

Visual Guide to Computer Systems

MCP A Visual Guide

Sample Book — Chapters 1 to 2

A Beginner-Friendly Foundation for Practical MCP Engineering

Reiner Braun

Introduction

This book is designed for readers who are taking their first steps with MCP, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

This is a beginner-friendly book focused on the essential fundamentals. The chapters are arranged so that each one supplies background knowledge used by the chapters that follow.

- **Read the chapters in order.** The sequence moves from the MCP mental model to protocol behavior, capability design, transports, production engineering, and an integrated capstone.
- **Review “What You’ll Be Able to Do” before each chapter.** This short section identifies the practical abilities the chapter is intended to build.
- **Use diagrams to understand relationships, processes, and structures.** The surrounding prose remains authoritative; each diagram is a visual aid for concepts that are harder to picture from text alone.
- **Try the code and hands-on examples whenever possible.** Running the examples, inspecting the messages, and deliberately testing failure cases will make the protocol behavior more concrete.
- **Use the chapter summary and review questions to check retention.** Confirm that you can explain the central ideas in your own words before moving on.
- **Treat advanced-topic overviews as signposts.** You do not need to memorize them now; return to them when a project introduces the corresponding operational need.

Table of Contents

Introduction

How to Use This Book

Part I. Foundations, Core Capabilities, and Local Integration

Chapter 1. MCP in One Picture

The big picture

Why a common protocol matters

What MCP standardizes

What MCP deliberately does not standardize

The systems engineer's view

Continuing example: an operations assistant

The first misconception to remove

What to carry forward

Chapter 2. Hosts, Clients, Servers, and Trust Boundaries

Start with the roles

One client per server connection

Local and remote describe placement, not trust

The two protocol layers

Every connection crosses trust boundaries

Responsibility does not stop at the SDK

Map the operations assistant

Names and identity require care

The local-server misconception

The architectural takeaway

Protocol baseline: Model Context Protocol revision **2026-07-28**.

MCP is a standardized integration layer between an AI application and external capabilities. It gives hosts, clients, and servers a common way to discover data and operations, exchange structured messages, request user input, and carry those messages over local or remote transports. It does not decide how a model reasons, and it does not make an integration secure by itself.

A production MCP integration is ordinary systems integration with an unusual source of input: some requests may be proposed or shaped by a language model. Production quality therefore depends on familiar disciplines such as identity, authorization, validation, threat modeling, timeouts, retry safety, observability, testing, deployment control, and operational ownership.

This guide develops that foundation through sixteen chapters:

1. MCP in One Picture
2. Hosts, Clients, Servers, and Trust Boundaries
3. Requests, Results, Discovery, and Versioning
4. Choosing Between Tools, Resources, and Prompts
5. Designing Safe and Usable Tools
6. Delivering Context with Resources
7. Prompts, Elicitation, and Multi-Round Interaction
8. Local Integrations with the stdio Transport
9. Remote Integrations with Streamable HTTP
10. Build, Inspect, and Test a Minimal Integration
11. Identity, Authorization, and Secret Handling
12. Security Engineering for MCP Systems
13. Reliability, Performance, and Long-Running Work
14. Observability, Testing, and Troubleshooting
15. Deployment, Compatibility, and Operational Governance
16. Capstone: From Read-Only Operations Assistant to Guarded Remediation

The continuing example is an **operations assistant** used by a systems engineering team. It begins with read-only access to approved runbooks and service health, then moves to a remote deployment and, only after stronger controls are in place, gains one narrowly scoped remediation capability.

Part I. Foundations, Core Capabilities, and Local Integration

Chapter 1. MCP in One Picture

What You'll Be Able to Do

After completing this chapter, you will be able to:

- Explain what MCP standardizes and what remains outside the protocol.
- Describe MCP as an integration layer rather than a model, agent, or security guarantee.
- Recognize the main systems-engineering concerns behind an MCP-enabled application.

The big picture

MCP solves an integration problem, not an intelligence problem.

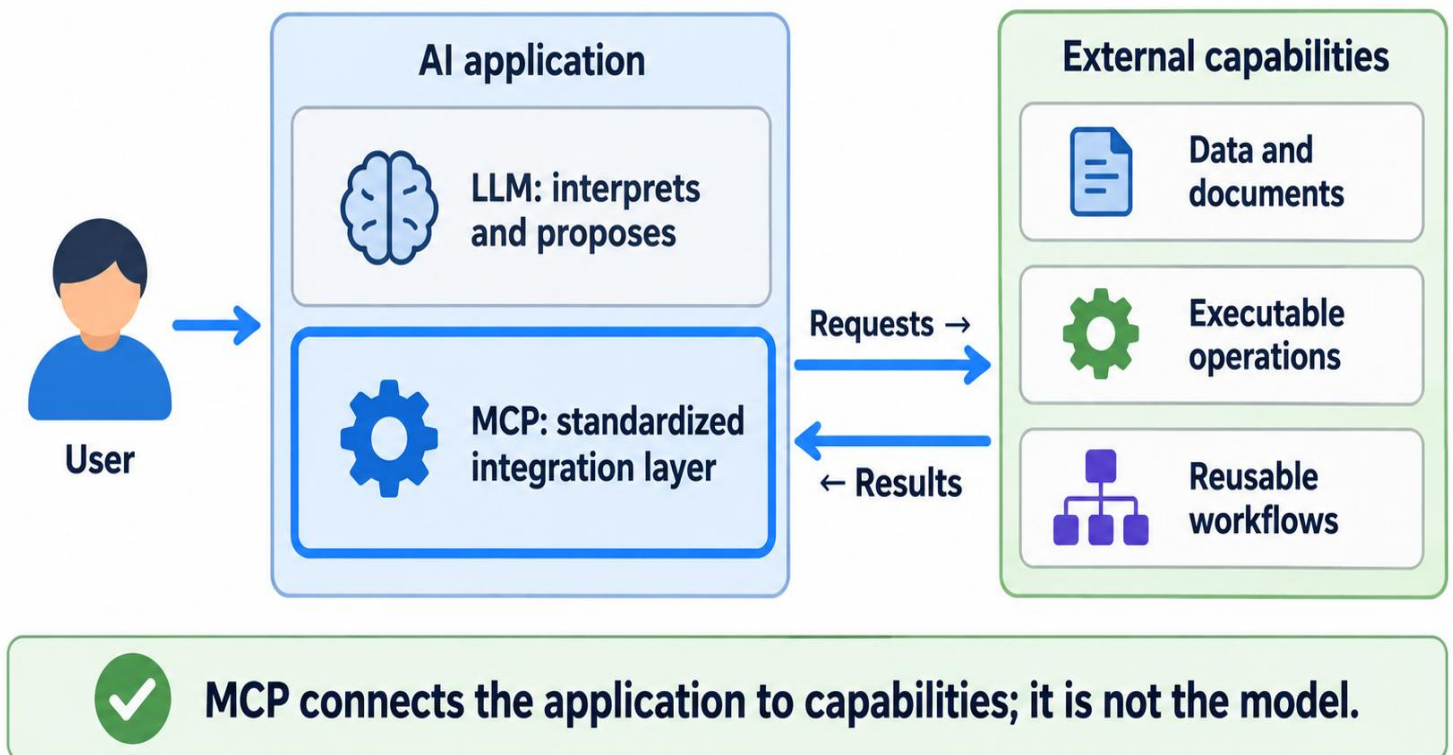
The Model Context Protocol, usually shortened to **MCP**, is an open protocol for connecting AI applications to external systems. A **protocol** is an agreed set of message formats and interaction rules that allows independently developed software to communicate.

An AI application often needs more than the knowledge built into its model. It may need to read a project document, query a monitoring service, create a ticket, or start a controlled operational action. MCP provides a standard interface for exposing those capabilities.

A useful first model is:

Chapter 1: MCP in One Picture

1-1 The big picture



001

Figure 1-1: The big picture

The **large language model (LLM)** inside an AI application is a model that processes and generates language. The model may help select an operation or interpret returned data, but MCP does not define the model itself. MCP defines how the surrounding application communicates with external capability providers.

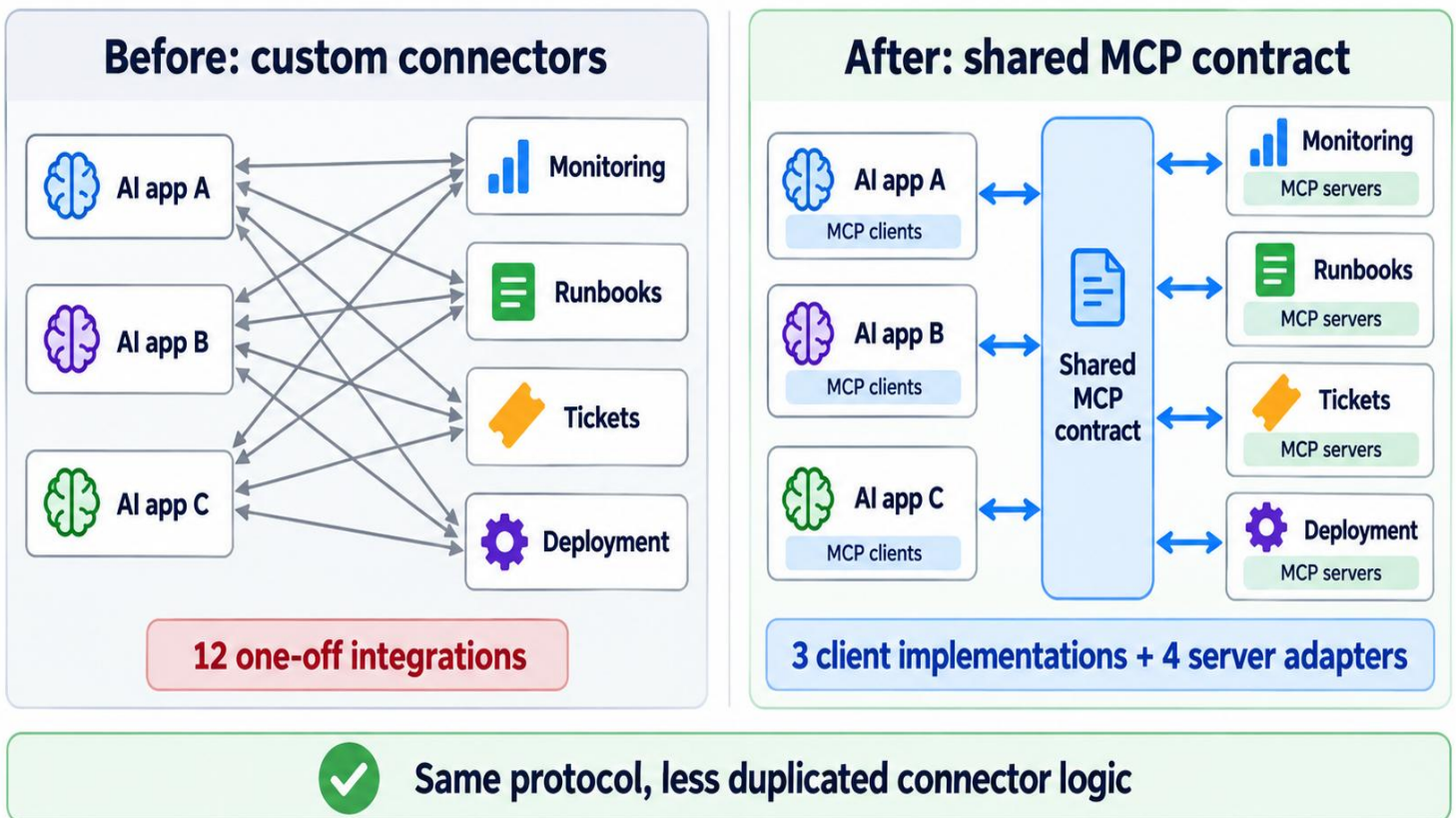
The practical conclusion for a systems engineer is direct: **treat an MCP integration as ordinary distributed or local systems integration, with processes, endpoints, credentials, schemas, logs, timeouts, permissions, and failure modes.**

Why a common protocol matters

Without a common protocol, each AI host and each external system may require a custom connector. Suppose a team uses three AI applications and wants to connect each of them to four operational systems:

Chapter 1: MCP in One Picture

1-2 Why a common protocol matters



002

Figure 1-2: Why a common protocol matters

- a monitoring platform,
- a runbook repository,
- an incident-management system,
- and a deployment service.

A fully custom approach can produce many separate integrations. Each connector may use different conventions for describing operations, validating parameters, reporting errors, requesting approval, and returning data. The result is duplicated code and inconsistent operational behavior.

MCP reduces that duplication by giving applications and capability providers a common contract. The relationship is similar to a standard electrical connector: equipment from different vendors can interoperate because both sides implement the same interface.

The analogy has an important limit. A physical connector mainly standardizes shape and electrical behavior. **MCP standardizes communication, but it does not guarantee that the connected capability is trustworthy, authorized, correct, or safe.** A standardized connector can still be connected to a dangerous device. In the same way, an MCP server can expose privileged or poorly designed operations.

What MCP standardizes

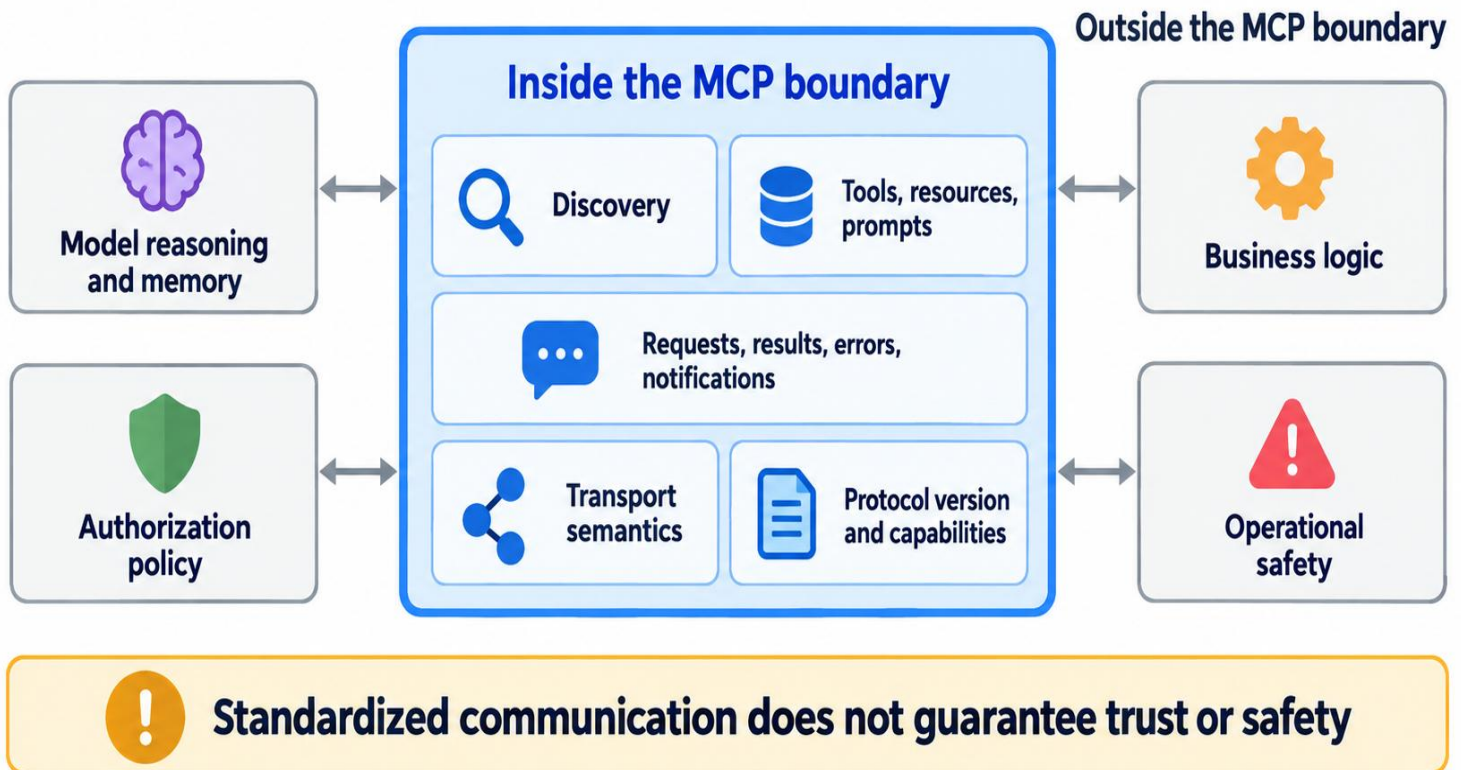
MCP standardizes the parts of an integration that must be understood by both sides.

Capability discovery

A **capability** is a feature that one participant declares it can support. For example, a server may declare that it exposes tools and resources, while a client may declare that it can collect additional user input.

Chapter 1: MCP in One Picture

1-3 What MCP standardizes



003

Figure 1-3: What MCP standardizes

Discovery lets a client learn information such as:

- which protocol versions the server supports,
- which feature families the server exposes,
- whether change notifications are available,
- and how the server identifies its implementation for display and diagnosis.

Structured operations

MCP defines common operations for listing and using the main server features:

- **tools** for executable operations,
- **resources** for addressable context data,
- **prompts** for reusable interaction templates.

Each feature has a distinct purpose. Keeping those purposes separate makes integrations easier to understand and govern.

Message structure

MCP messages use JSON-RPC 2.0. **JSON-RPC** is a lightweight remote-procedure-call format in which a sender names a method, supplies parameters, and correlates a response by an identifier.

This gives implementations a shared model for:

- requests,
- successful results,
- errors,
- and one-way notifications.

Transport-independent semantics

A **transport** is the communication mechanism that carries protocol messages. The same MCP concepts can be carried through a local subprocess stream or through HTTP. The transport changes how bytes move, but the core meaning of methods such as `tools/list` and `resources/read` remains the same.

Version and capability metadata

Modern MCP requests are self-contained. Each request states the protocol version and the client capabilities relevant to that request. This avoids relying on hidden connection state established earlier.

What MCP deliberately does not standardize

Understanding the boundary of MCP prevents many design mistakes.

Model reasoning

MCP does not define how an LLM decides whether to call a tool, which tool to choose, or how to interpret a result. Those decisions belong to the host application, its model orchestration, its prompts, and its policy layer.

Conversation memory

MCP does not define how a host stores chat history, summarizes prior messages, manages a model context window, or separates one user conversation from another.

Business logic

A tool named `restart_service` may call a deployment API, execute an orchestration workflow, or submit a request to another system. MCP defines the interface, not the internal implementation.

Organizational policy

MCP does not decide:

- who is allowed to restart production services,
- whether a change ticket is required,
- which environments are in scope,
- how approvals are recorded,
- or which credentials may be used.

These are authorization and operating-policy decisions that the server and surrounding infrastructure must enforce.

Safety enforcement by standardization alone

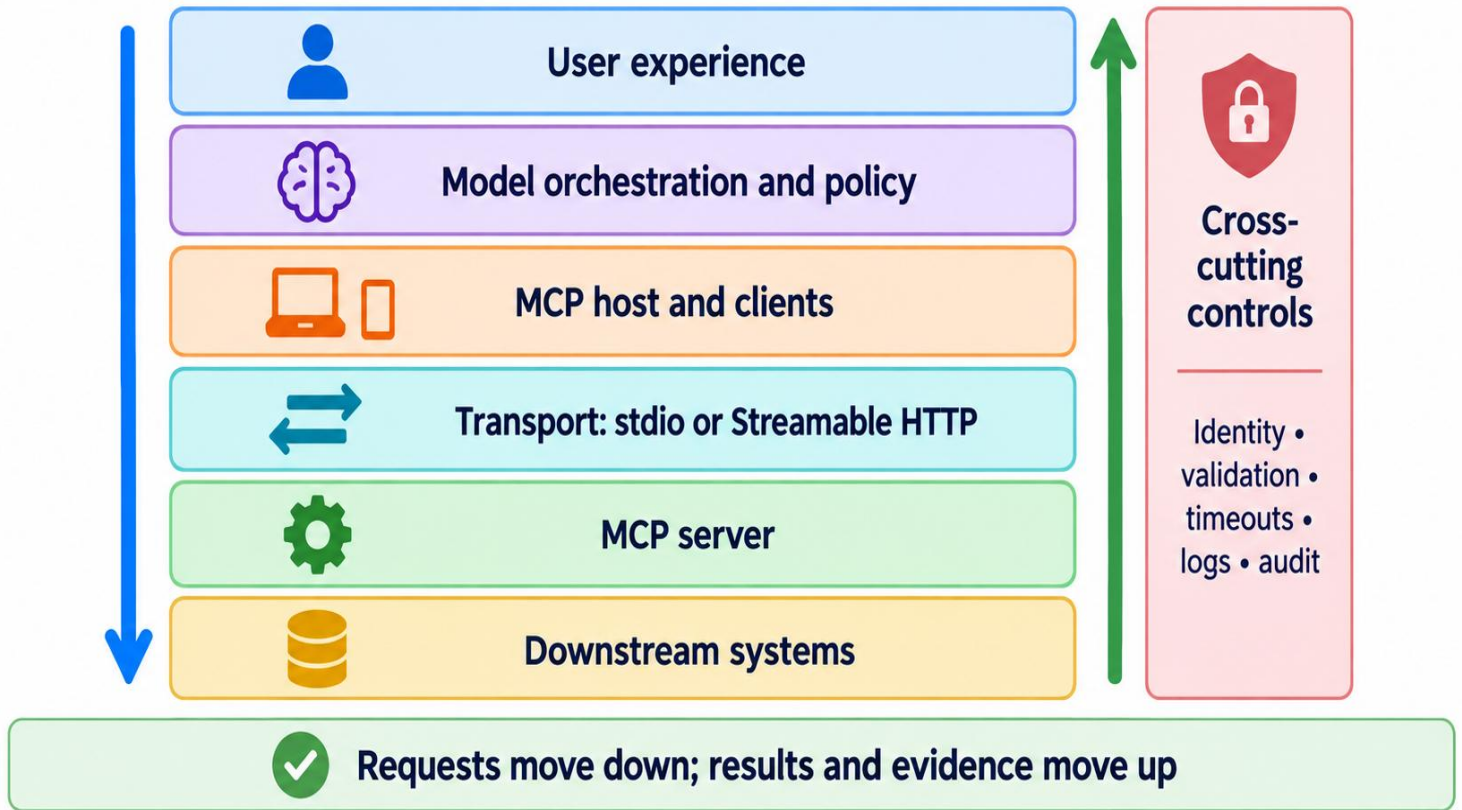
MCP supports structures that can help an application present operations clearly and request user input. It does not automatically provide isolation, least privilege, package trust, secret management, or safe execution.

The systems engineer's view

A beginner may first encounter MCP through an AI product interface. A systems engineer should immediately look behind that interface and identify the operational components.

Chapter 1: MCP in One Picture

1-5 The systems engineer's view



004

Figure 1-4: The systems engineer's view

For every integration, ask:

- What process or service implements the server?
- Who launches or deploys it?
- Which user or service identity does it run as?
- Which files, APIs, databases, and networks can it reach?
- Where do its credentials come from?
- What operations can cause side effects?
- What timeouts and retry rules apply?
- Where are logs, metrics, and audit events recorded?
- How is the server upgraded or disabled?

These questions are not secondary. They are part of the foundation because an MCP server may bridge model-influenced input to real infrastructure.

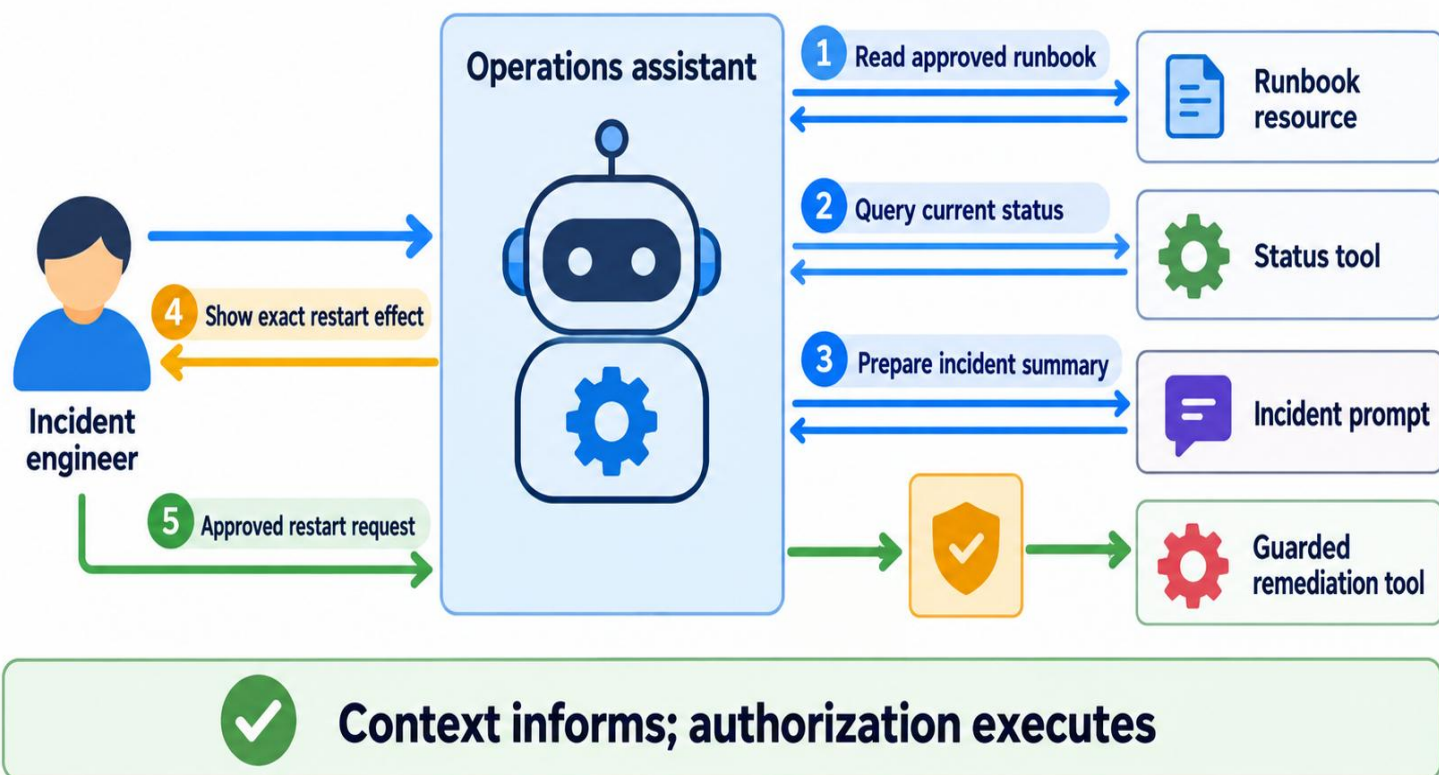
Continuing example: an operations assistant

Consider an internal assistant used during an incident. An engineer asks:

Investigate why the checkout service is unhealthy. Use the approved runbook, summarize the current status, and prepare a restart only if it is necessary.

Chapter 1: MCP in One Picture

1-6 Continuing example: an operations assistant



005

Figure 1-5: Continuing example: an operations assistant

A well-structured MCP environment might provide the following capabilities:

Need	MCP primitive	Example
Read the approved procedure	Resource	runbook://services/checkout
Query current service health	Tool	get_service_status
Start a standard incident workflow	Prompt	prepare_incident_summary
Perform a controlled restart	Tool	restart_service
Ask for a missing change-ticket number	Elicitation	Structured user-input request

The host may retrieve the runbook and make the status tool available to the model. The model may then recommend a restart. That recommendation is not authorization. Before a restart occurs, the host and server still need to enforce policy, show the target and effect, collect any required approval, and use credentials with appropriate scope.

This distinction is central:

- **Context can inform a decision.**
- **A model can propose an action.**
- **Only an authorized system should execute the action.**

The first misconception to remove

A frequent misconception is:

| MCP makes an integration safe because it is standardized.

The correct conclusion is:

MCP improves interoperability. Safety depends on the design and operation of the connected system.

A safe integration requires controls such as authentication, authorization, input validation, least-privileged credentials, user consent, isolation, timeouts, auditability, and controlled software supply chains. Later chapters examine these controls in their appropriate contexts.

What to carry forward

MCP is best understood as **a common protocol boundary between an AI application and external capabilities**. It standardizes discovery and structured interaction, while leaving model behavior, business logic, authorization policy, and operational safety to the systems around it.

That boundary is valuable precisely because it lets each side evolve independently. It is also dangerous to misunderstand, because a clean protocol interface can hide powerful effects behind a simple tool name.

Chapter Summary

- MCP standardizes communication between an AI application and external capabilities.
- It does not define model reasoning, business policy, or security controls.
- Systems engineers should treat every MCP integration as real software with identities, dependencies, and failure modes.

Review Questions

1. What integration problem does MCP solve?
2. Which important responsibilities remain outside MCP?
3. Why does protocol standardization not make a capability safe by itself?

Chapter 2. Hosts, Clients, Servers, and Trust Boundaries

What You'll Be Able to Do

After completing this chapter, you will be able to:

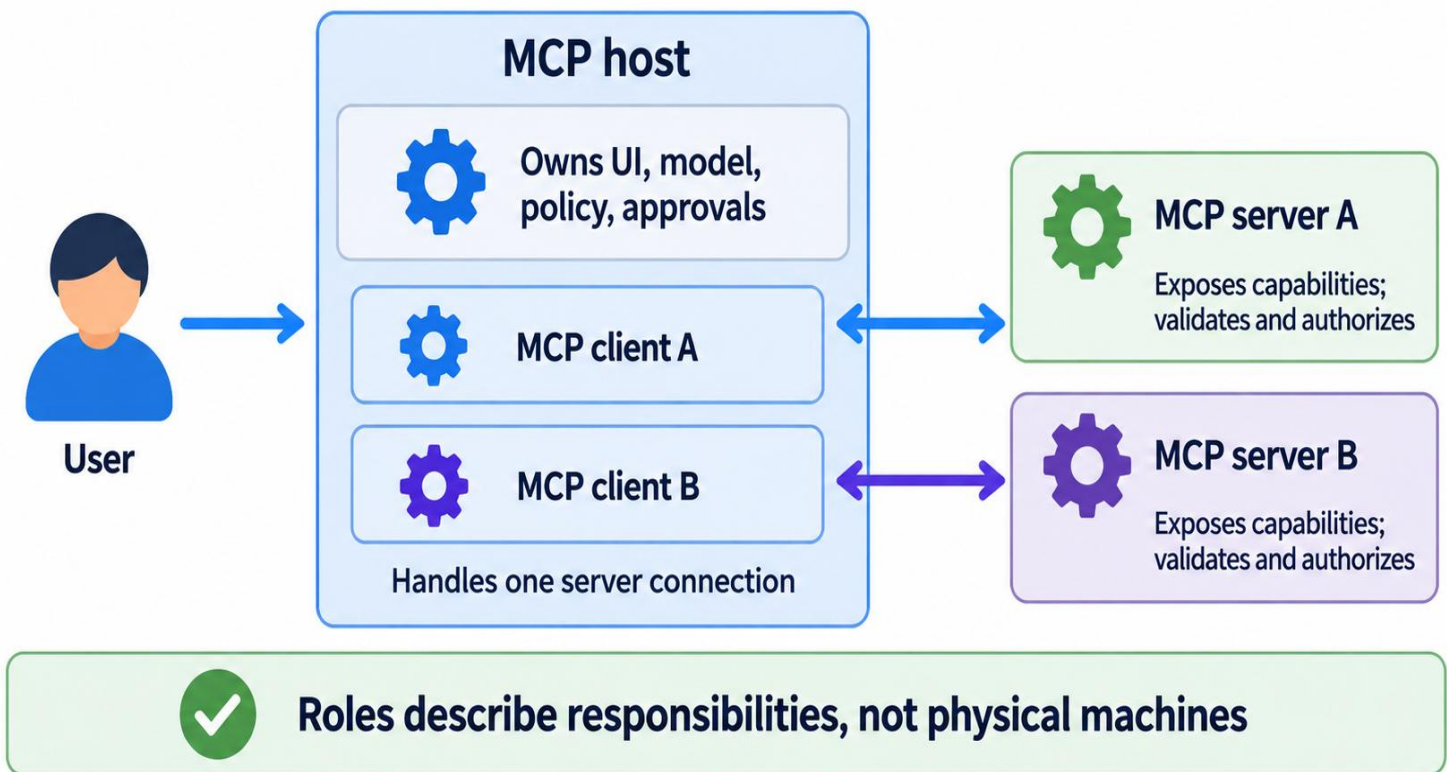
- Distinguish the host, client, and server roles without treating them as interchangeable.
- Separate the data layer from the transport layer.
- Map local and remote trust boundaries and assign responsibilities to the correct component.

Start with the roles

A user interacts with a host; the host creates an MCP client for each connected server; the server exposes capabilities.

Chapter 2: Hosts, Clients, Servers, and Trust Boundaries

2-1 Start with the roles



006

Figure 2-1: Start with the roles

These three terms describe protocol roles, not necessarily three separate machines.

Host

An **MCP host** is the AI application that owns the user experience and coordinates one or more MCP connections. It may be a desktop assistant, an IDE, an operations console, or an internal application.

The host typically decides:

- which servers are configured or allowed,
- which capabilities are shown to the user or model,
- which retrieved data enters model context,

- when to request approval,
- and how results appear in the interface.

The host also contains the model-facing orchestration that MCP does not standardize.

Client

An **MCP client** is the protocol component inside the host that communicates with one MCP server. It sends requests, receives results, tracks request identifiers, handles capability information, and uses a transport implementation.

The client is not normally the whole application. It is one connector managed by the host.

Server

An **MCP server** is a program that exposes capabilities through MCP. A server may provide data, executable operations, reusable prompts, or a combination of them.

A server can be:

- a local process launched by the host,
- a remote service shared by many clients,
- a thin adapter in front of an existing API,
- or a larger application with its own storage and downstream dependencies.

The word *server* describes the protocol role. It does not imply that the program runs on a separate physical server.

One client per server connection

A host commonly connects to several servers. It creates a separate MCP client connection for each one.

Chapter 2: Hosts, Clients, Servers, and Trust Boundaries

2-2 One client per server connection

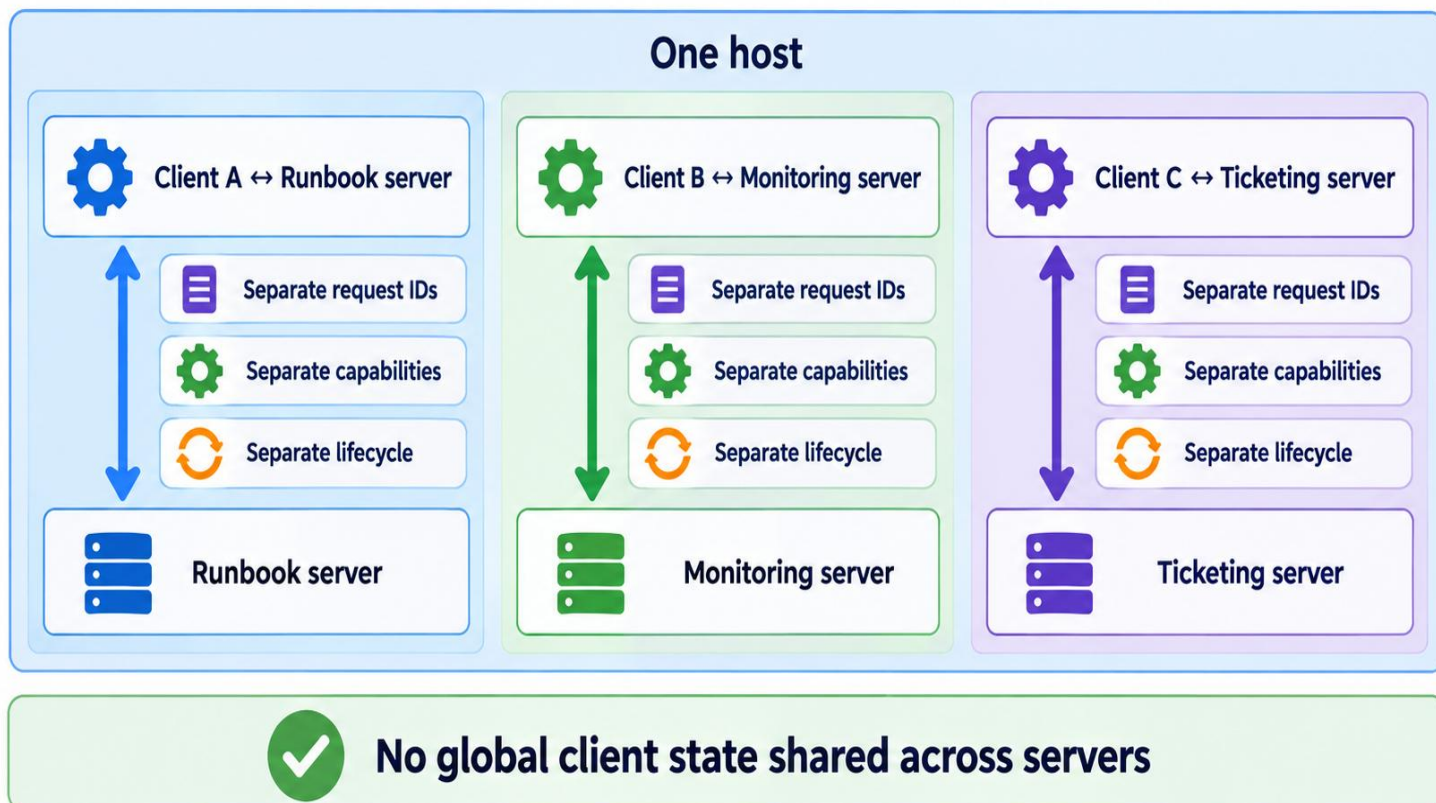


Figure 2-2: One client per server connection

This architecture has several consequences.

First, **host** and **client** are not interchangeable. The host coordinates the overall experience; each client handles one server connection.

Second, a host may combine capability lists from multiple servers. If two servers both expose a tool named `search`, the host needs a stable way to preserve server identity. A user interface might display names such as “Runbooks: search” and “Tickets: search,” while an internal registry may associate each tool with both a server identifier and a tool name.

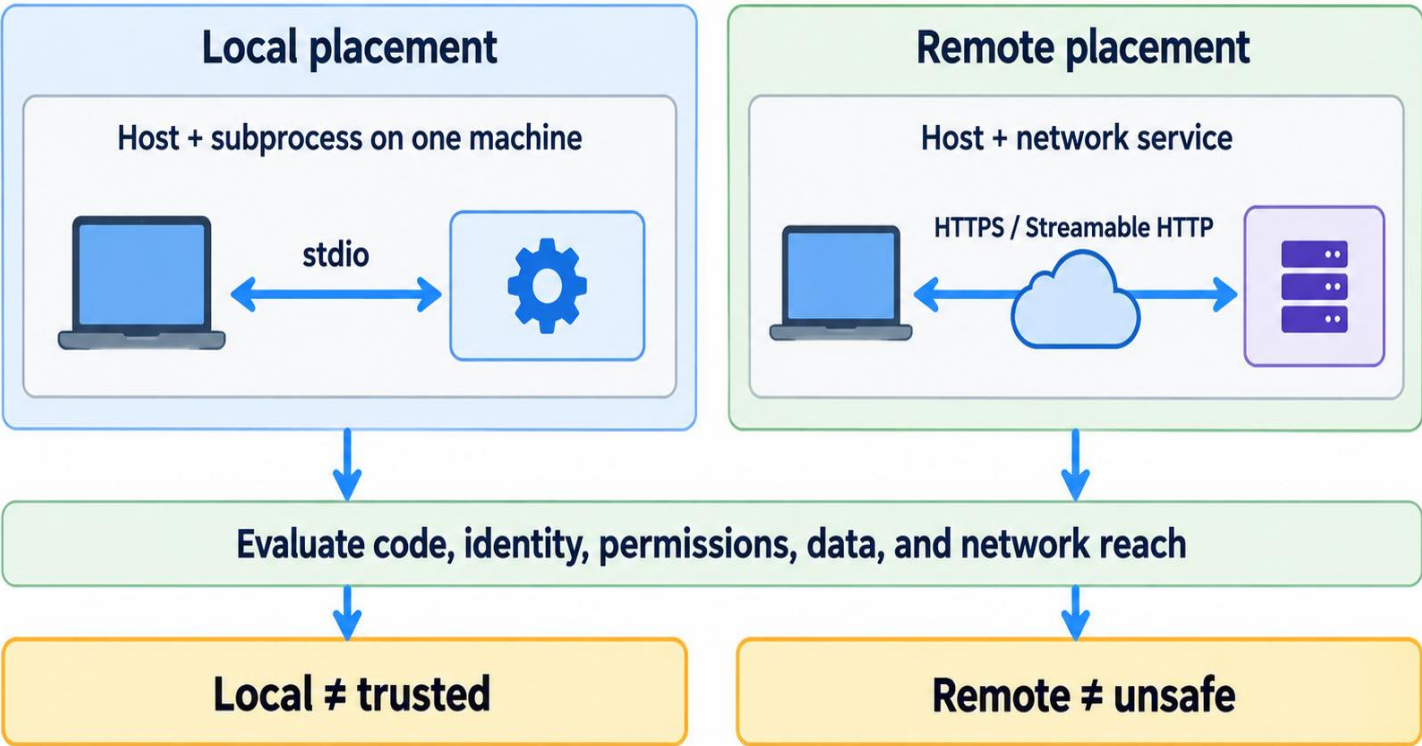
Third, one server should not assume it knows what other servers the host has connected. Coordination across servers is normally a host responsibility.

Local and remote describe placement, not trust

A local MCP server usually runs as a subprocess on the same machine as the host. A remote MCP server runs behind a network endpoint.

Chapter 2: Hosts, Clients, Servers, and Trust Boundaries

2-3 Local and remote describe placement, not trust



008

Figure 2-3: Local and remote describe placement, not trust

Property	Local server	Remote server
Typical transport	stdio	Streamable HTTP
Typical client count	One host-launched client	Many independent clients
Main boundary	Process and operating system	Network and service boundary
Common credential source	Process environment or local secret mechanism	HTTP authorization mechanism
Common operational risk	Package, process, filesystem, environment access	Network exposure, identity, gateway, multi-tenancy

These are typical patterns, not universal rules.

A local process can be highly privileged. It may inherit environment variables, read files available to the user, execute child processes, or access local network services. A remote service can be tightly isolated and strongly authenticated. **Location alone does not establish trust.**

The two protocol layers

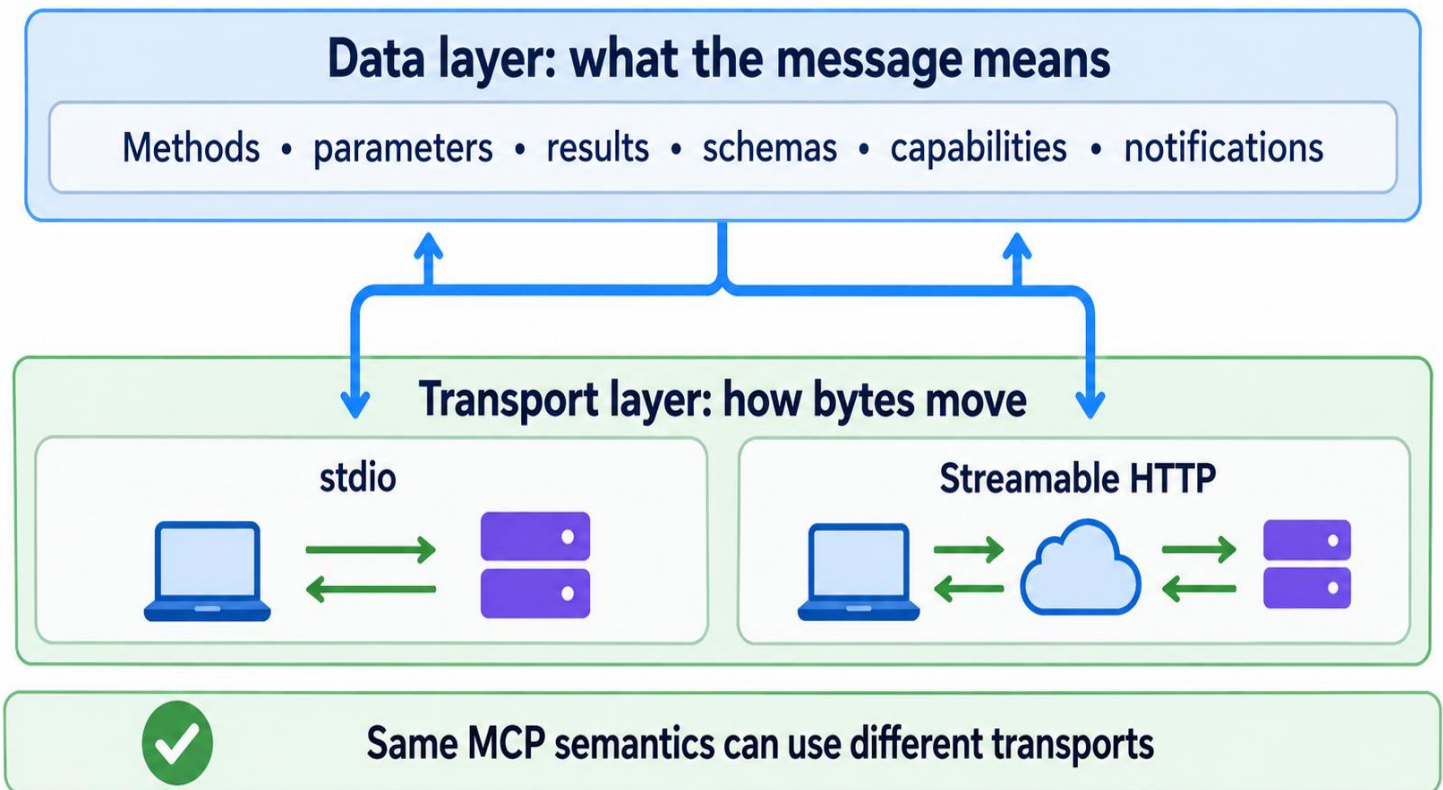
MCP separates the meaning of messages from the channel that carries them.

Data layer

The **data layer** defines the JSON-RPC methods, parameters, results, schemas, capability declarations, and notifications.

Chapter 2: Hosts, Clients, Servers, and Trust Boundaries

2-4 The two protocol layers



009

Figure 2-4: The two protocol layers

Examples include:

- `server/discover`,
- `tools/list`,
- `tools/call`,
- `resources/read`,
- `prompts/get`.

When an engineer asks, “What does this request mean?” the answer usually belongs to the data layer.

Transport layer

The **transport layer** defines how messages move between the client and server. It covers message framing, process streams or HTTP behavior, connection handling, and transport-specific security.

When an engineer asks, “How does the message reach the server?” the answer belongs to the transport layer.

The separation can be compared to sending a parcel:

- The data layer is the contents, labels, and agreed form inside the parcel.
- The transport layer is the delivery service that moves it.

The same data-layer request can be carried by different transports without changing its core purpose.

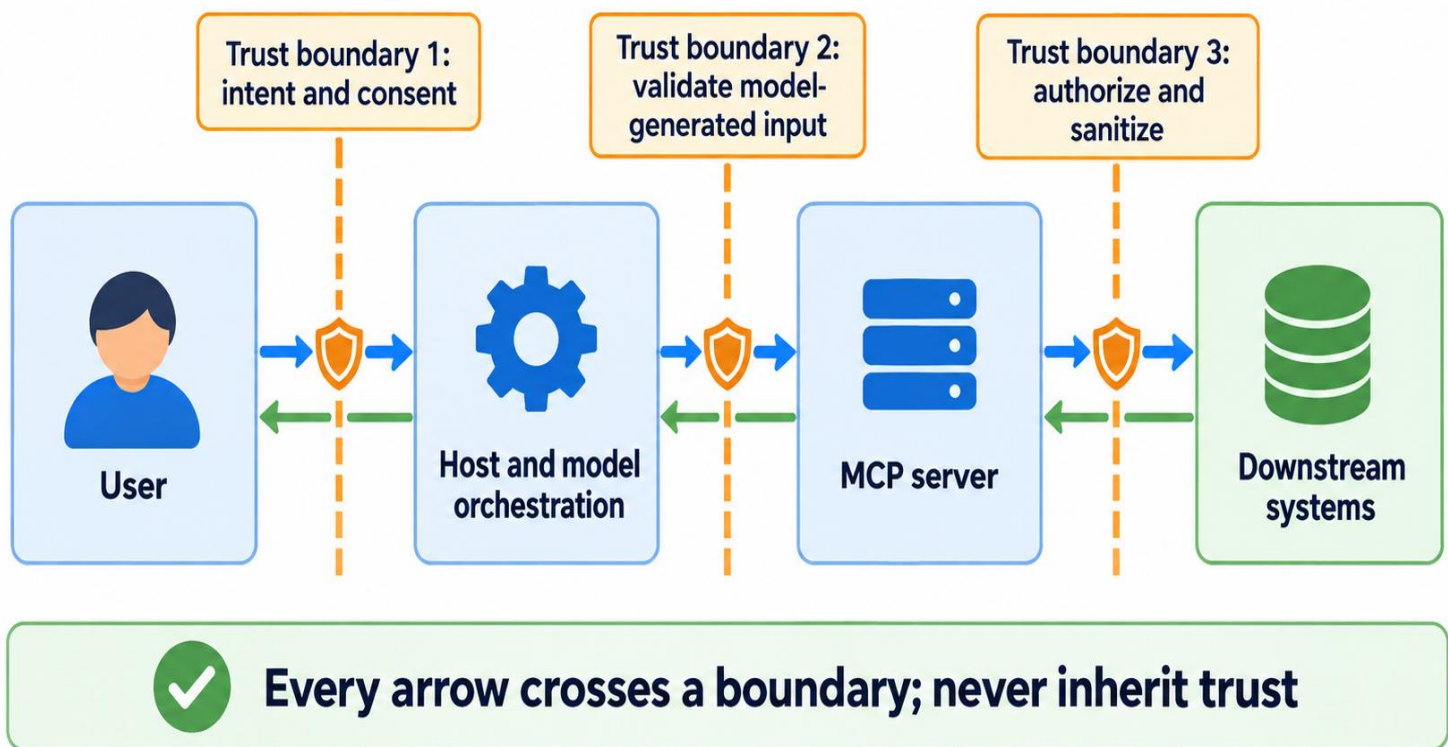
Every connection crosses trust boundaries

A **trust boundary** is a point where data or control passes between components that should not automatically trust one another.

In the operations-assistant example, important boundaries include:

Chapter 2: Hosts, Clients, Servers, and Trust Boundaries

2-5 Every connection crosses trust boundaries



010

Figure 2-5: Every connection crosses trust boundaries

Each arrow deserves explicit review.

User to host

The host receives the user’s request and may display proposed operations. Risks include ambiguous consent, misleading summaries, and accidental disclosure of user data to a server.

The host should make the requesting server and proposed effect visible. It should not turn a vague user statement into a privileged action without an appropriate control step.

Host to local subprocess

The host may launch code from a package or executable and pass it arguments, a working directory, and environment variables. Risks include malicious packages, replaced binaries, inherited secrets, excessive file access, and command injection in launch configuration.

Host to remote endpoint

The host sends requests over a network. Risks include endpoint impersonation, token leakage, incorrect authorization audience, insecure redirects, proxy misconfiguration, and unexpected data retention by the remote service.

Server to downstream systems

The MCP server may call a monitoring API, database, deployment controller, or operating-system command. This is often the most consequential boundary because the server converts protocol input into real effects.

Model-generated input to business logic

Even when the server itself is trusted, its inputs may be proposed by a model. The model can misunderstand the user, select the wrong target, produce malformed values, or be influenced by untrusted content. **Model-generated arguments must be treated as untrusted input.**

Responsibility does not stop at the SDK

An SDK can implement message serialization, method dispatch, schema helpers, transport behavior, and common error handling. It cannot take ownership of the entire security and operational design.

A useful responsibility split is:

Participant	Primary responsibilities
Host	Server selection, user experience, context control, approval presentation, client lifecycle
MCP client component	Protocol encoding, request correlation, capability handling, transport integration
MCP server	Input validation, business rules, authorization enforcement, safe downstream access, result sanitization
Infrastructure	Process isolation, network controls, secret delivery, deployment, monitoring, recovery
Organization	Access policy, ownership, change management, audit requirements, incident response

Responsibilities may overlap, but no layer should assume another layer has already enforced every control.

For example, the host may validate a service argument against a displayed schema. The server must still validate it because a different client may call the same server, a client may be compromised, or the value may change between validation and execution.

Map the operations assistant

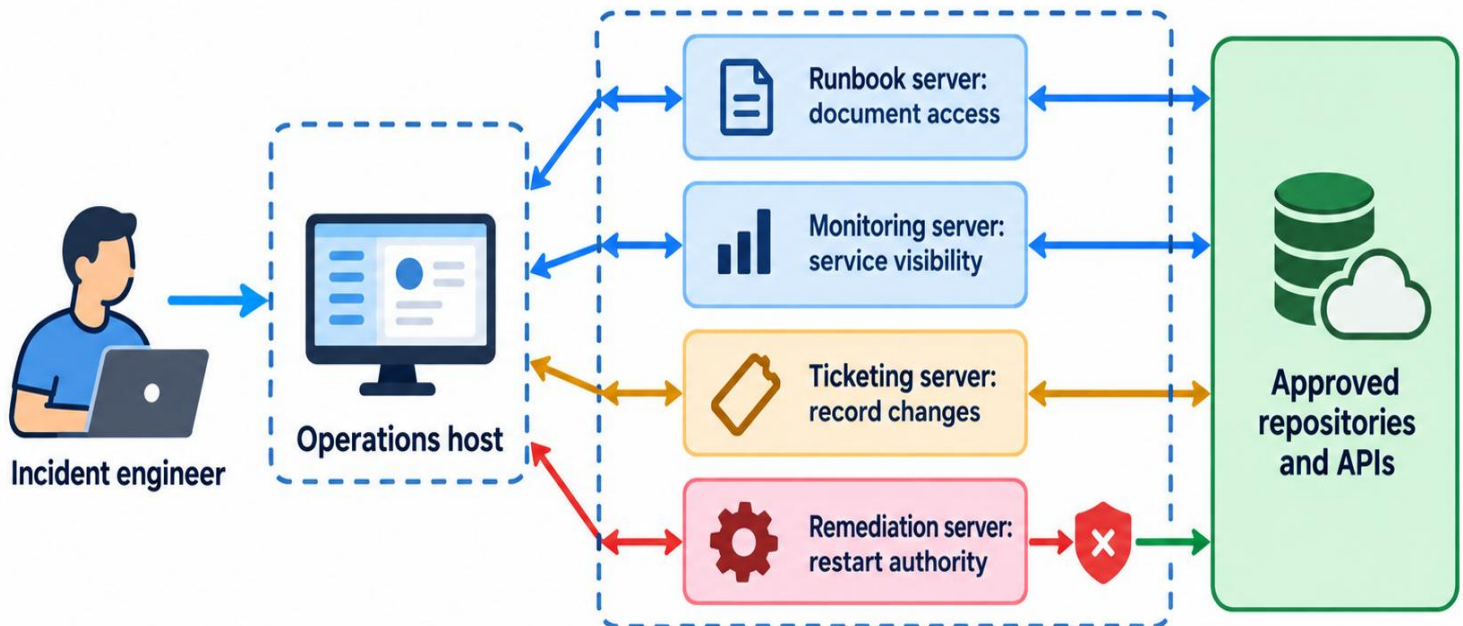
A simple trust-boundary inventory for the continuing example might look like this:

Component	Data or authority handled	Main question
Host	User request, model context, approvals	What is shown to the model and user?
Runbook client/server	Internal procedures	Which documents may this user read?
Monitoring client/server	Service metrics and status	Which services and environments are visible?
Ticketing client/server	Incident notes and change records	Who may create or modify records?
Remediation server	Restart authority	Which targets can be changed, under what approval?
Downstream APIs	Operational systems and credentials	Are permissions narrower than the server's total function?

The diagram and table do not solve the security problem. They make the problem visible enough to solve.

Chapter 2: Hosts, Clients, Servers, and Trust Boundaries

2-7 Map the operations assistant



Each path has its own identity, policy, and audit trail

011

Figure 2-6: Map the operations assistant

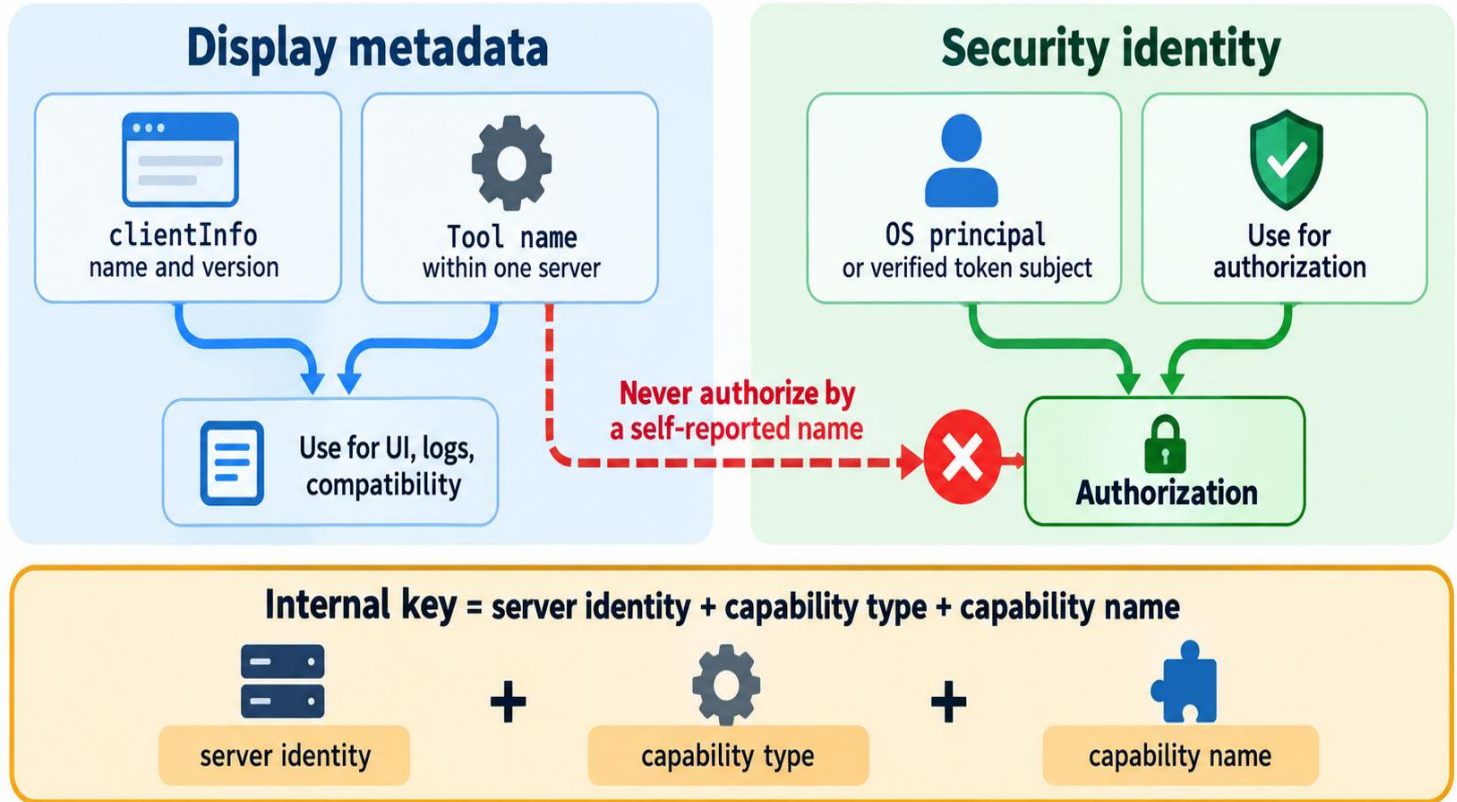
Names and identity require care

MCP implementation metadata can include client and server names and versions. These values are useful for display, logging, and compatibility diagnosis. They are self-reported and should not be treated as authenticated security identities.

Similarly, a tool name identifies an operation within a server's namespace. It does not prove that two tools with the same name have the same behavior. The host must preserve the server association when presenting or invoking a capability.

A useful internal identifier is conceptually:

2-8 Names and identity require care



012

Figure 2-7: Names and identity require care

not only:

The local-server misconception

A frequent misconception is:

A local MCP server is automatically trusted because it runs on the same machine.

The correct conclusion is:

A local server replaces some network risks with process, package, filesystem, and environment risks.

Before approving a local server, inspect its source or provenance, its launch command, the identity under which it runs, the directories it can access, the secrets it receives, and the network destinations it can reach.

The architectural takeaway

The MCP architecture is simple only at the highest level: host, client, and server. Operationally, each connection crosses several trust boundaries and leads to downstream systems with their own authority.

The most important distinctions are:

- the host is not the same as an individual MCP client,
- a server role does not imply a separate machine,
- the data layer is not the transport layer,
- and local execution is not automatic trust.

These distinctions make the message model in the next chapter much easier to reason about.

Chapter Summary

- The host owns the user experience, each client connection speaks to one server, and each server exposes capabilities.
- Local and remote describe placement, not trustworthiness.
- Trust boundaries and enforcement responsibilities must be mapped explicitly.

Review Questions

1. How do the host, client, and server roles differ?
2. Why can a local server still cross important trust boundaries?
3. Which component must enforce authorization for a privileged operation?