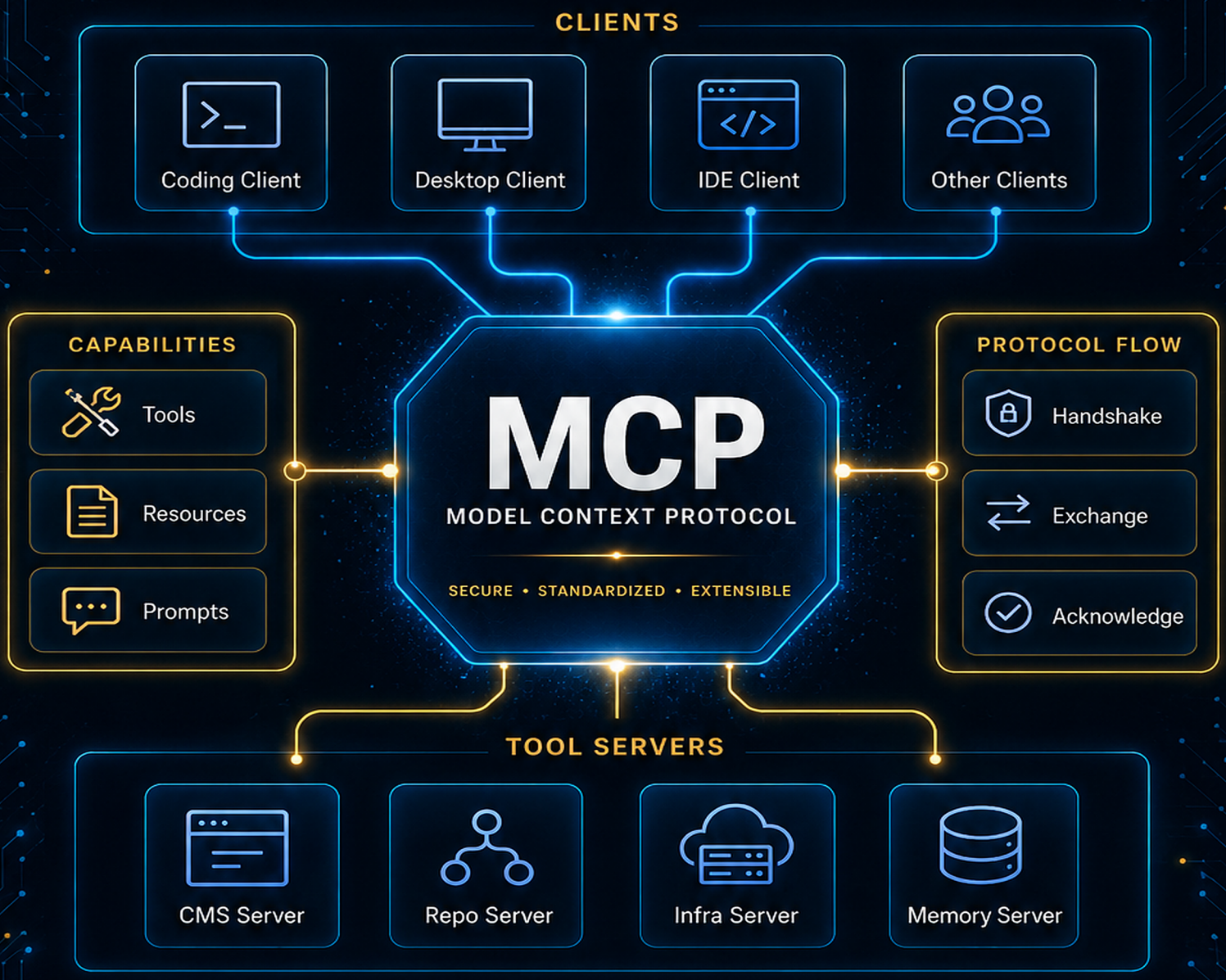
The Model Context Protocol (MCP) in Practice

BUILDING, INTEGRATING, AND SCALING CUSTOM TOOL SERVERS FOR AI AGENTS



YOHAN J. RODRÍGUEZ

Preface

> "A language model can reason about your problem. It cannot reach your systems — until you give it a way."

A language model, on its own, can only talk. It can reason about your code, draft your content, and explain your stack trace, but it cannot read the file you are pointing at, query the database that holds the answer, or create the draft you just asked for. Every useful AI feature bridges that gap by connecting the model to a real tool, a live data source, or a system that does something. The question this book answers is how to build those connections *well*: safely, reusably, and without rewriting them for every model and every editor.

The Model Context Protocol (MCP) is the answer the industry converged on. Before it, each AI client invented its own way to expose tools, and each capability had to be re-implemented for each client: N clients times M capabilities, a tangle that grows multiplicatively. MCP replaces it with one open protocol. You build a *server* that exposes a capability once, and any MCP-aware client can use it, including Claude Code, Claude Desktop, Cursor, and Zed. The tangle becomes $N + M$. That is the central idea, and the rest of the book is about carrying it out properly.

This is a hands-on book for engineers who build things. It is written for backend, platform, DevOps, and AI engineers who are comfortable with HTTP APIs, JSON, and the command line, and who have been handed some version of "connect this system to our AI tooling." You do not need a machine-learning background, because MCP is an integration protocol and not a modeling one. The code is in Python and TypeScript, and the book follows both. It builds its first server twice so you can see what belongs to the protocol and what belongs to one SDK.

The book follows the real arc of building a server, in five parts. **Part I** explains the protocol: the JSON-RPC foundation, the stdio and SSE transports, the initialization handshake, and the three primitives a server is built from, which are tools, resources, and prompts. **Part II** builds a small working server in Python and again in TypeScript. **Part III** is the craft of design and security: tool surfaces that a model uses correctly, resources that serve live data without flooding the

context window, prompt templates that package workflows, and a defense-in-depth treatment of a model that can be manipulated by the content it reads. **Part IV** connects servers to real clients and then builds four reference servers against real kinds of systems: WordPress content, GitLab issues and pipelines, host infrastructure behind a strict allowlist, and a persistent memory store. **Part V** proves and ships them, with layered testing, the MCP Inspector, Docker packaging, and deployment over SSE behind a reverse proxy.

Two habits run through every chapter. The first is that safety is designed in rather than added later. The WordPress server can create drafts but has no way to publish, the GitLab server previews before it acts, the DevOps server runs only named operations, and the memory server refuses to store secrets, and each chapter explains why. From Chapter 6 on, each chapter provides a reusable checklist you can run against your own servers. The second is honesty about what was verified. The first servers were run by hand and driven over stdio, and the safety helpers and storage layers of the reference servers have tests that were executed. Running a full reference server against a live WordPress site, GitLab project, or systemd host is left to you, in the labs and in the release checklist of the final chapter, and none of those labs needs production access. Every chapter includes a hands-on lab. Some are run at a terminal, and others are design and review exercises.

A word on scope and on time. This book teaches the *server* side of MCP. Writing a custom client or host is a different job and stays out of scope. Because MCP, its SDKs, and the clients that speak it all move quickly, the book pins exact versions for its code (the Python `mcp` SDK and the TypeScript `@modelcontextprotocol/sdk`) and gathers the fast-moving details into clearly marked callouts. The SSE transport it demonstrates is now considered a legacy option, and the final part explains the newer Streamable HTTP direction. Treat the callouts as pointers to confirm against current documentation, and treat the durable ideas as the part that does not drift: a protocol beats bespoke integration, design the surface a model can use, and never trust the model with more than it needs.

By the end, you should be able to design, implement, test, secure, connect, package, and deploy MCP servers, and to look at any system with an API or a local capability and put a safe, well-designed agent interface in front of it. Let us build it.

Yohan J. Rodriguez

Contents

Preface i

1 The MCP Landscape: Why Protocol Beats Integration **1**

1.1 The N-by-M Integration Problem 1
1.2 What a Protocol Buys You 3
1.3 The Three Primitives at a Glance 3
1.4 Where MCP Fits in the Stack 4
1.5 MCP, Function Calling, and Framework Tools 5
1.6 The MCP Client Ecosystem 6
1.7 The Four Servers You Will Build 7
1.8 Should You Use MCP at All? 7
1.9 Hands-On: Connect Your First Server 9
1.10 What You Will Be Able to Do 10

1 | The MCP Landscape: Why Protocol Beats Integration

A language model, on its own, can only talk. It can reason about your problem, draft your email, and explain your stack trace, but it cannot read the file you are referring to, query the database that holds the answer, or create the draft post you just asked for. To do anything in the world, the model needs tools — and somebody has to build the bridge between the model and each tool. For the first few years of the modern era of large language models, that bridge was built by hand, one integration at a time, and rebuilt every time either end of it changed. The Model Context Protocol exists to stop that rebuilding. This chapter is about why a protocol is the right answer to a problem that, at first, does not look like it needs one, and about what you will build over the rest of the book once you accept that it does.

This is the orientation chapter. It does not teach you to write a server yet — that starts in Chapter 4. Its job is to give you the mental map: the problem MCP solves, the shape of the solution, the ecosystem you are stepping into, and an honest account of when the protocol earns its keep and when a simpler approach is fine. Readers who understand *why* the protocol exists make better design decisions than readers who treat it as just another library to import, so it is worth a few pages before the code begins.

1.1 The N-by-M Integration Problem

Consider a single, modest capability: "create a blog post draft." The work itself is small — a function that takes a title and a body and writes a row to a database or calls a content API. The hard part was never the function. The hard part was telling a model that the function exists, in the exact dialect that the model's client expects.

Before MCP, every AI integration was bespoke. If you wanted the OpenAI function-calling API to use your capability, you wrote a JSON function definition in its schema. If you wanted a different vendor's model to use the same capability, you wrote a second definition in that vendor's schema. If you wanted a proprietary chat product's plugin system to expose it, you wrote a manifest in a third format. Three descriptions of one function, none of them reusable, each of

them coupled to a product you do not control. Listing 1.1 shows the duplication in miniature.

Listing 1.1: One capability, redefined once per provider. Add a third client and you write a third wrapper.

```python
"""The N-by-M problem, in code.

The same capability --- "create a blog post" --- has to be redefined once per model
provider and once per client, because each one expects a different tool-calling shape.
Two providers are shown here; a third would be a third near-duplicate. Multiply by every
client that needs the capability and the maintenance cost grows with the product, not the
sum, of the two counts.
"""

from typing import Any


def create_post(title: str, body: str, status: str = "draft") -> dict[str, Any]:
    """The actual capability. This part is small and provider-agnostic."""
    return {"id": 123, "title": title, "status": status}


# — Provider A: OpenAI-style function calling ——————————
openai_tool = {
    "type": "function",
    "function": {
        "name": "create_post",
        "description": "Create a blog post draft.",
        "parameters": {
            "type": "object",
            "properties": {
                "title": {"type": "string"},
                "body": {"type": "string"},
                "status": {"type": "string", "enum": ["draft", "publish"]},
            },
            "required": ["title", "body"],
        },
    },
}

# — Provider B: a different vendor's bespoke schema ————————
vendor_b_tool = {
    "tool_name": "create_post",
    "summary": "Create a blog post draft.",
    "args": [
        {"key": "title", "kind": "string", "required": True},
        {"key": "body", "kind": "string", "required": True},
        {"key": "status", "kind": "string", "default": "draft"},
    ],
}

# Same function. Two hand-written wrappers. Change the function signature and you edit
# every wrapper. Add a client that speaks neither shape and you write a third.
```

Now scale it up. Suppose your organization has M capabilities worth exposing — post creation, issue lookup, log retrieval, a metrics query — and there are N different clients or models that should be able to use them. The naive approach requires up to $N \times M$ hand-written integrations, and the maintenance cost grows with that product rather than with the sum. Change one capability's signature and you touch every client that uses it. Adopt one new client and

you re-wrap every capability. When the model changes, the wrapper changes; when the client changes, the manifest changes. The combinatorial blow-up is not a hypothetical — it is the predictable consequence of letting each end of the bridge define its own dialect.

This is the same problem that protocols have solved over and over in computing history. Before a common protocol, every device driver was written for every operating system; every database had its own wire format that every language needed a bespoke client for. The escape from $N \times M$ is always the same move: define a standard in the middle, and make each side implement it once.

1.2 What a Protocol Buys You

MCP solves the integration problem the way the web solved document delivery and the way SQL clients solved database access: with a client–server protocol and a defined wire format.

In MCP's model, the **server** exposes capabilities and the **client** consumes them. A server advertises what it can do — its tools, its readable resources, its prompt templates — using a standardized format built on JSON-RPC 2.0 (the subject of Chapter 2). A client is the program the model runs inside: Claude Code, Claude Desktop, an editor with agent features. Any MCP-compliant client can consume any MCP-compliant server without custom glue. You implement the bridge once, on the server side, and every client that speaks the protocol can cross it.

That single sentence is the whole value proposition, so it is worth restating precisely. You write *one* server. It works in every client that supports MCP, today and in the future, including clients that did not exist when you wrote it. The capability is no longer coupled to a model or a product. The $N \times M$ matrix collapses into $N + M$: each client implements the protocol once, each server implements it once, and the middle is shared (as illustrated in Figure 1.1).

There is a second, subtler benefit. Because the protocol defines a handshake in which the client and server negotiate capabilities explicitly (Chapter 2), and because the server is a separate process with its own permissions (Chapter 9), the protocol gives you natural seams for security and versioning that a pile of inline function definitions never had. A bespoke wrapper has no boundary; an MCP server is a boundary. We will lean on that boundary heavily when we get to the security chapter.

1.3 The Three Primitives at a Glance

An MCP server exposes exactly three kinds of capability. Chapter 3 is devoted to them in depth; here is the one-paragraph version so the rest of this chapter has vocabulary to use.

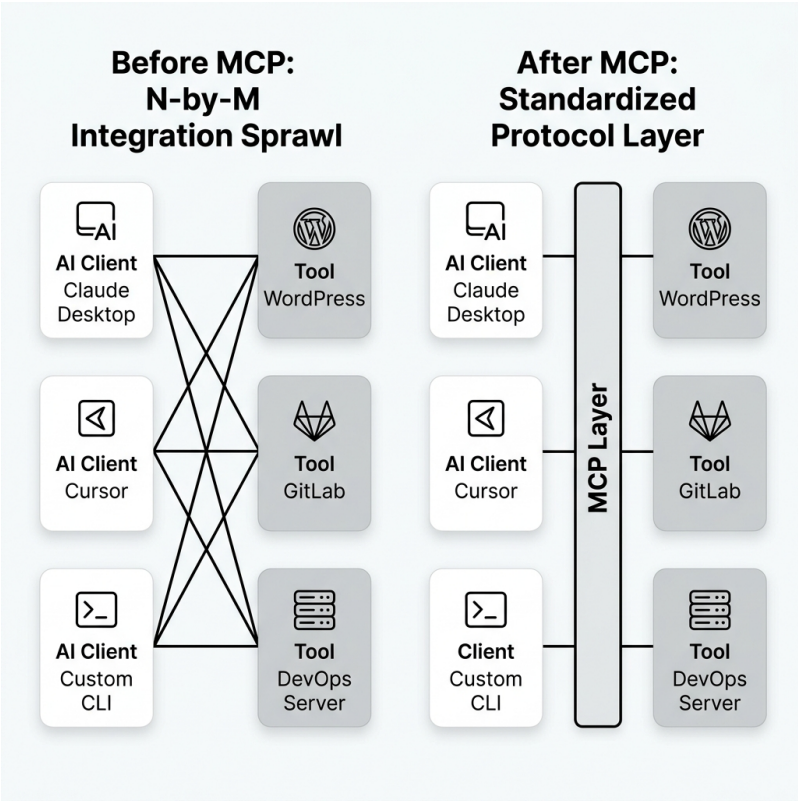


Figure 1.1: Comparison of ad-hoc integration complexity ($N \times M$ connections) versus standardized protocol-based scaling ($N + M$ connections) using the Model Context Protocol.

Primitive	What it is	When to use it
Tool	A callable function the model invokes	To *do* something: create, trigger, query, modify
Resource	Readable data at a URI	To *read* context: a file, a record, a log, a config
Prompt template	A parameterized message sequence	To *frame* a request the server knows how to phrase

Table 1.1: The three MCP primitives. Tools are the write path, resources are the read path, and prompt templates package reusable expertise. Choosing the right one is the core design skill of MCP development, covered fully in Chapter 3.

Most servers start with tools, because "let the model call my function" is the most obvious need. Resources and prompt templates are underused by beginners precisely because their value is less obvious — and that is exactly why Chapters 7 and 8 give them their own treatment.

1.4 Where MCP Fits in the Stack

It helps to place MCP in the larger picture of an AI application. At the center sits the model. Around it sits the client host — the program that manages the conversation, decides when to call a tool, and renders results to the user. The host is where agent behavior lives: planning, looping, deciding. MCP is the layer *between the host and the outside world* (as illustrated in Figure 1.2). When the host decides the model needs to take an action or read some data, it speaks MCP to a

server, and the server does the real work against a filesystem, a database, an API, or a shell.

This is why the book is about building servers, not hosts. The host is the consumer; the server is where your capabilities live. Writing a custom client host — implementing the model loop, the tool-call dispatch, the UI — is a different and larger project, and it is explicitly out of scope here (we note this again in Chapter 2). Multi-agent orchestration, where several models coordinate, sits even further up the stack; this book references it where relevant but treats it as the subject of a later volume. Your job, throughout this book, is to build the thing on the far side of the protocol: a reliable, secure, well-designed server.

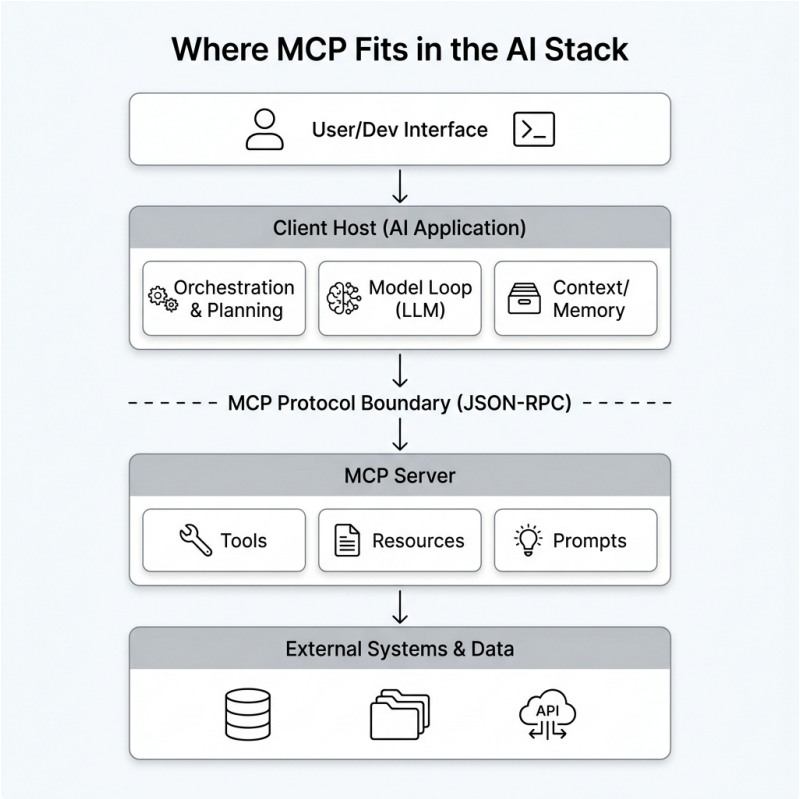


Figure 1.2: The architectural layers of an AI application showing the Model Context Protocol as the secure communication boundary between the client host and external servers exposing capabilities.

1.5 MCP, Function Calling, and Framework Tools

If you have already wired a model to a tool, you have probably done it one of two other ways, and it is worth being clear about how MCP relates to them.

Approach	Coupling	Reuse across clients
Provider function calling	Tied to one model API	None — redefine per provider
Framework tools (LangChain, etc.)	Tied to one framework's abstractions	Within that framework only
Model Context Protocol	Tied to the protocol, not a vendor	Any MCP client, any MCP model

Table 1.2: MCP compared with the two integration styles most readers arrive with. The difference that matters is where the coupling lives.

Provider function calling — the OpenAI-style `tools` array, and its equivalents — is the mechanism by which a single model API learns about your functions. It is perfectly good, and under the hood an MCP client often translates the server's tools into exactly such a structure before handing them to the model. The limitation is reuse: a function definition written for one provider's API is married to that API. MCP sits one level up, providing the function definitions in a vendor-neutral way that the client then adapts to whatever model it is driving.

Framework tools, as in LangChain or LlamaIndex, are abstractions a framework offers so your Python or TypeScript code can register callables the framework will expose to a model. If your whole application lives inside that framework, this works. But the abstractions do not map cleanly onto the MCP wire format — there is no transport concept, no capability negotiation, no resource URI templates — so framework tool code is not portable to other clients. Readers coming from a LangChain background will find conceptual overlap (the idea of a "tool"), and real structural differences; the book points these out as they arise.

The honest summary: MCP does not replace function calling, it standardizes and shares it. It does not compete with agent frameworks, it gives them a vendor-neutral way to acquire capabilities. The win is that the capability you write is decoupled from both the model and the framework.

1.6 The MCP Client Ecosystem

A protocol is only useful if things speak it. MCP's adoption is what turns the single-implementation promise from a nice idea into a practical reality, and as of this writing the protocol has been adopted broadly across the major AI coding and chat tools — including Claude Code and Claude Desktop, and editor and IDE integrations such as Cursor and Zed, among a growing list. Alongside the clients, public registries of community servers have emerged, so that connecting an existing server is often a matter of a single command rather than writing any code at all.

A word of caution about this section in particular. The client list and the maturity of the registries are a fast-moving target; the specific names and their exact level of support will have shifted by the time you read this. Treat this paragraph as a snapshot, not a specification, and verify the current state against each tool's own documentation. Throughout the book, anything tied to the moving ecosystem — this client list, the exact configuration commands in Chapter 10, the SDK details in Chapter 5 — is flagged as version-sensitive for exactly this reason. The durable knowledge is the protocol itself; the surrounding tooling is where the churn lives.

What matters for your purposes is the structural fact underneath the names: there is now more than one client, those clients agree on a protocol, and the number is growing rather than shrinking. That is precisely the condition under which writing a server, rather than a stack of per-client wrappers, is the correct investment.

1.7 The Four Servers You Will Build

This book is relentlessly practical: every chapter ships working code, and the second half is organized around four complete reference servers. They are introduced here so you have a destination in mind from the first page.

- **A WordPress content server** (Chapter 11, Python) — read, draft, and update posts through the WordPress REST API, with a default-to-draft safety pattern so the model cannot publish by accident.
- **A GitLab server** (Chapter 12, TypeScript) — issues, merge requests, and CI/CD, including the single most useful agent capability in a developer's day: reading a failing pipeline's job log and reasoning about it.
- **A DevOps infrastructure server** (Chapter 13, Python) — service status, logs, and disk and memory checks behind a strict command allowlist, with a dry-run mode for anything that changes state.
- **A memory and knowledge server** (Chapter 14, Python) — persistent storage and retrieval across conversations, backed by SQLite for structured recall and pgvector for semantic search.

Two of these are in Python and two in TypeScript on purpose. The book teaches both SDK tracks in parallel — the Python `mcp` package and the TypeScript `@modelcontextprotocol/sdk` — so that whichever language you reach for, you have a complete foundation and a worked example. By the final chapter you will package one of these servers as a Docker image and deploy it over a network transport behind a reverse proxy.

1.8 Should You Use MCP at All?

A book about MCP has an obvious incentive to tell you the answer is always yes. It is not, and pretending otherwise would cost you trust on the first page. MCP adds real value in three situations, and adds mostly ceremony outside them.

It is worth the investment when a capability must be *reused across more than one client or model*, because that is the $N \times M$ problem the protocol was built to kill. It is worth it when the capability

needs a genuine *security boundary* — a separate process with its own permissions, an explicit allowlist, a place to validate every argument — because the protocol gives you that boundary for free where inline function definitions give you nothing. And it is worth it when you benefit from *standardized capability negotiation*, so that a client can discover at runtime what your server offers rather than being hard-coded against it.

It is overkill when you are wiring a single capability to a single client in a single model, as a prototype you expect to delete. There, a provider's native function-calling API is less code and fewer moving parts, and reaching for a protocol buys you nothing you will use. Listing 1.2 captures the trade-off as a blunt heuristic — not a law, but an honest one.

Listing 1.2: A rule of thumb for the "MCP or just call the API?" question.

```python
"""A blunt heuristic for the question this chapter ends on.

MCP earns its keep when a capability must be reused across clients or models, when it needs
a real security boundary, or when it benefits from standardized capability negotiation. A
one-off, single-client prototype usually does not need it. This is a rule of thumb, not a
law --- but it captures the trade-off honestly.
"""

from dataclasses import dataclass


@dataclass
class Integration:
    clients: int            # how many distinct clients will consume the capability
    models: int             # how many model providers must it work with
    needs_isolation: bool   # does it touch a filesystem, shell, or network it must guard?
    is_throwaway: bool      # a demo you will delete next week?


def should_use_mcp(i: Integration) -> bool:
    if i.is_throwaway and i.clients <= 1:
        return False
    return i.clients > 1 or i.models > 1 or i.needs_isolation


if __name__ == "__main__":
    prototype = Integration(clients=1, models=1, needs_isolation=False, is_throwaway=True)
    platform = Integration(clients=4, models=2, needs_isolation=True, is_throwaway=False)
    print("prototype ->", should_use_mcp(prototype))   # False
    print("platform  ->", should_use_mcp(platform))    # True
```

The reason the answer is usually "yes" for the readers of this book is that real organizational work almost never stays a single-client prototype. The integration you build this quarter for one editor is the integration three teams want next quarter for three others, and the model you target today is not the only model your company will use. MCP is the bet that there will be more than one consumer — and that bet pays off far more often than not.

1.9 Hands-On: Connect Your First Server

You learn a protocol fastest by watching it work, so before writing a single line of server code in Chapter 4, connect an existing community server and observe its capabilities appear inside a real client. Nothing here requires you to understand the wire format yet; that is Chapter 2's job. This is the "it already works" moment.

The community filesystem server is a good first target: it exposes tools for reading files under a directory you choose, which makes its effect easy to see. For a graphical client like Claude Desktop, you register a server by adding an entry to its configuration file. The shape of that entry is the same across clients: a name, a command to launch, and the arguments to pass. Listing 1.3 shows a minimal entry.

Listing 1.3: A Claude Desktop configuration entry that launches the community filesystem server over stdio.

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "/Users/you/Documents/notes"
      ]
    }
  }
}
```

For a command-line client like Claude Code, the same registration is a single command, which is faster for experimentation. Listing 1.4 adds the server, lists what is registered, and removes it again.

Listing 1.4: Registering, listing, and removing a community server with the Claude Code CLI.

```
# Add a community MCP server to Claude Code (user scope) and confirm it is registered.
# The server is launched on demand over stdio; npx fetches it the first time.
claude mcp add filesystem -- npx -y @modelcontextprotocol/server-filesystem ~/notes

# List the servers Claude Code knows about and their transport.
claude mcp list

# Remove it again when you are done experimenting.
claude mcp remove filesystem
```

Once a server is connected, the client asks it what it can do by sending a `tools/list` request, and the server answers with the names, descriptions, and input schemas of its tools. That exchange — which you will send by hand in Chapter 2 and implement yourself in Chapter 4 — is what populates the model's awareness of your capabilities. Listing 1.5 shows a representative request and response so you can see the shape of what is flowing underneath the friendly UI.

Listing 1.5: A `tools/list` request and a representative response. This is what the client turns into the model's tool awareness.

```
{
```

```
  "request": {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "tools/list"
  },
  "response": {
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
      "tools": [
        {
          "name": "read_file",
          "description": "Read the complete contents of a file from the allowed directory.",
          "inputSchema": {
            "type": "object",
            "properties": {
              "path": { "type": "string" }
            },
            "required": ["path"]
          }
        }
      ]
    }
  }
}
```

With the server connected, ask the model in plain language to read a file from the directory you allowed, and watch it call the tool. The model did not know that tool existed a minute ago; the server told it, through the protocol, the moment they were introduced. That is the entire idea of the book, demonstrated before you have written any code. The exact configuration-file locations and CLI flags vary by client and version — Chapter 10 covers every connection surface in detail, including how to diagnose it when the server stubbornly refuses to appear.

1.10 What You Will Be Able to Do

By the end of this book you will be able to design, build, test, package, and deploy a production-grade MCP server in either Python or TypeScript. You will understand the JSON-RPC 2.0 wire format well enough to debug a raw message, choose correctly between tools, resources, and prompt templates, write input schemas that guide a model reliably, and defend a server's security design in a code review. You will have shipped at least one of the four reference servers to a real environment.

For now, the only thing you need to carry into Chapter 2 is the central claim of this one: the problem with AI integrations was never the integration, it was the *re*-integration — the endless re-wrapping of the same capability for each new model and client. A protocol ends that. You build the bridge once, on the server side, and let every client that speaks the protocol walk across it. The next chapter opens up the protocol itself and shows you exactly what crosses that bridge, field by field, on the wire.