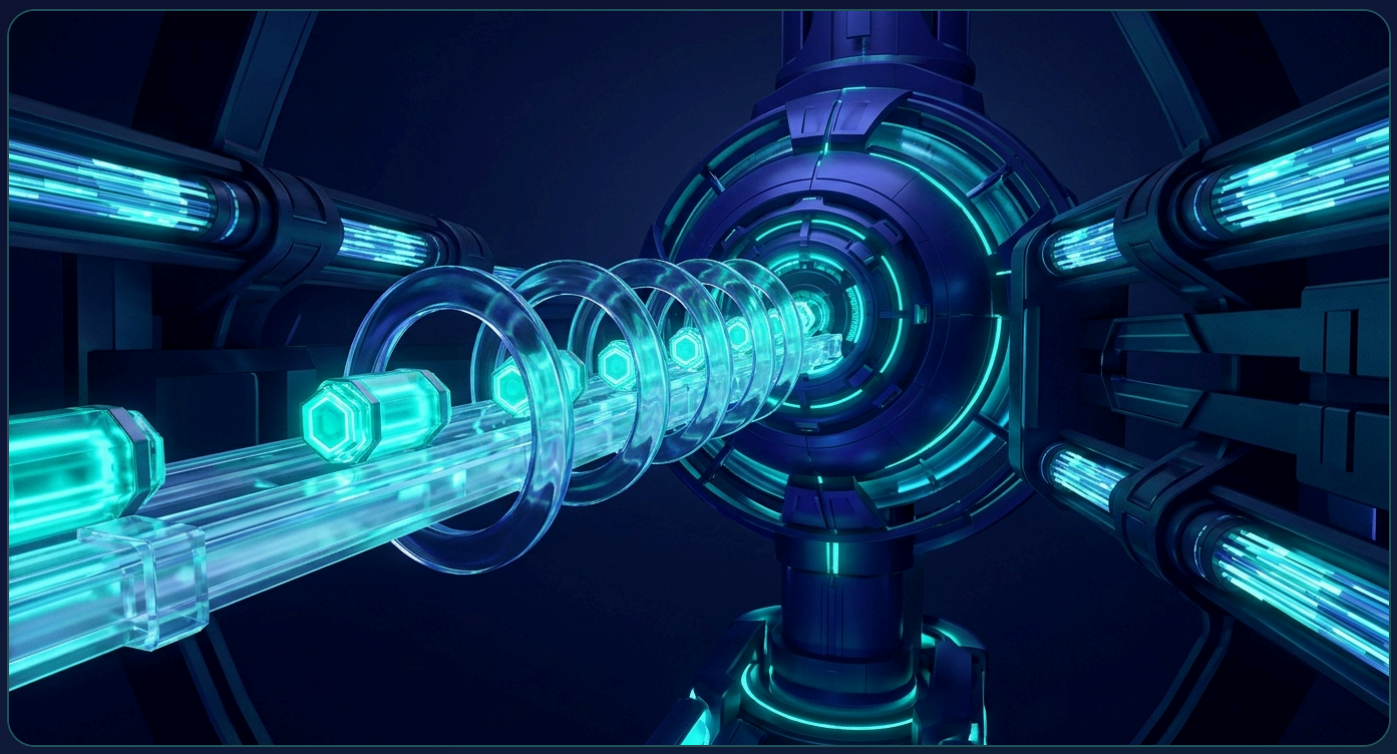
AI AGENT ENGINEERING & ARCHITECTURE • FIELD GUIDE

Building Production MCP Servers

The complete 12-page visual handbook to designing, validating, securing, and evaluating Model Context Protocol servers using TypeScript SDK v2 and Python FastMCP.

TypeScript SDK v2 | Python FastMCP | Streamable HTTP & stdio | Zod v4 & outputSchema | OAuth 2.1 & Tool Hints | MCP Inspector & Evals



SDK v2 & FastMCP

Build schema-strict tools in TypeScript `@modelcontextprotocol/server` v2 or Python `FastMCP` with Pydantic v2 validation.

Stateless Factories

Spawn a fresh `McpServer` per HTTP request while pooling upstream clients at module scope for effortless horizontal scale.

Evaluated Reliability

Verify contracts with `scripts/smoke.ts`, MCP Inspector, and a 10-question multi-hop evaluation harness before shipping.

Agent-Centric Design vs. Raw REST Wrappers

An MCP server is not a thin proxy over HTTP endpoints—it is an **ergonomic user interface for a language model**. Great servers pack high-signal data into tight context budgets while keeping destructive actions crystal clear.

Quick Navigation

1. Agent-Centric Design & Primitives p. 2
2. SDK v2 Setup & Transport Matrix p. 3
3. MCP Inspector & Trace UI Tour p. 4
4. Stateless Factory & Dual Transports p. 5
5. Tool Schemas & structuredContent p. 6
6. Troubleshooting Clinic & Checklist p. 12
7. Python FastMCP & Pydantic v2 p. 7
8. Errors, OAuth 2.1 & Security p. 8
9. Five Copy-Paste Production Recipes p. 9
10. Packaging & Multi-Host Rollout p. 10
11. 10-Question Eval Pipeline p. 11

T Tools (Model-Controlled)

Executable functions the LLM chooses to invoke—from read-only searches to side-effecting writes—guided by JSON schemas and annotations.

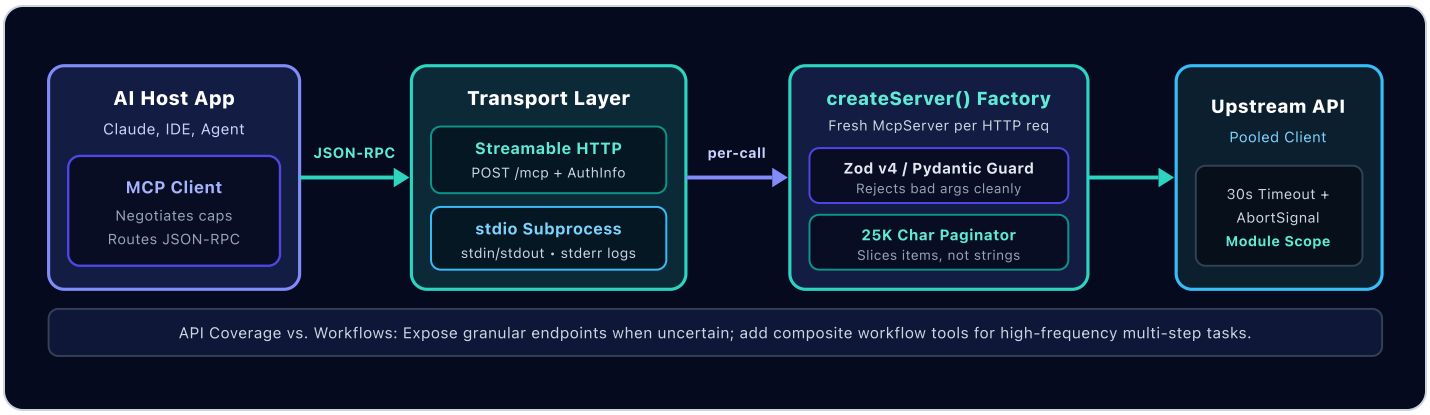
R Resources (App-Controlled)

Read-only data addressed by URI or `ResourceTemplate` (e.g., `example://teams`) that host apps fetch and attach as context.

P Prompts (User-Controlled)

Named message templates with `completable()` argument autocomplete that users trigger explicitly via slash commands or menus.

Architecture: Host → Client → Transport → Stateless Factory → Upstream



Context Efficiency Rule: Respect CHARACTER_LIMIT = 25_000

Never dump raw 100KB REST payloads into a model's context window. Cap every tool response around `25_000` characters by reducing returned items (e.g., halving the array and setting `next_offset`), defaulting to concise Markdown, and offering `response_format="json"` for programmatic chains.

SDK v2 Setup, v1-to-v2 Migration & Transports

Always start new TypeScript servers on **SDK v2** (`@modelcontextprotocol/server`) with `zod ^4.2.0` and Node `>=20.3`, or build in Python with **FastMCP** and Pydantic v2. Never mix v1 and v2 imports inside one repository.

TypeScript SDK v2 Install & Imports

```
# Core v2 server + Zod 4.2+ (Node >=20.3)
npm install @modelcontextprotocol/server zod@^4.2.0
npm install @modelcontextprotocol/express
@modelcontextprotocol/node express
npm install -D @modelcontextprotocol/client typescript
tsx

import { McpServer, createMcpHandler } from
"@modelcontextprotocol/server";
import { serveStdio } from
"@modelcontextprotocol/server/stdio";
import * as z from "zod/v4";
```

Migrating Legacy `@modelcontextprotocol/sdk` v1 Projects

Legacy v1 receives bug and security fixes only. Upgrade cleanly with the official codemod, resolve any flagged call sites, and verify types:
`npx @modelcontextprotocol/codemod@latest v1-to-v2 .`
Key rewrites: `server.tool` → `registerTool`, `StdioServerTransport` → `serveStdio`, `McpError` → `ProtocolError`, `extra.signal` → `ctx.mcpReq.signal`.

Python FastMCP + Pydantic v2 Stack

```
# Python 3.10+ with official mcp SDK & httpx
pip install mcp pydantic httpx

from mcp.server.fastmcp import FastMCP, Context
from pydantic import BaseModel, Field, ConfigDict,
field_validator
import httpx

mcp = FastMCP("github_mcp") # {service}_mcp convention
```

Naming & Schema Discipline

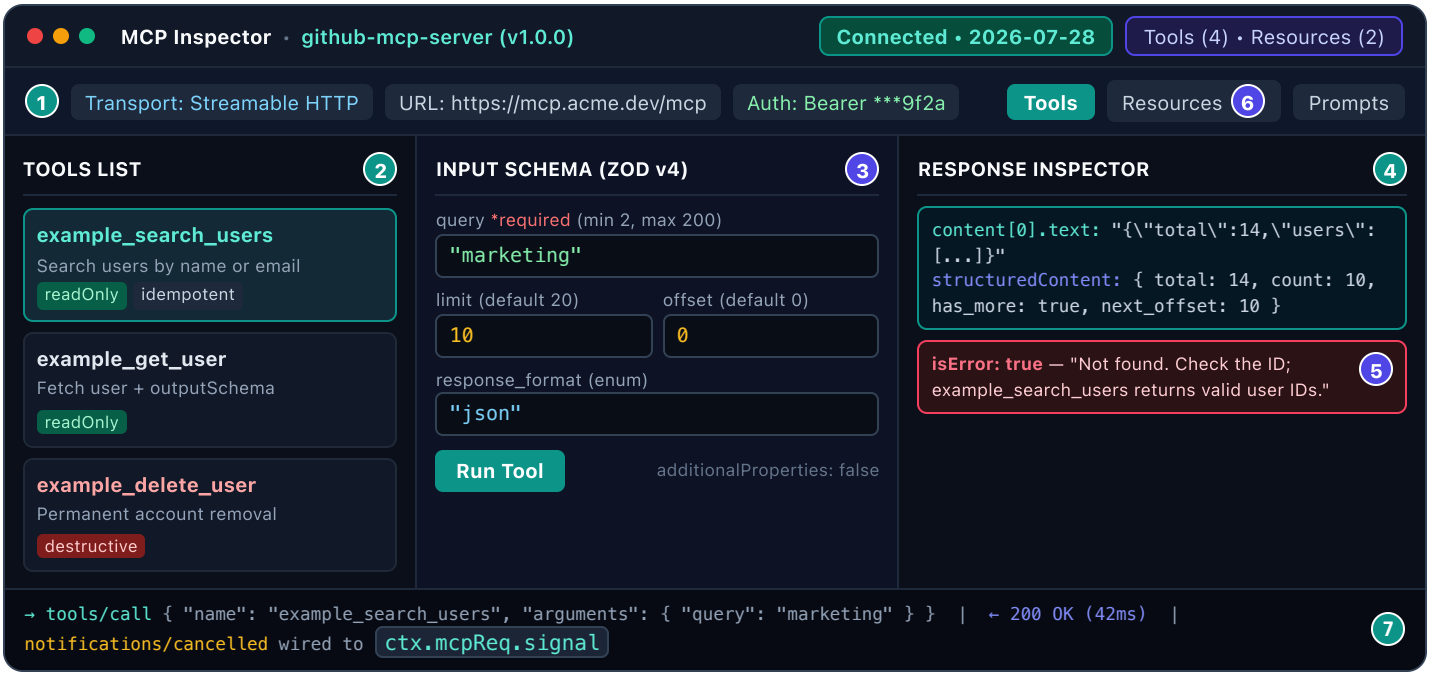
- **Node/TS package:** `{service}-mcp-server` (lowercase hyphens, no versions).
- **Python server:** `{service}_mcp` (lowercase underscores).
- **Zod version floor:** Keep `zod` at `^4.2.0`; versions before `4.1.13` silently drop `.describe()` text from advertised JSON schemas!

Transport Decision Matrix: Pick the Right Wire Protocol

Transport Mode	Best Use Case	SDK v2 / FastMCP Wiring	Operational Tradeoffs
`stdio` Local Default	CLI tools, desktop apps, and local IDEs spawning the server as a child process.	`serveStdio(createServer)` in TS v2; `mcp.run()` in FastMCP.	Zero network config; single client per process. **Never write to stdout**—log strictly to `console.error` / `stderr`.
Stateless Streamable HTTP Remote Recommended	Cloud APIs, multi-user deployments, containers, serverless, and load-balanced pods.	`createMcpHandler(createServer)` + `createMcpExpressApp()` + `toNodeHandler()`.	Zero session affinity required; scales horizontally out of the box; supports 2025-era and 2026-07-28 clients alike.
Stateful HTTP Sessions	Long-lived subscriptions or server-initiated requests needing sticky in-memory session state.	Session store + `StreamableHTTPServerTransport` or Hono/Fastify session adapters.	Requires sticky routing or external Redis state; avoid unless tools genuinely need multi-turn server memory.
Legacy SSE Deprecated	Only for maintaining compatibility with un-upgraded 2024 MCP clients.	Separate `GET /sse` + `POST /messages` legacy endpoints.	Two-endpoint complexity, proxy buffering issues, and deprecated in the modern MCP spec. Migrate to Streamable HTTP.

Annotated Tour: MCP Inspector & JSON-RPC Trace

Launch `npx @modelcontextprotocol/inspector node dist/index.js` to test every schema, behavioral hint, error banner, and live JSON-RPC frame before connecting an agent host.

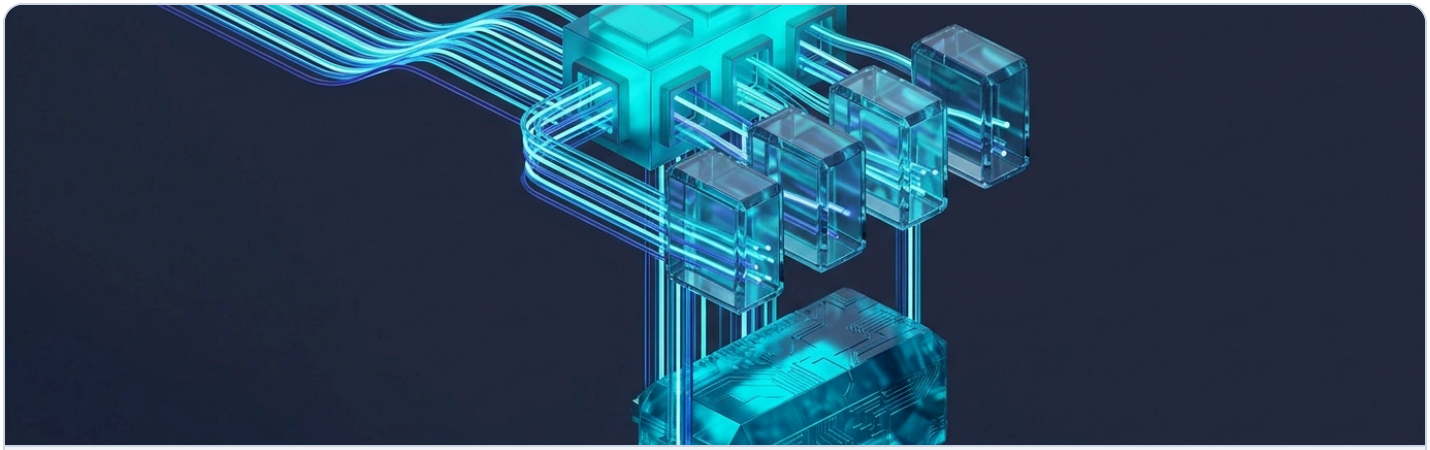


Inspector Control Legend: What to Verify on Every Build

1. **Transport & Bearer Auth Bar:** Switch between `stdio` subprocess launch and remote Streamable HTTP with custom `Authorization: Bearer` headers.
2. **Tool List & Hint Badges:** Confirm snake_case service prefixes and accurate `readOnlyHint` / `destructiveHint` badges appear immediately.
3. **Zod v4 / Pydantic Form:** Verify every parameter displays rich `.describe()` help text, defaults, min/max bounds, and strict rejection of unknown keys.
4. **Dual Response Inspector:** Ensure `structuredContent` matches your `outputSchema` and mirrors the serialized JSON in `content[0].text`.
5. **`isError: true` Recovery Banner:** Trigger a bad ID or 429 deliberately and read the message as an agent would—it must name the exact next tool or fix.
6. **Resources & URI Templates Tab:** Test static URIs and `ResourceTemplate` placeholders, ensuring missing IDs raise `ResourceNotFoundError`.
7. **JSON-RPC Frame & Cancellation Log:** Confirm zero stray stdout noise on `stdio` and verify client cancellation aborts upstream HTTP calls cleanly.

The Stateless `createServer()` Factory & Dual Transports

The single most important architectural rule in TypeScript SDK v2 is isolating per-request protocol state inside a cheap `createServer()` factory while keeping heavy upstream clients pooled at module scope.



Stateless Concurrency Model: Each incoming HTTP request spawns a lightweight, isolated `McpServer` pod via `createServer()` while closing over a single shared upstream API client, connection pool, and token verifier at module scope.

Why Shared Singletons Crash Under Load

> **The Concurrency Trap: Error `-32603`**
> If you instantiate `const server = new McpServer(...)` once at module scope and return it from `createServer()`, your smoke test passes because it sends requests sequentially. In production, the moment two HTTP calls overlap, the second request fails with `-32603 Internal server error`!

- **Per-connection vs. per-request:** `serveStdio(createServer)` invokes your factory once for the subprocess; `createMcpHandler(createServer)` invokes it fresh on **every HTTP request**.
- **Keep factories fast:** Build HTTP agents, Redis pools, and config once at module scope, then close over them inside `createServer()`.
- **Host/Origin safety:** `createMcpExpressApp()` binds `127.0.0.1` and blocks DNS rebinding. On `0.0.0.0`, pass `allowedHosts: ["mcp.example.com"]`.

One File Serving Both Transports

```
// Shared client lives at module scope; factory stays cheap
function createServer(): McpServer {
  const server = new McpServer({ name: "example-mcp-server", version: "1.0.0" });
  registerUserTools(server);
  return server;
}
if (process.env.TRANSPORT === "http") {
  const handler = createMcpHandler(createServer);
  const app = createMcpExpressApp();
  const nodeHandler = toNodeHandler(handler);
  app.all("/mcp", (req, res) => void nodeHandler(req, res, req.body));
  app.listen(Number(process.env.PORT ?? 3000), "127.0.0.1");
} else {
  serveStdio(createServer);
  console.error("example-mcp-server running on stdio");
// stderr only!
}
```

Tool Registration Anatomy: Schemas & Hints

Every tool registration defines a machine-verifiable contract: strict Zod v4 input validation, behavioral hints for host approval UIs, cancellation-aware upstream calls, and item-halving pagination under `CHARACTER_LIMIT`.

Color-Coded `server.registerTool(...)` Anatomy

1. **Service-Prefixed Name:** `"github_search_issues"` (snake_case, verb-first, zero ambiguity across multi-server hosts) → 2. **Description & Examples:** State purpose, return shape, sibling tool to use instead, and 2 concrete example queries (JSDoc is ignored!) → 3. **z.strictObject Input:** Rejects unknown keys; every field uses `.describe()` with examples → 4. **Annotations:** `readOnlyHint`, `destructiveHint`, `idempotentHint`, `openWorldHint` → 5. **Cancellation Signal:** `AbortSignal.any([ctx.mcpReq.signal, AbortSignal.timeout(30_000)])` → 6. **Dual Output:** Return `structuredContent` validated by `outputSchema` plus serialized JSON or Markdown text.

```
const SearchInput = z.strictObject({
  query: z.string().min(2).max(200).describe("Match against names/emails, e.g. 'marketing'"),
  limit: z.number().int().min(1).max(100).default(20).describe("Max users to return"),
  offset: z.number().int().min(0).default(0).describe("Pagination offset"),
  response_format: z.enum(ResponseFormat).default(ResponseFormat.MARKDOWN)
});
server.registerTool("example_search_users", {
  title: "Search Users",
  description: "Search users by name or email. Use example_get_user if you already know the ID.",
  inputSchema: SearchInput,
  annotations: { readOnlyHint: true, destructiveHint: false, idempotentHint: true, openWorldHint: true }
}, async ({ query, limit, offset, response_format }, ctx) => {
  const raw = await apiRequest("users/search", { q: query, limit, offset }, ctx.mcpReq.signal);
  const data = UserPageSchema.parse(raw); // Strips unadvertised upstream fields
  return paginateUnderCharLimit(data, offset, response_format, 25_000);
});
```

Behavioral Annotations Matrix

Hint	Default	When to Set `true`
`readOnlyHint`	`false`	Never mutates state (safe for auto-approve).
`destructiveHint`	`true`	Deletes, overwrites, or triggers irreversible changes.
`idempotentHint`	`false`	Repeating identical args produces no extra effect.
`openWorldHint`	`true`	Talks to external APIs, web, or third-party systems.

`outputSchema` & Item-Halving Rules

- **Always-JSON tools only:** Declare `outputSchema` only when a tool always returns structured JSON (omit on dual Markdown/JSON tools).
- **Mirror text & object:** When returning `structuredContent`, serialize the same object as JSON in `content[0].text` for backward-compatible clients.
- **Zod 4 API shifts:** Replace v3 `.strict()` with `z.strictObject({...})`, `z.nativeEnum(E)` with `z.enum(E)`, and `z.string().email()` with `z.email()`.
- **Slice arrays, never strings:** Halve items in a `while` loop until `text.length <= 25_000` so `next_offset` aligns cleanly.

Python `FastMCP` & Pydantic v2 Production Patterns

In Python, `mcp.server.fastmcp` turns typed Pydantic v2 models and docstrings directly into JSON Schemas. Pair strict `ConfigDict` settings with a shared `httpx.AsyncClient` and async lifespan hooks.

Pydantic v2 Strict Input & Validators

```python
class ResponseFormat(str, Enum):
    MARKDOWN = "markdown"; JSON = "json"

class UserSearchInput(BaseModel):
    model_config = ConfigDict(
        str_strip_whitespace=True,
validate_assignment=True, extra="forbid"
    )
    query: str = Field(..., min_length=2,
max_length=200, description="Query e.g. 'team:mkt'")
    limit: int = Field(default=20, ge=1, le=100)
    response_format: ResponseFormat =
ResponseFormat.MARKDOWN

    @field_validator("query")
    @classmethod
    def non_blank(cls, v: str) -> str:
        if not v.strip(): raise ValueError("Query cannot
be empty")
        return v
```

`@mcp.tool` & Shared `httpx.AsyncClient`

```python
mcp = FastMCP("example_mcp")
_http = httpx.AsyncClient(base_url=API_BASE_URL,
timeout=30.0)

@mcp.tool(name="example_search_users", annotations={
    "title": "Search Example Users", "readOnlyHint":
True,
    "destructiveHint": False, "openWorldHint": True
})
async def example_search_users(params: UserSearchInput,
ctx: Context) -> str:
    """Search users by name, email, or team."""
    await ctx.report_progress(0.2, "Querying upstream
directory")
    try:
        resp = await _http.get("/users/search",
params=params.model_dump())
        resp.raise_for_status()
        return format_users(resp.json(),
params.response_format)
    except (httpx.HTTPStatusError,
httpx.TimeoutException) as exc:
        return handle_api_error(exc)
```

Resources, Prompts & Lifespan State

```python
@asynccontextmanager
async def app_lifespan():
    pool = await create_db_pool()
    yield {"db": pool}
    await pool.close()

@mcp.resource("config://settings/{key}")
async def get_setting(key: str) -> str:
    return json.dumps(load_safe_setting(key))

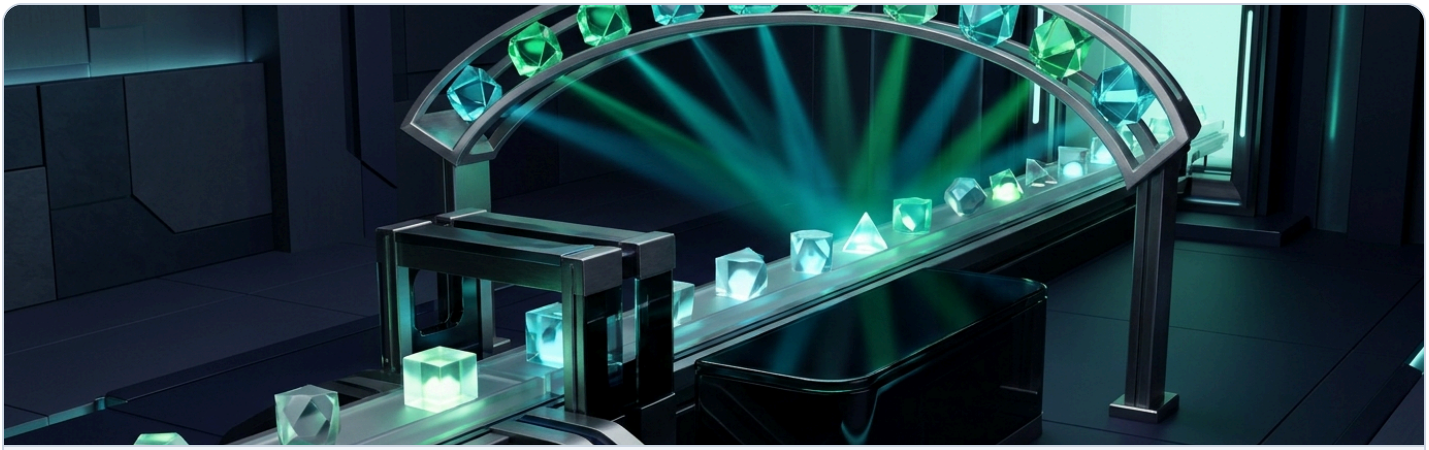
@mcp.prompt()
def team_audit_prompt(team: str) -> str:
    return "Audit access for team " + team + " using
example_search_users."
```

Python Production Rules

- **Never block the event loop:** Always use `httpx.AsyncClient` inside `async def` tools—never call synchronous `requests.get()`.
- **Use Pydantic v2 idioms:** Use `model_config = ConfigDict(...)`, `@field_validator` with `@classmethod`, and `model_dump()` (never deprecated v1 `Config` or `.dict()`).
- **Markdown vs. JSON formatting:** Format Markdown with human-readable UTC timestamps and `Name (ID)` headers; strip bulky avatar arrays.
- **Context superpowers:** Inject `ctx: Context` for `ctx.report_progress()`, `ctx.log_info()`, `ctx.elicit()`, and lifespan connections.
- **Dual run modes:** Default to `mcp.run()` for local `stdio`, or pass `mcp.run(transport="streamable_http", port=8000)` for HTTP.

Error Channels, OAuth 2.1 Auth & Hardening

Remote MCP servers must separate model-facing recovery hints from client-level JSON-RPC errors, enforce resource-scoped OAuth 2.1 tokens, and treat every upstream payload and path parameter as untrusted.



Defense-in-Depth Validation Gate: Every inbound call clears Host/Origin checks, OAuth 2.1 token audience verification, strict schema validation, and path/SSRF allowlists before touching downstream services.

Two Distinct MCP Error Channels

- **Tool Errors (`isError: true`):** Seen directly by the LLM! Return `{ isError: true, content: [...] }` or throw a normal `Error` with an actionable next step (e.g., *"Rate limit exceeded. Wait 60s."*).
- **Protocol Errors (`ProtocolError`):** JSON-RPC errors handled by the host application; the model never sees them. Throw `ProtocolError(ProtocolErrorCode.InvalidParams, ...)` or `ResourceNotFoundError(uri.href)` inside resource and prompt handlers.

Hardening Checklist

- **Path traversal:** Resolve paths with `realpath()` and assert `resolved.startsWith(ROOT + path.sep)`.
- **SSRF & URL safety:** Allowlist schemes (`https:`) and block RFC1918 / link-local metadata IPs (`169.254.169.254`).
- **Upstream sanitization:** Validate API JSON via `Schema.parse(raw)` and strip raw HTML/script payloads.

OAuth 2.1 Bearer Auth & Per-Caller Context

```
const serverUrl = new
URL("https://mcp.example.com/mcp");
const verifier: OAuthTokenVerifier = {
  async verifyAccessToken(token): Promise<AuthInfo> {
    const claims = await verifyJwt(token);
    if (!claims) throw new
OAuthError(OAuthErrorCode.InvalidToken, "Bad token");
    return {
      token, clientId: claims.clientId, scopes:
claims.scopes,
      expiresAt: claims.exp, // Required: tokens without
exp are rejected
      resource: new URL(claims.aud) // Must match
expectedResource
    };
  }
};
const app = createMcpExpressApp({ host: "0.0.0.0",
allowedHosts: ["mcp.example.com"] });
const auth = requireBearerAuth({ verifier,
requiredScopes: ["mcp"], expectedResource: serverUrl });
const handler = createMcpHandler(({ authInfo }) =>
createServer(authInfo));
app.all("/mcp", auth, (req, res) => void
toNodeHandler(handler)(req, res, req.body));
```

Inside tools, inspect `ctx.http?.authInfo` (undefined on `stdio`) to enforce tenant isolation and per-tool scopes.

5 Real-World Copy-Paste MCP Recipes

Drop these five production primitives directly into your TypeScript v2 or Python FastMCP server to eliminate common protocol bugs, context overflows, and silent startup failures.

Pattern & Goal	Copy-Paste TypeScript v2 / FastMCP Snippet	Production Guarantee

1. Dual-Transport Entrypoint
Serve local `stdio` by default and Streamable HTTP when `TRANSPORT=http`.
	`if (process.env.TRANSPORT === "http") {`
	`  const h = createMcpHandler(createServer);`
	`  const app = createMcpExpressApp();`
	`  app.all("/mcp", (q, s) => void toNodeHandler(h)(q, s, q.body));`
	`  app.listen(3000, "127.0.0.1");`
	`} else { serveStdio(createServer); }`
		One build artifact runs locally in IDEs and remotely in Docker/cloud with DNS-rebinding guards intact.

2. `CHARACTER_LIMIT` Safe Paginator
Stay under 25,000 chars without slicing JSON mid-field.
	`let items = data.users, res = render(items, offset);`
	`while (items.length > 1 && res.text.length > 25_000) {`
	`  items = items.slice(0, Math.ceil(items.length / 2));`
	`  res = render(items, offset);`
	`}`
	`return { content: [{ type: "text", text: res.text }] };`
		Guarantees valid JSON/Markdown and sets `next_offset` accurately to the first omitted record.

3. `outputSchema` + `structuredContent`
Return machine-validated JSON for typed downstream pipelines.
	`server.registerTool("example_get_user", {`
	`  inputSchema: z.strictObject({ user_id: z.string().min(1) }),`
	`  outputSchema: UserSchema,`
	`  annotations: { readOnlyHint: true }`
	`}, async ({ user_id }, ctx) => {`
	`  const u = UserSchema.parse(await fetchUser(user_id, ctx.mcpReq.signal));`
	`  return { content: [{ type: "text", text: JSON.stringify(u) }], structuredContent: u };`
	`});`
		Strips unexpected upstream keys and satisfies both structured-output clients and plain-text agents.

4. `scripts/smoke.ts` Self-Check
Verify built server over real `stdio` before shipping.
	`const client = new Client({ name: "smoke", version: "1.0.0" });`
	`await client.connect(new StdioClientTransport({`
	`  command: "node", args: ["dist/index.js"],`
	`  env: { ...getDefaultEnvironment(), API_KEY: process.env.API_KEY ?? "" }`
	`}));`
	`const bad = await client.callTool({ name: "example_search_users", arguments: { query: "j" } });`
	`assert.equal(bad.isError, true);`
		Catches broken builds, missing env forwarding, and silent Zod schema regressions in under 2 seconds.

5. HTTP Status Error Mapper
Turn 401/403/404/429 into model-actionable recovery advice.
	`function httpErrorMessage(status: number): string {`
	`  switch (status) {`
	`  case 401: return "Invalid API key. Check EXAMPLE_API_KEY.";`
	`  case 404: return "Not found. Run example_search_users for IDs.";`
	`  case 429: return "Rate limited. Wait 60s before retrying.";`
	`  default: return "Upstream HTTP " + status + ".";`
	`  }`
	`}`
		Prevents blind retry loops by telling the LLM whether to stop, wait, or call a discovery tool first.

Pro Tip: Pair `npm run build && npm run smoke` in Every CI Run

Because `tsx` skips TypeScript type-checking during local dev, always run `npm run build` (`tsc`) followed by `tsx scripts/smoke.ts` against `dist/index.js` so you verify the exact compiled binary hosts execute.

Packaging, Distribution & Multi-Host Rollout

Ship one predictable server artifact across local laptops and cloud clusters. Clean entrypoints, minimal Docker images, and portable JSON configs let teams adopt your tools in seconds.

1. Local `npx` & `uvx` Package Metadata

```
// package.json (Node/TypeScript)
{
  "name": "example-mcp-server",
  "type": "module",
  "bin": { "example-mcp-server": "dist/index.js" },
  "files": ["dist"],
  "engines": { "node": ">=20.3" }
}
```

Start `src/index.ts` with `#!/usr/bin/env node` so `npx -y example-mcp-server` works immediately. For Python, declare `[project.scripts]` in `pyproject.toml` so hosts can run `uvx example-mcp`, or zip into a desktop `.mcpb` bundle with a `manifest.json`.

2. Stateless Docker Image (Streamable HTTP)

```
FROM node:22-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY dist ./dist
ENV TRANSPORT=http PORT=3000
USER node
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

Bind to `0.0.0.0` inside containers and configure `allowedHosts` + `requireBearerAuth`. Never bake API secrets into image layers—inject them at runtime.

3. Registering Your Server Across Major AI Hosts

Host Environment	Configuration / CLI Registration	Key Environment Notes
Claude Code	`claude mcp add example -- npx -y example-mcp-server` Remote: `claude mcp add --transport http example https://mcp.acme.dev/mcp`	Supports project-scoped `.mcp.json` checked into Git (pass secrets via `${EXAMPLE_API_KEY}`).
Claude Desktop	Edit `claude_desktop_config.json` under `"mcpServers"` with `"command": "npx"`, `"args": ["-y", "example-mcp-server"]`, or drop a `.mcpb` bundle.	Desktop GUI spawns isolated env—list required API keys explicitly inside the entry's `"env"` object.
Google Antigravity	Add server under MCP settings (`mcp_config.json`) using either `command`/`args` for `stdio` or `serverUrl` for Streamable HTTP.	Exposes tools alongside built-in workspace capabilities; pair with `SKILL.md` workflows for higher-level playbooks.
Codex / OpenAI Agents SDK	`MCPServerStdio(params={"command": "uvx", "args": ["example-mcp"]})` or `MCPServerStreamableHttp(url=...)`.	Supports explicit tool filtering and tracing spans across multi-agent handoffs.

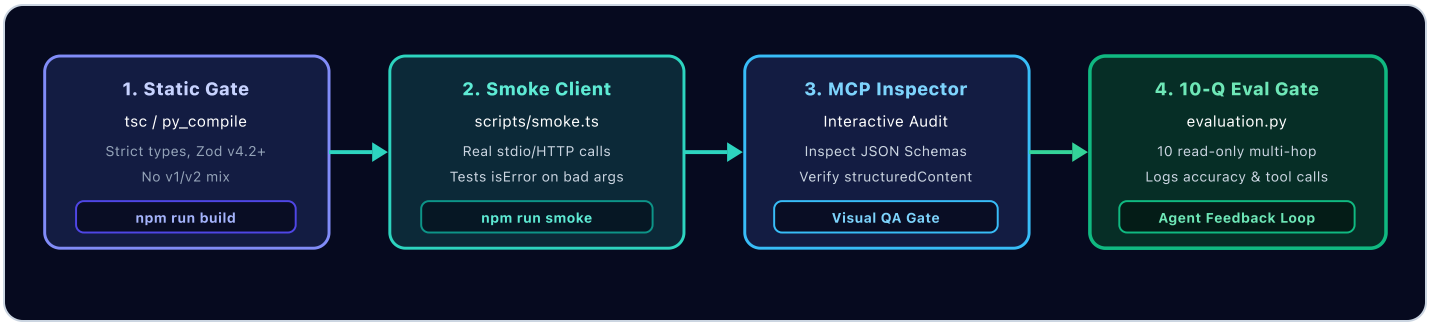
Security Boundary for Shared Team Configs

Commit server coordinates (command, URL, and variable names like `${EXAMPLE_API_KEY}`) to repository configs, but keep actual tokens inside user-local `.env` or OS keychain vaults.

Cross-Tool Synergy & 10-Question Eval Pipeline

Compiling cleanly only proves syntax. True MCP quality is measured by whether a cold LLM—given *only* your tool schemas and descriptions—can solve realistic, multi-hop research tasks without getting lost.

Four-Stage Verification & Evaluation Pipeline



Designing 10 High-Signal Evaluation Questions

- **Read-only & independent:** Never require writes or depend on earlier question state.
- **Multi-hop & non-keyword:** Paraphrase concepts so the agent must page, cross-reference IDs, and synthesize across multiple tool calls.
- **Stable & verifiable:** Query closed/historical records (e.g., Q4 archived repos) whose single string answer never drifts over time.
- **Format-explicit:** State exact output formats in the question (e.g., *"Use YYYY/MM/DD"* or *"Provide only the username"*).

XML Harness Format & Runner

```
<evaluation>
  <qa_pair>
    <question>Find the repository archived in Q3 2023
with the most prior forks. What was its primary
language?</question>
    <answer>Python</answer>
  </qa_pair>
</evaluation>

# Run against stdio or live Streamable HTTP endpoint
python scripts/evaluation.py -t http -u
https://mcp.acme.dev/mcp \
  -H "Authorization: Bearer $TOKEN" -o eval_report.md
evaluation.xml
```

Mine the Eval Report for Tool-Design Bugs

When accuracy dips or tool-call counts spike, read the agent's per-task feedback in `eval_report.md`: missing pagination hints, vague parameter descriptions, or oversized responses almost always explain why an agent stalled.

Troubleshooting Clinic & Pre-Flight Checklist

Diagnose the seven most common MCP production failures in ten seconds, then clear the six-point pre-flight gate before publishing or deploying.

Symptom / Error	Root Cause	10-Second Fix
1. Broken `stdio` JSON-RPC Host disconnects with parse error.	Code or a dependency called `console.log()` or `print()`, writing logs to `stdout`.	Route every log exclusively to `console.error()` / `sys.stderr`; keep `stdout` pure JSON-RPC.
2. Concurrent `-32603` Failures Smoke test passes; parallel HTTP fails.	Factory returns one shared module-level `McpServer` instance across HTTP requests.	Instantiate `new McpServer(...)` and register tools *inside* `createServer()` on every call.
3. Empty Schema Descriptions Model guesses field formats blindly.	Project installed Zod `<4.2.0` (pre-`4.1.13` strips `.describe()` text) or omitted `.describe()`.	Pin `"zod": "^4.2.0"`, import via `import * as z from "zod/v4"`, and add `.describe()` everywhere.
4. v1 / v2 Type & Import Clashes `registerTool` missing or type errors.	Mixed legacy `@modelcontextprotocol/sdk` v1 imports with v2 `@modelcontextprotocol/server`.	Remove v1 packages or run `npx @modelcontextprotocol/codemod@latest v1-to-v2 .` across the project.
5. Truncated Mid-Word Payloads Invalid JSON or lost records.	Sliced response text at 25,000 characters instead of trimming whole result items.	Halve the array in a `while` loop until rendered output fits `25_000`, then set `next_offset`.
6. Hanging Upstream Fetches Cancelled calls leak sockets & quota.	`fetch()` lacked a timeout and ignored the client's MCP cancellation token.	Pass `signal: AbortSignal.any([ctx.mcpReq.signal, AbortSignal.timeout(30_000)])`.
7. Wrong Tool Invoked Agent calls sibling server's `search`.	Unprefixed generic names (`list_items`, `create_issue`) collide across connected servers.	Namespace every tool as `{service}_{action}_{resource}` and mention sibling tools in descriptions.

6-Point Production Pre-Flight Checklist

- ☐ **1. Clean Factory Isolation:** `createServer()` constructs a fresh `McpServer` per call; shared HTTP pools live at module scope.
- ☐ **2. Strict Contracts & Hints:** Zod v4 `z.strictObject` / Pydantic v2 `extra="forbid"`, rich `.describe()`, and accurate annotations on every tool.
- ☐ **3. Bounded Context Payloads:** Pagination (`limit` / `offset` / `next_offset`) and item-halving cap responses under `25_000` characters.
- ☐ **4. Auth & Signal Wiring:** `requireBearerAuth` + `expectedResource` on remote HTTP; `ctx.mcpReq.signal` + 30s timeout on upstream calls.
- ☐ **5. Build & Smoke Verified:** `npm run build && npm run smoke` (or `py_compile`) and MCP Inspector checks pass with zero stdout pollution.
- ☐ **6. 10-Question Eval Passed:** Multi-hop read-only `evaluation.xml` passes with high accuracy and crisp tool-call efficiency.