Day 4: Tools, MCP & Agents

Function Calling, Model Context Protocol, and Autonomous Tool-Using Systems

Isham Rashik · AI Engineer
Engineering AI with Clarity
NLP • Computer Vision • Fine-Tuning • RAG • Agentic AI

Version: v1.0 · April 25, 2026

Acknowledgment

I am deeply grateful to Dr. Sreedath Panat and **Vizuara Technologies** for creating and generously sharing the *Context Engineering* course. These notes would not exist without his exceptional teaching, clear explanations, and dedication to making cutting-edge AI concepts accessible to everyone.

Thank you, Dr. Panat, for the time, effort, and passion you have poured into this course. Your work has been instrumental in shaping my understanding of context engineering, tool use, and the Model Context Protocol.

I also extend my gratitude to Nitish Singh and the **CampusX** team for their outstanding MCP deep-dive series. Their detailed walkthroughs of MCP architecture, transport layers, the lifecycle protocol, connectors, and hands-on demos (newsletter automation, Manim, Blender) provided invaluable supplementary context that enriched these notes significantly.

→ LLM Context Engineering Bootcamp

→ CampusX: MCP Deep Dive Series

Contents

1 Prerequisites

- Understanding of context windows, context rot, and the six core elements (Day 1)
- Familiarity with `CLAUDE.md`, system prompts, and RAG (Days 2-3)
- Understanding of the WSCI framework: WRITE, SELECT, COMPRESS, ISOLATE [1] (Day 3)
- Basic understanding of APIs and JSON format
- Familiarity with client-server architecture concepts

2 Overview

This guide covers **tools**, the **Model Context Protocol (MCP)**, and **agents** — the three pillars that turn LLMs from text predictors into systems that can perceive, reason, and act on the real world. It begins with the context assembly problem that made developers "human APIs," then introduces tool schemas, the five-step tool life cycle pipeline, and strategies for scaling tool descriptions with RAG. The core of the guide is a comprehensive walkthrough of MCP: the N × M integration problem it solves, the three-layer architecture (Host, Client, Server), the three primitives (tools, resources, prompts), JSON-RPC 2.0 messaging, transport layers, the three-stage lifecycle, error handling, server design principles, and connectors vs. JSON configuration. It then covers the MCP development ecosystem (FastMCP, MCP Inspector, LangChain MCP Adapters) and grounds the theory with three real-world demos: Blender 3D animation, a multi-source AI newsletter pipeline, and Manim mathematical visualizations. The guide closes with Just-In-Time instructions for context efficiency, four tool execution patterns (sequential, parallel, conditional, iterative), and two agent architectures — the Tool Calling Agent and the ReAct Agent — setting the stage for multi-agent orchestration in Day 5.

3 Why LLMs Need Tools

Without tools, an LLM is a very sophisticated text predictor. With tools, it becomes an agent that can perceive the world, reason about it, and act on it. LLMs are next-token prediction models: despite emergent capabilities that improve with scale, they have three fundamental limitations that tools address.

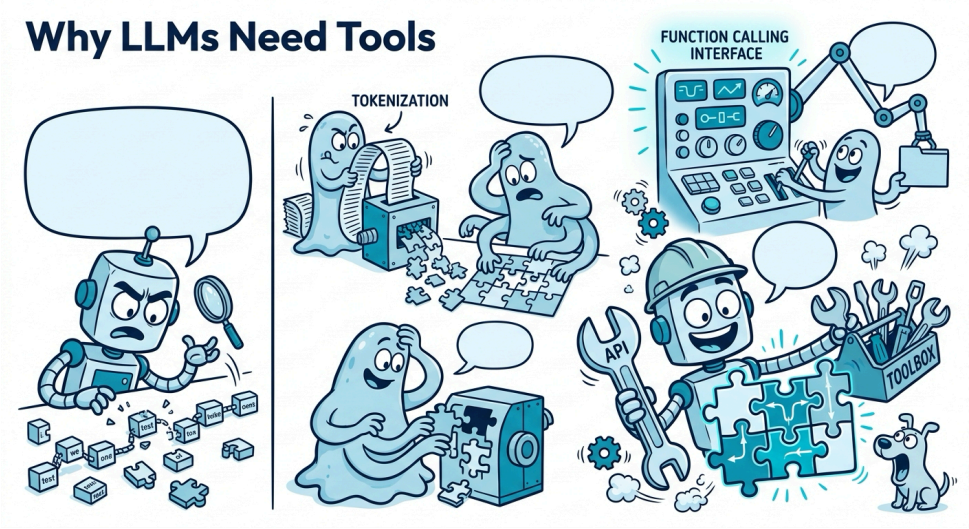

Figure 1: Smart enough to plan dinner, but without hands to cook it: LLMs need tools to act on the world

3.1 The Three Fundamental Limitations

Table 1: Three fundamental limitations of LLMs that tools solve

Limitation	Description	Example
No access to live data	LLMs have no connection to the internet or real-time databases	Cannot fetch current weather, stock prices, or exchange rates
Unreliable computation	LLMs are not deterministic calculation engines; they can hallucinate mathematical results	7321 × 8761 may produce errors; a calculator tool gives a deterministic answer
No real-world effects	LLMs cannot independently take actions in external systems	Cannot send an email, place an Amazon order, post to Slack, or create a 3D model

> **Key Insight**
>
> LLMs do not *execute* tools. LLMs *decide* which tool to call and produce a structured response specifying the tool name and arguments. The actual execution is performed by the tool's own logic.

3.2 The Context Assembly Problem: Developers as Human APIs

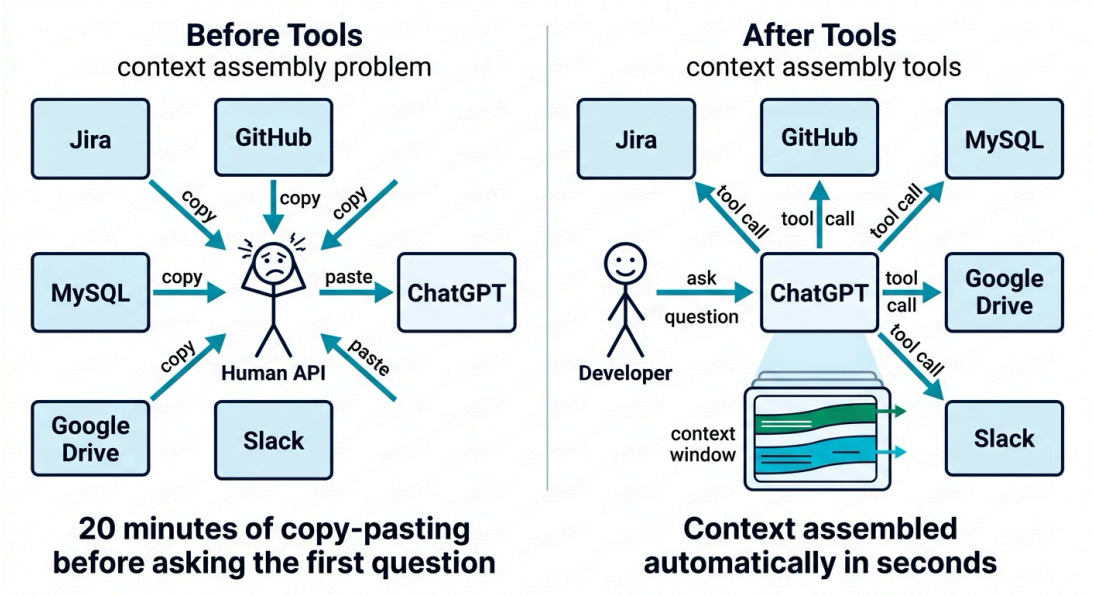

Figure 2: Before tools: 20 minutes of copy-pasting across five apps. After tools: one question and the LLM fetches everything itself

Before tools existed, using an LLM for real work meant *manually assembling context* from scattered sources. Consider a concrete scenario: a software engineer is assigned a ticket to implement two-factor authentication (2FA) for their company's product.

To develop this feature with AI assistance, the developer must:

1. **Fetch the ticket from Jira** — Copy the requirements, acceptance criteria, and implementation notes

2. **Pull the codebase from GitHub** — Copy the relevant authentication code from 10-12 files

3. **Study the database schema from MySQL** — Copy the existing user table structure and relationships

4. **Retrieve security guidelines from Google Drive** — Copy the company's security compliance document

5. **Check team discussions on Slack** — Copy relevant messages about the 2FA approach

6. **Paste everything into ChatGPT** — After 20 minutes of copy-pasting, finally ask the first question

In this workflow, the developer has become a **human API**: their entire job is assembling context for the LLM. More time is spent on context assembly than on actual development. Worse, this approach does not scale: a codebase with thousands of files cannot be manually summarized and pasted into a chat window.

> ⚠ **Warning**
>
> The context assembly problem is not just inconvenient — it is the fundamental bottleneck that prevented LLMs from becoming true development partners. Before tools, the human was the integration layer between every data source and the AI.

With tools, the same 2FA scenario transforms completely. ChatGPT is now connected to Jira, GitHub, MySQL, Google Drive, and Slack through tool integrations. The developer simply asks: *"I have a new ticket assigned. Help me implement two-factor authentication."* The LLM autonomously fetches the ticket details, pulls the relevant code, reads the database schema, retrieves the security guidelines, and checks Slack discussions — all through tool calls. The context that was previously scattered across five platforms is now assembled automatically in seconds.

3.3 Tools as the Solution

Tools bridge the gap between what an LLM can reason about and what it can accomplish:

- **Live data access:** Weather APIs, financial data APIs, database queries
- **Computation:** Calculator tools, code execution environments
- **Real-world actions:** Email sending (Gmail API), messaging (Slack MCP), 3D modeling (Blender MCP), file system operations

With the motivation established, the next question is: how do tool results fit into the context window?

3.4 Tool Results and Context Window Impact

Every tool call result becomes new context. When an LLM makes a tool call, the result is stored back in the context window. This has direct consequences for context engineering:

1. Tool results consume tokens from the context budget
2. Large tool responses (e.g., a 5,000-token JSON from Slack) can bloat the context
3. Tool descriptions themselves consume tokens: concise schemas are essential
4. The context must reserve space for tool results alongside RAG results, system prompts, and conversation history

Consider a typical context window **before** a tool call: the system prompt occupies 20%, conversation history 15%, RAG results 30%, leaving 35% available. After a single tool call returns a large result (occupying 25% of the window), only 10% remains for the LLM to reason and generate a response.

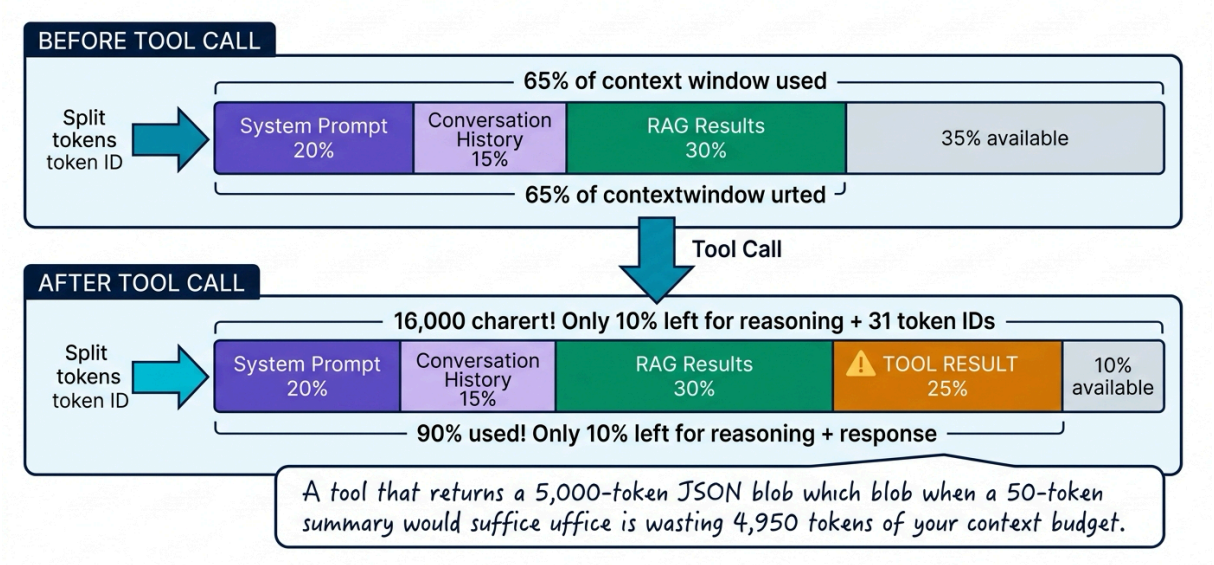

Figure 3: Before a tool call, 35% of the context window is free; after a large tool result consumes 25%, only 10% remains for reasoning and response generation

> **Danger**
>
> A tool that returns a 5,000-token JSON blob when a 50-token summary would suffice is wasting 4,950 tokens of your context budget. Design tool responses to be as concise as possible; return only the fields the LLM needs.

> **Warning**
>
> Tool results add to the context window. If a tool returns a massive JSON response, not all of it needs to enter the LLM's context. The MCP client is responsible for deciding what subset of tool results to pass to the LLM.

3.5 RAG vs. Tool Results

A common question is how RAG results differ from tool results. The distinction is fundamental.

Table 2: RAG retrieves from a static knowledge base; MCP tools access live dynamic data

Aspect	RAG Results	Tool Results (via MCP)
Data source	Pre-indexed documents stored as vector embeddings	Live data from APIs, databases, or external services
Update frequency	Static or infrequently updated (e.g., a 2025 shareholder report)	Real-time or near-real-time (e.g., current stock price, live weather)
Mechanism	Query → embed → cosine similarity → retrieve top-K chunks	Structured tool call → function execution → JSON response
LLM agnostic?	Depends on implementation	MCP is inherently LLM-agnostic by protocol design

Memorize

RAG retrieves from a *static knowledge base* (documents converted to embeddings). MCP tools access *live, dynamic data* that changes in real time. Both contribute context to the LLM, but they serve fundamentally different purposes.

With the motivation for tools and their context impact established, the next section covers how tools are described to the LLM so it can decide which to call.

4 Tool Schemas: Describing Tools to the LLM

A tool schema is a JSON description that tells the LLM what a tool does, what inputs it expects, and what outputs it returns. The schema is what enables the LLM to decide *which* tool to call and *how* to call it.

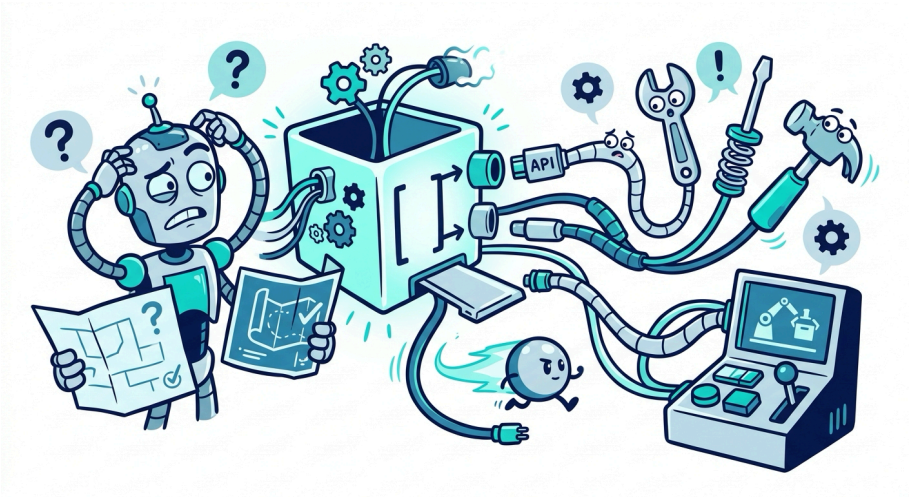

Figure 4: A JSON blueprint tells the LLM what each tool does, so it picks the right one instead of guessing

4.1 Anatomy of a Tool Schema

Example Tool Schema: search_products

```
{
  "name": "search_products",
  "description": "Search the product catalog from the
    e-commerce database. Returns top 50 matching items
    with price, availability, and category.",
  "parameters": {
    "type": "object",
    "properties": {
      "query": {
        "type": "string",
        "description": "Search keyword to match against
          product name, description, or SKU"
      },
      "category": {
        "type": "string",
        "enum": ["electronics", "clothing", "home",
                 "sports", "books"],
        "description": "Product category to filter results"
      },
      "max_results": {
        "type": "integer",
        "description": "Maximum number of results to
          return (default: 50)"
      }
    },
    "required": ["query"]
  }
}
```

The schema contains a **name** in snake_case (verb-noun format), a natural language **description**, typed **parameters** with constraints, and a distinction between **required** and **optional** fields.

4.2 Tool Schema Design Principles

Six principles govern good tool schema design:

Table 3: Six principles for tool schema design

Principle	Description	Example
Verb-noun naming	Tool name follows action_entity format in snake_case	search_missions, get_weather_forecast, calculate_fuel_estimate
Clear description	Explain what the tool does, what it returns, and when to use it	Search the AstroLog mission database by keywords, status, etc. Returns matching missions with status, crew, and destination.
Enumerate constraints	Use enum for categorical parameters instead of free-text strings	enum: [planned, in_transit, completed, delayed, aborted]
Required vs. optional	Only mark truly required parameters as required; allow defaults	Marking everything as required forces the LLM to hallucinate values for unknown fields
Return descriptions	Document what the tool returns so the LLM can interpret results	returns: { “missions”: [{ “id”: “”, “name”: “”, “status”: “” }] }
Low parameter count	Keep parameters under 5; combine related fields into objects	More parameters = more chances for the LLM to make errors

> 🔥 **Tip**
>
> Enumerating constraints is the single most impactful principle. If a parameter has 5 valid values, listing them explicitly in the schema prevents the LLM from hallucinating a sixth. Free-text strings invite creativity; enums enforce correctness.

4.3 Vague vs. Well-Defined Schema

Vague Schema

- Name: getMissionData (camelCase, ambiguous)
- Description: Gets mission data
- Parameters: q: { "type": "string" } (cryptic)
- Constraints: status: { "type": "string" } (no enum)
- Token cost: ~1/3 of well-defined

Well-Defined Schema

- Name: search_missions (snake_case, verb-noun)
- Description: Search the AstroLog mission database by keywords, status, etc.
- Parameters: query: { "type": "string", "description": "Search keyword..." }
- Constraints: status: { "enum": ["planned", "in_transit", "completed", "delayed", "aborted"] }
- Token cost: ~3x more but dramatically more useful

Memorize

A well-defined schema costs more tokens but prevents hallucination, reduces errors, and enables the LLM to make correct tool calls on the first attempt. This is a worthwhile trade-off.

With schemas defined, the next question is how the LLM actually uses them to interact with tools.

5 The Tool Life Cycle Pipeline

Every tool interaction between an LLM and an external function follows a fixed five-step pipeline. Understanding this pipeline is essential because each step has a distinct owner (the LLM, the client, or the tool itself), and errors at any stage propagate downstream. The pipeline applies universally, whether using raw function calling or MCP.

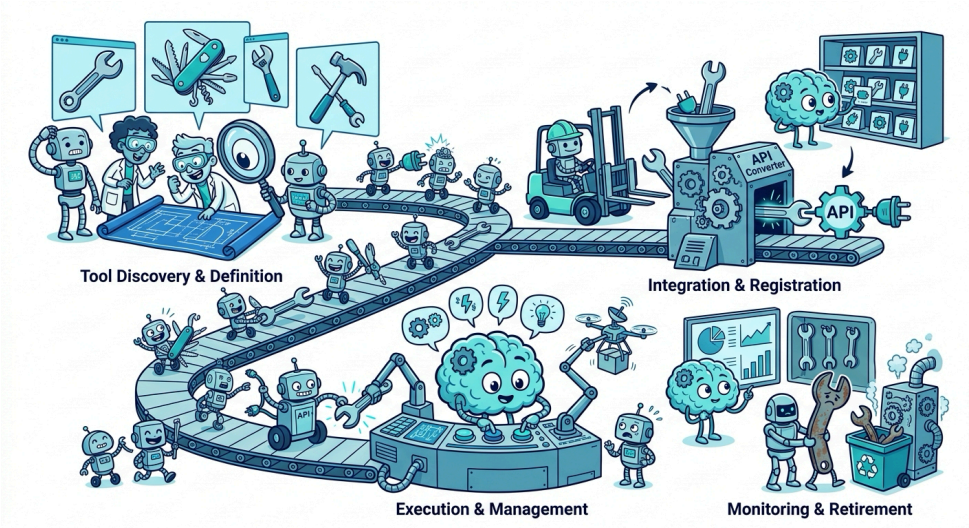

Figure 5: DESCRIBE, DECIDE, CALL, RETURN, REASON: every tool interaction follows the same five-step dance

5.1 Five Steps of Tool Interaction

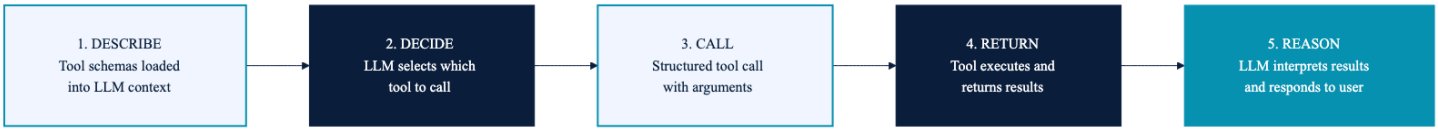

Figure 6: The five-step tool life cycle: DESCRIBE, DECIDE, CALL, RETURN, REASON

1 **DESCRIBE** — Tool schemas (names, descriptions, parameter definitions) are loaded into the LLM's context

2 **DECIDE** — The LLM reads the user query and available schemas, then decides which tool(s) to call

3 **CALL** — The LLM produces a structured output specifying the tool name and arguments (hidden from the user)

4 **RETURN** — The tool executes and returns its result

5 **REASON** — The LLM reasons over the tool result and produces a natural-language response for the user

> ⚠ **Warning**
>
> The tool schema most heavily impacts **Step 2 (DECIDE)**. A poor schema causes the LLM to pick the wrong tool or pass incorrect arguments. The rest of the pipeline inherits that error.

Each stage of the pipeline raises a distinct context engineering question. The following table maps each stage to the question it poses and the strategy that addresses it:

Table 4: Context engineering questions and strategies at each stage of the tool life cycle

Stage	Context Engineering Question	Strategy
DESCRIBE	How many tool descriptions can fit in my token budget?	Keep descriptions concise; use RAG for tool selection at 50+ tools
DECIDE	Will the model pick the right tool?	Clear descriptions, distinct names, constrained enums
CALL	Will the model generate valid arguments?	Strong typing, required fields, examples in descriptions
RETURN	Is the result too large for the context window?	Compress results, return summaries not raw data
REASON	Does the model know how to interpret the result?	Include interpretation instructions

5.2 Who Runs the Tool?

This is a critical distinction that many practitioners get wrong:

- **The LLM does NOT execute tools.** The LLM produces a structured JSON describing which tool to call and what arguments to pass.
- **The tool's own logic executes the function.** The structured output from the LLM is used to invoke the function programmatically.
- The LLM produces **two types of output** during a tool interaction: (1) a structured tool call (hidden from user), specifying tool name and arguments, and (2) a natural language response (shown to user), the final answer after reasoning over tool results.

With the tool pipeline understood for a small number of tools, the next challenge is: what happens when there are hundreds of tools?

6 Scaling Tool Descriptions: RAG for Tool Selection

The tool life cycle assumes the LLM can see all available tool schemas at once. This works for a handful of tools, but modern production systems are exposed to 50, 100, or even 200+ tools. Loading every schema into the system prompt creates severe context bloat, leaving little room for the actual conversation, RAG results, or tool responses. The solution borrows directly from RAG: instead of loading all tool descriptions upfront, retrieve only the relevant ones for each query.

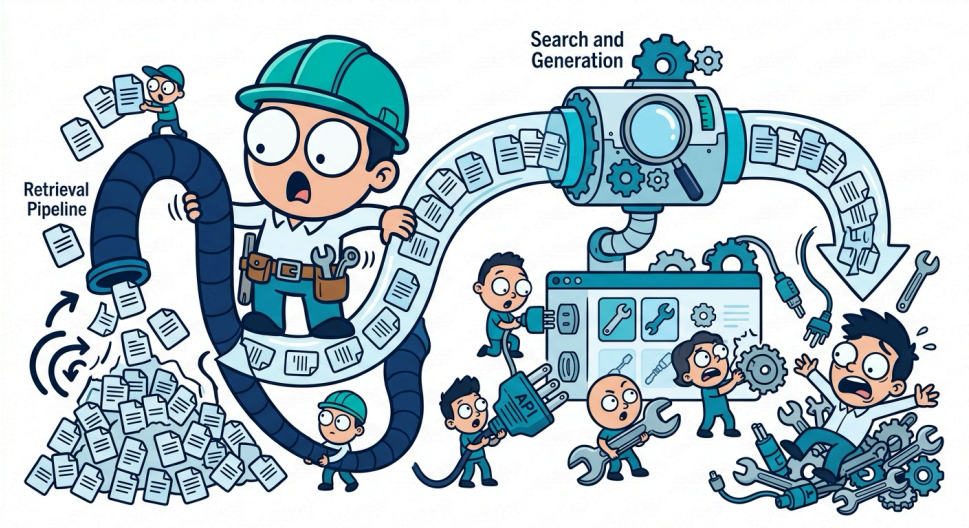

Figure 7: 200 tool descriptions walk into a context window and nobody fits: use RAG to load only what you need

6.1 The Problem: Too Many Tools

Each tool description consumes approximately **100 to 200 tokens**. With scaling, the context window fills quickly.

Table 5: Tool count vs. context impact

Number of Tools	Token Consumption	Impact
5-10	500-2,000 tokens	Acceptable: include all in system prompt
50-100	5,000-20,000 tokens	Context bloat: need selective loading
200+	20,000+ tokens	Severe context rot: mandatory RAG for tool selection

The solution: use RAG (vector database or keyword matching) on tool descriptions to load only the **top 5-10 most relevant** tool schemas into the LLM's context for the current task.

6.2 Grouped Tool Descriptions

For complex software (e.g., Slack with 3,000+ functions), organize tools into **named groups**:

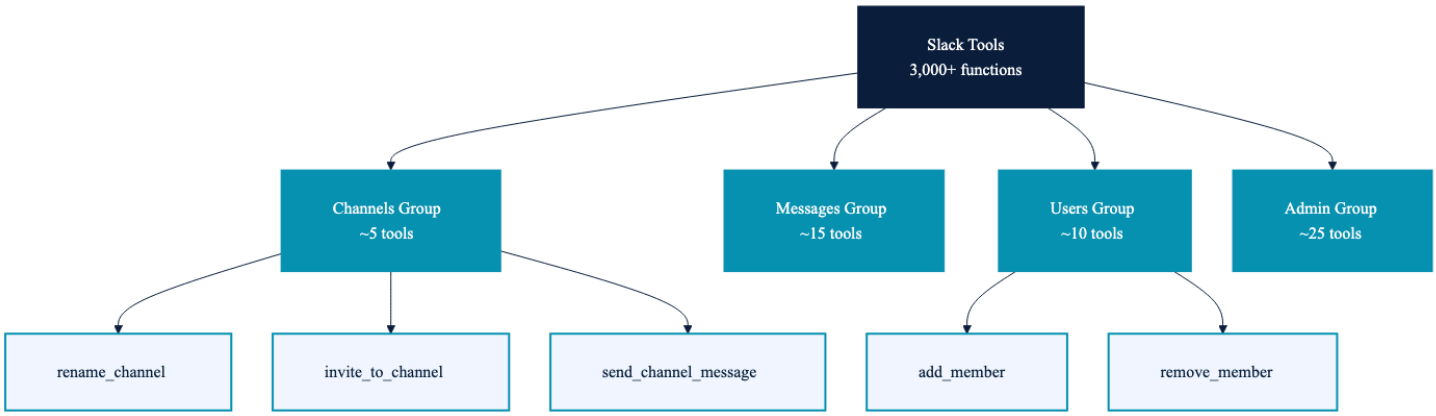

Figure 8: Grouped tool descriptions: the LLM loads only the relevant group

Instead of loading all 3,000 Slack function descriptions, the LLM first identifies the relevant group (e.g., Channels) based on the current task context. Only the tools within that group (5 tools) are loaded. As the task evolves (e.g., after adding a user to a channel, needing to notify an admin), the LLM can switch groups, loading Admin tools while unloading Channels tools. JIT instructions from the last tool result can guide the LLM toward the next relevant group.

With tool schemas and scaling strategies covered, the guide now turns to the protocol that standardizes all LLM-tool communication: MCP.

7 Model Context Protocol (MCP)

The **Model Context Protocol (MCP)** is a standardized protocol for LLM-tool communication. It became one of the most discussed topics in the AI community in 2025 and has since matured from a trend into a core infrastructure component for agentic applications.

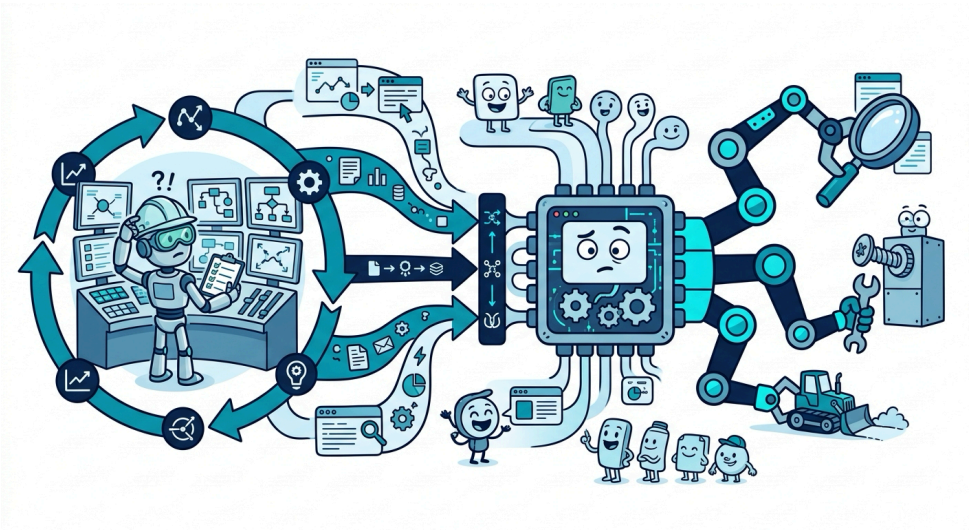

Figure 9: One protocol to rule them all: MCP replaces a thousand custom wires with a single universal plug

7.1 The Evolution: From ChatGPT to Function Calling to MCP

The path to MCP followed a clear progression rooted in how LLMs entered the world.

7.1.1 Three Waves of LLM Adoption

On November 30, 2022, ChatGPT launched [2] and crossed 1 million users in five days and 100 million users within two months, numbers no software had ever achieved. Its adoption unfolded in three distinct waves:

Table 6: Three waves of ChatGPT adoption that set the stage for function calling and MCP

Wave	Name	What Happened	Outcome
1	Pure Wonder	Users asked random questions (explain quantum physics from a cat's perspective), shared screenshots on social media	Collective realization that machines could converse naturally for the first time in 500+ years of human-machine interaction
2	Professional Adoption	Lawyers summarized contracts, coders debugged code, teachers planned curricula	Productivity doubled across industries; ChatGPT proved it was a serious work tool, not just entertainment
3	API Revolution	OpenAI released GPT APIs; Microsoft added Copilot to Office; Google integrated AI into Gmail/Docs/Drive; new-age tools (Cursor, Perplexity) emerged	AI became accessible beyond ChatGPT; every existing software became AI-enabled

7.1.2 The Fragmentation Problem

After Wave 3, a new problem emerged: **AI fragmentation**. Every software became AI-enabled, but each AI existed in isolation. Notion's AI had no idea what was happening in Slack's AI. VS Code's coding assistant had no awareness of discussions in Microsoft Teams. Users found themselves living in **multiple disconnected AI worlds**, manually juggling information across tools to complete even simple tasks.

7.1.3 Function Calling: The First Solution

In mid-2023, OpenAI introduced **function calling** [3]: LLMs could invoke external functions by producing structured output with the function name and arguments. For the first time, LLMs could not only chat but also execute tasks. Instead of the user manually copy-pasting from Jira, GitHub, MySQL, Google Drive, and Slack into ChatGPT, the AI could fetch from each system automatically.

Function calling triggered an explosion of tool integrations. Enterprise software (Salesforce, Slack, GitHub), internal department tools (HR data, finance), and AI-first products (Cursor, Perplexity) all added function calling support.

But function calling had a critical flaw.

7.2 The N × M Problem: Why MCP Exists

Each tool integration had to be custom-coded on the **client side**. If a company had 3 AI chatbots and wanted to connect to 10 services, they needed 3 × 10 = 30 unique integrations. Each integration required writing custom code for each API, handling authentication and

error handling, maintaining and updating when APIs change, and different code for different LLM providers.

Without MCP, connecting N LLMs to M tools requires **N × M** custom integrations. Each LLM handles API calls and responses differently. Switching from Claude to GPT requires rewriting every integration.

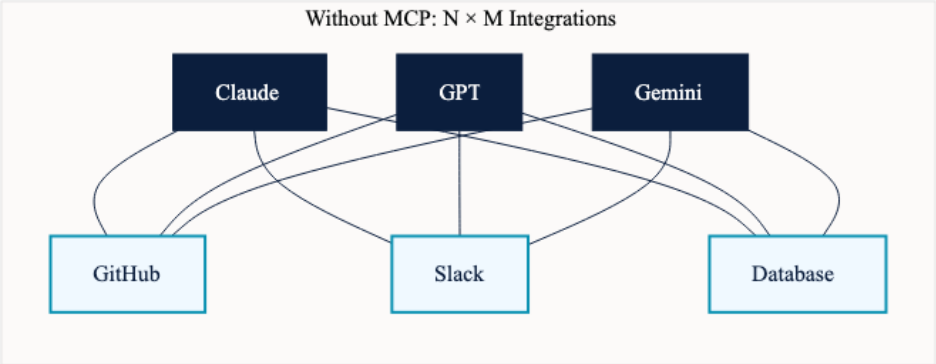

Figure 10: Without MCP: every LLM needs a custom integration with every tool (N × M connections)

With MCP, the problem becomes **N + M**. The protocol standardizes how instructions are sent to the server and how results are returned.

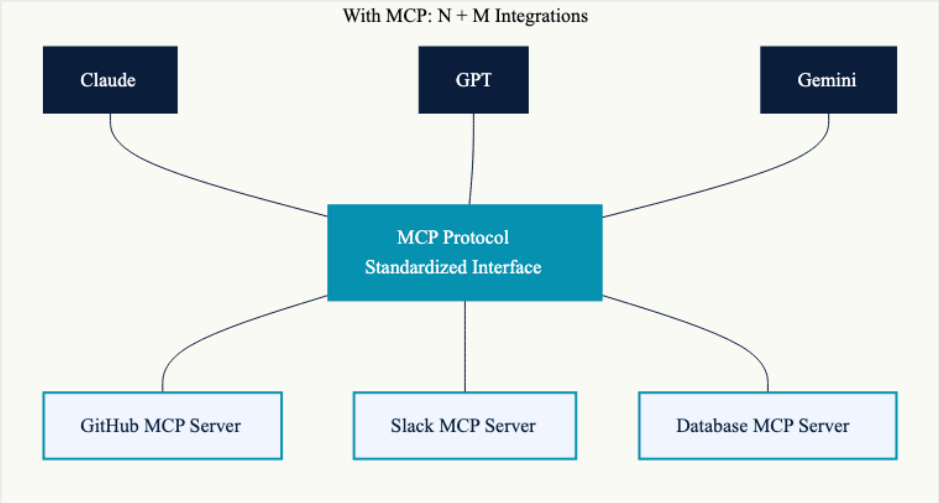

Figure 11: With MCP: a standardized protocol reduces connections to N + M

Without MCP: N × M

- 3 LLMs × 3 tools = 9 custom integrations
- Each integration is unique
- Switching LLM requires rewriting every tool connection
- Developer must know each LLM's API format

With MCP: N + M

- 3 LLM clients + 3 MCP servers = 6 integrations
- Common protocol for all
- Switching LLM only requires updating the client
- Server developers need not know which LLM will connect

7.3 MCP vs. API: What Is the Difference?

This is the most frequently asked question about MCP. The short answer: MCP does not replace APIs — it **standardizes how LLMs interact with them**. An API is the raw interface to a service; MCP is the protocol layer that wraps APIs so any LLM can discover and use them without custom integration code.

Table 7: MCP vs. API: MCP does not replace APIs — it standardizes how LLMs consume them

Aspect	Traditional API	MCP
Purpose	Expose a service's functionality to any consumer (web apps, mobile apps, scripts)	Standardize how LLMs specifically discover and invoke tools
Consumer	Any software: browsers, mobile apps, backend services, CLI tools	LLM-based applications (Claude Desktop, Cursor, custom agents)
Discovery	Manual: read API docs, write integration code, handle authentication	Automatic: client calls tools/list and receives all available tools with schemas
Integration effort	Custom code per API per client (N × M problem)	One protocol for all: connect once, access everything (N + M)
Schema format	Varies: OpenAPI/Swagger, GraphQL, custom docs	Standardized JSON tool schemas with name, description, parameters, and return type
Who writes the glue code?	The client developer writes API calls, auth, error handling, rate limiting	The server developer wraps their API once; clients connect with zero custom code
Communication	HTTP request/response (REST) or query/mutation (GraphQL)	JSON-RPC 2.0 over stdio (local) or HTTP/SSE (remote) with bidirectional messaging
Context awareness	None: APIs have no concept of LLM context windows or token budgets	Built-in: the MCP client manages what schemas and results enter the LLM's context

Memorize

Think of it this way: Slack still has a REST API. The Slack MCP server is a thin wrapper around that API that exposes Slack's functions as MCP tools. The API does the work; MCP makes it discoverable and usable by any LLM without writing Slack-specific integration code on the client side. MCP is to APIs what USB-C is to hardware peripherals: a universal connector, not a replacement for the devices themselves.

7.4 The USB-C Analogy

MCP is to LLM-tool connections what USB-C is to hardware connections.

Table 8: USB-C analogy: MCP standardizes the LLM-tool connection just as USB-C standardized hardware connections

Before USB-C	After USB-C
Different ports for HDMI, VGA, micro-USB, etc.	One universal port connects to everything
Each device needs a specific cable	One cable + adapters for all devices
N × M cable combinations	N + M (one port standard + adapters)

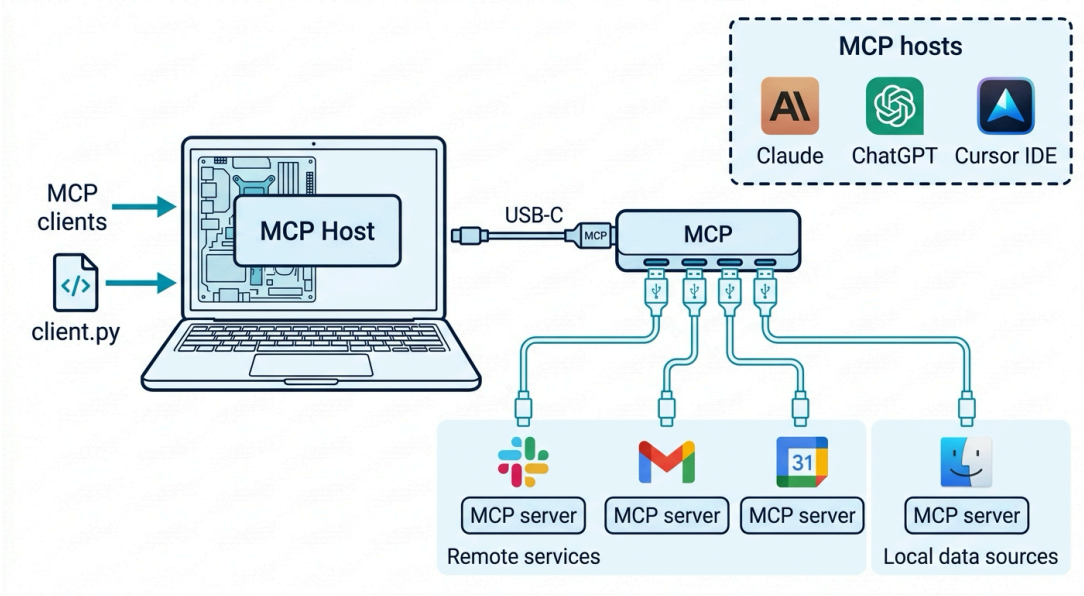

Figure 12: One port to rule them all: MCP hosts plug into remote and local services through a single standardized protocol — just like USB-C

Just as USB-C standardizes the connector between a laptop and any peripheral, MCP standardizes the connector between any LLM and any tool.

> **Info**
>
> MCP was released by Anthropic on November 25, 2024 as an open-source standard [4]. The key architectural shift from function calling: in function calling, all integration logic (API calls, authentication, error handling, rate limiting) is written on the CLIENT side. In MCP, all heavy lifting is done on the SERVER side. Service providers (GitHub, Google Drive, Slack) build their own MCP servers. The client side only needs a connection configuration.

The **network effect** drives MCP ecosystem growth: famous AI chatbots (Claude Desktop, Cursor, Windsurf, Perplexity) announced MCP support, which pressured service providers

to build MCP servers (future user traffic comes through AI tools). More servers attract more clients, more clients attract more servers.

With the motivation for MCP established, the next question is: what does the architecture actually look like?

7.5 MCP Host, Client, and Server: Roles Defined

The MCP architecture has three layers. The **Host** is the AI application the user interacts with (Claude Desktop, Cursor IDE, or a custom chatbot); it contains the LLM. The **MCP Client** is a helper entity inside the host that translates between the host's language and the MCP protocol. The **MCP Server** hosts and executes tools.

Each client connects to exactly one server. If the host needs multiple servers (GitHub, Slack, Google Drive), it creates one client per server. This is analogous to a phone (host) needing one SIM card (client) per network (server).

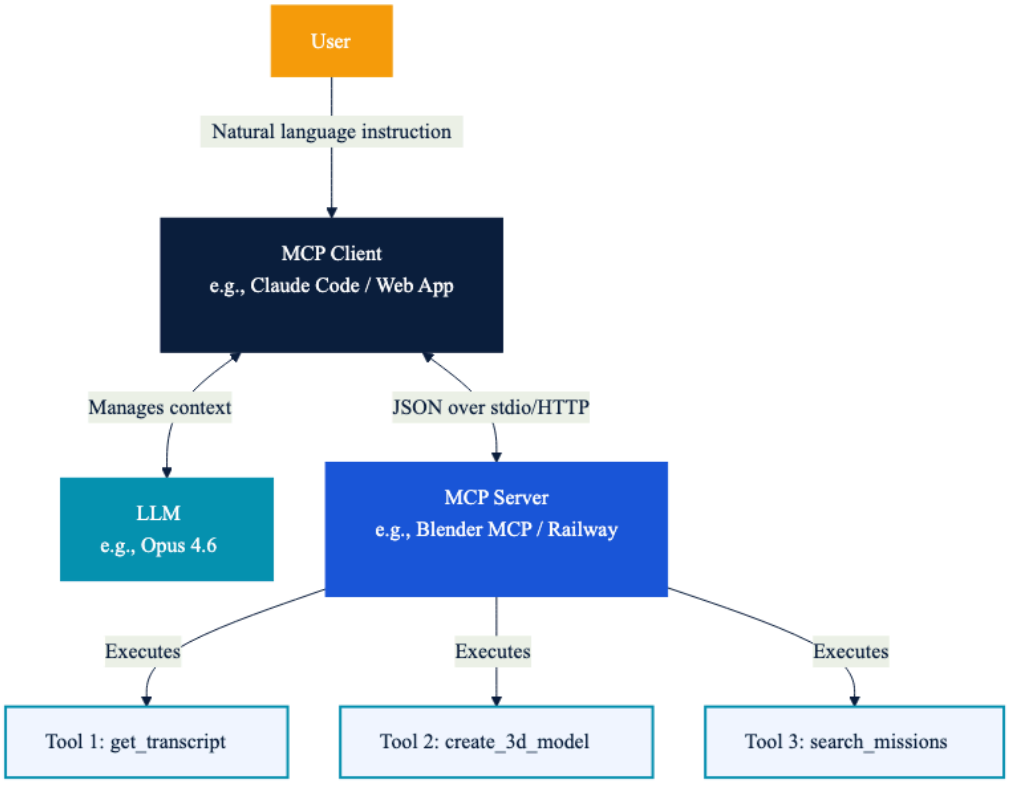

Figure 13: MCP architecture: User communicates with Client, which manages LLM context and forwards tool calls to Server

Table 9: MCP component roles

Component	Role	Key Responsibilities
MCP Client	Intermediary between user/LLM and server	Manages LLM's context window; decides what tool schemas to expose to LLM; decides what portion of tool results to feed to LLM; maintains conversation loop
MCP Server	Hosts and executes tools	Exposes tool schemas, resources, and prompts; executes tool functions; returns results to client; does NOT manage the LLM's context
LLM	The reasoning brain	Decides which tools to call and in what order; produces structured tool call responses; reasons over tool results; produces final natural language output

> **Key Insight**
>
> The LLM *never* communicates directly with the MCP server. All communication flows through the MCP client. The server does not care about or manage the LLM's context window: that is exclusively the client's responsibility.

The diagram below shows the full three-layer MCP architecture in detail: the Host (containing the LLM and its clients), the transport layer (stdio for local, HTTP/SSE for remote), and the MCP Servers (each exposing tools, resources, and prompts). Notice the strict 1:1 mapping between clients and servers, while the LLM fans out to all clients simultaneously.

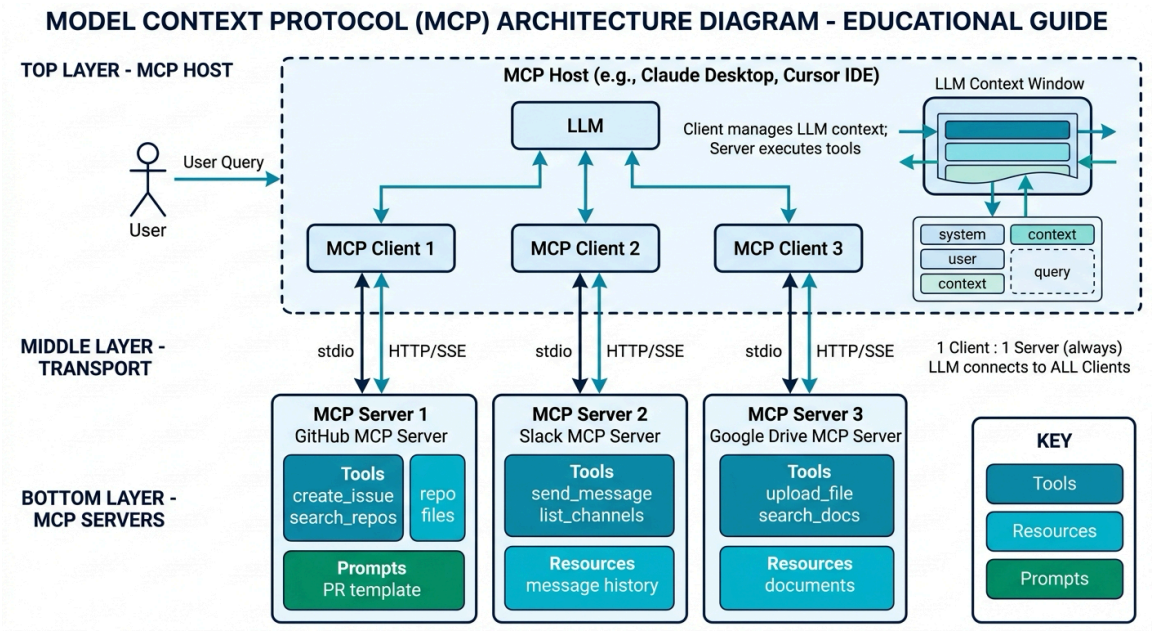

Figure 14: The full blueprint: Host holds the brain, Clients are the translators, Servers do the heavy lifting — all connected by one protocol

7.6 The Three MCP Primitives

MCP servers expose three types of capabilities, called primitives:

Table 10: Three MCP primitives: tools, resources, prompts

Primitive	Decorator	Description	Example
Tools	@server.tool	Functions the LLM can invoke via structured calls	get_transcript(video_url), search_missions(query, status)
Resources	@server.resource	Documents exchanged between server and client for later reference	Mission logs, crew schedules, transcripts stored as markdown for RAG
Prompts	@server.prompt	Predefined, reusable system prompts associated with tool actions	Summarize this transcript in one paragraph

Resources explained: After a tool returns a large result (e.g., a full video transcript), the client can store it as a resource, a document that can be queried later via RAG or keyword search without making another server call. Key difference from tools: resources are **read-only data**; tools **perform actions**.

Prompts explained: Reusable system prompts that define *how* to use tool results. For example, a button labeled "Summarize" triggers a predefined prompt that pairs a transcript resource with a summarization instruction.

Each primitive has **standard operations** defined by the protocol:

Table 11: Standard MCP operations for each primitive

Primitive	Standard Operations
Tools	tools/list (discover available tools), tools/call (invoke a specific tool with arguments)
Resources	resources/list, resources/read, resources/subscribe, resources/unsubscribe
Prompts	prompts/list (discover templates), prompts/get (retrieve a specific template)

With the three primitives defined, the next question is how client and server physically communicate.

7.7 Transport Layer: stdio vs. HTTP/SSE

The transport layer determines how client and server communicate. The choice depends on whether they are on the same device or across a network.

Table 12: MCP transport options

Transport	When Used	Example
stdio (Standard I/O)	Client and server on the same device	Claude Code controlling Blender installed locally
HTTP/HTTPS	Client and server on different devices (over network)	Web app on Vercel communicating with MCP server on Railway
SSE (Server-Sent Events)	Real-time streaming over network	Streaming tool results as they become available
WebSocket	Bidirectional real-time communication	Interactive applications requiring constant updates

With the transport layer handling *how* bytes move, the next question is *what* those bytes contain.

7.8 JSON-RPC 2.0: The Communication Protocol

All MCP messages between client and server use **JSON-RPC 2.0** [5] as the wire protocol. Every message is a JSON object with a specific structure. A natural question is: why not use REST APIs, which are already the standard for web communication? Anthropic chose JSON-RPC for five specific reasons:

Table 13: Why JSON-RPC over REST: five architectural reasons behind MCP's protocol choice

Reason	JSON-RPC 2.0	REST API
Lightweight	Minimal overhead: just method, params, and id — no HTTP headers or metadata per message	Every request carries HTTP headers, status codes, and URL routing overhead
Bidirectional	Both client and server can initiate requests (server can request sampling from client)	Strictly one-way: client sends request, server responds — server cannot initiate
Transport-agnostic	Works over stdio, HTTP, WebSockets, or any stream — not tied to a specific transport	Tied to HTTP by definition; cannot run over stdio or raw streams
Batching	Multiple requests can be sent in a single array and processed together	One request per HTTP call; batching requires custom implementation
Notifications	Built-in fire-and-forget messages (no ID, no response expected) for progress updates and alerts	No native notification concept; requires workarounds like webhooks or polling

Table 14: Three JSON-RPC 2.0 message types in MCP

Message Type	Direction	Has ID?	Purpose
Request	Client ⟶ Server	Yes	Client sends a request expecting a response (e.g., tools/list, tools/call)
Response	Server ⟶ Client	Yes (matches request)	Server sends back the result of a request
Notification	Either direction	No	One-way message, no response expected (e.g., initialized, notifications/cancelled)

7.8.1 Requests vs. Notifications: Why the Distinction Matters

The difference between requests and notifications is fundamental to MCP's design:

Table 15: Requests require a response; notifications are fire-and-forget

Aspect	Request	Notification
Has ID field?	Yes (e.g., id: 1)	No
Expects response?	Yes, always	No, never (fire-and-forget)
Direction	Typically Client → Server (but Server → Client for sampling/elicitation)	Either direction
Examples	tools/list, tools/call, initialize, resources/read	initialized, notifications/cancelled, notifications/progress
When to use	When the sender needs a result or confirmation	When the sender is informing the other party without needing acknowledgment

Memorize

The `initialized` message (Step 3 of the handshake) is a **notification**, not a request. The client does not wait for a response; it simply informs the server that setup is complete. If you accidentally implement it as a request (with an ID), the server will try to respond, causing protocol confusion.

JSON-RPC Request vs. Notification

Request (has ID, expects response):

```
{
  "jsonrpc": "2.0",
  "method": "tools/list",
  "params": {},
  "id": 1
}
```

Server responds:

```
{
  "jsonrpc": "2.0",
  "result": {
    "tools": [...]
  },
  "id": 1
}
```

The `id` in the response **must** match the request `id`. This is how the client matches responses to requests.

Notification (no ID, no response):

```
{
  "jsonrpc": "2.0",
  "method": "notifications/progress",
  "params": {
    "progressToken": "abc",
    "progress": 60,
    "total": 100
  }
}
```

No response is sent or expected.

With the message format understood, the next question is how a full MCP session unfolds from start to finish.

7.9 The MCP Lifecycle: Three Stages

An MCP **session** is one continuous connection between client and server [6]. The **lifecycle** is the complete sequence of steps governing how they establish, use, and end that connection.

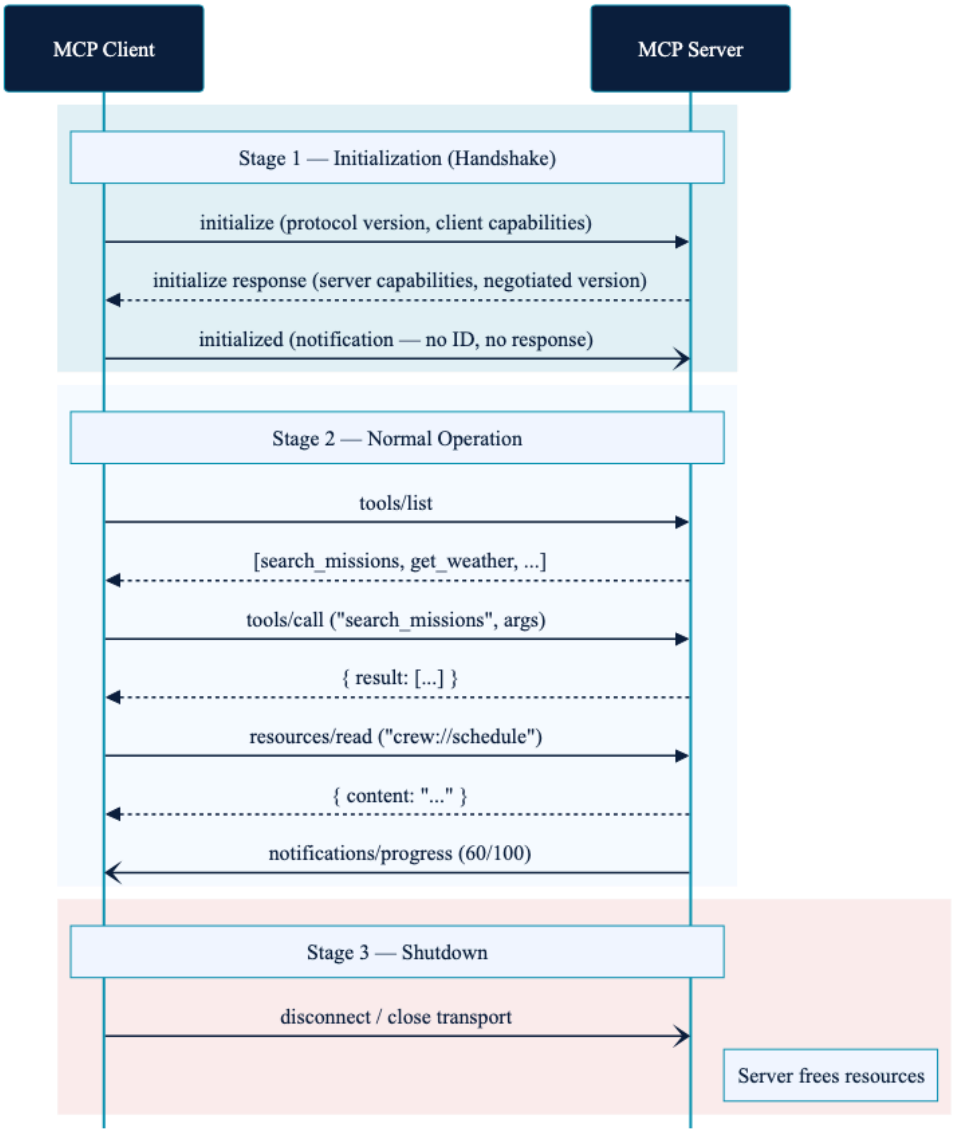

Figure 15: The full MCP session: handshake with version negotiation, normal operation with tool discovery and usage, then graceful shutdown

7.9.1 Stage 1: Initialization (The Handshake)

The initialization must be the **first** interaction between client and server. No other communication is allowed beforehand (except pings).

1. **Client sends Initialize Request** — Includes protocol version (e.g., 2024-11-05), client capabilities (roots, sampling), and implementation info (client name/version)

2 **Server responds with Initialize Response** — Returns its protocol version, server capabilities (tools, resources), and implementation info (server name/version)

3 **Client sends Initialized Notification** — A notification (no ID) confirming the connection is successful. After this, the session is active.

> ⚡ **Two Critical Rules During Initialization**
>
> The Client MUST NOT send requests other than pings before the Server has responded to the initialize request. The Server MUST NOT send requests other than pings and logging before receiving the initialized notification.

7.9.2 Version Compatibility and Protocol Negotiation

During initialization, both parties exchange **protocol version strings** (e.g., `"2024-11-05"`). Version negotiation follows strict rules:

1. The client sends its supported protocol version in the initialize request
2. The server compares against its own supported version(s) and responds with the negotiated version
3. If versions are incompatible (e.g., client sends `"2025-03-15"` but server only supports `"2024-11-05"`), the handshake **fails** and the session does not start
4. Both parties must respect the negotiated version for all subsequent messages

> 🔥 **Tip**
>
> Version negotiation ensures forward compatibility. As the MCP specification evolves (adding new primitives, new message types, or new capabilities), older clients and newer servers (or vice versa) can detect incompatibilities early rather than failing silently during operation.

7.9.3 Capabilities Exchanged During Handshake

During initialization, client and server exchange **capabilities**: declarations of what advanced features each side supports. These capabilities govern what requests are allowed during normal operation.

Client capabilities (what the client tells the server it supports):

Table 16: Client capabilities: what the client tells the server it can do

Capability	Description	Example
Roots	The client grants the server access to specific filesystem directories (project folders, workspaces)	A coding assistant grants the MCP server access to the current project root so it can read and modify files
Sampling	The client allows the server to request LLM generation from the host's model	A data analysis server asks the host's LLM to summarize a dataset before returning results
Elicitation	The client allows the server to prompt the user for interactive input	A deployment server asks the user to confirm before deleting a production database

Server capabilities (what the server tells the client it supports):

Table 17: Server capabilities: what primitives the server exposes

Capability	Description
Tools	The server has tools the client can invoke (tools/list, tools/call)
Resources	The server has documents or data the client can read (resources/list, resources/read)
Prompts	The server has prompt templates the client can use (prompts/list, prompts/get)

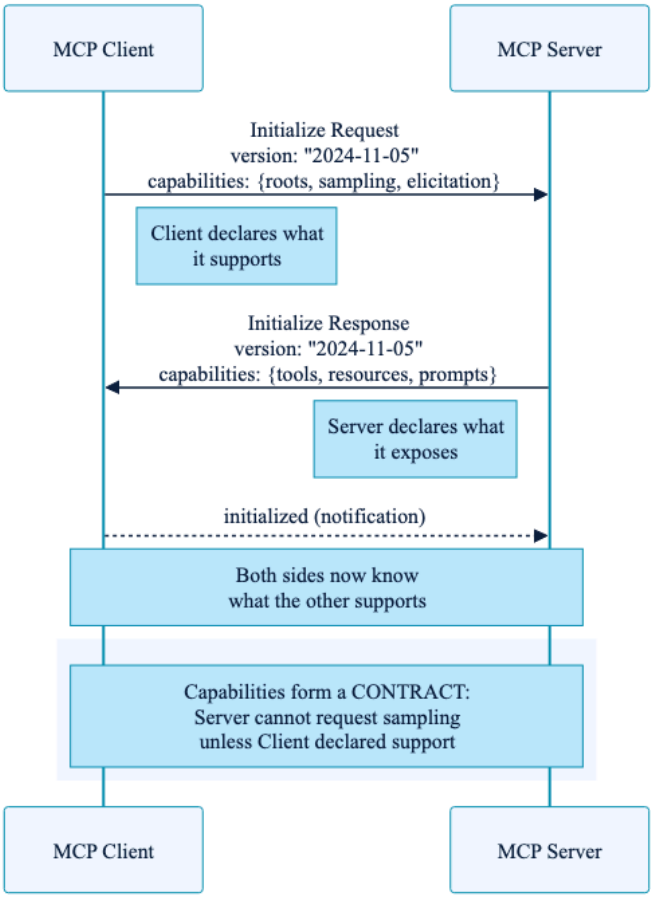

Figure 16: Capabilities exchange during handshake: client declares roots/sampling/elicitation; server declares tools/resources/prompts. These form a contract.

> **Key Insight**
>
> Capabilities are a **contract**. If the client does not declare sampling support, the server must not request LLM generation. If the server does not declare tools support, the client must not call tools/list. This prevents runtime errors from unsupported operations.

7.9.4 Server-to-Client Requests (Bidirectional MCP)

MCP is not strictly unidirectional. After the handshake, the server can **initiate requests** to the client (not just respond to client requests). This is a fundamental inversion of the typical request-response flow.

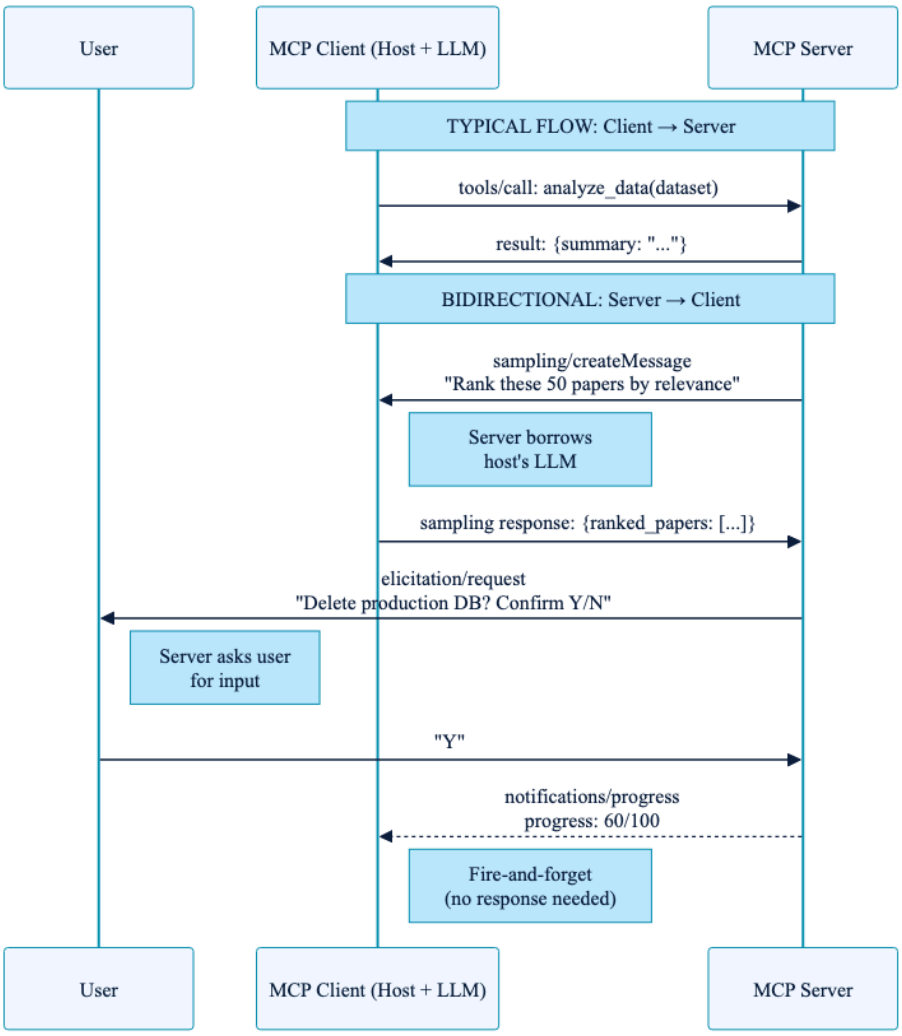

Figure 17: Bidirectional MCP: the server can request sampling (LLM generation), elicitation (user input), and send notifications (progress updates) back to the client

Table 18: Server-to-client requests: the server can initiate communication back to the client

Server-to-Client Request	Requires Capability	Description
Sampling (LLM Generation)	Client must declare sampling	The server asks the host's LLM to generate content. Example: a research server fetches 50 papers and asks the host LLM to rank them before returning results. The server does not need its own LLM.
Elicitation (User Input)	Client must declare elicitation	The server asks the user a question interactively. Example: a file management server asks the user to confirm before overwriting an existing file.
Notifications	Always allowed	One-way messages: progress updates (notifications/progress), log messages (notifications/message), resource change alerts. No response expected.

> ⚠ **Warning**
>
> Sampling is powerful but risky. If a server can request LLM generation from the host, it effectively gains partial control over the host's AI. Clients should implement guardrails: rate limiting on sampling requests, user approval for sensitive generations, and cost monitoring.

7.9.5 Stage 2: Normal Operation

After initialization, the session enters normal operation. This has two parts:

Part 1: Capability Discovery. This happens **automatically** as soon as initialization completes. The client knows the server supports tools/resources/prompts (from the handshake), but does not know the *specific* tools available. So it sends three discovery requests as a batch: `tools/list` to discover exact tool names and descriptions, `resources/list` for available resources, and `prompts/list` for prompt templates. If the server does not support a particular primitive, it returns a "method not found" error for that request, which the client silently ignores.

Part 2: Tool and Resource Usage. Based on user queries, the client calls specific tools (`tools/call`) or reads specific resources (`resources/read`).

7.9.6 Stage 3: Shutdown

The session terminates when the client shuts down (user closes the host application) or the server shuts down (crash, network failure). Either party can initiate termination.

With the lifecycle stages defined, the next concern is what happens when things go wrong.

7.10 Error Handling, Timeouts, and Progress

7.10.1 Ping/Pong (Health Check)

Either client or server can send a `ping` method to check if the other party is alive. The other party responds with `pong`. This is allowed even during initialization.

7.10.2 Error Handling

When errors occur, the server returns a standard JSON-RPC error object:

Table 19: Standard JSON-RPC error codes used in MCP

Error Code	Meaning
-32601	Method not found (e.g., server does not support prompts/list)
-32602	Invalid parameters sent for a tool
-32600	Invalid request format
-32700	Parse error (not valid JSON)
-32000 and above	Application-specific (authentication failure, rate limit, quota error)

7.10.3 Timeouts and Cancellation

Clients set a timeout threshold per request (e.g., 30 seconds). If the server exceeds it, the client sends `notifications/cancelled` with the request ID and reason. The server stops processing and frees resources.

7.10.4 Progress Notifications

For long-running tasks, the server sends periodic progress updates using the **progressToken** mechanism:

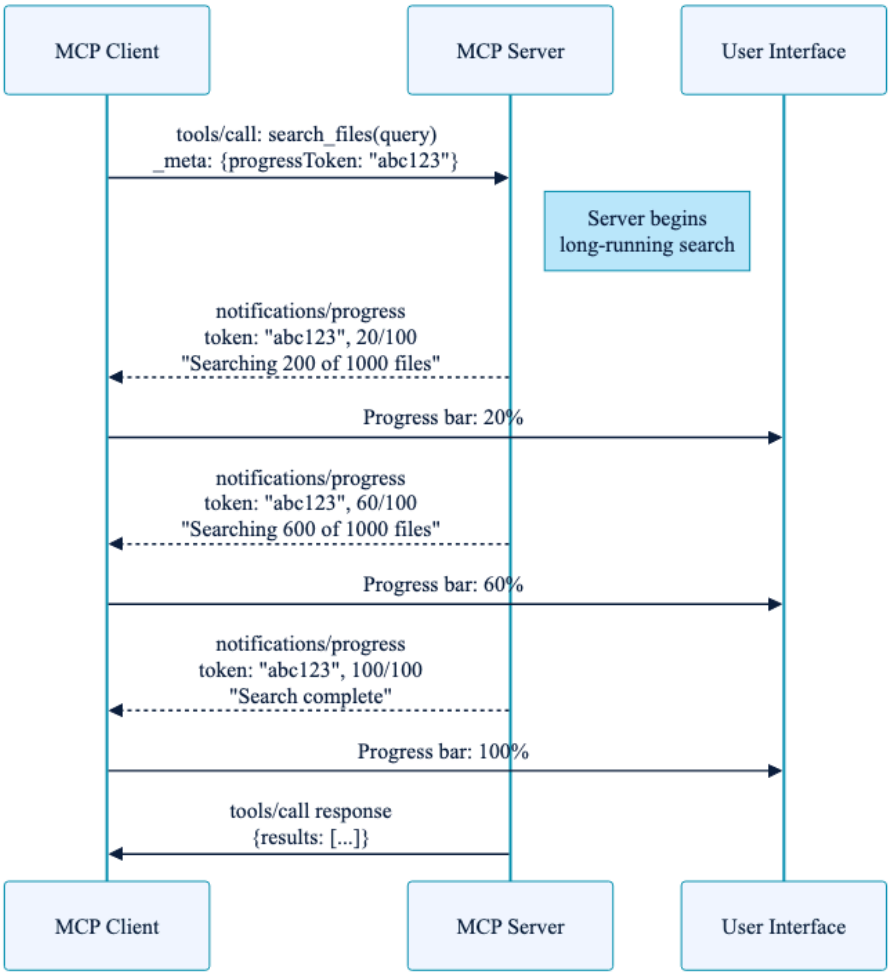

Figure 18: progressToken flow: client includes token in request; server sends periodic notifications/progress; client renders progress bar

1 **Client includes progressToken in request** — When calling a tool, the client adds a progressToken to the request's _meta field:

```
{ "_meta": { "progressToken": "abc123" } }
```

2 **Server sends periodic notifications/progress** — The server uses the same token to send updates:

```
{
  "method": "notifications/progress",
  "params": {
    "progressToken": "abc123",
    "progress": 60,
    "total": 100,
    "message": "Searching 600 out of 1000 files"
```

```
  }
}
```

3 **Client displays progress to user** — The client matches the progressToken to the original request and renders a progress bar or status message in the UI

</> Progress Notification Example

```
{
  "jsonrpc": "2.0",
  "method": "notifications/progress",
  "params": {
    "progressToken": "abc123",
    "progress": 60,
    "total": 100,
    "message": "Searching 600 out of 1000 files"
  }
}
```

! Memorize

Progress notifications are **notifications** (no ID, no response expected). The server can send as many as needed during a single tool execution. If the client did not include a progressToken, the server should not send progress notifications for that request.

With error handling, timeouts, and progress understood, the next section puts all the pieces together to show how a complete tool call flows through MCP.

7.11 Tool Call Flow in MCP

After the handshake, individual tool calls follow a four-step pattern:

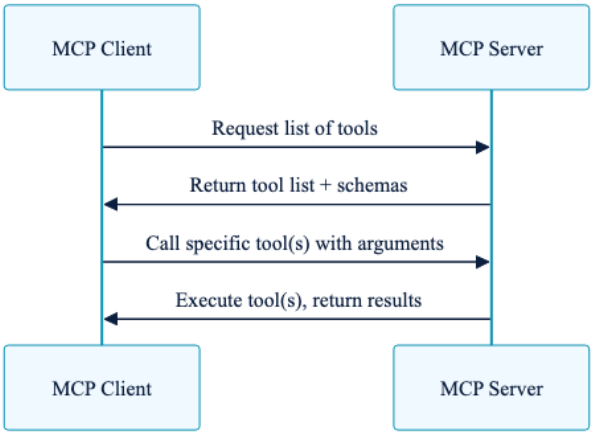

Figure 19: Tool call flow: Client requests tools, Server returns schemas, Client calls, Server executes and returns

7.12 MCP Data Flow: Complete Walkthrough

The full data flow for a single tool interaction:

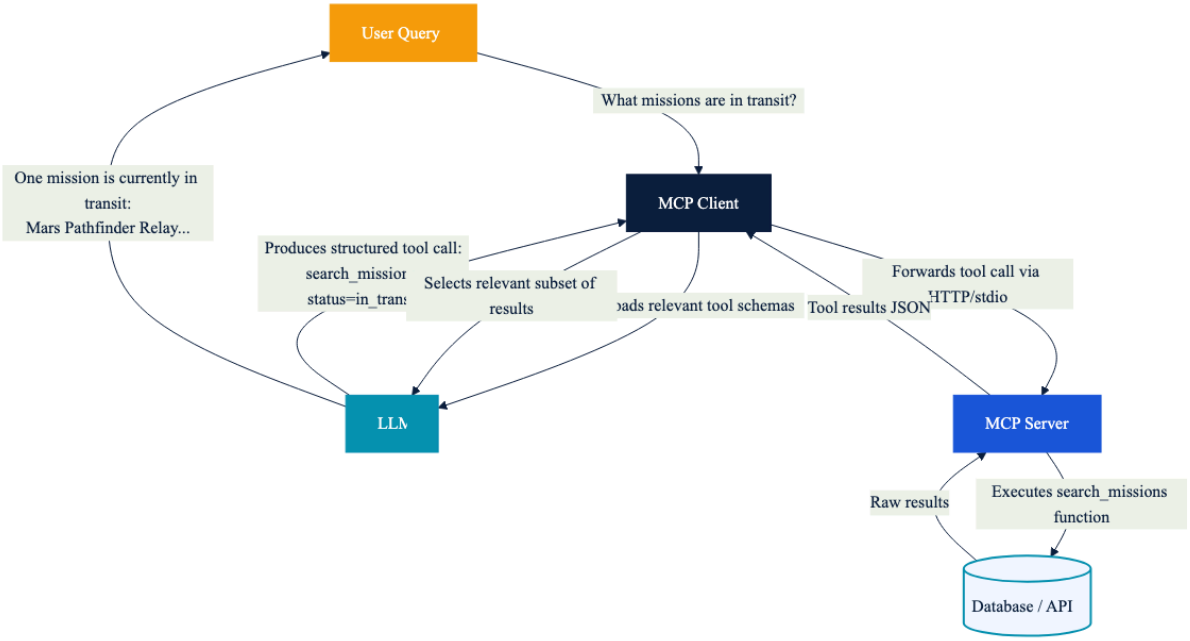

Figure 20: Complete MCP data flow

1. **User asks a question** — e.g., What missions are currently in transit?

2. **Client loads relevant tool schemas into LLM context** — Based on the query, the client selects which schemas the LLM should see

3. **LLM produces a structured tool call** — e.g., search_missions(status=in_transit)

4. **Client forwards tool call to MCP Server** — Via stdio (local) or HTTP (remote)

5 **Server executes the function** — Against its data source (database, API, etc.)

6 **Server returns tool results to Client** — As JSON, not directly to the LLM

7 **Client selects relevant subset of results** — Context management: not all results need to enter the LLM's window

8 **LLM reasons over results and responds** — Produces natural language answer for the user

7.13 Who Controls the LLM's Context?

This is the most frequently confused aspect of MCP. The following table clarifies every responsibility:

Table 20: MCP responsibility matrix

Question	Answer
Who decides which tool schemas the LLM sees?	MCP Client
Who decides which tool to call?	LLM
Who decides the order of tool calls?	LLM
Who executes the tool?	MCP Server
Where do tool results go first?	MCP Client (not directly to the LLM)
Who decides what portion of results enters the LLM context?	MCP Client
Who maintains the conversation loop?	MCP Client
Can the server access the LLM's context?	No: the server does not care about or manage the LLM's context

Memorize

If the MCP server exposes 10 tools but the client sends only 3 tool schemas to the LLM, the LLM can only access those 3 tools. The client is the gatekeeper of what the LLM knows about.

With context ownership clarified, the next question is how to design good MCP servers.

7.14 MCP Server Design: Principles and Anti-Patterns

Good MCP server design follows three principles:

If The tool needs server-side resources (APIs, databases, hardware) → Keep on the server — Only the server has access to these resources
If The tool needs LLM-based processing (summarization, analysis) → Move to the client — The client already has an LLM; duplicating it on the server couples server to provider
If A tool can be decomposed into fetch data + process with LLM → Split: only fetch data on the server — Keeps the server stateless, simple, and LLM-agnostic

⚡ **Anti-Pattern: LLM on the Server**

Building tools like summarize_video on the server (which internally fetches a transcript AND calls an LLM) is an anti-pattern. Only get_transcript belongs on the server. Summarization is the client's job: it already has an LLM. Server-side LLM coupling makes the server dependent on a specific provider and duplicates reasoning capacity.

7.15 MCP Server Safety and Guardrails

When MCP tools can make destructive changes (e.g., deleting database entries), implement defense in depth:

1. **Triple confirmation:** Client-side guardrails requiring 3 user confirmations before delete/edit operations
2. **Eliminate dangerous operations:** Remove delete/edit tools from the server entirely if not needed
3. **Server-side validation:** Tools themselves reject invalid operations (e.g., refusing to change an aborted mission to completed)
4. **Read-only by default:** MCP servers for sensitive systems should default to read-only; write access requires explicit, guarded tools

🔥 **Tip**

The client-server architecture naturally supports safety. The client provides guardrails *before* instructions reach the server, and the server provides guardrails via tool-level validation. This is defense in depth.

7.16 Key Conceptual Clarifications

Several commonly confused points about MCP, drawn from quiz discussions:

? If an MCP server exposes a tool but the LLM never sees the tool schema

The LLM cannot use the tool. The LLM has no awareness of the tool's existence unless the client communicates the schema.

? Who decides when to invoke a tool vs. when to include a resource?

The **LLM** decides when to invoke tools (it is the brain). The **client** decides when to include resources in the LLM's context.

? An MCP server exposes 15 tools with detailed descriptions. What is the main risk?

Context rot. Loading all 15 tool descriptions consumes significant tokens and may degrade LLM output quality if the context becomes bloated.

? Does everything from the MCP server count as output tokens?

No. The MCP client ultimately decides what goes into the LLM's token context. Server results go to the client first, and the client controls what subset reaches the LLM.

? Can RAG be wrapped in a tool call?

Yes. RAG has no inherent dependency on the LLM: it is a retrieval mechanism (query, embed, similarity, retrieve). A tool function can implement RAG internally: accept a query, run vector retrieval, and return the top-K chunks as the tool result.

7.17 Connectors vs. JSON Configuration

There are two ways to connect MCP servers to a host application like Claude Desktop:

Connectors (App Store Model)

- Built-in feature that links to MCP servers automatically
- One-click installation with OAuth sign-in
- Anthropic writes, hosts, and maintains connector code
- Available for popular SaaS tools (Google Drive, Notion, Slack)
- Curated and security-reviewed

JSON Configuration File

- Manually add server details to a JSON config file
- Full flexibility for any MCP server (custom or third-party)
- Immediate connectivity without waiting for approval
- Required for custom/personal MCP servers
- Settings > Developer > Edit Config in Claude Desktop

Info

Connectors cannot fully replace JSON config for two reasons: (1) Anthropic cannot write connector code for the thousands of MCP servers appearing daily. (2) MCP is an open standard: forcing all servers through an approval process would centralize control and defeat the purpose. Both options coexist: Connectors for popular SaaS, JSON config for everything else.

Discovering MCP Servers

Search **Awesome MCP Servers** on GitHub [7] - a curated repository listing MCP servers by category (productivity, AI/ML, databases, etc.) with 89K+ stars. This is the discovery platform for finding servers to connect to.

With the MCP protocol fully covered, the next section addresses the practical ecosystem for building MCP servers and clients.

8 Building with MCP: The Ecosystem

Understanding the protocol is only half the story — you also need to know **what to build with**. This section surveys the practical tooling that has grown around MCP: the official SDKs, the high-level FastMCP decorator pattern that collapsed server creation from hundreds of lines to a handful, the MCP Inspector for debugging, and the deployment options for taking a server from localhost to production. Whether you are writing your first tool or wiring an existing API into the MCP ecosystem, the libraries and workflows covered here are your starting kit.

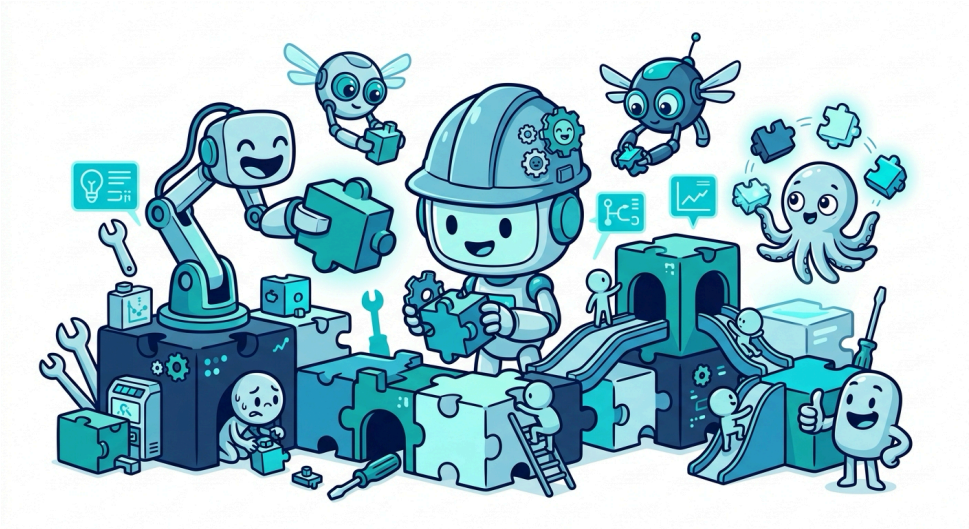

Figure 21: Some assembly required: SDKs, decorators, an Inspector, and a cloud to deploy it all

8.1 The MCP Library Ecosystem

The MCP development tooling evolved through three stages:

Table 21: Evolution of the MCP library ecosystem

Stage	Library	Experience
1. MCP SDK (Late 2024)	pip install mcp[cli]	Official Anthropic SDK [8] with mcp.server, mcp.client, mcp.cli. Verbose and boilerplate-heavy: even a simple add-two-numbers server required extensive code for manual transport handling.
2. FastMCP v1 (Early 2025)	Part of MCP SDK	Created by Jeremiah Lowin (CEO of Prefect) as an abstraction on top of MCP SDK [9]. Much simpler code. Became so popular it was adopted into the official SDK.
3. FastMCP v2 (2025)	pip install fastmcp	Independent library. Install via pip install fastmcp (gets v2) vs. pip install mcp[cli] (gets SDK with FastMCP v1 inside). Both produce virtually identical code.

🔥 **Analogy: TensorFlow and Keras**

TensorFlow was complex; Keras simplified it; eventually Keras became the default way to use TensorFlow. Same happened here: FastMCP simplified MCP SDK and was absorbed into it. Another analogy: WSGI was the raw protocol specification; Flask was the developer-friendly abstraction. MCP SDK is like WSGI; FastMCP is like Flask.

With FastMCP established as the primary development library, the following sections demonstrate its core patterns for server and tool development.

8.2 FastMCP Patterns

The core pattern for building MCP servers with FastMCP:

</> FastMCP Server: Core Pattern

```python
from fastmcp import FastMCP

mcp = FastMCP("expense-tracker")

@mcp.tool(description="Add a new expense")
def add_expense(date: str, amount: float,
                category: str, note: str) -> str:
    # Insert into SQLite database
    ...
    return "Expense added successfully"

@mcp.resource("categories://list")
def get_categories() -> str:
    # Return valid categories from JSON file
    ...

mcp.run()  # STDIO transport (local)
```

8.2.1 The Decorator Pattern: How FastMCP Works

The `@mcp.tool()` and `@mcp.resource()` decorators are the core abstraction that makes FastMCP powerful. Understanding what happens behind the scenes clarifies why so little code is needed:

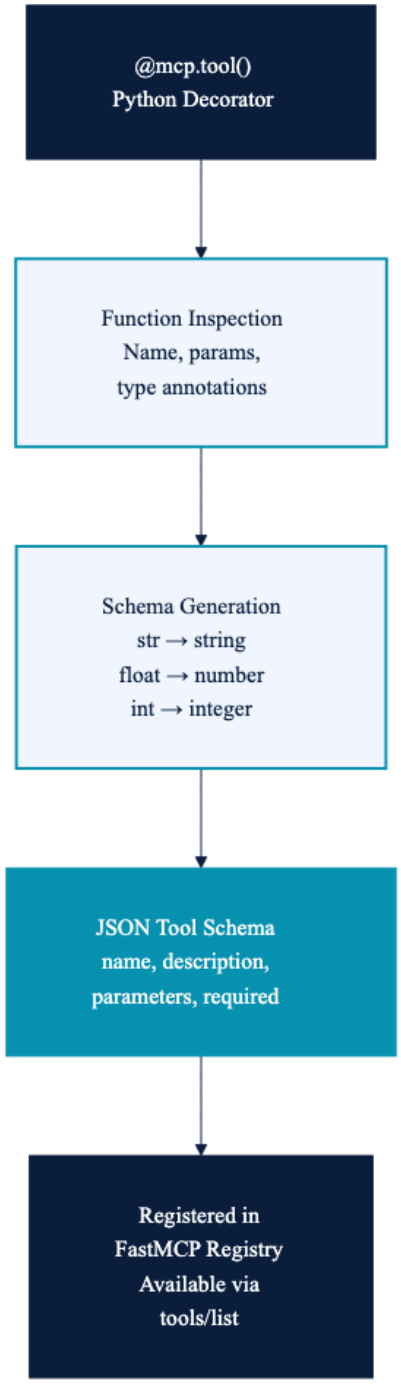

Figure 22: How @mcp.tool() works: Python decorator inspects function type annotations and auto-generates a JSON tool schema registered in the FastMCP tool registry

1 **Decorator registration** — When Python encounters @mcp.tool(), it registers the function with FastMCP's internal tool registry. The function itself is unchanged.

2 **Automatic schema generation** — FastMCP inspects the function's type annotations (str, float, int, etc.) and generates a complete JSON tool schema. Parameter names, types, and descriptions are extracted automatically.

3 **Description injection** — The description= argument in the decorator becomes the tool's natural language description in the schema, which the LLM uses during DECIDE.

4 **Transport handling** — mcp.run() starts the server with the chosen transport (STDIO or HTTP). All JSON-RPC protocol details (handshake, capability exchange, message routing) are handled automatically.

> ! **Memorize**
>
> This is why Python type annotations matter in FastMCP. If you write `amount: float`, FastMCP generates `"type": "number"` in the schema. If you write `category: str`, it generates `"type": "string"`. The LLM sees these types and produces correctly typed arguments. Missing type annotations produce ambiguous schemas.

8.2.2 MCP Inspector: Debugging Workflow

MCP Inspector is the debugging tool for MCP servers (like Postman for APIs). It is essential for verifying server behavior before connecting to Claude Desktop.

1 **Launch the Inspector** — Run: uv run fastmcp dev main.py — this starts the server and opens the Inspector in your browser

2 **Verify transport and connection** — The Inspector shows the active transport type (STDIO or HTTP) and connection status. If the server fails to start, errors appear here.

3 **Discover available capabilities** — The Inspector lists all registered tools, resources, and prompts with their schemas. Verify that tool names, descriptions, and parameter types match your expectations.

4 **Test individual tools** — Click any tool to open a test panel. Enter input values and execute the tool. The Inspector shows the full JSON-RPC request sent and response received.

5 **Review message history** — The Inspector maintains a complete log of all JSON-RPC messages exchanged during the session. This is invaluable for debugging protocol issues, incorrect parameter types, or unexpected error codes.

> 🔥 **Always Test in Inspector First**
>
> Test all tools in the MCP Inspector before connecting to Claude Desktop. Issues like wrong parameter types, missing fields, or incorrect return formats are much easier to diagnose in the Inspector than through Claude Desktop's error messages.

Installing in Claude Desktop: Run `uv run fastmcp install claude-desktop main.py` to automatically add the server to Claude Desktop's JSON config. Always restart Claude Desktop after adding new servers.

Once a server works locally, the next decision is whether it needs to be accessible over the network.

8.3 Local vs. Remote Servers

The **only code change** to make a server remote:

Table 22: Local vs. remote MCP servers: the only code difference is the transport parameter

Aspect	Local Server	Remote Server
Code change	mcp.run()	mcp.run(transport="http", host="0.0.0.0", port=8000)
Transport	STDIO	Streamable HTTP
Location	Same machine as host/client	Different machine on internet
Speed	Faster (same machine)	Slower (network communication)
Users	Single user	Multiple concurrent clients
Enterprise use	Personal/dev only	Most enterprise servers will be remote

All other tool/resource code remains identical. However, remote servers introduce a concurrency challenge that local servers do not face.

8.3.1 Async/Await for Production Remote Servers

Local servers handle one user (you), so synchronous code is fine. Remote servers handle **multiple concurrent users**. If tool functions are synchronous, every user's request blocks all others, creating a queue:

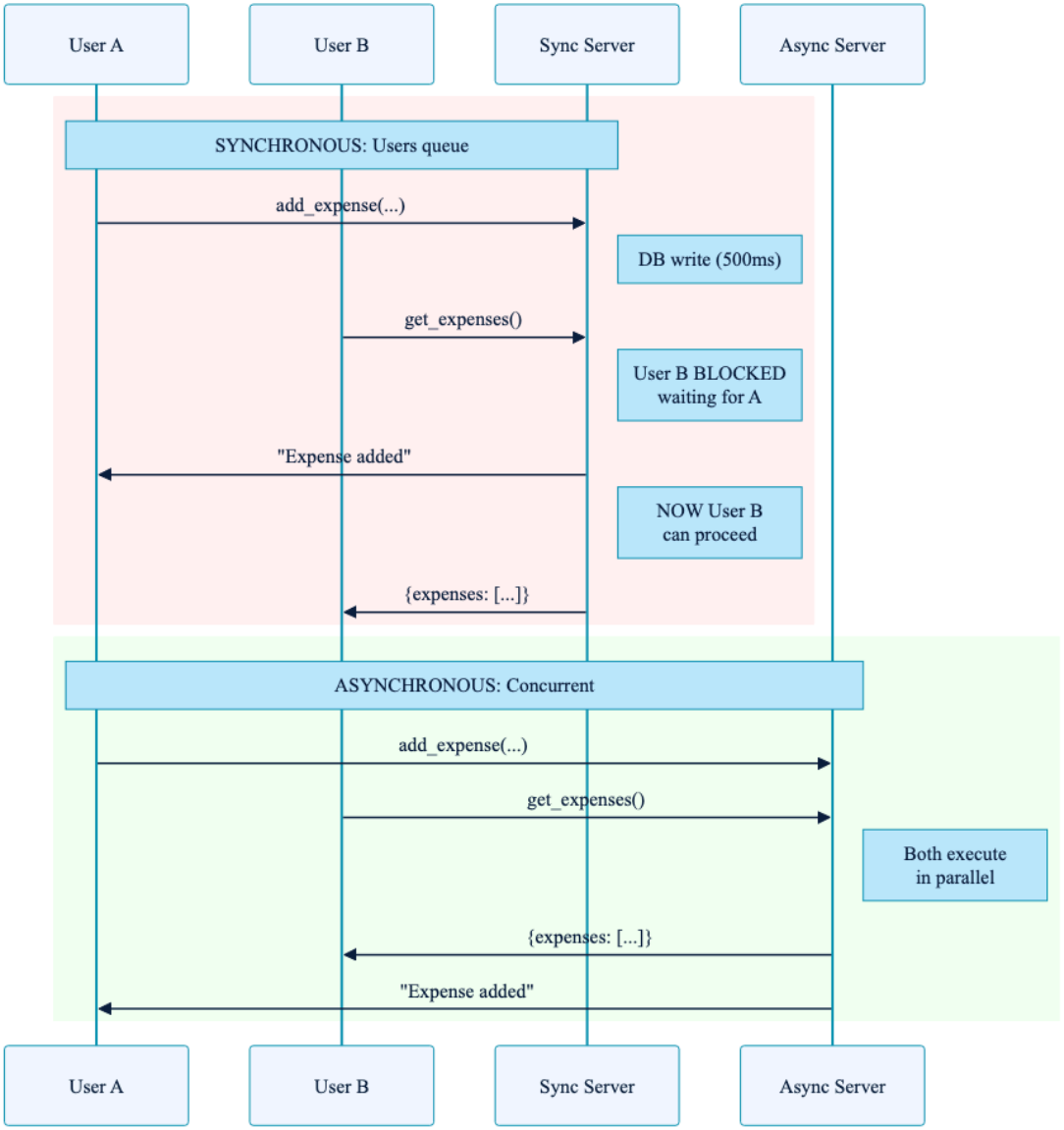

Figure 23: Synchronous vs. asynchronous server: sync blocks User B while User A's request processes; async handles both concurrently

Synchronous (Local Only)

- import sqlite3
- def add_expense(...): conn = sqlite3.connect('db.sqlite')
- One user at a time
- Second user waits until first completes
- Acceptable for personal use

Asynchronous (Production Remote)

- import aiosqlite
- async def add_expense(...): async with aiosqlite.connect('db.sqlite') as conn:
- Multiple concurrent users
- Requests processed in parallel
- Required for shared remote servers

</> Converting Sync to Async: Before and After

```python
# BEFORE: Synchronous (blocks other users)
import sqlite3

@mcp.tool(description="Add a new expense")
def add_expense(date: str, amount: float,
                category: str, note: str) -> str:
    conn = sqlite3.connect("expenses.db")
    conn.execute("INSERT INTO expenses ...", ...)
    conn.commit()
    conn.close()
    return "Expense added"

# AFTER: Asynchronous (concurrent users)
import aiosqlite

@mcp.tool(description="Add a new expense")
async def add_expense(date: str, amount: float,
                      category: str, note: str) -> str:
    async with aiosqlite.connect("expenses.db") as conn:
        await conn.execute(
            "INSERT INTO expenses ...", ...)
        await conn.commit()
    return "Expense added"
```

⚠ Warning

The conversion pattern: (1) Replace `import sqlite3` with `import aiosqlite`. (2) Add `async` before `def`. (3) Replace `sqlite3.connect()` with `async with aiosqlite.connect()`. (4) Add `await` before every database operation. Python async/await is a prerequisite for MCP remote server development.

Even with async code in place, deploying to cloud platforms introduces its own set of issues.

8.3.2 Production Deployment Gotchas

Three common issues emerge when deploying remote MCP servers to cloud platforms:

Table 23: Common production deployment issues and solutions

Issue	Cause	Solution
Database becomes read-only	Cloud platforms (Render, Railway) may mount the filesystem as read-only by default	Use a writable directory (e.g., /tmp) or switch to a managed database service (PostgreSQL, Supabase)
All users queue (blocking)	Synchronous tool functions block the event loop; while User A's database query runs, User B waits	Convert all tool functions to async/await (see conversion pattern above)
No user isolation	Without authentication, all users share the same database and see each other's data	Implement user authentication; partition data by user ID; future MCP specification updates may standardize auth flows
Missing dependencies	Cloud environments may lack system libraries or Python packages present locally	Use requirements.txt or pyproject.toml; test deployment in a clean environment before going live

Once these issues are addressed, the server is ready for deployment. Several platforms support MCP server hosting, each suited to different use cases.

8.3.3 Deployment Platforms and FastMCP Cloud

MCP servers can be deployed to any platform that supports Python:

Table 24: Deployment platforms for MCP servers and clients

Platform	Use Case	Notes
FastMCP Cloud (fastmcp.cloud)	Quick deployment for FastMCP servers	Free hosting; auto-deploys from GitHub (watches for new commits and rebuilds automatically); ideal for prototyping and personal servers
Railway	Production MCP servers	Supports persistent databases, custom domains, environment variables
Render	Production MCP servers	Similar to Railway; free tier available for testing
AWS / GCP / Azure	Enterprise MCP servers	Full control over infrastructure; required for compliance-sensitive deployments
Vercel	MCP client frontends	Best for hosting the web UI that connects to remote MCP servers

> 🔥 **FastMCP Cloud Workflow**
>
> Push your MCP server code to GitHub. Connect the repository to fastmcp.cloud. The platform watches for commits and auto-rebuilds on every push. If a deployment fails (missing dependencies, build errors), fix the issue locally, push again, and the platform rebuilds automatically.

Regardless of which platform you deploy to, the development process itself should follow a structured, incremental approach.

8.4 Incremental Server Development Workflow

Building an MCP server should follow an incremental pattern that minimizes debugging friction:

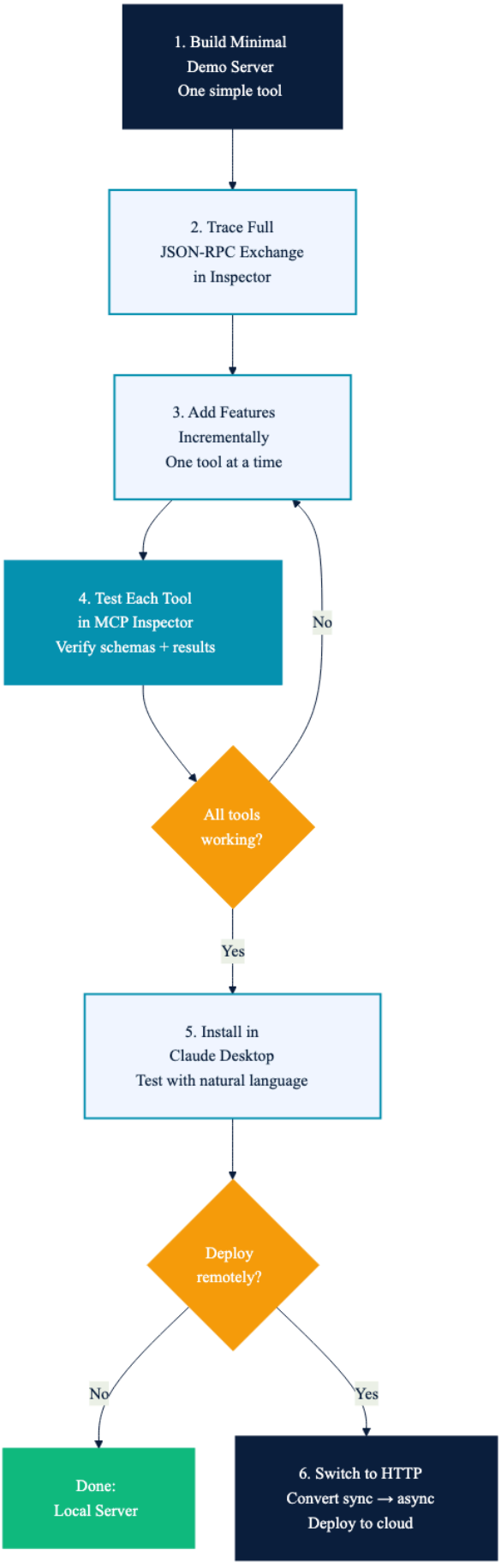

Figure 24: Incremental MCP server development: build minimal demo, trace protocol, add features one at a time, test in Inspector, then deploy

1. **Build a minimal demo server** — Start with one simple tool (e.g., an add function) to verify the full pipeline works: server starts, Inspector connects, tool executes, result returns

2. **Understand the complete process** — Trace the full JSON-RPC exchange in the Inspector: handshake → capability discovery → tool call → result. This builds intuition about the protocol.

3. **Add features incrementally** — Add one tool at a time. After each addition, test in the Inspector before proceeding. Common progression: basic read → filtered read → write → update → delete.

4. **Test in MCP Inspector** — Verify every tool works correctly in the Inspector with various inputs, edge cases, and error conditions before connecting to Claude Desktop.

5. **Install in Claude Desktop** — Run uv run fastmcp install claude-desktop main.py. Restart Claude Desktop. Test with natural language queries.

6. **Deploy remotely (if needed)** — Switch transport to HTTP, convert sync to async, deploy to FastMCP Cloud or Railway, test with remote connections.

> **Memorize**
>
> Do not skip the Inspector step. Connecting a broken server to Claude Desktop produces cryptic errors. The Inspector shows exactly what JSON-RPC messages are being exchanged, making issues immediately visible.

For teams with existing web backends, there is a shortcut: FastMCP can convert FastAPI applications directly into MCP servers.

8.5 FastAPI to FastMCP Conversion

FastMCP is compatible with FastAPI. Companies with existing FastAPI backends can instantly create MCP servers:

</> Converting a FastAPI App to an MCP Server

```python
from fastmcp import FastMCP
from my_app import app  # existing FastAPI app

mcp = FastMCP.from_fastapi(app,
    name="My Expense Tracker")

# All API endpoints automatically become MCP tools
mcp.run()
```

This means existing web/mobile backends can be exposed as MCP servers without rewriting code. The reverse is also possible: `FastMCP.as_fastapi()` converts an MCP server to a FastAPI app.

With server development covered, the final piece of the ecosystem is building custom MCP clients that connect to these servers programmatically.

8.6 Building MCP Clients

While Claude Desktop and Cursor are pre-built MCP clients, custom clients can be built programmatically. There are three approaches:

Table 25: Three approaches to building MCP clients

Approach	Library	Best For
Official MCP SDK	pip install mcp[cli]	Maximum control; verbose and complex; direct JSON-RPC handling; rarely needed for most use cases
FastMCP Client	pip install fastmcp	Simple single-server clients; good for testing your own servers programmatically
LangChain MCP Adapters	pip install langchain-mcp-adapters	Recommended for production clients; lightweight wrapper that makes MCP tools compatible with the LangChain/LangGraph agent ecosystem; supports multi-server connections

The recommended approach uses **LangChain MCP Adapters** [10]: a lightweight wrapper that makes MCP tools compatible with LangChain and LangGraph.

The client-building pattern:

1. **Configure servers** — Define server connections in a dictionary: STDIO (command + args) for local, Streamable HTTP (URL) for remote

2. **Connect via MultiServerMCPClient** — Creates one client per server and fetches all available tools from all connected servers

3. **Bind tools to LLM** — llm.bind_tools(tools) gives the LLM awareness of available MCP tools

4. **Send prompt and get tool call** — If LLM determines a tool is needed, response.tool_calls contains tool name, args, and ID

5. **Execute tool call** — Look up the tool by name and invoke it with the arguments from the LLM's response

6. **Send result back to LLM** — Create a ToolMessage with the result and tool_call_id; send full history (prompt + LLM response + tool result) back to the LLM

7. **LLM produces final answer** — With the tool result in context, the LLM generates a human-readable response

</> MCP Client: Core Pattern with LangChain

```
from langchain_mcp_adapters.client import (
    MultiServerMCPClient)
from langchain_openai import ChatOpenAI
from langchain_core.messages import ToolMessage

servers = {
    "math": {"transport": "stdio",
             "command": "uv", "args": [...]},
    "expense": {"transport": "streamable-http",
                "url": "https://my-server.cloud/mcp"}
}

async with MultiServerMCPClient(servers) as client:
    tools = await client.get_tools()
    llm = ChatOpenAI(model="gpt-4o-mini")
    llm_with_tools = llm.bind_tools(tools)

    response = await llm_with_tools.ainvoke(prompt)
    if response.tool_calls:
        # Execute each tool call
        for tc in response.tool_calls:
            result = await tools[tc["name"]].ainvoke(
                tc["args"])
            # Send result back as ToolMessage
```

! Memorize

The same client can connect to multiple servers (local + remote) simultaneously. Any MCP server, whether self-built or third-party, can be connected to a custom client. Python async/await is a prerequisite for MCP client development.

An alternative client approach uses `MCPToolkit` from the `langchain_mcp` package, which wraps MCP server communication in a simpler interface:

</> Alternative: MCPToolkit Client

```
from langchain_mcp import MCPToolkit

async def main():
    toolkit = MCPToolkit(
        server_command=["python", "my_server.py"])
    tools = toolkit.get_tools()

    # Use with any LangChain agent
    agent = create_tool_calling_agent(
        llm=llm, tools=tools, prompt=prompt)
```

Both approaches (`MultiServerMCPClient` for multi-server scenarios, `MCPToolkit` for single-server simplicity) produce LangChain-compatible tool objects that work with any agent pattern (Tool Calling or ReAct).

With the MCP ecosystem covered (protocol, lifecycle, building servers and clients), the next section demonstrates MCP's real-world impact through three live demos.

9 MCP in Action: Real-World Demos

The protocol, architecture, and tooling covered so far are abstract until you see them produce tangible output. This section presents three end-to-end demonstrations from the lecture: creating 3D animations in Blender through natural language, automating a full AI newsletter pipeline, and generating mathematical visualizations with Manim. Each demo uses a different combination of MCP servers and illustrates a distinct use case — creative production, multi-source research, and educational content generation.

9.1 Demo 1: Blender MCP — 3D Animation from Natural Language

The first demo uses Claude Code as the MCP client, Opus as the LLM, and Blender's built-in MCP server to create a spring-mass-damper animation entirely through natural language — no manual Blender interaction required.

9.1.1 Architecture

Table 26: Blender MCP demo architecture

Component	Role	Details
Claude Code	MCP Client	Receives user prompts, manages LLM context, forwards tool calls
Opus 4.6	LLM	Decides which Blender functions to invoke, produces structured tool calls
Blender MCP	MCP Server	Built-in plugin in Blender 4.4.3+; starts on port 9876; exposes hundreds of Blender functions as tools
Blender	Tool executor	Executes 3D modeling functions: create objects, set materials, define constraints, render animations

The communication uses **stdio** transport since Claude Code and Blender run on the same device. The MCP server exposes hundreds of Blender functions as tools — creating objects, defining constraints between objects, setting lighting, configuring the camera, and triggering renders.

9.1.2 The Prompt

The instructor sent a single natural language prompt to Claude Code:

Blender MCP Prompt

```
Make a new scene rendered as a 5-second video using EEVEE
(instead of Cycles). I want the rendering to happen in
2 minutes, so select the resolution accordingly.

Scene description: An animated scene that shows simple
harmonic motion using a spring-mass-damper system. Make
sure the lighting is bright but not overexposed. Make sure
the camera is pointed at the experimental setup.
```

Claude Code decomposed this into dozens of structured tool calls: creating a cube (the mass), a spring object, a damper, defining constraints between them (the cube must move with the spring as it compresses), setting up lighting, pointing the camera, configuring EEVEE render settings, and triggering the animation render.

9.1.3 Result and Significance

The rendered animation showed a functioning spring-mass-damper system with the mass oscillating, the spring compressing and extending, and damping visible over time. Key observations:

- **What was communicated:** Claude Code sent structured function calls to the MCP server (e.g., create a cube of specific dimensions at XYZ position). The server executed each function in Blender.
- **Difficulty:** Even for experienced Blender users, creating constrained animations (where objects move together physically) is time-consuming. Defining constraints between a spring and a mass so they move coherently requires significant manual effort.
- **Primitive prompt, impressive result:** The prompt followed no advanced prompting standards, yet produced a working physics animation. With detailed preferences and style guides, the output would improve significantly.

9.1.4 Capturing a Team's Visual Style with SKILL.md

The instructor described an ongoing experiment: feeding previous Blender files to Claude Code and asking it to generate **SKILL.md** files that summarize the team's visual style — lighting preferences, object definition patterns, constraint conventions, and rendering settings. These skill files would then allow Claude Code to produce new animations *in the team's established style* with minimal prompting.

> 🔥 **Tip**
>
> This workflow generalizes beyond Blender. Any creative tool with an MCP server can benefit from style capture: feed previous work to the LLM, extract patterns into skill files, then use those patterns as context for future generations. The SKILL.md approach turns tribal knowledge into reusable LLM context.

9.1.5 Industry Disruption Potential

The Blender demo sparked a broader discussion about MCP's disruption potential across industries:

Table 27: Industry disruption opportunities through MCP

Industry	Current Pain Point	MCP Opportunity
CAD / Manufacturing	CAD companies run years behind AI adoption; experts spend hours on manual 3D modeling	MCP servers for CAD software (SolidWorks, AutoCAD) enabling natural language design
Real Estate	3D house renderings for apartments require expensive manual work; demand exceeds supply	MCP server for rendering tools that generate walkthroughs from floor plans
Logistics / Hospitals	Internal software is clunky and hard to use; staff waste time navigating complex UIs	MCP servers wrapping internal APIs so staff interact via natural language
Enterprise (Salesforce, ERP)	Managing invoices, inventory, and CRM through complex software interfaces	MCP servers exposing Salesforce/ERP functions as tools for AI assistants

> ❗ **Memorize**
>
> The key insight: any software with functions that can be exposed as an API is a candidate for an MCP server. The server provides perfect access to every function the software offers. The question becomes: can you design an MCP client that interacts with the server effectively enough to produce real value? That is where the engineering skill lies.

9.2 Demo 2: AI Newsletter Pipeline — Multi-Source Research with MCP

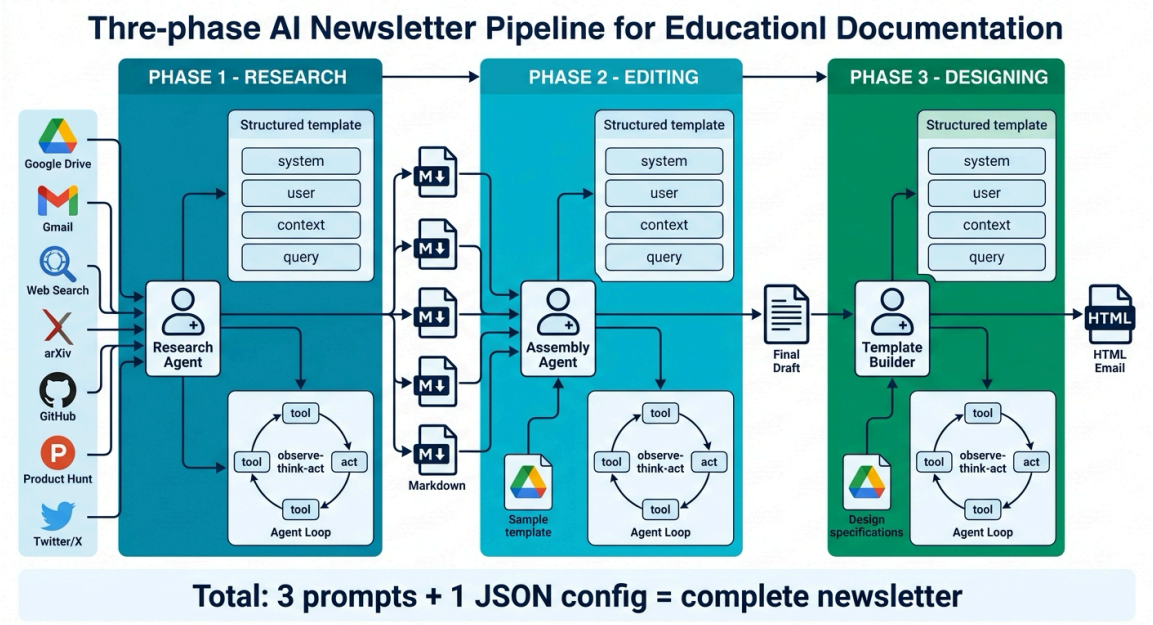

Figure 25: Three agents, seven MCP servers, one JSON config: from scattered sources to a production-ready HTML newsletter

The second demo automated the entire creation of a weekly AI newsletter using Claude Desktop connected to seven MCP servers. The workflow demonstrates MCP's power for **multi-source context assembly** — exactly the problem that made developers "human APIs."

9.2.1 Newsletter Structure

The newsletter (**CampusX AI Newsletter**) was designed with nine sections:

1. Introduction
2. Big Story of the Week
3. Quick Updates (3-4 bullet points)
4. Top Research Papers (from arXiv)
5. Top GitHub Repos
6. Learning Tutorial of the Week
7. Top AI Products (from Product Hunt)
8. Top Tweets
9. Closing Notes

9.2.2 Three-Phase Workflow

The entire newsletter was produced in three phases, each driven by a single prompt:

Table 28: Three-phase newsletter pipeline

Phase	Agent Role	MCP Tools Used	Output
1. Research	AI Newsletter Research Agent	Google Drive (read content ideas + past performance), Gmail (reader feedback), Web Search, arXiv, GitHub, Product Hunt, Twitter/X	5 markdown files (one per source category)
2. Editing	AI Newsletter Assembly Agent	Google Drive (read sample template)	Final draft combining 5 research files into newsletter format
3. Designing	Email Template Builder Agent	Google Drive (read design specs)	Production-ready HTML email

9.2.3 Phase 1: Research

The research prompt instructed Claude Desktop to act as an **AI newsletter research agent**. It first read two files from Google Drive: a content ideas document (topics the team wanted to cover) and a performance data spreadsheet (which past topics got the most engagement). It then checked Gmail for reader feedback and feature requests.

With this editorial context assembled, Claude conducted parallel research across five sources: web search for breaking AI news, arXiv for top papers, GitHub for trending repositories, Product Hunt for new AI products, and Twitter/X for viral AI discussions. Each source produced a separate markdown file saved to disk.

9.2.4 Phase 2: Editing

The editing prompt instructed Claude to read all five research markdown files plus a sample newsletter template from Google Drive (establishing tone, structure, and formatting conventions). Claude combined everything into a single final draft matching the template's style.

9.2.5 Phase 3: Designing

The design prompt took the final draft and converted it into production-ready HTML email, following design specifications (fonts, colors, layout) stored in Google Drive.

9.2.6 MCP Configuration

The entire setup required only a JSON configuration file in Claude Desktop listing each MCP server. The configuration was minimal — each server entry specified the command to launch it and any required environment variables (API keys). No custom integration code was written.

> **Info**
>
> Three prompts and a JSON config file replaced what would traditionally require a dedicated engineering team: data pipeline, content aggregation, editorial workflow, and email design. The MCP servers already existed; the value was in orchestrating them through well-crafted prompts.

9.3 Demo 3: Manim MCP — Mathematical Visualizations

The third demo connected the **Manim** MCP server to Claude Desktop. Manim is the Python visualization library created by Grant Sanderson (3Blue1Brown) for his mathematics YouTube channel — the same library behind the channel's signature animated explanations of linear algebra, calculus, and neural networks.

9.3.1 What Changed with MCP

Previously, creating Manim visualizations required writing Python code manually — a difficult and time-consuming process even for experienced developers. With the Manim MCP server, the workflow becomes:

1. Write a high-level English description of the mathematical concept you want to visualize
2. Send it as a prompt in Claude Desktop
3. Claude generates Manim code and sends it to the Manim MCP server
4. The server executes the code and produces a video
5. Output: a visualization that looks similar to what you see on 3Blue1Brown

9.3.2 Setup: JSON Config (No Connector Available)

Since Manim is a specialized local tool (not a major SaaS platform), there is no built-in connector. The setup uses manual JSON configuration:

1. **Install dependencies** — pip install manim and pip install mcp
2. **Clone the Manim MCP server repo** — From GitHub to a local directory
3. **Open Claude Desktop config** — Settings → Developer → Edit Config
4. **Add server entry to JSON** — Specify: (1) absolute path to python3 (from which python3), (2) path to manim executable (from which manim), (3) path to manim_server.py inside the cloned repo's src/ folder

5 **Save and restart Claude Desktop** — The Manim MCP server appears in the available tools list

9.3.3 Demo Result

The prompt asked for an animation showing **vector transformation in linear algebra**. Claude generated Manim code that:

- Drew a 2D coordinate grid with basis vectors **î** and **ĵ**
- Applied a matrix transformation to the grid
- Showed the grid bending under the transformation
- Displayed the new transformed vectors **î'** and **ĵ'**

The output was a video file — input was an English sentence, output was a publication-quality mathematical animation.

> 🔥 **Tip**
>
> Manim MCP exemplifies the **connector vs. JSON config** distinction covered earlier. Popular SaaS tools (Google Drive, Slack, Notion) get one-click connectors. Specialized or custom servers (Manim, Blender, your company's internal tools) use JSON configuration. Both connect identically at the protocol level — the only difference is the setup experience.

With MCP's real-world capabilities demonstrated, the next section addresses a technique for keeping tool-related context lean: Just-In-Time instructions.

10 Just-In-Time (JIT) Instructions

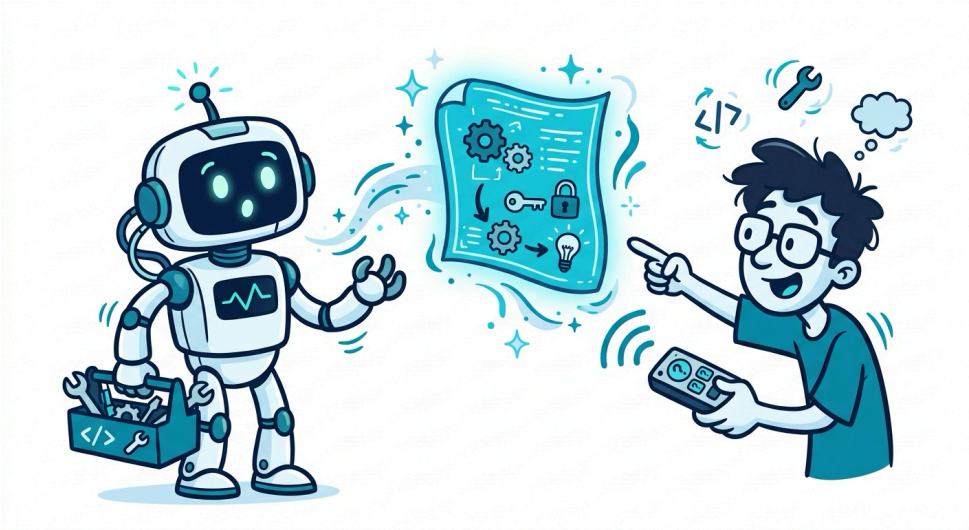

Figure 26: Why memorize the entire cookbook? JIT serves one recipe at a time, right when the chef needs it

10.1 The Problem JIT Solves

If 50-100 tools each have detailed usage instructions, loading all those instructions into the system prompt creates massive context bloat. JIT instructions solve this by delivering tool-specific guidance **only when that tool's results are being processed**. Anthropic (or Shopify; both are credited as pioneers) introduced this pattern.

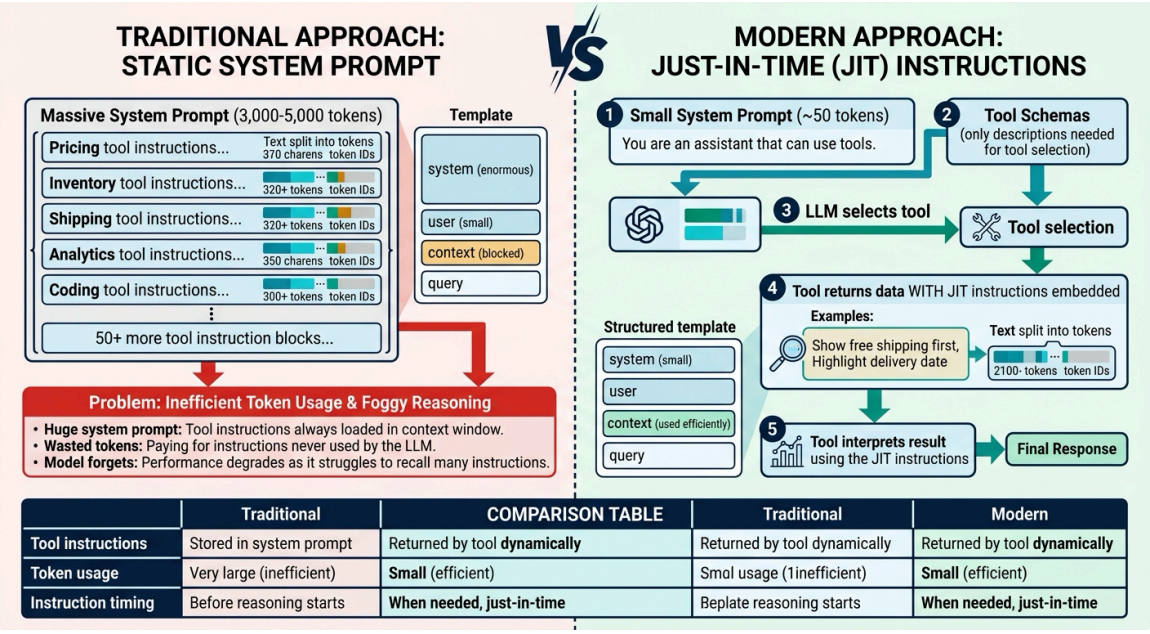

Figure 27: Traditional approach loads all tool instructions into a massive system prompt (3,000-5,000 tokens); JIT approach delivers instructions only when a tool is called

Without JIT

- System prompt contains ALL 100 tool instructions
- ~10K-20K tokens consumed permanently
- Context rot from instruction overload
- Instructions for irrelevant tools pollute the window

With JIT

- System prompt has minimal tool descriptions
- ~2K tokens for descriptions
- Tool-specific guidance arrives with tool results (~200 tokens)
- Only relevant instructions enter the context

10.2 JIT in Tool Results (Server to Client)

The MCP server embeds deterministic instructions alongside tool results using if/else logic:

JIT Instruction in Fuel Estimation Tool Result

```
{
  "result": {
    "fuel_kg": 52500,
    "cost_usd": 2625000,
    "safety_margin": 0.15
  },
  "jit_instructions": "IMPORTANT: 15% safety margin
    already included. If fuel > 50,000 kg, warn that
    this requires a heavy-lift vehicle. Present cost
    in millions (e.g., $2.6 million). Suggest checking
    weather at the launch site."
}
```

The server uses deterministic if/else conditions to select the appropriate JIT instruction. If fuel > 50,000 kg, include heavy-lift warning. If fuel < 10,000 kg, include light-payload note. This is not LLM-based: it is keyword/threshold logic on the server side.

10.3 JIT in Tool Calls (Client to Server)

The LLM can also embed JIT instructions in its tool call to tell the server to filter its response:

JIT Instruction in a Tool Call (Requesting Filtered Response)

```
{
  "tool": "search_missions",
  "arguments": { "status": "in_transit" },
  "jit_instructions": "Return only parameter A
    and parameter B"
}
```

Instead of the server returning 500 values from 10 tools (10 × 50), the JIT instruction tells the server to return only 20 values (10 × 2). The client receives a pre-filtered response and can directly pass it to the LLM without additional filtering logic.

10.4 JIT Concrete Example: Fuel Estimation

JIT-Enhanced Fuel Estimation Tool Result

```
{
  "tool_result": {
    "fuel_kg": 52500,
    "cost_usd": 2625000,
    "safety_margin": 0.15
  },
  "jit_instructions": "IMPORTANT: 15% safety margin
    already included. Fuel > 50,000 kg requires
    heavy-lift vehicle. Present cost as $2.6 million.
    Suggest checking weather at launch site."
}
```

For a weather tool, the JIT instruction becomes critical for safety:

Safety-Critical JIT Instruction for Weather Tool

```
{
  "tool_result": {
    "location": "Cape Canaveral",
    "condition": "Partly Cloudy",
    "wind_speed_kmh": 18,
    "launch_safe": true
  },
  "jit_instructions": "If launch_safe is false:
    CRITICAL. Do not suggest waiting for weather to
    clear. Policy requires 48-hour clear forecast.
    Recommend postponement and monitoring."
}
```

Safety-Critical Tools: Use Both

For safety-critical applications, put instructions in **both** the tool schema description (static guidance) **and** JIT results (dynamic, condition-dependent guidance). The schema says "always verify fuel sufficiency before recommending launch." The JIT provides specific thresholds based on the actual returned value.

10.5 When JIT Is Needed vs. Not

If 5-10 tools → **No JIT needed** — Include full descriptions in system prompt (~500-1,000 tokens)
If 50-100 tools → **JIT essential** — Loading all descriptions causes context bloat; use JIT for active tools only
If Safety-critical tools → **Both schema and JIT** — JIT in results for dynamic guidance + schema description for static safety rules
If Simple data retrieval → **No JIT needed** — Result is self-explanatory; no additional interpretation guidance needed

Info

Modern coding agents like Claude Code already handle context management well. JIT becomes essential primarily at scale (100+ tools) or for safety-critical applications where deterministic post-processing guidance is required.

With JIT addressing token efficiency, the next section covers how tools are orchestrated into different execution patterns before examining full autonomous agents.

11 Tool Execution Patterns

Before introducing autonomous agents, it is important to understand the four fundamental patterns for orchestrating tool calls. These patterns describe **how** an LLM (or an agent framework) arranges one or more tool invocations to accomplish a task. Every agent, regardless of architecture, uses one or more of these patterns under the hood.

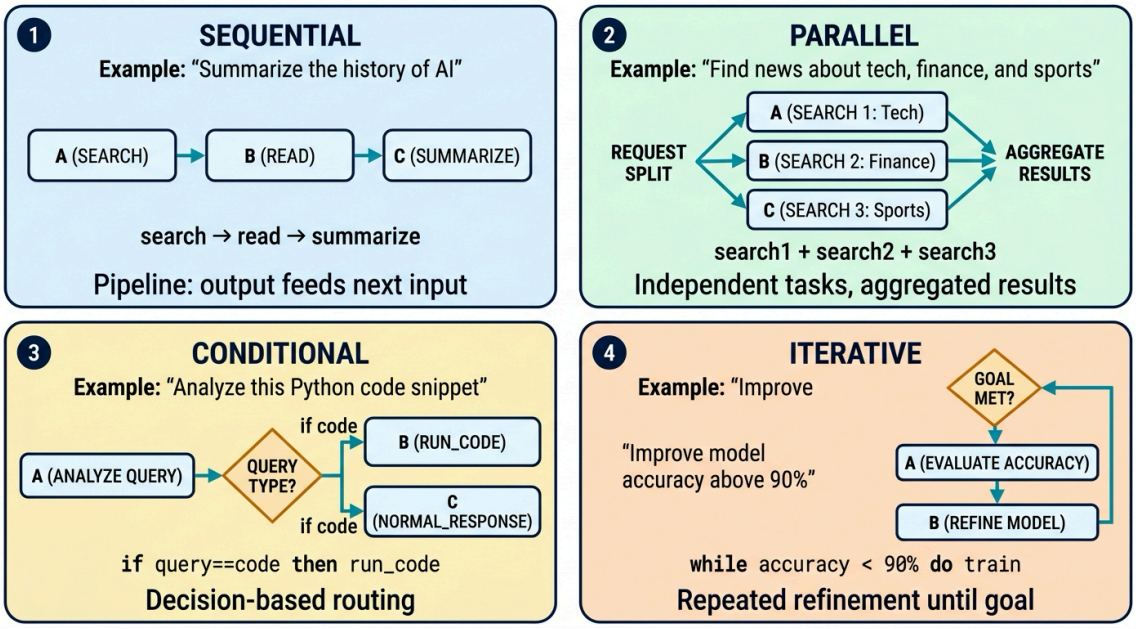

Figure 28: Four ways to wire tools: in a line, in parallel, through a decision gate, or in a loop — every agent is built from these primitives

11.1 Sequential (Pipeline)

In the sequential pattern, tool calls execute **one after another**, where each tool's output feeds directly into the next tool's input. This is the simplest and most common pattern.

Sequential Example: Summarize a Topic

```
User: "Summarize the history of AI"

Step 1: search("history of AI")       → [list of URLs]
Step 2: read(urls[0])                 → [article text]
Step 3: summarize(article_text)       → [summary]
```

When to use: Tasks with natural dependency chains where each step requires the output of the previous step. Most single-purpose tool interactions are sequential.

11.2 Parallel (Fan-Out / Fan-In)

In the parallel pattern, multiple independent tool calls execute **simultaneously**, and their results are aggregated. Modern LLMs with native tool calling (Claude, GPT-4) can emit multiple tool calls in a single response, enabling true parallelism.

Parallel Example: Multi-Topic News Search

```
User: "Find news about tech, finance, and sports"

Step 1 (parallel):
  search("tech news")      → [tech results]
  search("finance news")   → [finance results]
  search("sports news")    → [sports results]

Step 2: aggregate(tech, finance, sports) → [combined summary]
```

When to use: Tasks where multiple independent pieces of information are needed. Parallel execution reduces latency proportionally to the number of concurrent calls.

11.3 Conditional (Router)

In the conditional pattern, the LLM evaluates the input and **routes** to different tools based on a decision. This is analogous to an `if-else` branch in programming.

Conditional Example: Query Type Routing

```
User: "Analyze this Python code snippet"

Step 1: analyze(query)                → type = "code"
Step 2: if type == "code":
            run_code(snippet)         → [execution result]
        else:
            generate_response(query)  → [text response]
```

When to use: Tasks where the appropriate tool depends on input classification. Common in multi-modal agents that handle different query types (code, search, calculation, creative writing).

11.4 Iterative (Loop)

In the iterative pattern, tools execute in a **loop** that repeats until a goal condition is met. Each iteration refines the result, and the loop terminates when quality thresholds are satisfied or a maximum iteration count is reached.

Iterative Example: Model Training Loop

```
User: "Improve model accuracy above 90%"

Loop:
  Step 1: evaluate_accuracy(model)      → accuracy = 85%
  Step 2: refine_model(model, params)   → updated model
  Step 3: evaluate_accuracy(model)      → accuracy = 88%
  Step 4: refine_model(model, params)   → updated model
  Step 5: evaluate_accuracy(model)      → accuracy = 91% ✓ (goal met, exit)
```

When to use: Tasks requiring progressive improvement: code debugging (fix → test → fix), data cleaning (validate → correct → validate), or any workflow where quality is assessed and refined incrementally.

Warning

Iterative patterns require explicit stopping criteria (maximum iterations, token budget, or quality threshold). Without them, an agent can loop indefinitely, burning tokens and API credits. Always set both a maximum iteration count AND a maximum token budget.

Table 29: Four tool execution patterns and their characteristics

Pattern	Tool Calls	Data Flow	Best For
Sequential	One at a time, in order	Output of step N → input of step N+1	Dependency chains (search → read → summarize)
Parallel	Multiple simultaneously	Fan-out → aggregate	Independent queries (multi-topic search)
Conditional	One path chosen	Decision gate → branch	Input-dependent routing (code vs. text)
Iterative	Repeated until goal	Loop with exit condition	Progressive refinement (debug, train, clean)

> **Memorize**
>
> Real-world agents rarely use a single pattern in isolation. A ReAct agent is fundamentally **iterative** (thought-action-observation loop), but within each iteration it may use **sequential** tool chains or **parallel** tool calls. The Tool Calling Agent leverages **parallel** execution natively and can be wrapped in an **iterative** executor loop. Understanding these primitives helps you design and debug complex agent workflows.

With the execution patterns established, the next section covers how these patterns are embodied in full autonomous agent architectures.

12 Agents: Autonomous Tool-Using Systems

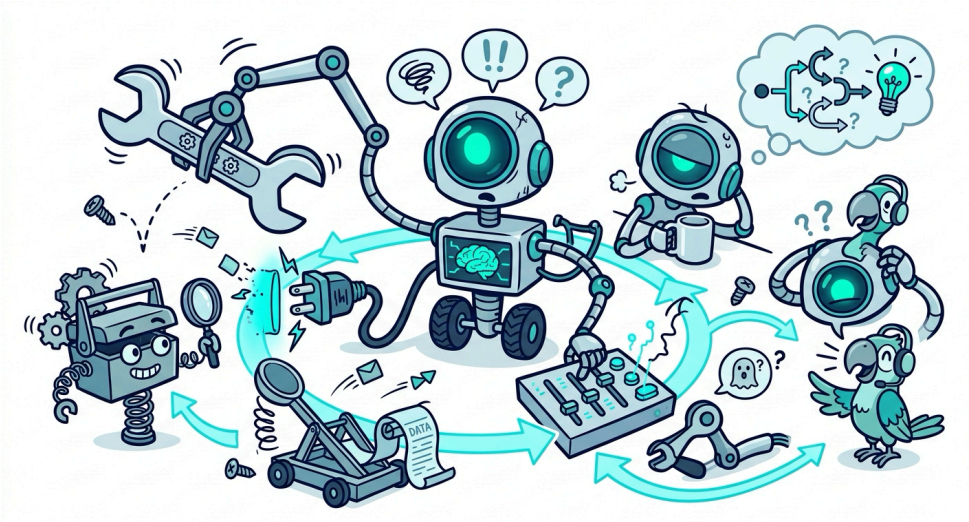

Figure 29: Think, act, observe, repeat: an agent loops through tools until the job is done (or the tokens run out)

An **agent** is an autonomous system that uses an LLM to decide which tools to use and how to use them. Unlike a single tool call, an agent can chain multiple tool calls, reason about intermediate results, and adapt its strategy based on observations.

12.1 Agent Components: The Four Building Blocks

Every agent consists of four components:

Table 30: Four components of an agent

Component	Role	Example
LLM Core	The reasoning engine that processes inputs and makes decisions	GPT-4, Claude Opus, Gemini
Tool Set	Collection of available tools the agent can use	Search, Calculator, Database, MCP tools
Prompt Template	Instructions that guide the agent's behavior and reasoning strategy	System prompt with role, constraints, output format
Memory	Stores conversation history and intermediate results for context	Chat history, agent scratchpad, prior observations

The `agent_scratchpad` is a special placeholder in the prompt template where intermediate reasoning steps (thoughts, actions, observations) are accumulated during execution. This is how the agent maintains state across a multi-step tool interaction.

Two dominant agent patterns exist: the **Tool Calling Agent** and the **ReAct Agent**. They differ in how the LLM decides which tools to use and how reasoning is structured.

12.2 The Tool Calling Agent

The Tool Calling Agent is the modern approach where the LLM **natively generates structured tool calls** using its built-in function calling capability. Instead of producing a text-based reasoning trace (like ReAct), the LLM directly emits a structured JSON object specifying which tool to call and what arguments to pass. The reasoning is implicit: it happens inside the LLM's forward pass, not as visible text.

12.2.1 How It Works

1. **User sends a prompt to the agent** — e.g., Search for LangChain tutorials and calculate 42 × 17

2. **LLM analyzes the prompt and available tool schemas** — The LLM's native function calling capability identifies which tools are needed

3. **LLM generates structured tool call(s)** — Outputs JSON with tool name, arguments, and a unique call ID. Can generate MULTIPLE tool calls in a single response (parallel execution).

4 **AgentExecutor dispatches each tool call** — The executor invokes each tool with its arguments and collects results

5 **Results are fed back to the LLM** — Each tool result is packaged as a ToolMessage (with matching call ID) and appended to the conversation

6 **LLM produces final answer or additional tool calls** — If more tools are needed, the loop continues. Otherwise, the LLM generates a natural language response.

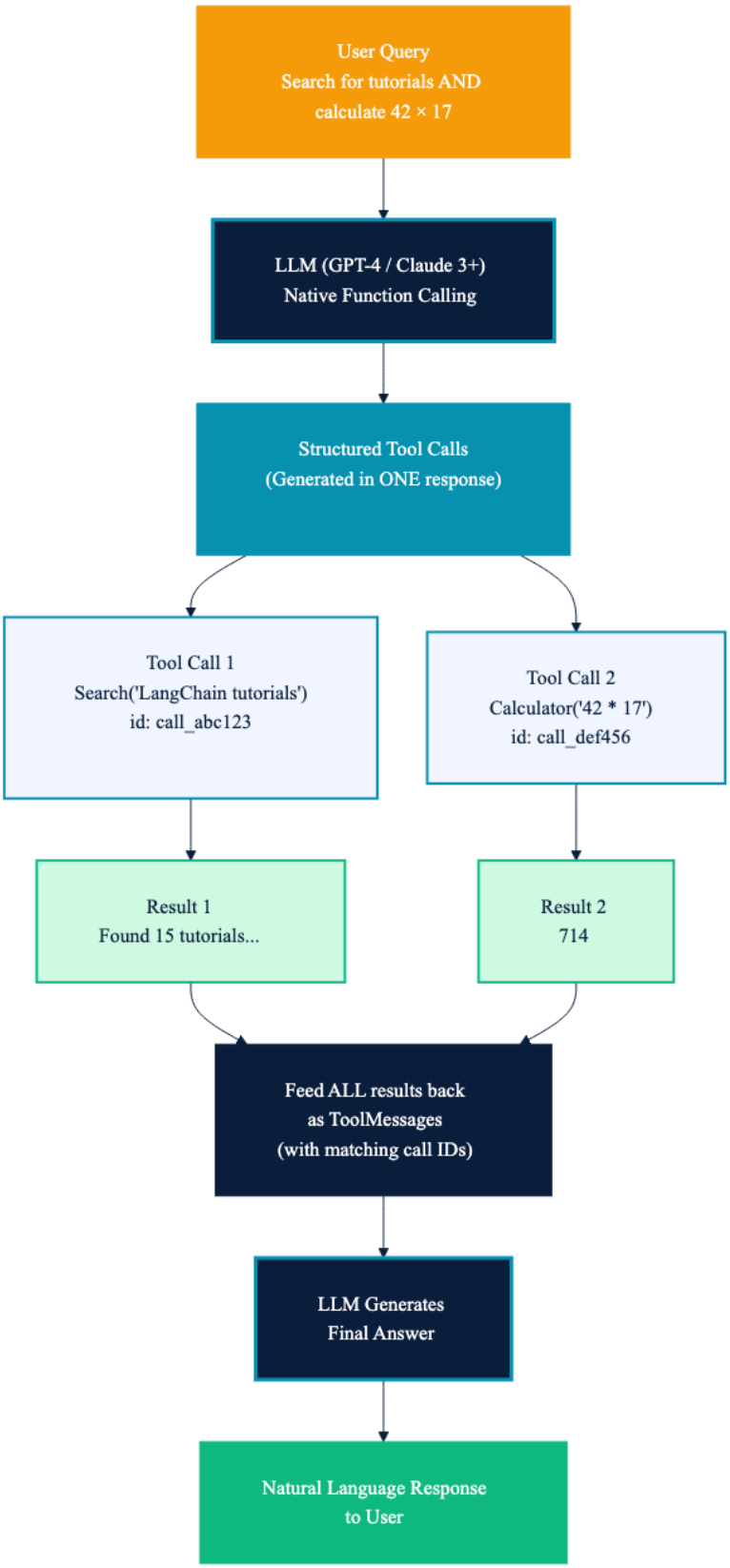

Figure 30: Tool Calling Agent: the LLM generates multiple structured tool calls in ONE response, enabling parallel execution. Both results feed back simultaneously.

12.2.2 Parallel Execution: The Key Advantage

Unlike ReAct (which is strictly sequential), the Tool Calling Agent can invoke **multiple tools in a single LLM call**. When the user asks "Search for LangChain tutorials AND calculate 42 × 17," the LLM generates both tool calls simultaneously:

Tool Calling Agent: Parallel Execution Trace

```
# LLM generates TWO tool calls in one response:
tool_calls: [
  {name: "Search", args: {query: "LangChain tutorials"},
   id: "call_abc123"},
  {name: "Calculator", args: {expression: "42 * 17"},
   id: "call_def456"}
]

# Both execute in parallel, results return together:
ToolMessage(content="Found 15 tutorials...",
            tool_call_id="call_abc123")
ToolMessage(content="714",
            tool_call_id="call_def456")

# LLM sees both results and generates final answer
```

With ReAct, this same task would require two sequential loops (search first, wait for result, think, then calculate, wait for result, think, answer): at least 6 steps and 2 full round trips to the LLM. The Tool Calling Agent does it in 2 steps and 1 round trip.

12.2.3 Implementation in LangChain

LangChain: Tool Calling Agent

```python
from langchain.agents import (
    create_tool_calling_agent, AgentExecutor)
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

# Prompt with agent_scratchpad placeholder
# for intermediate tool results
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    ("placeholder", "{chat_history}"),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
])

llm = ChatOpenAI(model="gpt-4", temperature=0)
agent = create_tool_calling_agent(llm, tools, prompt)

# AgentExecutor runs the loop:
# prompt -> LLM -> tool calls -> execute -> feed
# back -> LLM -> final answer (or more tool calls)
executor = AgentExecutor(
    agent=agent, tools=tools, verbose=True)
result = executor.invoke({
    "input": "Search for LangChain tutorials "
             "and calculate 42 * 17"
})
```

12.2.4 Concrete Trace: Single-Step Tool Calling

Query: “What is the weather in Cape Canaveral?”

Table 31: Tool Calling Agent trace: the LLM generates a structured tool call (not a text-based reasoning trace)

Step	Component	Content
1	User Input	What is the weather in Cape Canaveral?
2	LLM Output	tool_calls: `[{ "name": "get_weather", "args": { "location": "Cape Canaveral" }, "id": "call_001" }]`
3	Tool Execution	get_weather(location=Cape Canaveral) returns: `{ "condition": "Sunny", "temp": "72F" }`
4	ToolMessage	content: Sunny, 72F — tool_call_id: call_001
5	LLM Final	The weather in Cape Canaveral is sunny and 72°F. Great conditions for a launch.

12.2.5 Strengths and Limitations

Strengths

- Native LLM support (GPT-4, Claude 3+, Gemini)
- Parallel tool execution (multiple tools per call)
- Structured JSON output (built-in parameter validation)
- Fewer tokens (no explicit reasoning text generated)
- Cleaner implementation (less prompt engineering)
- Built-in error handling and retry mechanisms

Limitations

- Requires modern LLMs with function calling support
- Reasoning is implicit (harder to debug WHY a tool was chosen)
- API-specific features (different providers handle differently)
- Less control over reasoning strategy
- Cannot be used with older or open-source LLMs without tool support

The Tool Calling Agent is the recommended default for production applications where speed and token efficiency matter. The ReAct Agent, covered next, is preferred when reasoning transparency and debuggability are paramount.

12.3 The ReAct Agent

The **ReAct** (Reasoning + Acting) agent [11] uses an explicit **Thought → Action → Observation** loop. The reasoning is visible in the output, making it highly interpretable.

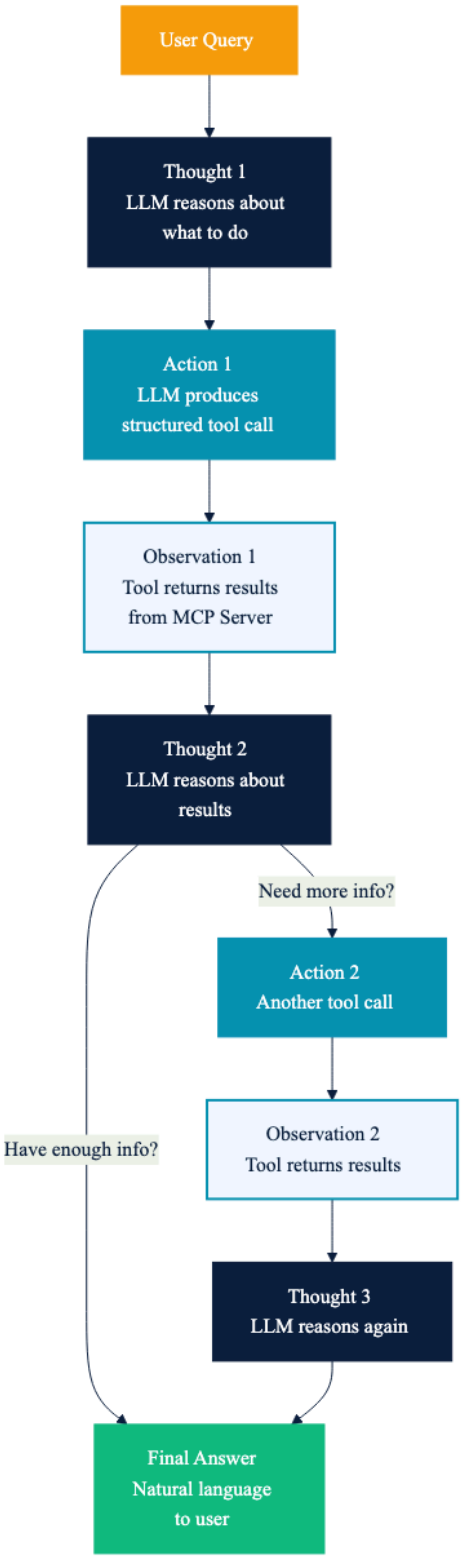

Figure 31: The ReAct loop: Thought, Action, Observation, repeat until the LLM has enough information to answer

</> LangChain: ReAct Agent

```python
from langchain.agents import (
    create_react_agent, AgentExecutor)
from langchain import hub

# ReAct prompt template from LangChain Hub
prompt = hub.pull("hwchase17/react")

agent = create_react_agent(
    llm=llm, tools=tools, prompt=prompt)

executor = AgentExecutor(
    agent=agent, tools=tools, verbose=True)

# Example ReAct trace in verbose output:
# Thought: I need to search for information
# Action: Search
# Action Input: "LangChain tutorials"
# Observation: LangChain is a framework...
# Thought: Now I need to calculate
# Action: Calculator
# Action Input: 42 * 17
# Observation: 714
# Thought: I have all the information
# Final Answer: ...
```

ReAct (Reasoning + Acting) is an agent loop pattern where the LLM alternates between explicit **thought** (reasoning) steps and **action** (tool call) steps, with **observation** (tool result) feeding back into the next thought.

The loop runs as follows: (1) **Thought**: the LLM reasons about what information it needs and which tool to call. (2) **Action**: the LLM produces a structured tool call. (3) **Observation**: the tool result (from the MCP server, via the client) is injected into the LLM's context. (4) The LLM then reasons again (next Thought), possibly calling another tool or producing the final answer. The loop can run 2, 3, or more times.

! Memorize

ReAct requires a model with internal reasoning capability. Regular models may keep relaying output to the user instead of entering the thought-action-observation cycle. Reasoning-capable models (Claude, GPT-o series) handle this naturally.

12.3.1 Single-Step ReAct Example

Query: "What missions are currently in transit?"

Table 32: Single-step ReAct trace: one thought-action-observation cycle

Step	Type	Content
1	Thought	The user wants missions currently in transit. I should search with status = in_transit.
2	Action	search_missions(query="in transit", status="in_transit")
3	Observation	{ "id": "M005", "name": "Mars Pathfinder Relay", "status": "in_transit", "crew": ["Chen", "Okafor"] }
4	Thought	I found one mission in transit: Mars Pathfinder Relay. I have all the details.
5	Answer	There is currently one mission in transit: Mars Pathfinder Relay, headed to Mars with crew members Chen and Okafor.

12.3.2 Multi-Step ReAct Example

Query: "Can we launch ISS Supply Delta on March 10th? Check the weather, cargo weight, then estimate fuel."

Table 33: Multi-step ReAct trace: three thought-action-observation cycles

Step	Type	Content
1	Thought	I need to: (1) check weather, (2) get cargo manifest, (3) estimate fuel.
2	Action	get_weather_forecast(location="Cape Canaveral", date="2026-03-10")
3	Observation	{ "condition": "Partly Cloudy", "wind_speed": 18, "launch_safe": true }
4	Thought	Weather is good, launch is safe. Now I need the cargo manifest.
5	Action	get_cargo_manifest(mission_id="M00003")
6	Observation	{ "total_weight_kg": 435 }
7	Thought	Cargo is 435 kg. Now I can estimate fuel for this payload to ISS.
8	Action	calculate_fuel_estimate(mission_id="M00003", destination="ISS", cargo_kg=435)
9	Observation	{ "fuel_kg": 52500, "cost_usd": 2625000, "safety_margin": 0.15 }
10	Thought	All clear: weather safe, cargo loaded, fuel estimated. Ready to answer.
11	Answer	Yes, ISS Supply Delta can launch on March 10th. Weather is clear at Cape Canaveral, cargo is 435 kg, estimated fuel is 52,500 kg (~\$2.6 million with 15% safety margin included).

> 🔥 **Tip**
>
> The thought step between each action is critical. After checking weather, the LLM reasoned that weather is safe. If it had been unsafe, the LLM would skip the cargo and fuel checks entirely, saving tokens and unnecessary tool calls.

12.3.3 ReAct with Error Handling

Query: "Update the Venus Atmosphere Probe mission to completed status."

Table 34: ReAct with error handling: the agent processes the error and explains it to the user

Step	Type	Content
1	Thought	User wants to update a mission status. First let me find the mission ID.
2	Action	search_missions(query="Venus Atmosphere Probe")
3	Observation	{ "id": "M008", "name": "Venus Atmosphere Probe", "status": "aborted" }
4	Thought	Found it: M008. Current status is aborted. Let me try to update it to completed.
5	Action	update_mission_status(mission_id="M008", status="completed")
6	Observation	{ "success": false, "error": "Invalid status transition. Cannot move from aborted to completed. Allowed transitions from aborted: none." }
7	Thought	The update failed because aborted is a terminal state. No further action is possible.
8	Answer	I cannot update mission M008 (Venus Atmosphere Probe). The mission status is aborted, which is a terminal state. It cannot be changed to completed or any other status.

12.3.4 Why the Thought Step Matters

Without the thought step, an agent with JIT instructions can still produce incorrect responses.

Scenario: A weather tool returns `launch_safe: false`. The JIT instruction says: "CRITICAL: Do not suggest waiting for weather to clear. Policy requires 48-hour clear forecast."

With Thought Step

- LLM reasons: JIT says I need a 48-hour clear forecast
- LLM calls the extended forecast tool before responding
- Correct behavior: makes another tool call

Without Thought Step

- LLM sees launch_safe: false
- Immediately outputs: Weather is bad, try again in a few hours
- Wrong behavior: skipped the JIT constraint entirely

> ⚡ **Danger**
>
> Without the thought step, the LLM may skip critical JIT constraints and produce incorrect outputs. The thought tokens consume more context but are of extremely high value: they improve the LLM's own context for the next decision.

12.4 Tool Calling Agent vs. ReAct Agent

Table 35: Tool Calling Agent vs. ReAct Agent comparison

Aspect	Tool Calling Agent	ReAct Agent
Approach	Direct function calling via LLM	Thought → Action → Observation loop
LLM Requirements	Modern LLMs with tool support (GPT-4, Claude 3+)	Any LLM with text generation
Reasoning	Implicit (internal to LLM)	Explicit (visible in output)
Performance	Faster, fewer tokens	3-5x more tokens, slower
Debugging	Less transparent	Highly transparent (full reasoning trace)
Parallel Execution	Native support (multiple tools in one call)	Sequential only
Error Recovery	Built-in retry mechanisms	Manual handling needed

If Using modern LLMs (GPT-4, Claude 3+) and need production efficiency → **Tool Calling Agent** — Faster, fewer tokens, parallel execution, cleaner implementation
If Need explicit reasoning traces or debugging complex workflows → **ReAct Agent** — Every reasoning step is visible; ideal for understanding agent behavior
If Working with older or open-source LLMs without native tool calling → **ReAct Agent** — Works with any LLM that can generate text; no special API features needed
If Teaching or demonstrating how agents work → **ReAct Agent** — The explicit thought-action-observation trace makes the process understandable

12.5 Token Consumption Trade-Off

Table 36: Token consumption trade-off between agent patterns

Aspect	Tool Calling Agent	ReAct Agent
Token usage	Lower: single action + response	3-5x higher: thought + action + observation per loop
Accuracy	Good: relies on LLM's internal reasoning	Higher: each step is explicitly reasoned; errors caught early
Best use case	Production agents where speed matters	Mission-critical tasks where reasoning transparency matters

> ⚠ **Warning**
>
> Tracking cumulative token consumption across a ReAct chain matters. Each loop adds thought tokens + tool result tokens. In the multi-step example above (weather → cargo → fuel), three full loops could consume significant context. If the context window fills, results may get truncated or context rot may degrade reasoning quality.

The following section consolidates the best practices for building MCP servers and clients, drawn from both instructors' recommendations.

13 MCP Best Practices

This section consolidates the most important do's and don'ts for building production MCP servers and clients, as emphasized throughout the lectures.

13.1 Tool Schema Design

Table 37: Tool schema design checklist

Practice	Why It Matters
Use verb-noun naming in snake_case	search_missions, get_weather — unambiguous names reduce LLM confusion
Write clear natural language descriptions	The LLM reads these to decide which tool to call; vague descriptions cause wrong tool selection
Enumerate categorical values explicitly	Listing allowed values (e.g., status: active, completed, aborted) drastically reduces hallucination
Mark required vs. optional parameters	Marking everything required forces the LLM to hallucinate values for fields it does not have
Keep parameters under 5	More parameters = more chances for the LLM to make errors; combine related fields into objects
Document return values	Describe what the tool returns so the LLM knows what to expect and can reason over results
Always add descriptions to every tool	A tool without a description is invisible to the LLM's decision-making process

13.2 Server Design

Table 38: MCP server design: do's and don'ts

Do	Don't
Keep tools minimal: fetch data only	Put LLM-dependent tasks (summarization, analysis) on the server
Handle authentication, rate limiting, and error handling on the server	Spread integration logic across client and server
Return raw data to the client; let the client decide what enters the LLM context	Return pre-processed or summarized results (that couples server to a specific LLM)
Use the prompt primitive to define server usage guidelines	Assume the LLM knows how to use your tools without guidance
Implement server-side validation (reject invalid operations)	Trust that the LLM will always send valid arguments
Default to read-only for sensitive systems	Expose delete/edit operations without explicit guardrails

13.3 Client Design

Table 39: MCP client design principles

Practice	Details
The client is the context gatekeeper	Control what tool schemas and what portion of results enter the LLM's context window
Use RAG for large tool sets (50+ tools)	Load minimum descriptions; let RAG select the 5-10 relevant tools per task
Group tools by category	Instead of loading all 3,000 Slack functions, load only the relevant group (Channels, Users, Admin)
Add triple confirmation for destructive operations	Require 3 user confirmations before delete/edit tool calls reach the server
Use JIT instructions at scale	At 100+ tools, deliver tool-specific guidance only when that tool's results are being processed
Make JIT instructions deterministic	No LLM logic needed in JIT — use an enumerated list of rules appended to tool results

13.4 Lifecycle and Operations

Table 40: Lifecycle and operational best practices

Practice	Details
Test with MCP Inspector before integrating	Debug your server in isolation (like Postman for MCP) before connecting to Claude Desktop
Always restart the host after config changes	Adding or modifying an MCP server in the JSON config requires a host restart
Use pings to keep connections alive	Periodic pings prevent OS/proxy from closing idle connections
Implement timeouts	Avoid requests hanging forever; free resources and inform the user when something fails
Send progress notifications for long tasks	Include a progressToken in requests; the server sends periodic updates so the client can show a progress bar
Respect the initialization handshake	No requests (except pings) before the handshake completes; violating this can crash the system
Handle shutdown gracefully	Client closes stdin → waits → SIGTERM → SIGKILL; client-side code should handle dropped connections and attempt reconnection

13.5 Security

Table 41: MCP security best practices

Practice	Details
Keep servers internal when possible	MCP servers do not need to be publicly accessible; run them locally or on internal networks
Centralize config in one JSON file	One file with all connections is easier to audit than API keys scattered across 10 different files
Restrict directory access for file system servers	Specify exactly which folders the server can access; never grant access to the entire machine
Eliminate dangerous operations if not needed	Remove delete/edit tools entirely from the server if the use case is read-only
Put safety instructions in both schema AND JIT	For safety-critical tools: static safety rules in the schema description + dynamic guidance in JIT results
Use the client-server split for defense in depth	Client provides guardrails before instructions reach the server; server validates at the tool level

> **Info**
>
> Use **FastMCP over raw MCP SDK** for server development. The MCP SDK is the low-level specification; FastMCP is the developer-friendly abstraction. In software, developer-friendly tooling tends to become the standard.

The following section collects practical implementation details, library references, and tooling recommendations.

14 Implementation Notes

- **FastMCP v2:** `pip install fastmcp` — the recommended library for building MCP servers. Handles protocol, transport, handshake, and tool registration via decorators.
- **MCP SDK:** `pip install mcp[cli]` — official Anthropic SDK with mcp.server, mcp.client, mcp.cli. More verbose than FastMCP but includes FastMCP v1 internally.
- **UV Package Manager:** `pip install uv` — faster than pip, recommended by the FastMCP ecosystem for project management.

- **MCP Inspector:** `uv run fastmcp dev main.py` — debugging tool for MCP servers (like Postman for MCP). Shows available tools, lets you test them, and displays JSON-RPC message history.
- **LangChain MCP Adapters:** `pip install langchain-mcp-adapters` — lightweight library for building custom MCP clients compatible with LangChain/LangGraph.
- **FastMCP Cloud:** fastmcp.cloud — free hosting for remote MCP servers with auto-deploy from GitHub.
- **aiosqlite:** Async SQLite library required for production remote servers handling concurrent users.
- **Vercel:** Common hosting for MCP client (web frontend).
- **Railway:** Common hosting for MCP server (Python backend).
- **Blender MCP:** Built-in plugin for Blender 4.4.3+; starts MCP server on a local port; communicates via stdio.
- **GitHub: awesome-mcp-servers:** Repository with 89K+ stars cataloging MCP servers by category.
- **PaperBanana:** Google Cloud AI Research tool using iterative ReAct loops for publication-quality figures; `pip install paperbanana`.
- **Claude Code and modern coding agents** handle context management automatically for most use cases; JIT is needed primarily at scale (100+ tools).

To consolidate, the following takeaways distill the most important concepts from the entire guide.

15 Key Takeaways

- LLMs have three fundamental limitations (no live data, unreliable computation, no real-world effects); tools solve all three
- LLMs do NOT execute tools; they produce structured tool calls; the tool's own logic runs the function
- Tool schemas must be well-designed: verb-noun naming, clear descriptions, enumerated constraints, required vs. optional parameters, return descriptions
- MCP standardizes LLM-tool communication, reducing the N × M integration problem to N + M
- MCP evolved from function calling's critical flaw: every tool integration was custom-coded on the client side; MCP shifts heavy lifting to the server side

- The MCP client is the gatekeeper of the LLM's context; it controls what tool schemas and results the LLM sees
- The MCP server executes tools and returns results but has no knowledge of or control over the LLM's context
- MCP servers should do the minimum necessary work; LLM-based processing belongs on the client side
- Three MCP primitives: tools (functions), resources (documents), prompts (reusable templates)
- MCP is bidirectional: servers can request LLM generation (sampling) and user input (elicitation) from the client
- Capabilities exchanged during handshake (roots, sampling, elicitation, tools, resources, prompts) form a contract governing what each party can request
- Version negotiation during initialization prevents incompatible client-server combinations from silently failing
- JSON-RPC 2.0 distinguishes requests (have ID, expect response) from notifications (no ID, fire-and-forget); confusing the two causes protocol errors
- Transport: stdio for local (same device), HTTP/SSE for remote (over network)
- Remote servers must use async/await to handle concurrent users; synchronous code blocks all users in a queue
- Always test servers in MCP Inspector before connecting to Claude Desktop; follow the incremental development workflow
- JIT instructions reduce context bloat by delivering tool-specific guidance only when needed; essential at 100+ tool scale
- Agents are autonomous systems with four components: LLM Core, Tool Set, Prompt Template, and Memory
- Two agent patterns: Tool Calling Agent (implicit reasoning, parallel execution, modern LLMs) and ReAct Agent (explicit reasoning, sequential, any LLM)
- Tool Calling Agent is faster and more token-efficient; ReAct Agent is more transparent and debuggable
- The thought step in ReAct is critical; without it, LLMs skip JIT constraints and produce incorrect outputs
- MCP enables multi-agent orchestration (covered in the companion Central Brain Pattern guide for Day 5)

16 Glossary

Term	Definition
Tool	An external function or API that an LLM can invoke to access data, perform computation, or take real-world actions
Tool Schema	A JSON description of a tool's name, purpose, parameters, and return format; enables the LLM to decide how to call it
Function Calling	The capability (introduced by OpenAI mid-2023) for LLMs to invoke external functions by producing structured output with function name and arguments
MCP	Model Context Protocol: open-source standard (Anthropic, Nov 2024) for LLM-tool communication; reduces N×M integrations to N+M
MCP Host	The AI application the user interacts with (Claude Desktop, Cursor, custom chatbot); contains the LLM
MCP Client	Helper entity inside the host that translates between the host's language and the MCP protocol; one client per server
MCP Server	Hosts and executes tools; exposes tools, resources, and prompts to the client
MCP Primitives	The three capabilities an MCP server exposes: tools (actions), resources (read-only data), prompts (templates)
MCP Capabilities	Features declared during handshake: client capabilities (roots, sampling, elicitation) and server capabilities (tools, resources, prompts)
AI Fragmentation	The problem where each AI-enabled software operates in isolation: Notion's AI has no awareness of Slack's AI, creating disconnected AI worlds
JSON-RPC 2.0	Wire protocol used for all MCP messages; defines Request (with ID), Response (matching ID), and Notification (no ID) message types
MCP Lifecycle	Three-stage session: Initialization (handshake), Normal Operation (discovery + usage), Shutdown
Connector	Built-in feature in Claude Desktop that links to MCP servers automatically (like an App Store for MCP servers)
stdio	Standard I/O transport for local (same-device) client-server communication
Streamable HTTP	Transport for remote (networked) client-server communication
FastMCP	Python library (v2) for building MCP servers with minimal boilerplate; \@mcp.tool() decorator pattern
MCP Inspector	Debugging tool for MCP servers; like Postman for MCP; shows tools, tests calls, displays JSON-RPC history

LangChain MCP Adapters	Library for building custom MCP clients compatible with LangChain/LangGraph ecosystem
MultiServerMCPClient	LangChain class that creates one client per server and fetches all tools from all connected servers simultaneously
MCPToolkit	Alternative LangChain class for single-server MCP client connections with a simpler interface
Agent	An autonomous system that uses an LLM to decide which tools to use and how; consists of LLM Core, Tool Set, Prompt Template, and Memory
Tool Calling Agent	Agent pattern where the LLM natively generates structured tool calls with implicit reasoning; supports parallel execution
ReAct Agent	Agent pattern using explicit Thought → Action → Observation loop; reasoning is visible in output
AgentExecutor	LangChain class that runs an agent in a loop: sends prompts, executes tool calls, feeds results back until a final answer
agent_scratchpad	Prompt placeholder where intermediate reasoning steps accumulate during multi-step agent execution
JIT Instructions	Just-In-Time instructions embedded in tool results or tool calls to provide context-efficient guidance
Central Brain	An orchestrator agent that delegates tasks to specialized agents via two-way MCP; holds global context but has no execution tools (covered in Day 5)
Two-way MCP	Bidirectional MCP communication where both parties can send and receive; enables multi-agent orchestration
Sampling	MCP capability where the server requests LLM generation from the host's model; inverts the typical client→server flow
Elicitation	MCP capability where the server prompts the user for interactive input via the client
Roots	MCP capability where the client grants the server access to specific filesystem directories
progressToken	Token included in tool call metadata that enables the server to send progress notifications back to the client
aiosqlite	Async SQLite library; required for production remote MCP servers to handle concurrent users without blocking
FastMCP Cloud	Free hosting platform (fastmcp.cloud) for MCP servers with auto-deploy from GitHub
Snake Case	Naming convention using underscores: search_missions (preferred for tool names)

PaperBanana	Google Cloud AI Research tool using iterative ReAct loops to generate publication-quality figures
Context Assembly Problem	The pre-tool bottleneck where developers manually copy-pasted context from scattered sources (Jira, GitHub, databases, Drive, Slack) into ChatGPT; made developers 'human APIs'
Human API	A developer whose primary role has become manually assembling context for an LLM by copy-pasting from multiple applications
Blender MCP	Built-in MCP server plugin for Blender 4.4.3+; exposes hundreds of 3D modeling functions as tools for natural language control
Manim	Python visualization library (3Blue1Brown); Manim MCP server enables mathematical animation generation from natural language prompts
SKILL.md	Markdown files that capture a team's working style and preferences; used to give LLMs consistent creative output matching established patterns

17 Notation Reference

Symbol / Term	Meaning
N × M	Number of custom integrations without MCP (N LLMs × M tools)
N + M	Number of integrations with MCP (N LLM clients + M MCP servers)
JSON-RPC 2.0	Wire protocol for MCP communication (Request, Response, Notification)
ReAct	Reasoning + Acting (agent loop pattern)
JIT	Just-In-Time (instructions delivered dynamically, not preloaded)
stdio	Standard Input/Output (local transport)
SSE	Server-Sent Events (streaming transport)
WSCI	Write-Select-Compress-Isolate (context engineering framework from Day 3)
@mcp.tool()	FastMCP decorator for exposing a function as a tool
@mcp.resource()	FastMCP decorator for exposing a document as a resource
tools/list, tools/call	Standard MCP operations for tool discovery and invocation
notifications/cancelled	Client cancellation notification (timeout exceeded)
notifications/progress	Server progress update for long-running tasks
progressToken	Client-assigned token in request metadata enabling server progress updates

initialized	Notification (not request) sent by client after successful handshake
sampling	Server-to-client capability: server requests LLM generation from host
elicitation	Server-to-client capability: server requests user input via client
roots	Client-to-server capability: grants filesystem access to server

18 Open Questions / Areas for Further Study

1. **MCP Host vs. MCP Client distinction:** An ongoing discussion in the community. The terms are sometimes used interchangeably but may have subtle architectural differences. *Monitor the MCP specification repository for clarification.*
2. **MCP Security:** How to prevent unauthorized tool execution, implement authentication between client and server, and enforce rate limiting on MCP servers. *Critical for production deployments.*
3. **Dynamic tool schema compression:** Can tool descriptions themselves be RAG-retrieved and compressed for token efficiency? *Approach: embed all tool descriptions in a vector database; retrieve only the top-K most relevant for each query.*
4. **ReAct vs. Plan-and-Execute:** When should an agent plan all steps upfront vs. reason step-by-step? *Trade-off: Plan-and-Execute reduces token overhead but cannot adapt to unexpected tool results. ReAct adapts but costs more tokens.*
5. **Multi-agent MCP topologies:** How do multiple agents share MCP servers? Can agents communicate with each other via MCP without a central brain? *Explore: peer-to-peer MCP patterns vs. hub-and-spoke orchestration.*
6. **Multi-agent orchestration:** The Central Brain pattern (covered in the Day 5 companion guide) uses two-way MCP to coordinate specialized agents. How does this scale to 10+ agents? *See: Day 5 guide for full treatment.*
7. **MCP versioning and backward compatibility:** How does the protocol handle server updates that change tool schemas? *Consideration: schema versioning, graceful degradation, and client-side fallback strategies.*
8. **Two-way MCP implementation:** Detailed protocol for bidirectional communication. How does state synchronization work between orchestrator and agents? *Read: MCP specification for server-to-client notifications.*

9. **Autonomous ReAct loop stopping criteria:** How many iterations should be allowed before forcing a response? *Approach: set a maximum iteration count AND a maximum token budget; stop at whichever is reached first.*
10. **MCP Authentication and Sessions:** How to implement user authentication on remote MCP servers? Without user isolation, all users share data. *Critical for any shared deployment. Future MCP specification updates may standardize auth flows.*
11. **MCP Sampling Guardrails:** How should clients implement guardrails for server sampling requests? Rate limiting, cost caps, user approval for sensitive generations, and audit logging are all open design questions. *Critical for preventing abuse in multi-tenant deployments.*

References

[1] L. Martin, "Context Engineering for Agents." [Online]. Available: https://blog.langchain.com/context-engineering-for-agents/

[2] OpenAI, "Introducing ChatGPT." [Online]. Available: https://openai.com/blog/chatgpt

[3] OpenAI, "Function Calling and Other API Updates." [Online]. Available: https://openai.com/index/function-calling-and-other-api-updates/

[4] Anthropic, "Introducing the Model Context Protocol." [Online]. Available: https://www.anthropic.com/news/model-context-protocol

[5] JSON-RPC Working Group, "JSON-RPC 2.0 Specification." [Online]. Available: https://www.jsonrpc.org/specification

[6] Model Context Protocol, "MCP Specification." [Online]. Available: https://spec.modelcontextprotocol.io/

[7] Community Contributors, "Awesome MCP Servers." [Online]. Available: https://github.com/punkpeye/awesome-mcp-servers

[8] Anthropic, "MCP Python SDK." [Online]. Available: https://github.com/modelcontextprotocol/python-sdk

[9] J. Lowin, "FastMCP: A Fast, Pythonic Way to Build MCP Servers and Clients." [Online]. Available: https://github.com/jlowin/fastmcp

[10] LangChain, "LangChain MCP Adapters." [Online]. Available: https://github.com/langchain-ai/langchain-mcp-adapters

[11] S. Yao *et al.*, "ReAct: Synergizing Reasoning and Acting in Language Models," in *International Conference on Learning Representations (ICLR)*, 2023.

Follow me for More AI Content

If you found these notes useful, connect with me on LinkedIn for more deep dives into Machine Learning, Artificial Intelligence, and Computer Vision.

Isham Rashik on LinkedIn

Scan the QR code or click the link above