

MCP (Model Context Protocol) — Overview, Architecture, How-To, and RAG Integration

MCP is an open standard for connecting AI applications to external tools and data sources via “MCP servers”. It standardizes discovery and access to Tools, Resources, and Prompts, so clients (chat apps, IDEs, agent runtimes) can plug into servers consistently—often described as a “USB-C for AI”. [1][2]

1 — OVERVIEW

Overview

Overview

MCP (Model Context Protocol) gives AI applications a standard way to talk to external tools, data, and prompts. With MCP, integrations become reusable “capability servers” that any MCP client can consume. Use the tabs above to jump to Core concepts, Transports, Architecture, How to use MCP, RAG + MCP, and Security.

2 — CORE CONCEPTS

Core concepts

Core concepts

Primitives

Tools

Callable actions with defined inputs/outputs (e.g., “search tickets”, “create issue”, “query view”).

Resources

Readable context the client can fetch and choose to include (docs, records, snapshots).

Prompts

Reusable prompt templates published by the server (triage, analysis, drafting patterns).

MCP defines a **client-server** architecture: the AI application is the **MCP client**, and each integration is an **MCP server** that exposes a catalog of tools/resources/prompts. This makes capability discovery and invocation consistent across clients. [2][1]

Client: chat / IDE / agent runtime

Server: tools + resources + prompts

Safer boundary for permissions

Bonus: the MCP ecosystem also includes an MCP Registry concept (like an app store of servers) which helps discovery of publicly available servers. [8]

3 — HOW IT CONNECTS

How it connects

Transports
MCP supports different transport mechanisms depending on the environment and deployment style.

stdio
Great for local integrations & CLIs; client spawns server process and communicates via stdin/stdout. [3]
HTTP-family transports

Common for production/shared services (remote servers). Many SDKs recommend HTTP for production use cases. [4]
Rule of thumb: use **stdio** when the server is local to the user (IDE workflows, local file access), and use **HTTP-style transports** for shared enterprise servers (central auth, logging, scaling). [3][4]

Transport selection (conceptual) if server_runs_on_user_machine: transport = "stdio" # local, simple, fast else: transport = "http / streamable http / sse" # production, shared
For debugging and validating your server schemas and responses, the MCP Inspector is a common dev tool. [5][6]

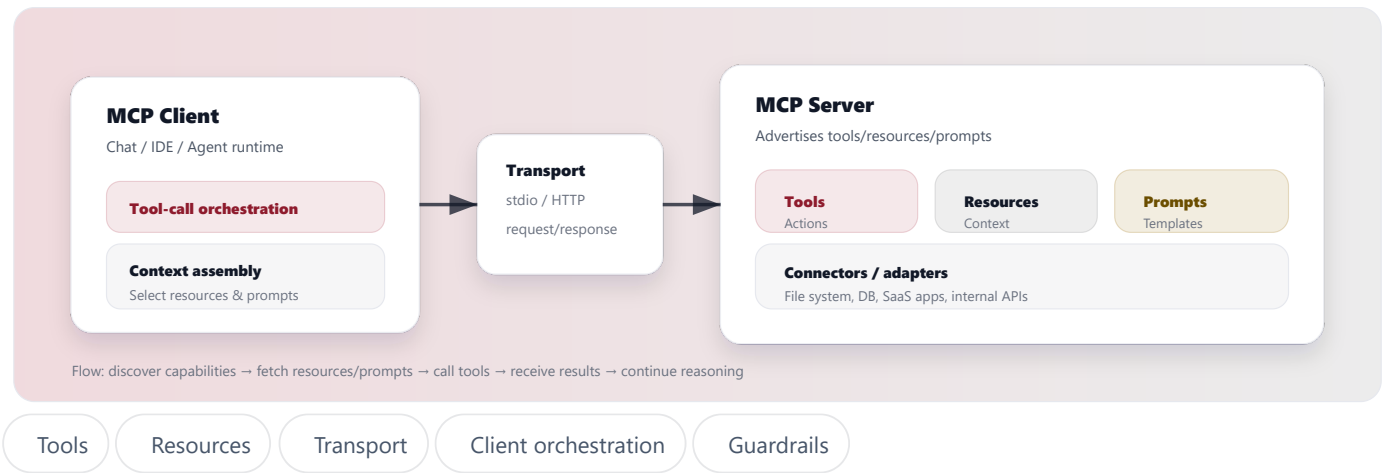
Architecture

Architecture

End-to-end
MCP sits between the AI client and your systems, standardizing discovery and calls.

Orchestration stays in client

Capability lives in servers



This client–server split is the key design point: **clients decide** what context to attach and when to call tools; **servers own** the integration logic and enforce boundaries. [2][1]

How to use MCP

Playbook

Step 1

Pick your first “high ROI” integration (KB, tickets, filesystem, metrics).

Step 2

Expose minimal tools + curated resources (read-only first).

Step 3

Test with MCP Inspector / dev client and harden security.

Start small: define a server that exposes a handful of **tools** and a few **resources**. Validate schemas and behavior with the **MCP Inspector**, then broaden capability surface gradually. [5][6]

Minimal MCP server contract (conceptual) server: "kb-and-tickets" tools: - name: "search_kb" input: { query: string } output: { hits: [{ title: string, url: string, snippet: string }] } - name: "search_tickets" input: { query: string, status?: string } output: { results: [{ id: string, title: string, status: string }] } resources: - uri: "kb://policies/ai-usage" description: "Approved AI usage policy" prompts: - name: "incident_triage" template: "Use search_tickets first. Then summarize risks and next steps..."

If you want a catalog of reference servers and examples, the MCP docs provide an “Example Servers” list. [7]

RAG + MCP

Example integration

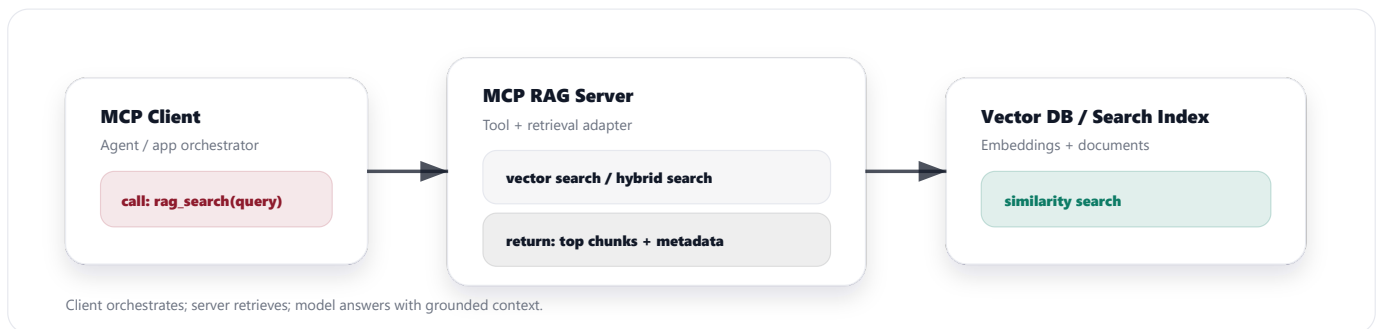
MCP is a clean way to “package” RAG as a reusable capability: the client calls a retrieval tool, receives top documents/chunks, then composes the final answer.

Tool: retrieve

Resource: passages

Output: citations-ready context

A standard RAG loop can be expressed through MCP as: **(1)** client asks the MCP server to retrieve relevant context, **(2)** server queries a vector database (or search index), **(3)** server returns chunks + metadata, **(4)** client injects those chunks into the model prompt and generates the answer. This isolates retrieval logic from the AI client while keeping orchestration in the client. [1][2]



Concrete MCP pattern for RAG: expose a single tool (e.g., `rag_search`) that takes a query and returns a bounded list of passages with doc IDs/URLs/titles. Some RAG MCP implementations explicitly require a `rag_search` tool. [9]

RAG over MCP (conceptual) tool: "rag_search" input: query: string k?: number filters?: object output: passages: - text: string source_id: string title?: string url?: string score?: number # Client loop 1) passages = call rag_search(user_question) 2) answer = LLM(user_question + passages as context) 3) include citations from (source_id/url/title)

If you want to implement quickly, there are community write-ups on building an MCP “RAG server” pattern (use as inspiration, then harden for security and governance). [10]