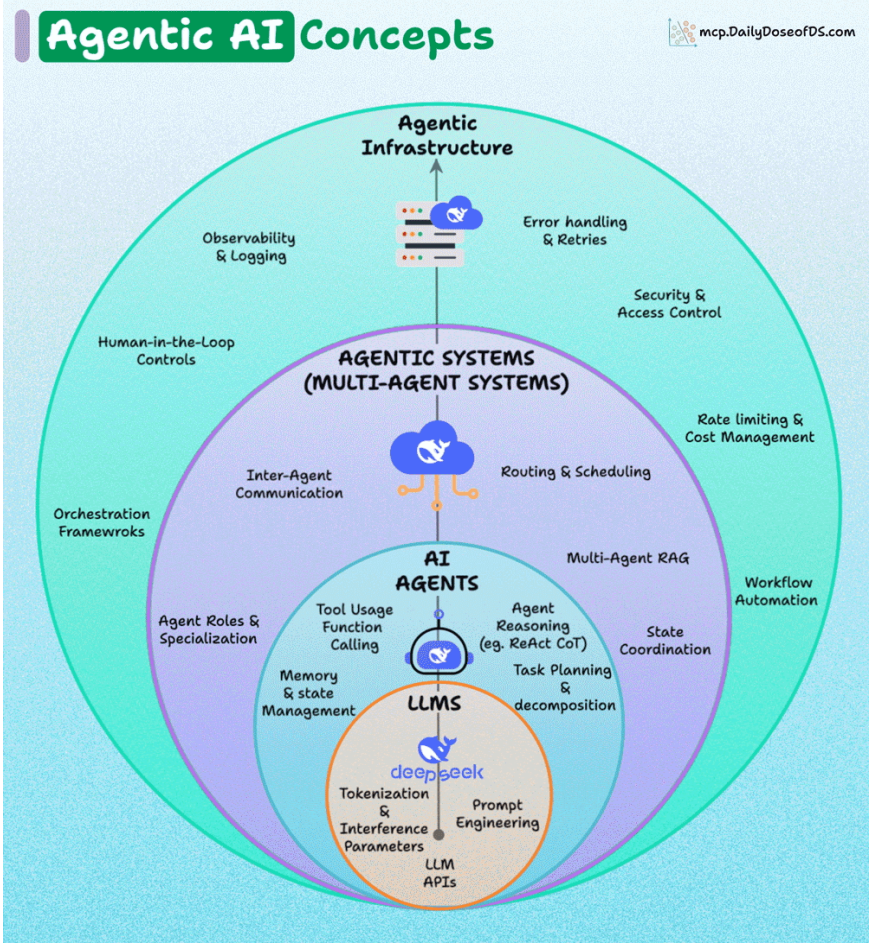
Router: A mechanism that directs tasks to the most appropriate agent or tool.

Each of these terms forms a key piece of the agentic AI ecosystem that AI engineers should know.

4 Layers of Agentic AI

The following graphic depicts a layered overview of Agentic AI concepts, depicting how the ecosystem is structured from the ground up (LLMs) to higher-level orchestration (Agentic Infrastructure).



Let's break it down layer by layer:

1) LLMs (foundation layer)

At the core, you have LLMs like GPT, DeepSeek, etc.

Core concepts here:

- Tokenization & inference parameters: how text is broken into tokens and processed by the model.
- Prompt engineering: designing inputs to get better outputs.
- LLM APIs: programmatic interfaces to interact with the model.

This is the engine that powers everything else.

2) AI Agents (built on LLMs)

Agents wrap around LLMs to give them the ability to act autonomously.

Key responsibilities:

- Tool usage & function calling: connecting the LLM to external APIs/tools.
- Agent reasoning: reasoning methods like ReAct (reasoning + act) or Chain-of-Thought.
- Task planning & decomposition: breaking a big task into smaller ones.
- Memory management: keeping track of history, context, and long-term info.

Agents are the brains that make LLMs useful in real-world workflows.

3) Agentic systems (multi-agent systems)

When you combine multiple agents, you get agentic systems.

Features:

- Inter-Agent communication: agents talking to each other, making use of protocols like ACP, A2A if needed.
- Routing & scheduling: deciding which agent handles what, and when.
- State coordination: ensuring consistency when multiple agents collaborate.
- Multi-Agent RAG: using retrieval-augmented generation across agents.

- Agent roles & specialization: Agents with unique purposes
- Orchestration frameworks: tools (like CrewAI, etc.) to build workflows.

This layer is about collaboration and coordination among agents.

4) Agentic Infrastructure

The top layer ensures these systems are robust, scalable, and safe.

This includes:

- Observability & logging: tracking performance and outputs (using frameworks like DeepEval).
- Error handling & retries: resilience against failures.
- Security & access control: ensuring agents don't overstep.
- Rate limiting & cost management: controlling resource usage.
- Workflow automation: integrating agents into broader pipelines.
- Human-in-the-loop controls: allowing human oversight and intervention.

This layer ensures trust, safety, and scalability for enterprise/production environments.

Overall, Agentic AI, as a whole, involves a stacked architecture, where each outer layer adds reliability, coordination, and governance over the inner layers.

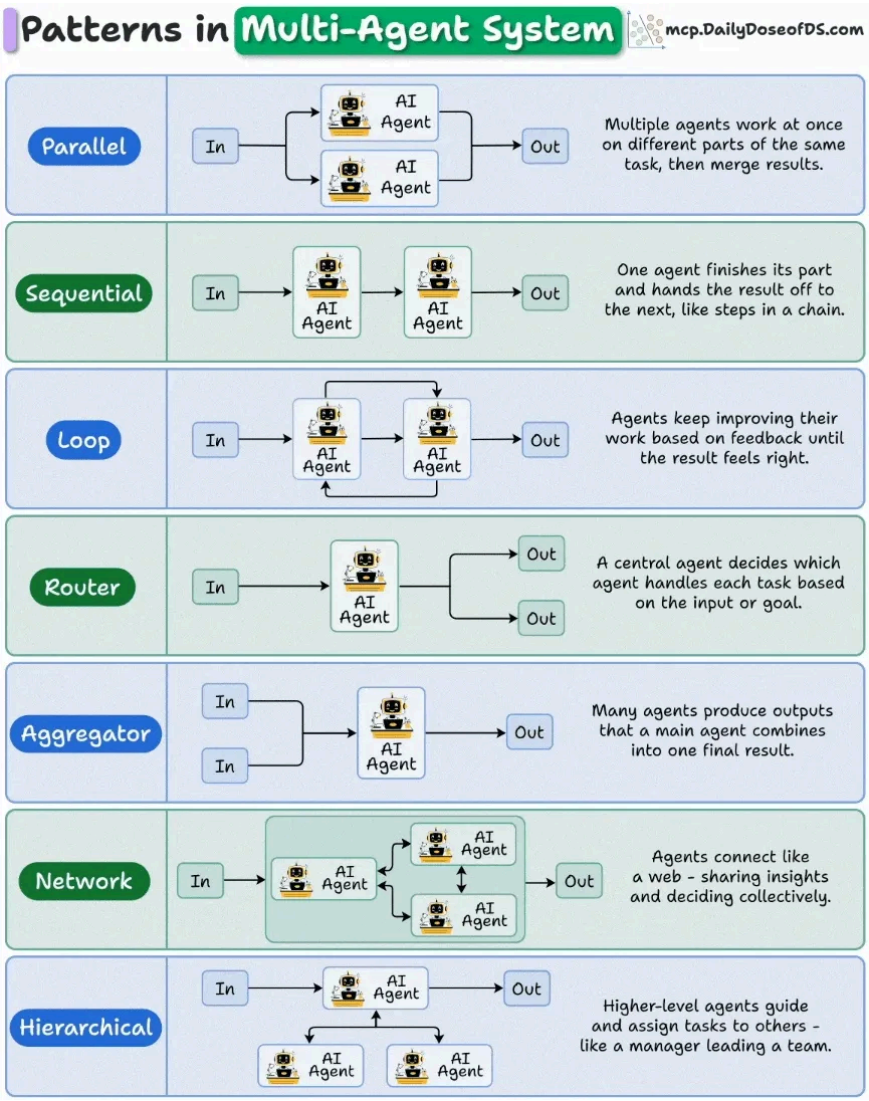
7 Patterns in Multi-Agent Systems

Monolithic agents (single LLMs stuffed with system prompts) didn't sustain for too long.

We soon realized that to build effective systems, we need multiple specialized agents that can collaborate and self-organize.

To achieve this, several architectural patterns have emerged for AI agents.

This visual explains the 7 core patterns of multi-agent orchestration, each suited for specific workflows:



1) Parallel

Each agent tackles a different subtask, like data extraction, web retrieval, and summarization, and their outputs merge into a single result.

Perfect for reducing latency in high-throughput pipelines like document parsing or API orchestration.

2) Sequential

Each agent adds value step-by-step, like one generates code, another reviews it, and a third deploys it.

You'll see this in workflow automation, ETL chains, and multi-step reasoning pipelines.

3) Loop

Agents continuously refine their own outputs until a desired quality is reached.

Great for proofreading, report generation, or creative iteration, where the system thinks again before finalizing results.

4) Router

Here, a controller agent routes tasks to the right specialist. For instance, user queries about finance go to a FinAgent, legal queries to a LawAgent.

It's the foundation of context-aware agent routing, as seen in emerging MCP/A2A-style frameworks.

5) Aggregator

Many agents produce partial results that the main agent combines into one final output. So each agent forms an opinion, and a central one aggregates them into a consensus.

Common in RAG retrieval fusion, voting systems, etc.

6) Network

There's no clear hierarchy here, and agents just talk to each other freely, sharing context dynamically.

Used in simulations, multi-agent games, and collective reasoning systems where free-form behavior is desired.

7) Hierarchical

A top-level planner agent delegates subtasks to workers, tracks their progress,

and makes final calls. This is exactly like a manager and their team.

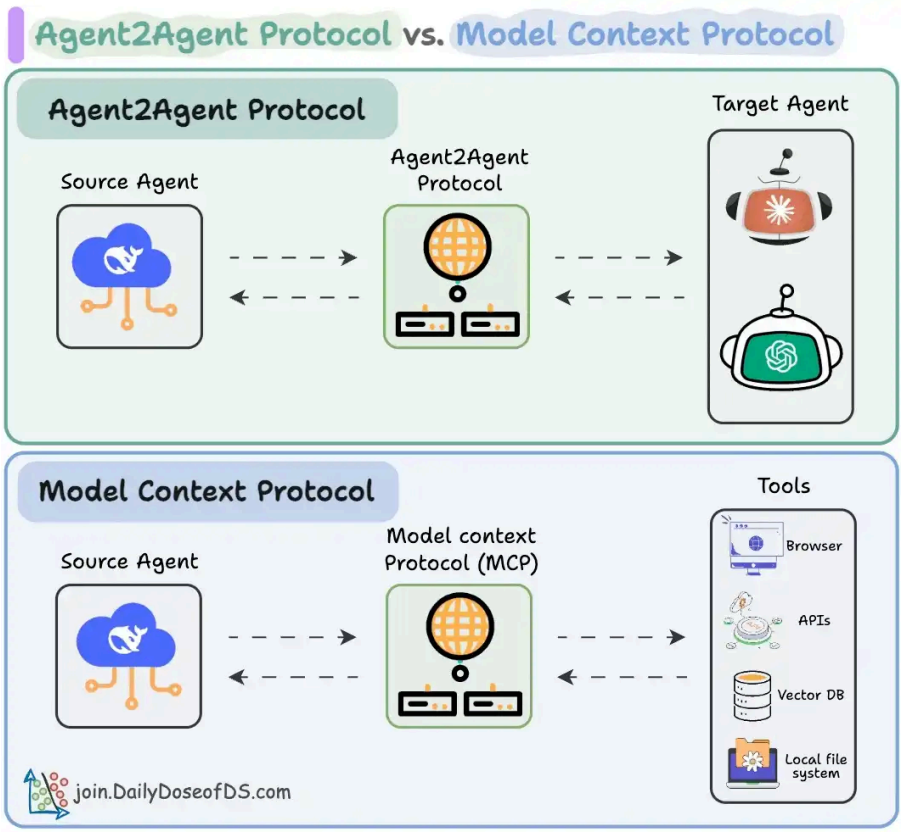
One thing we constantly think about when picking a pattern to build a multi-agent system (provided we do need an Agent and it has to be a multi-agent system) is not which one looks coolest, but which one minimizes friction between agents.

It's easy to spin up 10 agents and call it a team. What's hard is designing the communication flow so that:

- No two agents duplicate work.
- Every agent knows when to act and when to wait.
- The system collectively feels smarter than any individual part.

Agent2Agent(A2A) Protocol

Agentic applications require both A2A and MCP.



MCP provides agents with access to tools.

While A2A allows agents to connect with other agents and collaborate in teams.

Let's clearly understand what A2A is and how it can work with MCP.

If you don't know about MCP servers, we covered them in detail in the next section.

In a gist:

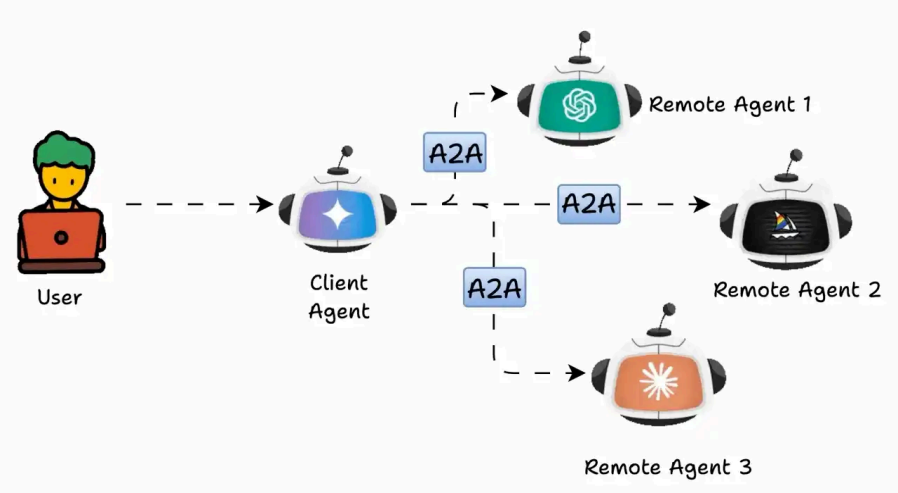
Agent2Agent (A2A) protocol lets AI agents connect to other Agents.

Model context protocol lets AI Agents connect to Tools/APIs.

So using A2A, while two Agents might be talking to each other...they themselves might be communicating to MCP servers.

In that sense, they do not compete with each other.

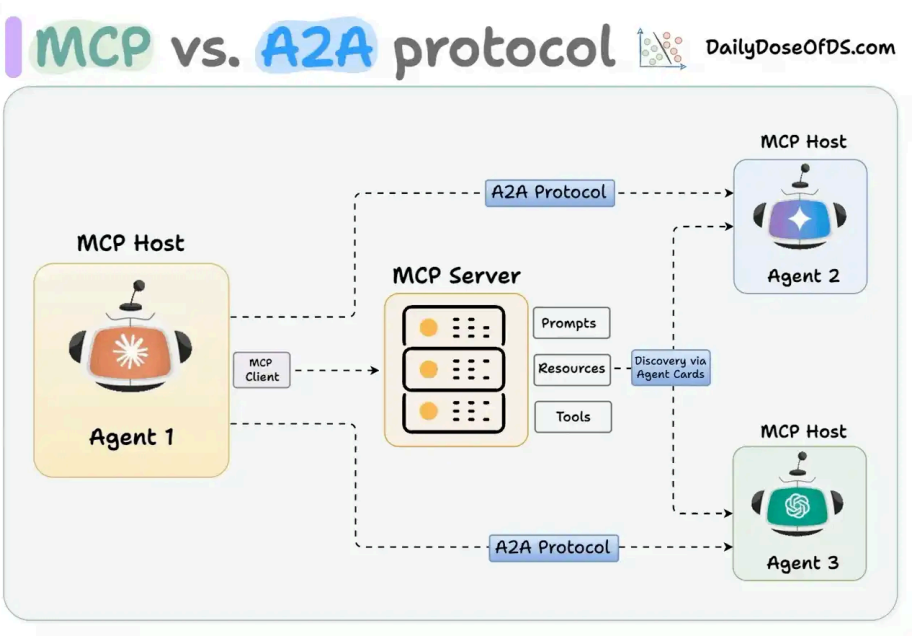
To explain further, Agent2Agent (A2A) enables multiple AI agents to work together on tasks without directly sharing their internal memory, thoughts, or tools.



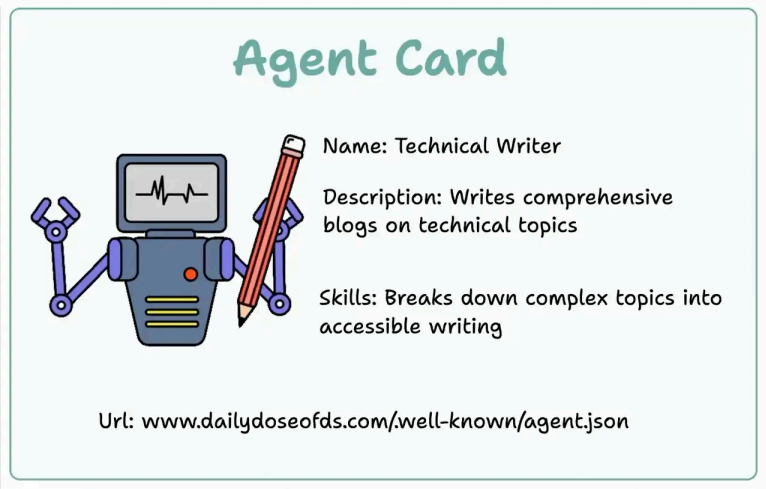
Instead, they communicate by exchanging context, task updates, instructions, and data.

Essentially, AI applications can model A2A agents as MCP resources,

represented by their AgentCard (more about it shortly).



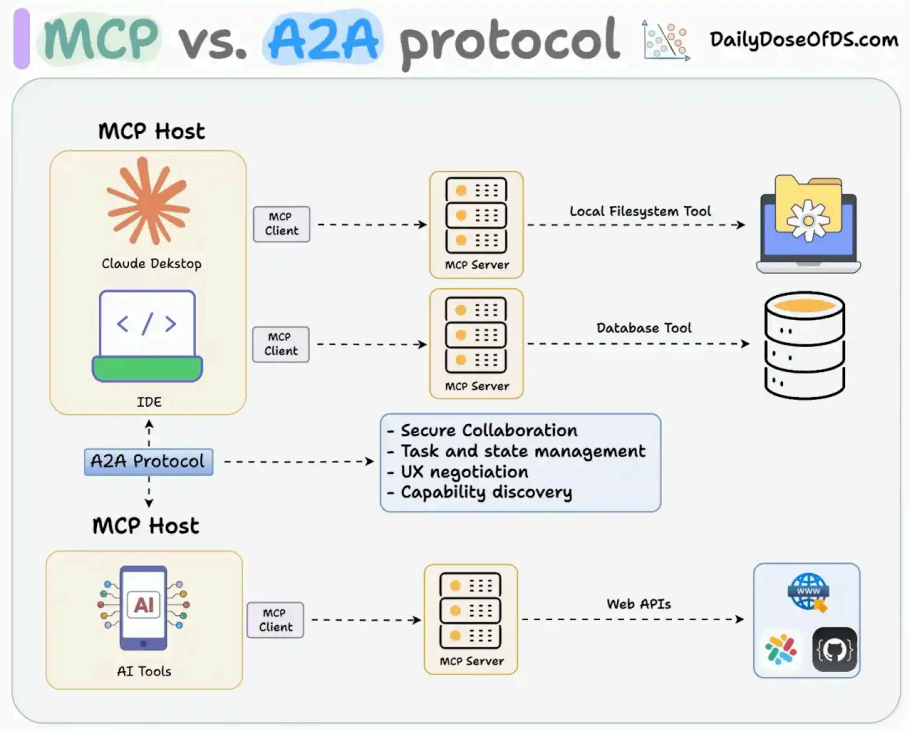
Using this, AI agents connecting to an MCP server can discover new agents to collaborate with and connect via the A2A protocol.



A2A-supporting Remote Agents must publish a "JSON Agent Card" detailing their capabilities and authentication.

Clients use this to find and communicate with the best agent for a task.

There are several things that make A2A powerful:



- Secure collaboration
- Task and state management
- Capability discovery
- Agents from different frameworks working together (LlamaIndex, CrewAI, etc.)

Additionally, it can integrate with MCP.

It's good to standardize Agent-to-Agent collaboration, similar to how MCP does for Agent-to-tool interaction.

Agent-User Interaction Protocol(AG-UI)

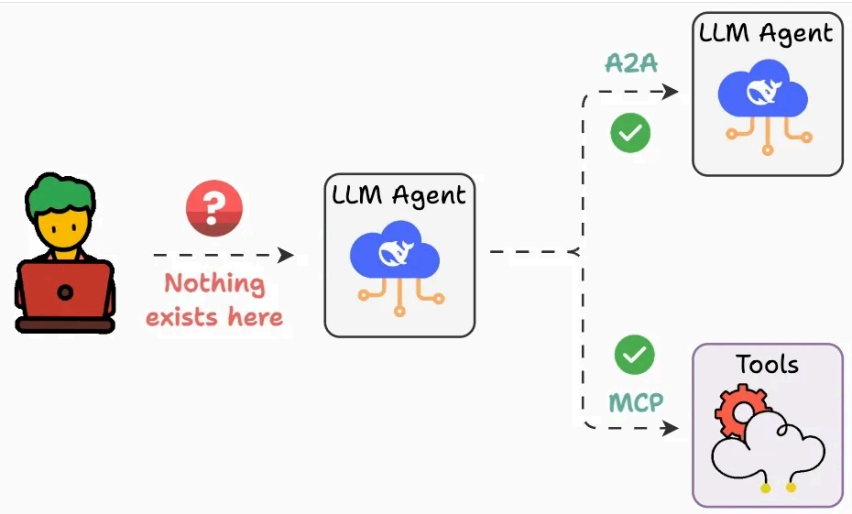
In the realm of Agents:

- MCP standardized Agent-to-Tool communication.

- Agent2Agent protocol standardized Agent-to-Agent communication.

But there's one piece still missing...

And that's a protocol for Agent-to-User communication:



Let's understand why this is important.

The problem

Today, you can build powerful multi-step agentic workflows using a toolkit like LangGraph, CrewAI, Mastra, etc.

But the moment you try to bring that Agent into a real-world app, things fall apart:

- You want to stream LLM responses token by token, without building a custom WebSocket server.

- You want to display tool execution progress as it happens, pause for human feedback, without blocking or losing context.
- You want to sync large, changing objects (like code or tables) without re-sending everything to the UI.
- You want to let users interrupt, cancel, or reply mid-agent run, without losing context.

And here's another issue:

Every Agent backend has its own mechanisms for tool calling, ReAct-style planning, state diffs, and output formats.

So if you use LangGraph, the front-end will implement custom WebSocket logic, messy JSON formats, and UI adapters specific to LangGraph.

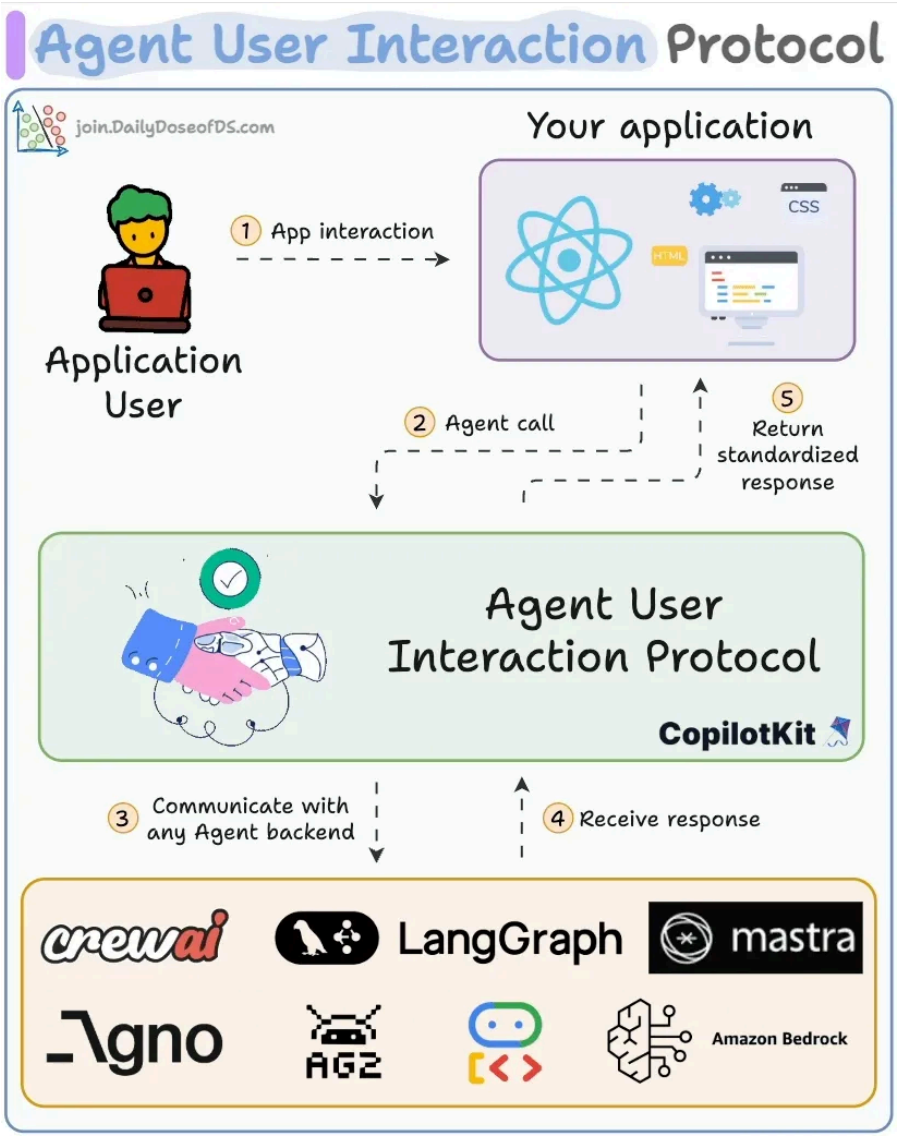
But to migrate to CrewAI, everything must be adapted.

This doesn't scale.

The solution: AG-UI

AG-UI (Agent-User Interaction Protocol) is an open-source protocol by CopilotKit that solves this.

It standardizes the interaction layer between backend agents and frontend UIs (the green layer below).



Think of it this way:

- Just like REST is the standard for client-to-server requests...
- AG-UI is the standard for streaming real-time agent updates back to the UI.

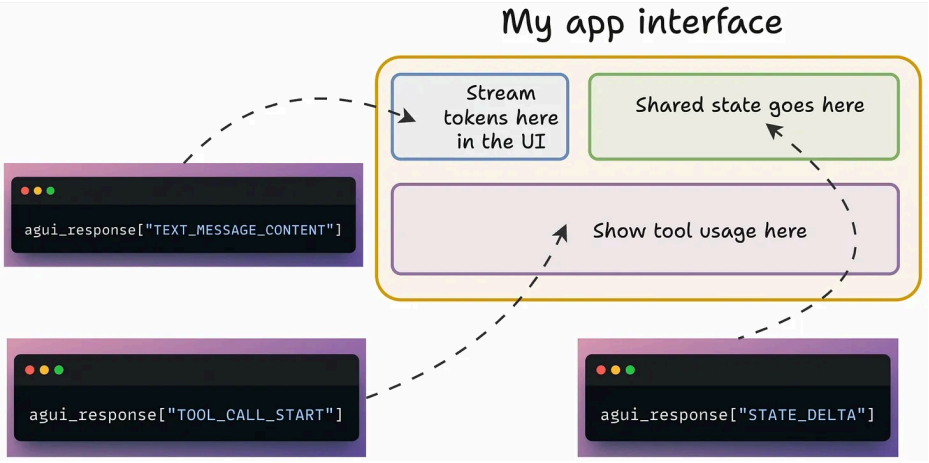
Technically speaking...

It uses Server-Sent Events (SSE) to stream structured JSON events to the frontend.

Each event has an explicit payload (like keys in a Python dictionary) like:

- TEXT_MESSAGE_CONTENT for token streaming.
- TOOL_CALL_START to show tool execution.
- STATE_DELTA to update shared state (code, data, etc.)
- AGENT_HANDOFF to smoothly pass control between agents

And it comes with SDKs in TypeScript and Python to make this plug-and-play for any stack, like shown below:



In the above image, the response from the Agent is not specific to any toolkit. It is a standardized AG-UI response.

This means you need to write your backend logic once and hook it into AG-UI, and everything just works:

- LangGraph, CrewAI, Mastra—all can emit AG-UI events.
- UIs can be built using CopilotKit components or your own React stack.
- You can swap GPT-4 for Llama-3 locally and not change anything in the frontend.

This is the layer that will make your Agent apps feel like real software, not just glorified chatbots.

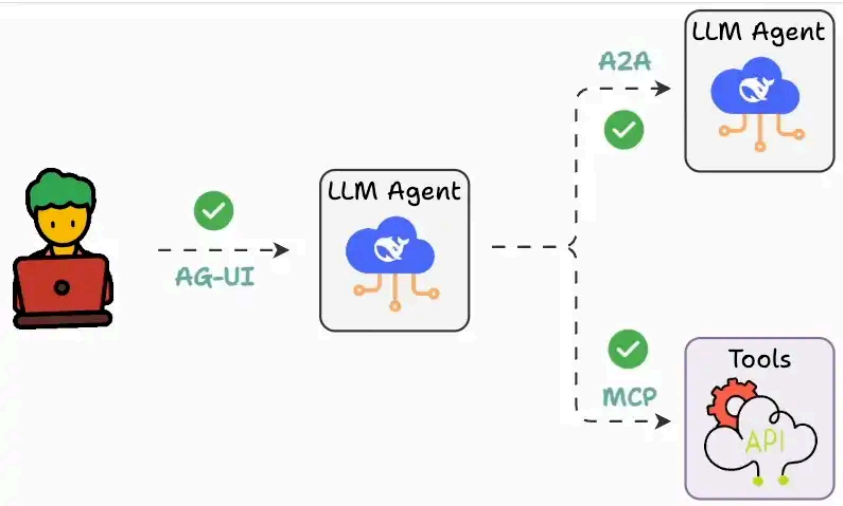
Agent Protocol Landscape

Something remarkable is happening in the AI industry.

Earlier the agent ecosystem was fragmented into dozens of incompatible frameworks.

But finally, the industry is converging around three protocols that work together.

These are:



AG-UI (Agent-User Interaction):

- The bi-directional connection between agentic backends and frontends.
- This is how agents become truly interactive inside your apps, not just as chatbots, but collaborative co-workers.

MCP (Model Context Protocol):

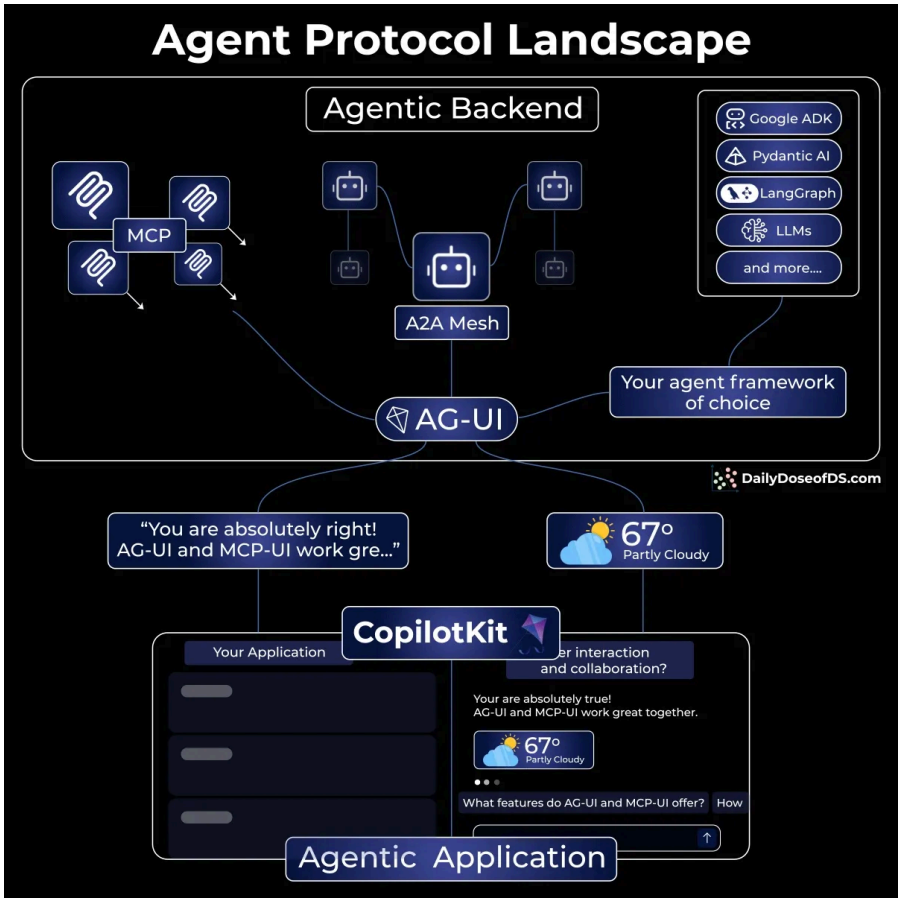
- The standard for how agents connect to tools, data, and workflows.
- Started by Anthropic, now adopted everywhere.

A2A (Agent-to-Agent):

- The protocol for multi-agent coordination.
- How agents delegate tasks and share intent across systems.

These aren't competing standards. They're layers of the same stack.

AG-UI can handshake with both MCP and A2A, meaning tool outputs and multi-agent collaboration can flow seamlessly to your user interface.



Your frontend stays connected to the entire agent ecosystem through one unified protocol layer.

CopilotKit sits above all three as the Agentic Application Framework.

It acts as the practical layer that lets you actually build with these protocols without dealing with the complexity.

So you get all three protocols, generative UI support and production-ready infrastructure in one framework.

The best part: All of this is 100% open-source!

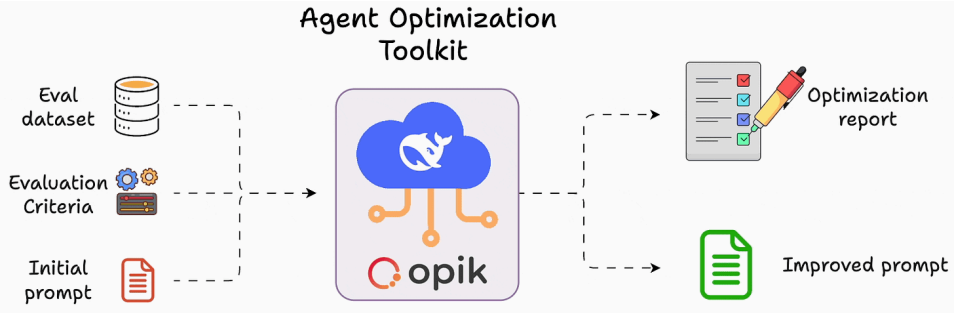
It breaks down handshakes, misconceptions and real examples and shows exactly how to start building.

Agent optimization with Opik

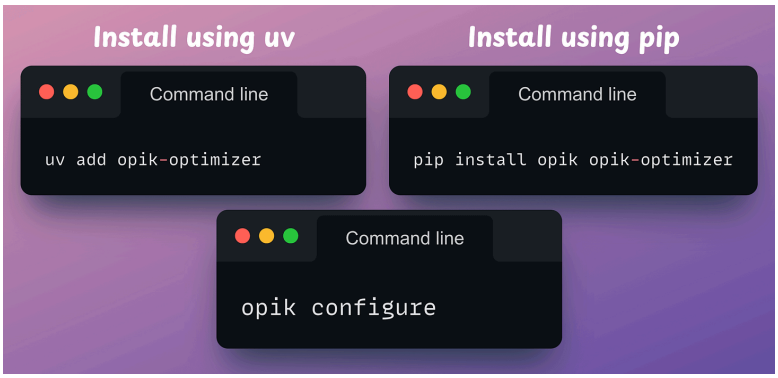
Developers manually iterate through prompts to find an optimal one. This is not scalable and performance can degrade across models.

Let's learn how to use the Opik Agent Optimizer toolkit that lets you automatically optimize prompts for LLM apps.

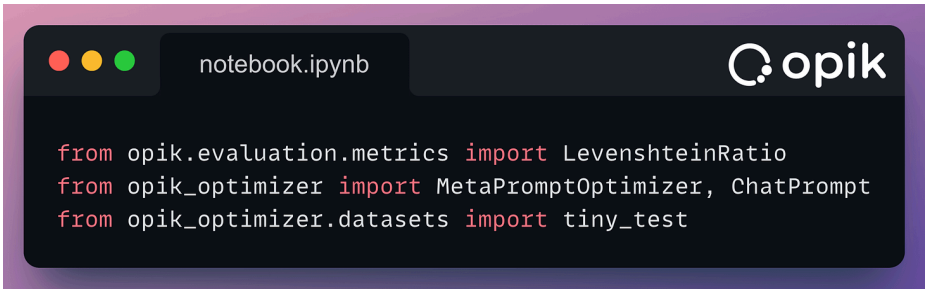
The idea is to start with an initial prompt and an evaluation dataset, and let an LLM iteratively improve the prompt based on evaluations.



To begin, install Opik and its optimizer package, and configure Opik:

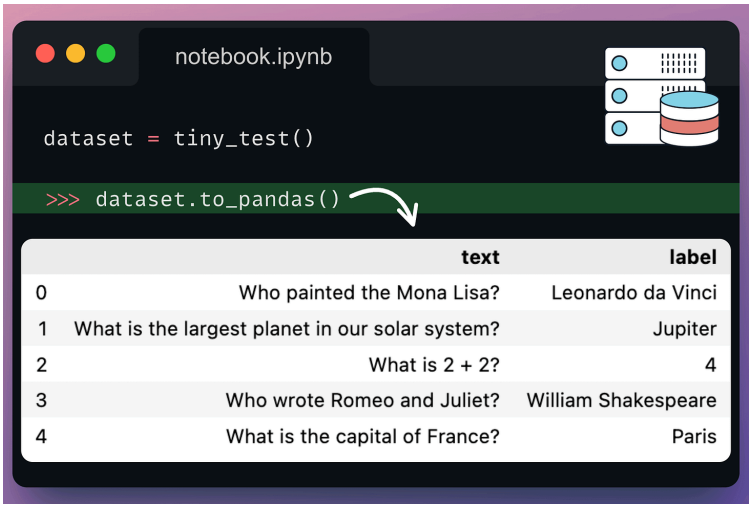


Next, import all the required classes and functions from opik and opik_optimizer:



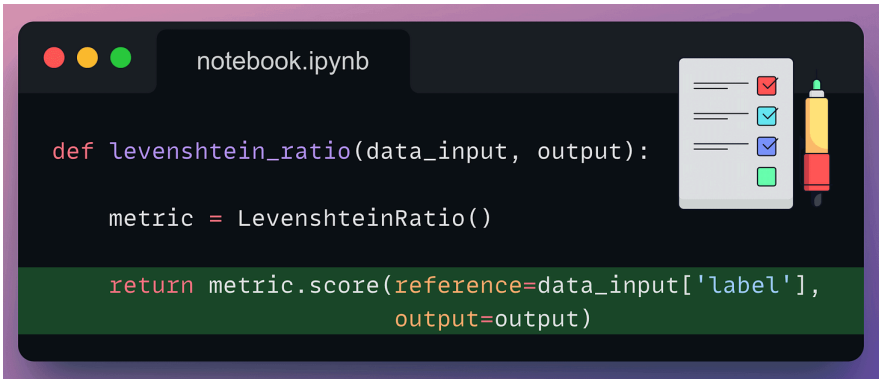
- LevenshteinRatio → Our metric to evaluate the prompt's effectiveness in generating a precise output for the given input.
- MetaPromptOptimizer → An algorithm that uses a reasoning model to critique and iteratively refine your initial instruction prompt.
- tiny_test → A basic test dataset with input-output pairs.

Next, define an evaluation dataset:

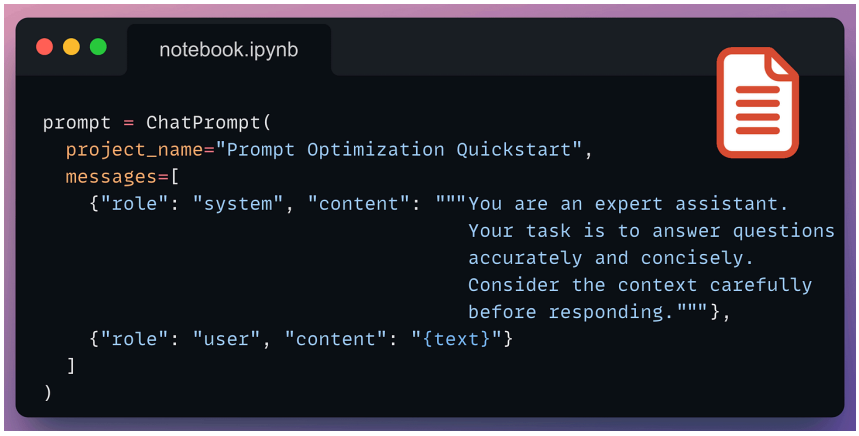


	text	label
0	Who painted the Mona Lisa?	Leonardo da Vinci
1	What is the largest planet in our solar system?	Jupiter
2	What is 2 + 2?	4
3	Who wrote Romeo and Juliet?	William Shakespeare
4	What is the capital of France?	Paris

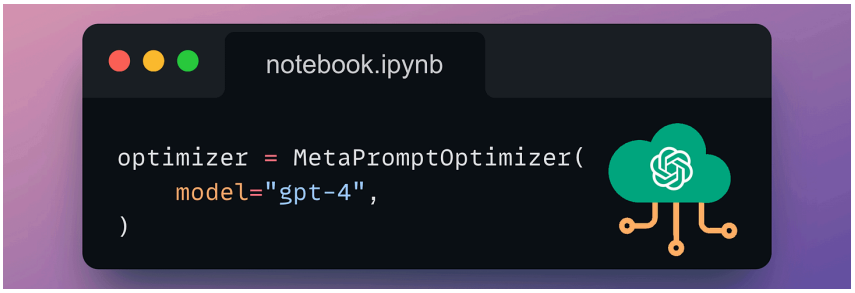
Moving on, configure the evaluation metric, which tells the optimizer how to score the LLM's outputs against the given label:



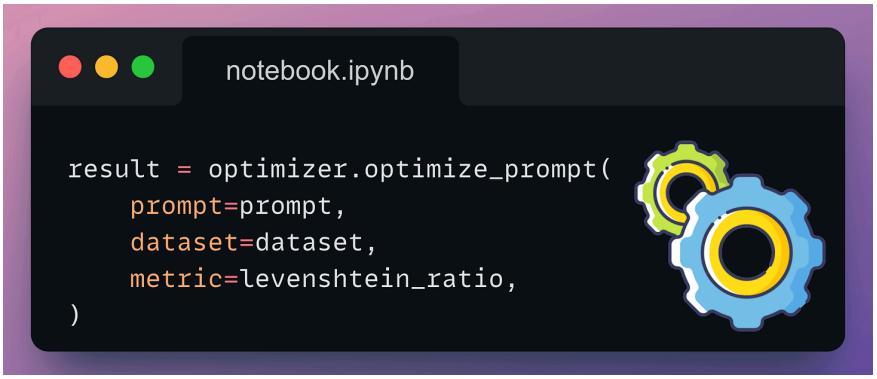
Next, define your base prompt, which is the initial instruction that the MetaPromptOptimizer will try to enhance:



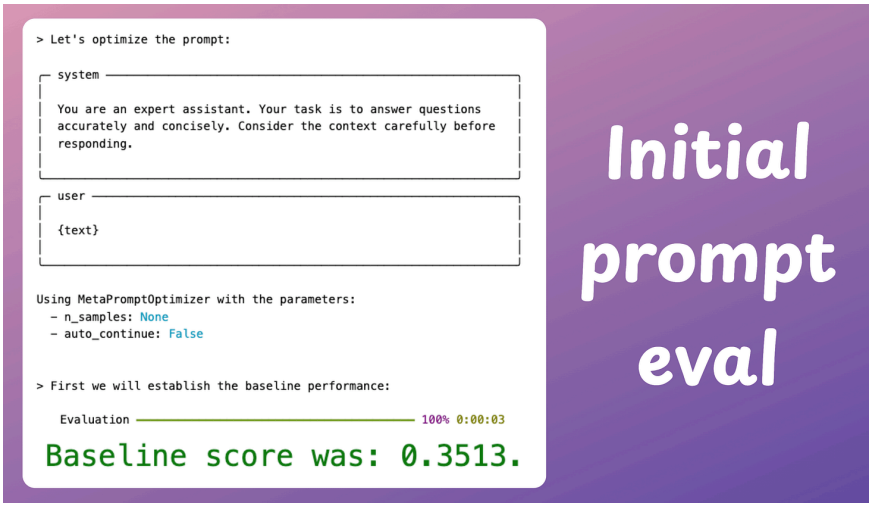
Next, instantiate a MetaPromptOptimizer, specifying the model to use in the optimization process:



Finally, the optimizer.optimize_prompt(...) method is invoked with the dataset, metric configuration, and prompt to start the optimization process:

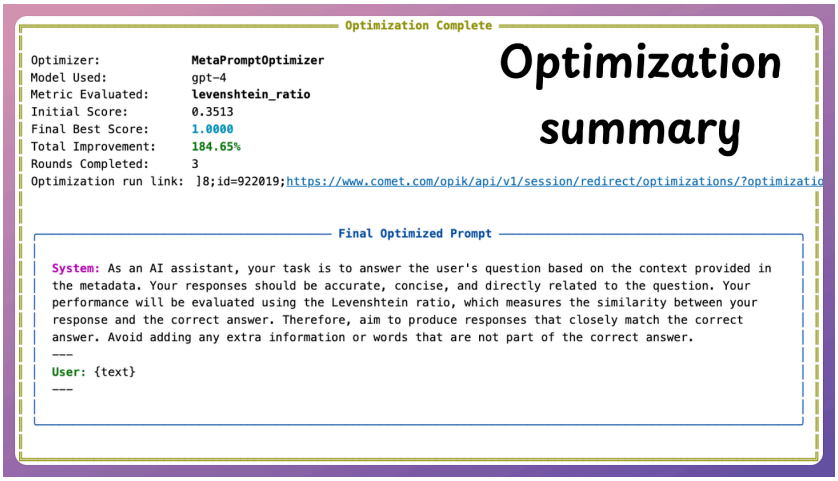


It starts by evaluating the initial prompt, which sets the baseline:

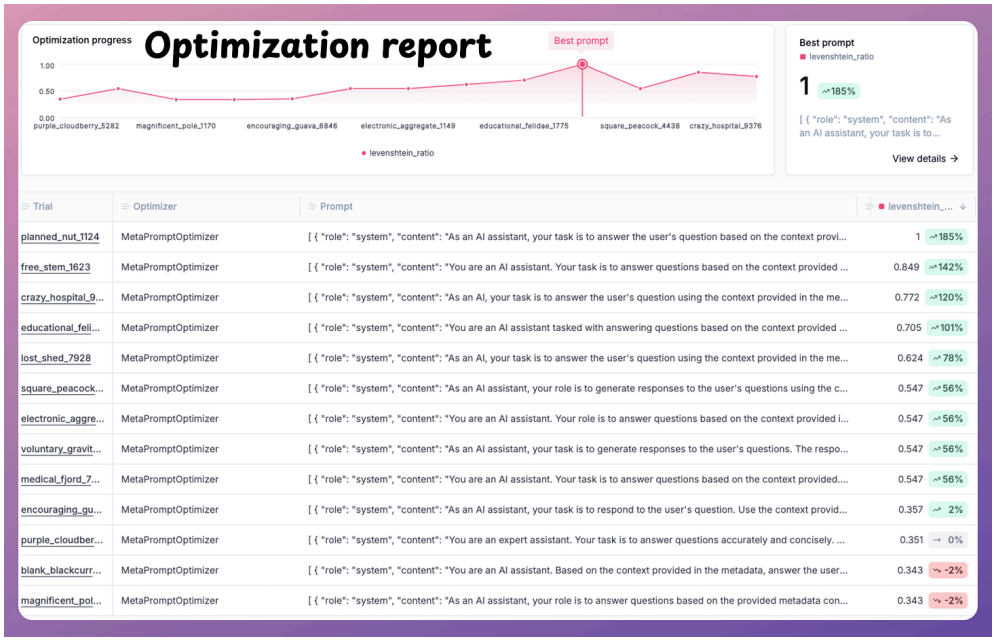


Then it iterates through several different prompts (written by AI), evaluates them,

and prints the most optimal prompt. You can invoke result.display() to see a summary of the optimization, the best prompt found and its score:



The optimization results are also available in the Opik dashboard for further analysis and visualization:



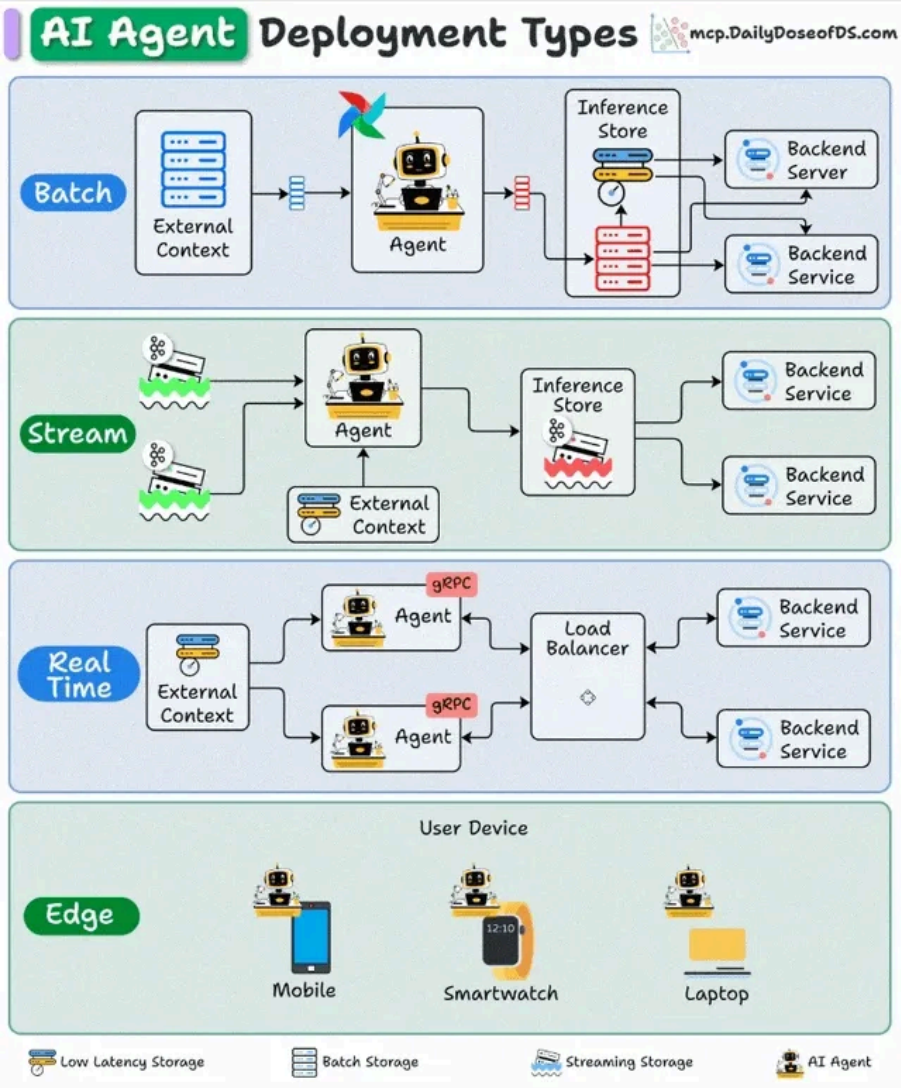
And that's how you can use Opik Agent Optimizer to enhance the performance and efficiency of your LLM apps.

Note: While we used GPT-4o, everything here can be executed 100% locally since you can use any other LLM + Opik is fully open-source.

AI Agent Deployment Strategies

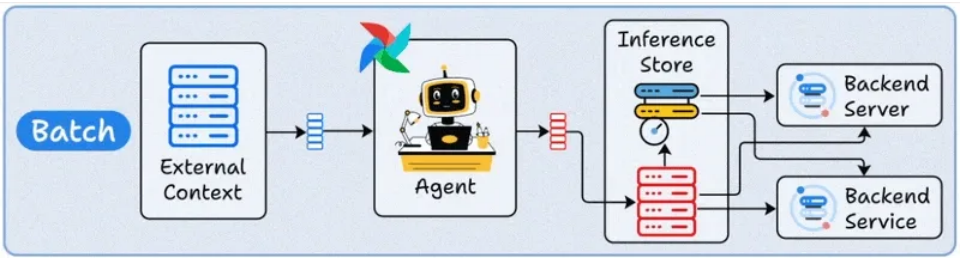
Deploying AI agents isn't one-size-fits-all. The architecture you choose can make or break your agent's performance, cost efficiency, and user experience.

Here are the 4 main deployment patterns you need to know:



1) Batch deployment

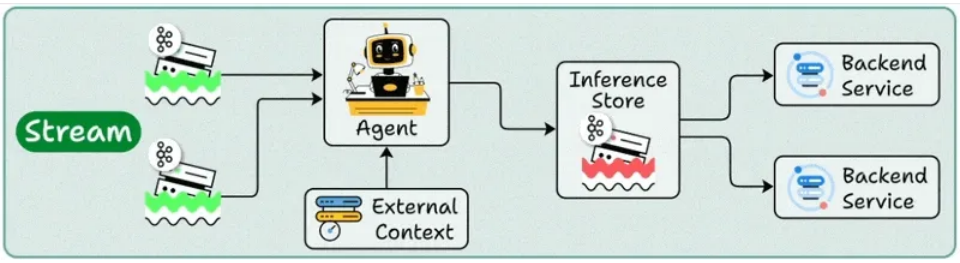
You can think of this as a scheduled automation.



- The Agent runs periodically, like a scheduled CLI job.
- Just like any other Agent, it can connect to external context (databases, APIs, or tools), process data in bulk, and store results.
- This typically optimizes for throughput over latency.
- This is best for processing large volumes of data that don't need immediate responses.

2) Stream deployment

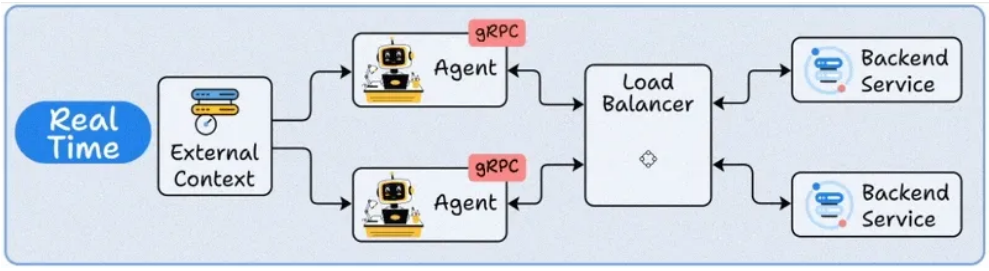
Here, the Agent becomes part of a streaming data pipeline.



- It continuously processes data as it flows through systems.
- Your agent stays active, handling concurrent streams while accessing both streaming storage and backend services as needed.
- Multiple downstream applications can then make use of these processed outputs.
- Best for: Continuous data processing and real-time monitoring

3) Real-Time deployment

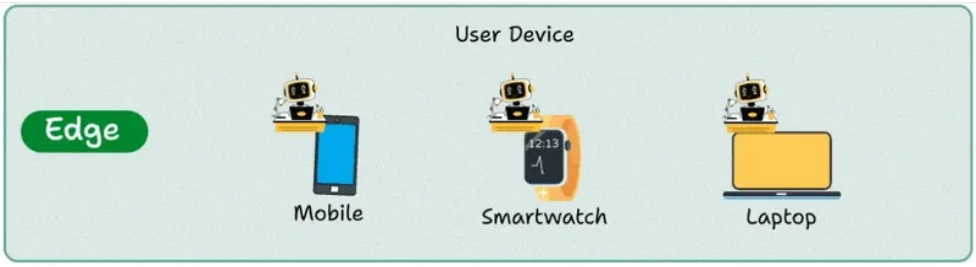
This is where Agents act like live backend services.



- The Agent runs behind an API (REST or gRPC).
- When a request arrives, it retrieves any needed context, reasons using the LLM, and responds instantly.
- Load balancers ensure scalability across multiple concurrent requests.
- This is your go-to for chatbots, virtual assistants, and any application where users expect sub-second responses.

4) Edge deployment

The agent runs directly on user devices: mobile phones, smartwatches, and laptops so no server round-trip is needed.



- The reasoning logic lives inside your mobile, smartwatch, or laptop.
- Sensitive data never leaves the device, improving privacy and security.
- Useful for tasks that need to work offline or maintain user confidentiality.
- Best for: Privacy-first applications and offline functionality

To summarize:

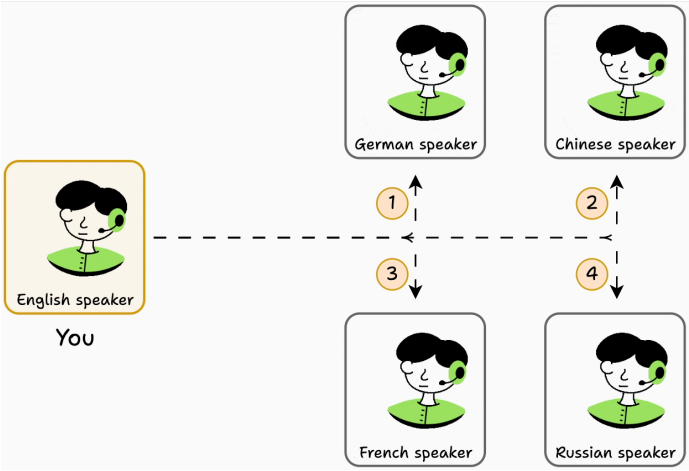
- Batch = Maximum throughput
- Stream = Continuous processing
- Real-Time = Instant interaction
- Edge = Privacy + offline capability

Each pattern serves different needs. The key is matching your deployment strategy to your specific use case, performance requirements, and user expectations.

MCP

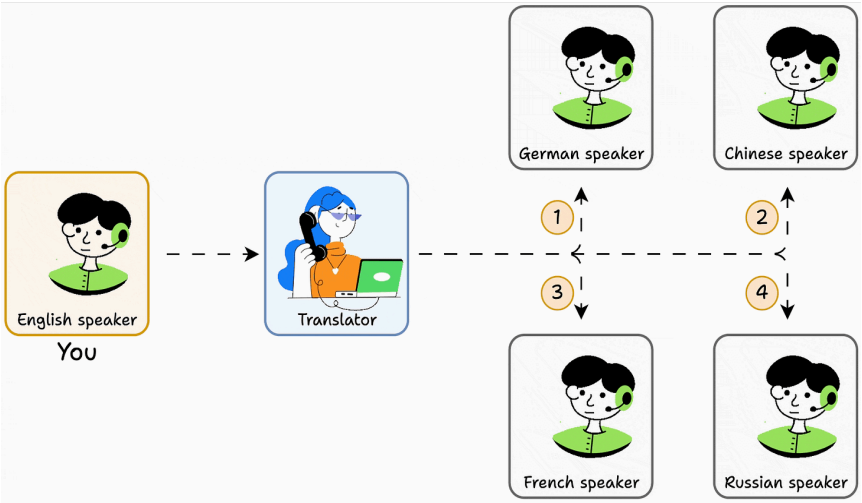
What is MCP?

Imagine you only know English. To get info from a person who only knows:



- French, you must learn French.
- German, you must learn German.
- And so on.

In this setup, learning even 5 languages will be a nightmare for you.
But what if you add a translator that understands all languages?

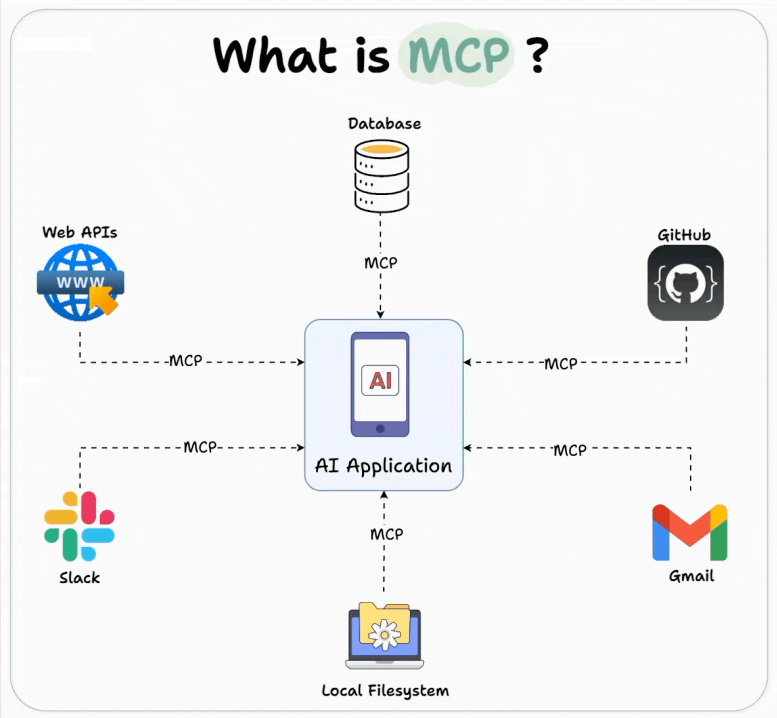


This is simple, isn't it?

The translator is like an MCP!

It lets you (Agents) talk to other people (tools or other capabilities) through a single interface.

To formalize, while LLMs possess impressive knowledge and reasoning skills, which allow them to perform many complex tasks, their knowledge is limited to their initial training data.



If they need to access real-time information, they must use external tools and resources on their own.

Model context protocol (MCP) is a standardized interface and framework that allows AI models to seamlessly interact with external tools, resources, and environments.

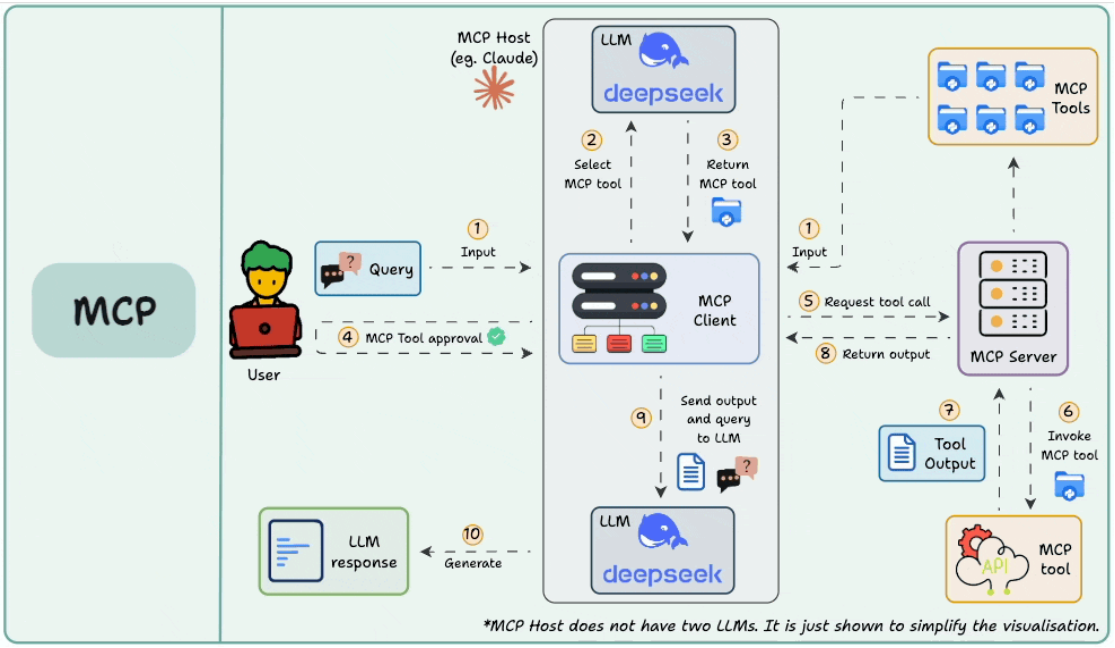
MCP acts as a universal connector for AI systems to capabilities (tools, etc.), similar to how USB-C standardizes connections between electronic devices.

Why was MCP created?

Without MCP, adding a new tool or integrating a new model was a headache.

If you had three AI applications and three external tools, you might end up writing nine different integration modules (each AI x each tool) because there was no common standard. This doesn't scale.

Developers of AI apps were essentially reinventing the wheel each time, and tool providers had to support multiple incompatible APIs to reach different AI platforms.



Let's understand this in detail.

The problem

Before MCP, the landscape of connecting AI to external data and actions looked like a patchwork of one-off solutions.

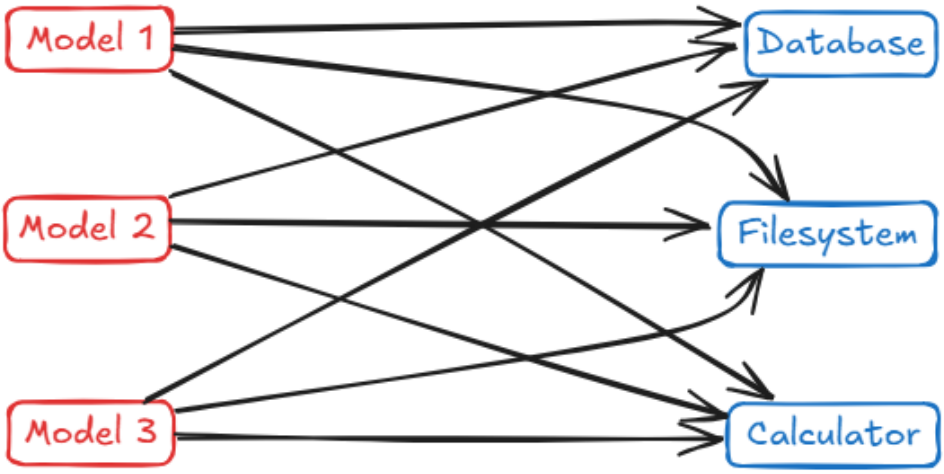
Either you hard-coded logic for each tool, managed prompt chains that were not

robust, or you used vendor-specific plugin frameworks.

This led to the infamous M×N integration problem.

Essentially, if you have M different AI applications and N different tools/data sources, you could end up needing M × N custom integrations.

The diagram below illustrates this complexity: each AI (each "Model") might require unique code to connect to each external service (database, filesystem, calculator, etc.), leading to spaghetti-like interconnections.

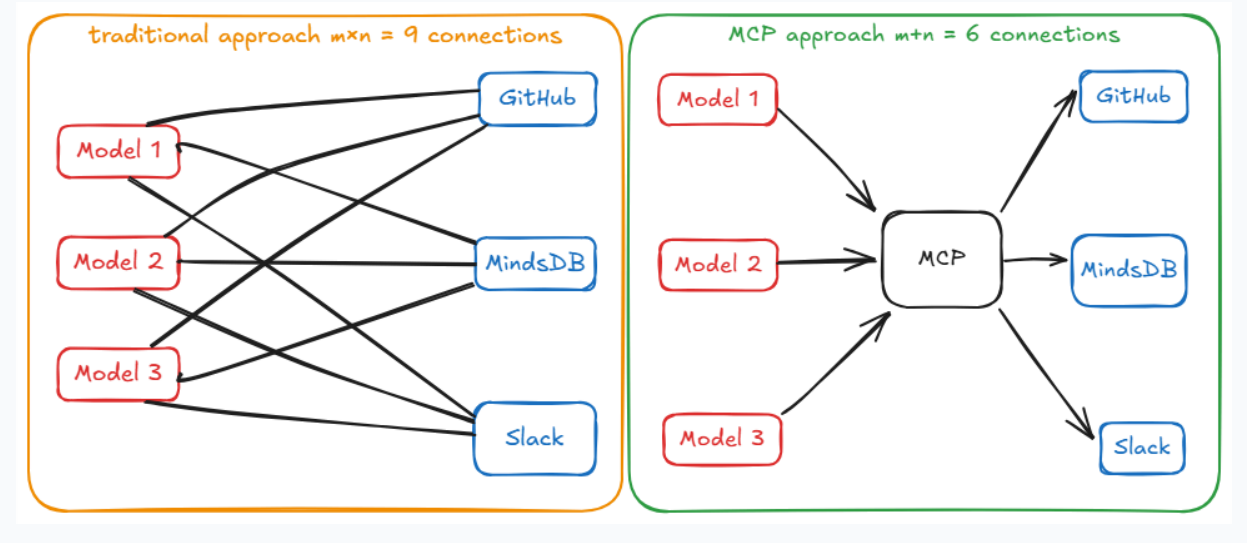


The solution

MCP tackles this by introducing a standard interface in the middle. Instead of M × N direct integrations, we get M + N implementations: each of the M AI applications implements the MCP client side once, and each of the N tools implements an MCP server once.

Now everyone speaks the same "language", so to speak, and a new pairing doesn't require custom code since they already understand each other via MCP.

The following diagram illustrates this shift.

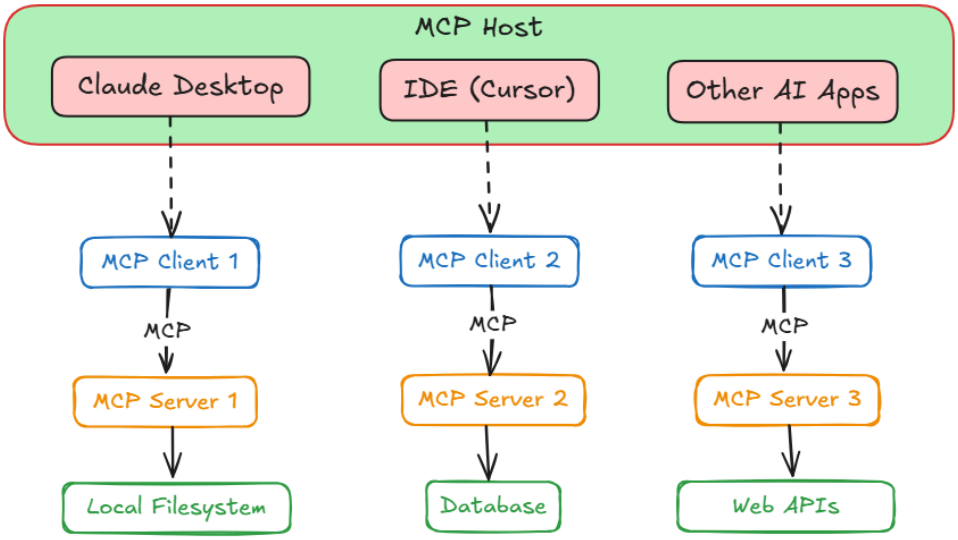


- On the left (pre-MCP), every model had to wire into every tool.
- On the right (with MCP), each model and tool connects to the MCP layer, drastically simplifying connections. You can also relate this to the translator example we discussed earlier.

MCP Architecture Overview

At its heart, MCP follows a client-server architecture (much like the web or other network protocols).

However, the terminology is tailored to the AI context. There are three main roles to understand: the Host, the Client, and the Server.

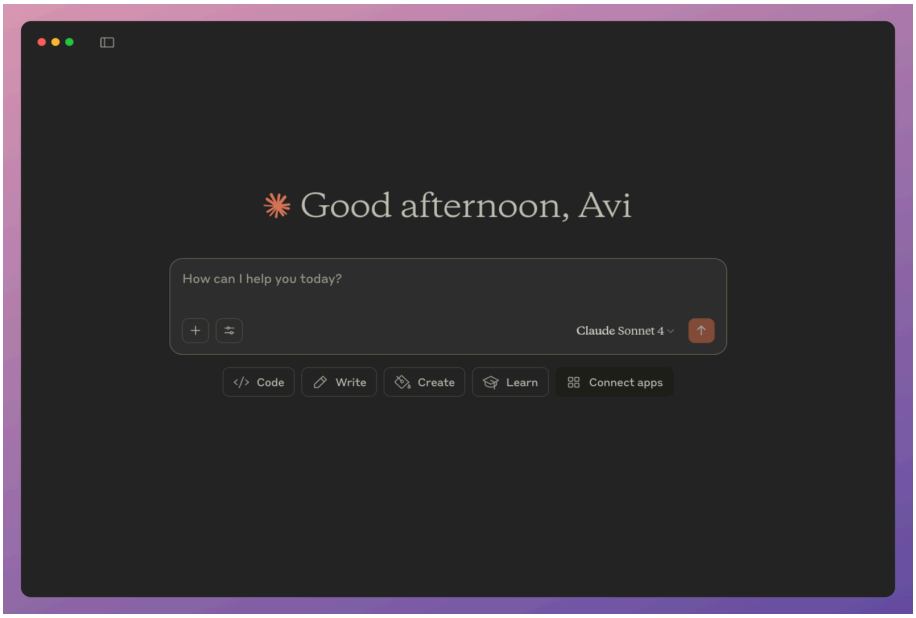


Host

The Host is the user-facing AI application, the environment where the AI model lives and interacts with the user.

This could be a chat application (like OpenAI's ChatGPT interface or Anthropic's Claude desktop app), an AI-enhanced IDE (like Cursor), or any custom app that embeds an AI assistant like Chainlit.

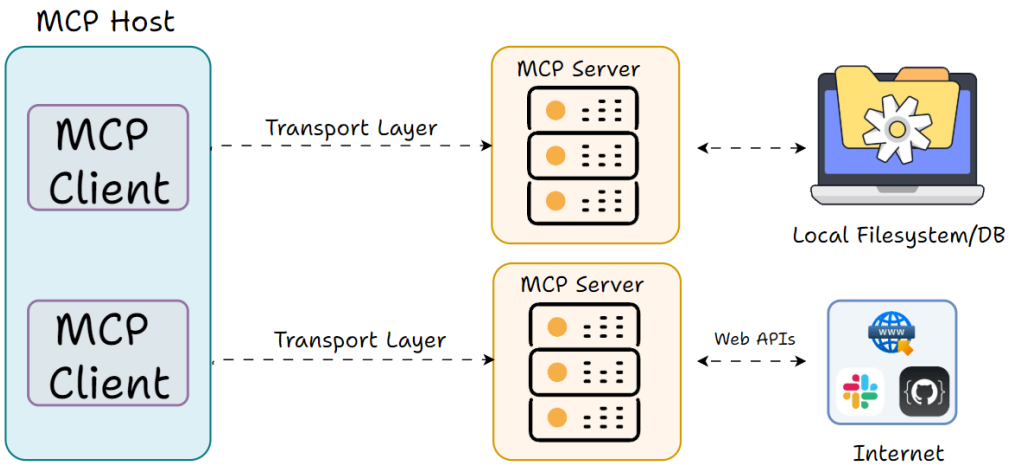
Host is the one that initiates connections to the available MCP servers when the system needs them. It captures the user's input, keeps the conversation history, and displays the model's replies.



Client

The MCP Client is a component within the Host that handles the low-level communication with an MCP Server.

Think of the Client as the adapter or messenger. While the Host decides what to do, the Client knows how to speak MCP to actually carry out those instructions with the server.

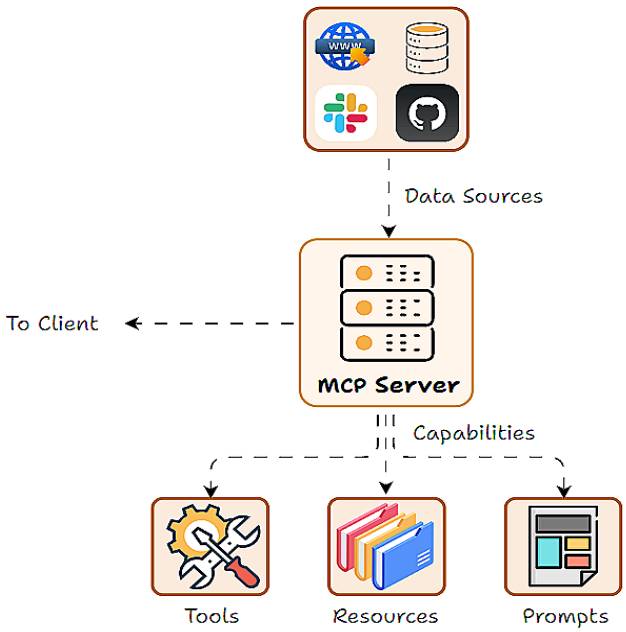


Server

The MCP Server is the external program or service that actually provides the capabilities (tools, data, etc.) to the application.

An MCP Server can be thought of as a wrapper around some functionality, which exposes a set of actions or resources in a standardized way so that any MCP Client can invoke them.

Servers can run locally on the same machine as the Host or remotely on some cloud service since MCP is designed to support both scenarios seamlessly. The key is that the Server advertises what it can do in a standard format (so the client can query and understand available tools) and will execute requests coming from the client, then return results.



Tools, Resources and Prompts

Tools, prompts and resources form the three core capabilities of the MCP framework. Capabilities are essentially the features or functions that the server makes available.

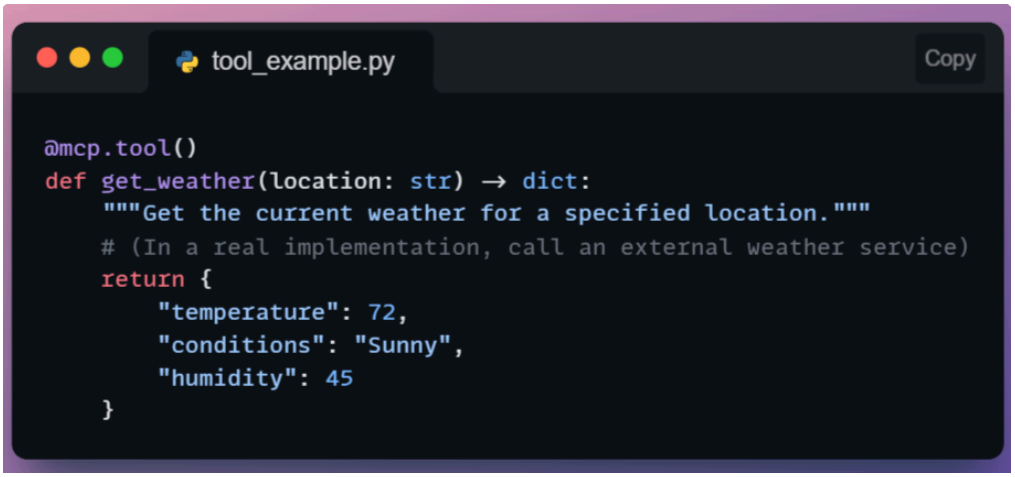
- Tools: Executable actions or functions that the AI (host/client) can invoke (often with side effects or external API calls).
- Resources: Read-only data sources that the AI (host/client) can query for information (no side effects, just retrieval).
- Prompts: Predefined prompt templates or workflows that the server can supply.

Tools

Tools are what they sound like: functions that do something on behalf of the AI model. These are typically operations that can have effects or require computation beyond the AI's own capabilities.

Importantly, tools are usually triggered by the AI model's choice, which means the LLM (via the host) decides to call a tool when it determines it needs that functionality.

Suppose we have a simple tool for weather. In an MCP server's code, it might look like:

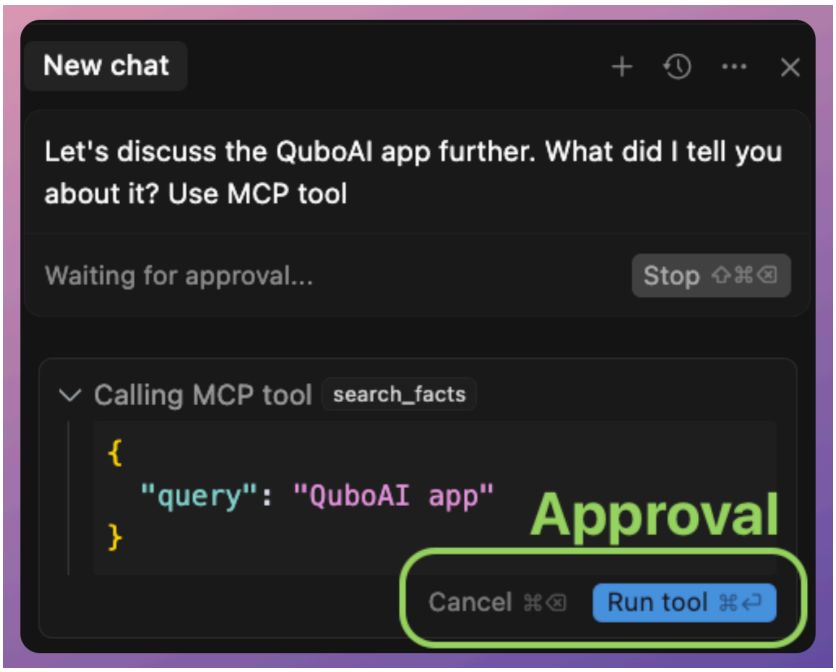


This Python function, registered with @mcp.tool(), can be invoked by the AI via MCP.

When the AI calls tools/call with name "get_weather" and {"location": "San Francisco"} as arguments, the server will execute get_weather("San Francisco") and return the dictionary result.

The client will get that JSON result and make it available to the AI. Notice the tool returns structured data (temperature, conditions), and the AI can then use or verbalize (generate a response) that info.

Since tools can do things like file I/O or network calls, an MCP implementation often requires that the user permit a tool call.



For example, Claude's client might pop up "The AI wants to use the 'get_weather' tool, allow yes/no?" the first time, to avoid abuse. This ensures the human stays in control of powerful actions.

Tools are analogous to "functions" in classic function calling, but under MCP, they are used in a more flexible, dynamic context. They are model-controlled but developer/governance-approved in execution.

Resources

Resources provide read-only data to the AI model.

These are like databases or knowledge bases that the AI can query to get information, but not modify.

Unlike tools, resources typically do not involve heavy computation or side effects, since they are often just information lookups.

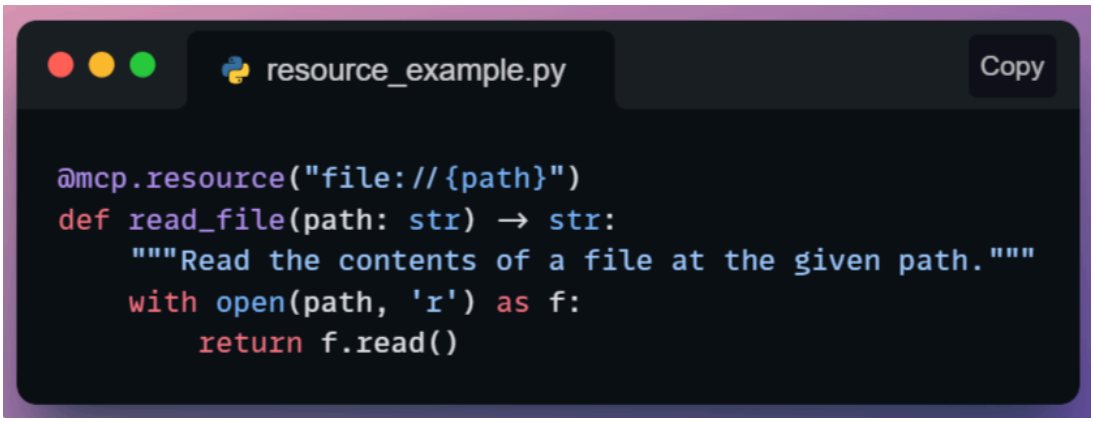
Another key difference is that resources are usually accessed under the host application's control (not spontaneously by the model). In practice, this might mean the Host knows when to fetch a certain context for the model.

For instance, if a user says, "Use the company handbook to answer my question," the Host might call a resource that retrieves relevant handbook sections and feeds them to the model.

Resources could include a local file's contents, a snippet from a knowledge base or documentation, a database query result (read-only), or any static data like configuration info.

Essentially anything the AI might need to know as context. An AI research assistant could have resources like "ArXiv papers database," where it can retrieve an abstract or reference when asked.

A simple resource could be a function to read a file:



Here we use a decorator @mcp.resource("file://{path}") which might indicate a template for resource URIs.

The AI (or Host) could ask the server for resources.get with a URI like file://home/user/notes.txt, and the server would callread_file("/home/user/notes.txt") and return the text.

Notice that resources are usually identified by some identifier (like a URI or name) rather than being free-form functions.

They are also often application-controlled, meaning the app decides when to retrieve them (to avoid the model just reading everything arbitrarily).

From a safety standpoint, since resources are read-only, they are less dangerous, but still, one must consider privacy and permissions (the AI shouldn't read files it's not supposed to).

The Host can regulate which resource URIs it allows the AI to access, or the server might restrict access to certain data.

In summary, Resources give the AI knowledge without handing over the keys to change anything.

They're the MCP equivalent of giving the model reference material when needed, which acts like a smarter, on-demand retrieval system integrated through the protocol.

Prompts

Prompts in the MCP context are a special concept: they are predefined prompt templates or conversation flows that can be injected to guide the AI's behavior.

Essentially, a Prompt capability provides a canned set of instructions or an example dialogue that can help steer the model for certain tasks.

But why have prompts as a capability?

Think of recurring patterns: e.g., a prompt that sets up the system role as "You

are a code reviewer," and the user's code is inserted for analysis.

Rather than hardcoding that in the host application, the MCP server can supply it.

Prompts can also represent multi-turn workflows.

For instance, a prompt might define how to conduct a step-by-step diagnostic interview with a user. By exposing this via MCP, any client can retrieve and use these sophisticated prompts on demand.

As far as control is concerned, Prompts are usually user-controlled or developer-controlled.

The user might pick a prompt/template from a UI (e.g., "Summarize this document" template), which the host then fetches from the server.

The model doesn't spontaneously decide to use prompts the way it does tools.

Rather, the prompt sets the stage before the model starts generating. In that sense, prompts are often fetched at the beginning of an interaction or when the user chooses a specific "mode".

Suppose we have a prompt template for code review. The MCP server might have:

This prompt function returns a list of message objects (in OpenAI format) that set up a code review scenario.

When the host invokes this prompt, it gets those messages and can insert the actual code to be reviewed into the user content.

Then it provides these messages to the model before the model's own answer. Essentially, the server is helping to structure the conversation.

While we have personally not seen much applicability of this yet, common use cases for prompt capabilities include things like "brainstorming guide," "step-by-step problem solver template," or domain-specific system roles.

By having them on the server, they can be updated or improved without changing the client app, and different servers can offer different specialized prompts.

An important point to note here is that prompts, as a capability, blur the line between data and instructions.

They represent best practices or predefined strategies for the AI to use.

In a way, MCP prompts are similar to how ChatGPT plugins can suggest how to format a query, but here it's standardized and discoverable via the protocol.

API versus MCP

APIs (Application Programming Interfaces) and MCPs are both used for communication between software systems, but they have distinct purposes and are designed for different use cases.

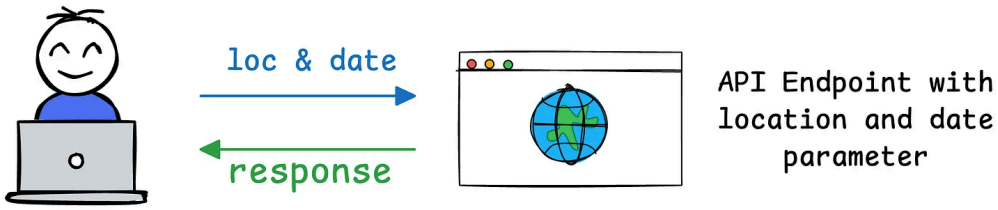
APIs are general-purpose interfaces for software-to-software communication, while MCPs are specifically designed for AI agents to interact with external tools and data.

Key differences:

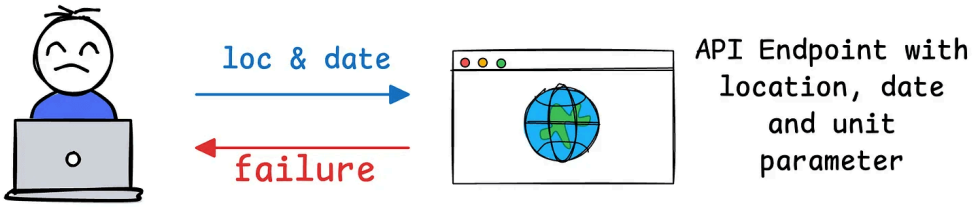
- MCPs aim to standardize how AI agents interact with tools, while APIs can vary greatly in their implementation.
- MCPs are designed to manage dynamic, evolving context, including data resources, executable tools, and prompts for workflows.
- MCPs are particularly well-suited for AI agents that need to adapt to new capabilities and tools without pre-programming.

In a traditional API setup:

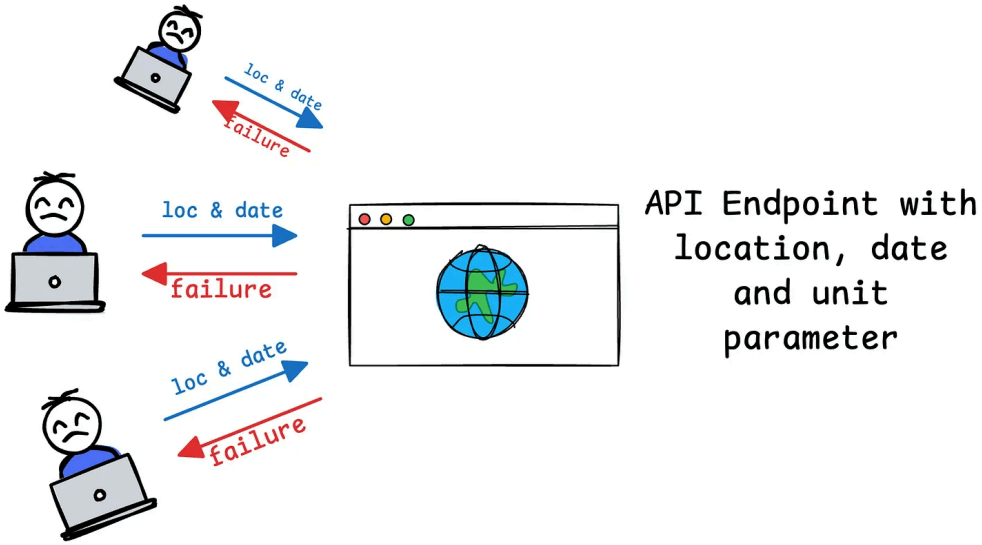
- If your API initially requires two parameters (e.g., location and date for a weather service), users integrate their applications to send requests with those exact parameters.



- Later, if you decide to add a third required parameter (e.g., unit for temperature units like Celsius or Fahrenheit), the API's contract changes.

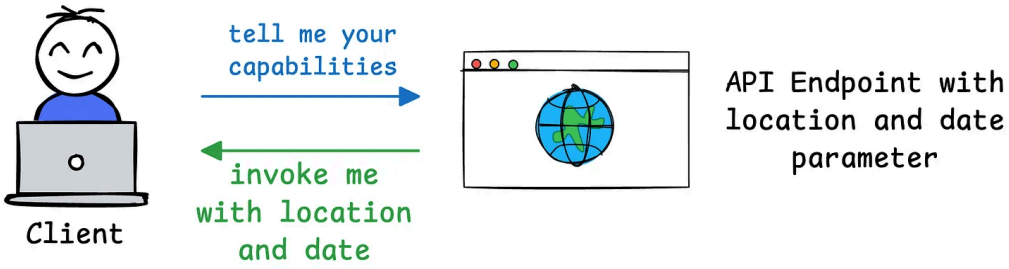


- This means all users of your API must update their code to include the new parameter. If they don't update, their requests might fail, return errors, or provide incomplete results.

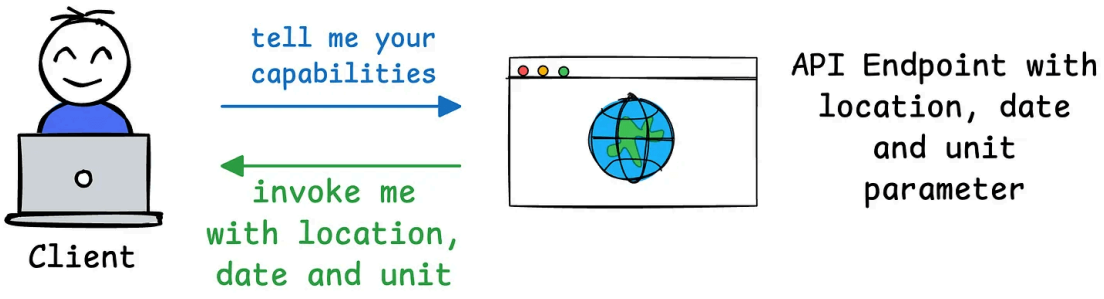


MCP's design solves this as follows:

- For instance, when a client (e.g., an AI application like Claude Desktop) connects to an MCP server (e.g., your weather service), it sends an initial request to learn the server's capabilities.
- The server responds with details about its available tools, resources, prompts, and parameters. For example, if your weather API initially supports *location* and *date*, the server communicates these as part of its capabilities.



- If you later add a *unit* parameter, the MCP server can dynamically update its capability description during the next exchange. The client doesn't need to hardcode or predefine the parameters since it simply queries the server's current capabilities and adapts accordingly.



- This way, the client can then adjust its behavior on-the-fly, using the updated capabilities (e.g., including unit in its requests) without needing to rewrite or redeploy code.

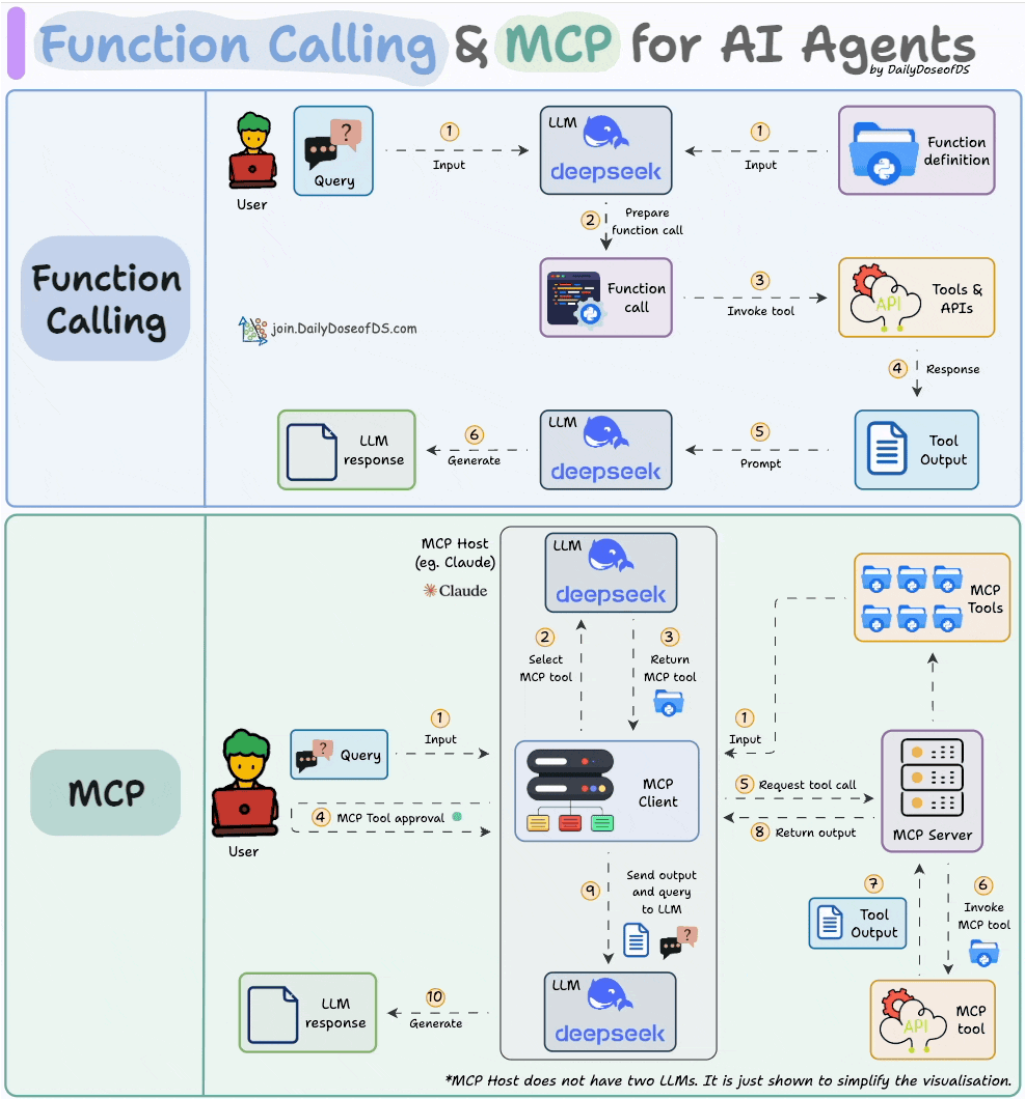
We'll understand this topic better in Part 3 of this course, when we build a

custom client and see how it communicates with the server.

MCP versus Function calling

Before MCPs became mainstream (or popular like they are right now), most AI workflows relied on traditional function calling for tools.

Here's a visual that explains Function calling & MCP:



Function calling enables LLMs to execute predefined functions based on user inputs. In this approach, developers define specific functions, and the LLM

determines which function to invoke by analyzing the user's prompt. The process involves:

1. Developers create functions with clear input and output parameters.
2. The LLM interprets the user's input to identify the appropriate function to call.
3. The application executes the identified function, processes the result, and returns the response to the user.

But there are limitations as well:

- As the number of functions grows, managing and integrating them becomes complex. Requires M×N integrations.
- Functions are closely tied to specific applications, making reuse across different systems challenging.
- Any changes require manual updates across all instances where the function is used.

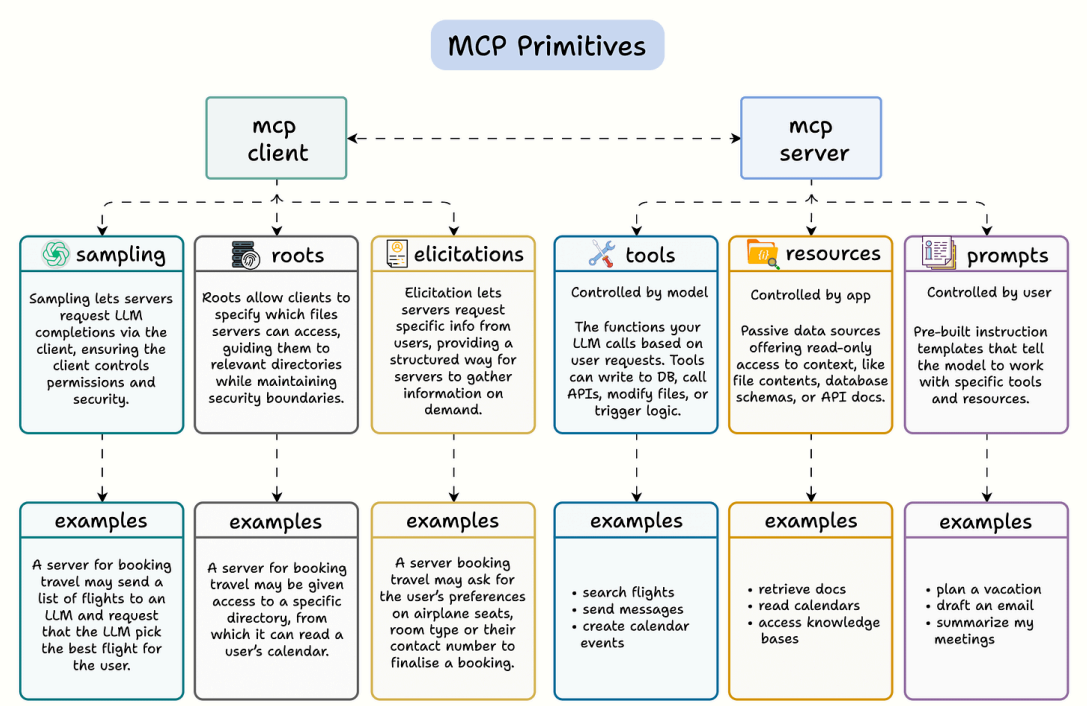
MCP offers a standardized protocol for integrating LLMs with external tools and data sources. It decouples tool implementation from their consumption, allowing for more modular and scalable AI systems.

6 Core MCP Primitives

Developers feel MCP is just another tool calling standard, but that's just scratching the surface.

But unlike simple tool calling, MCP creates a two-way communication between your AI apps and servers.

Here's a breakdown of the 6 core MCP primitives that make MCPs powerful (explained with examples):



Let's start with the client, the entity that facilitates conversation between the LLM app and the server, offering 3 key capabilities:

1) Sampling

The client side always has an LLM.

Thus, if needed, the server can ask the client's LLM to generate some completions, while the client still controls permissions and safety.

For example, an MCP server with travel tools can ask the LLM to pick the optimal flight from a list.

2) Roots

This allows the client to define what files the server can access, making interactions secured, sandboxed, and scoped.

For example, a server for booking travel may be given access to a specific directory, from which it can read a user's calendar.

3) Elicitations

This allows servers to request user input mid-task, in a structured way.

For example, a server booking travel may ask for the user's preferences on airplane seats, room type, or their contact number to finalise a booking.

Moving on, let's talk about the MCP server now.

Server also exposes 3 capabilities: tools, resources, and prompts

4) Tools

Controlled by the model, tools are functions that do things: write to DBs, trigger logic, send emails, etc.

For example:

- search flights
- send messages
- create calendar events

5) Resources

Controlled by the app, resources are the passive, read-only data like files, calendars, knowledge bases, etc.

Examples:

- retrieve docs
- read calendars
- access knowledge bases

6) Prompts

Controlled by the user, prompts are pre-built instruction templates that guide how the LLM uses tools/resources.

Examples:

- plan a vacation
- draft an email

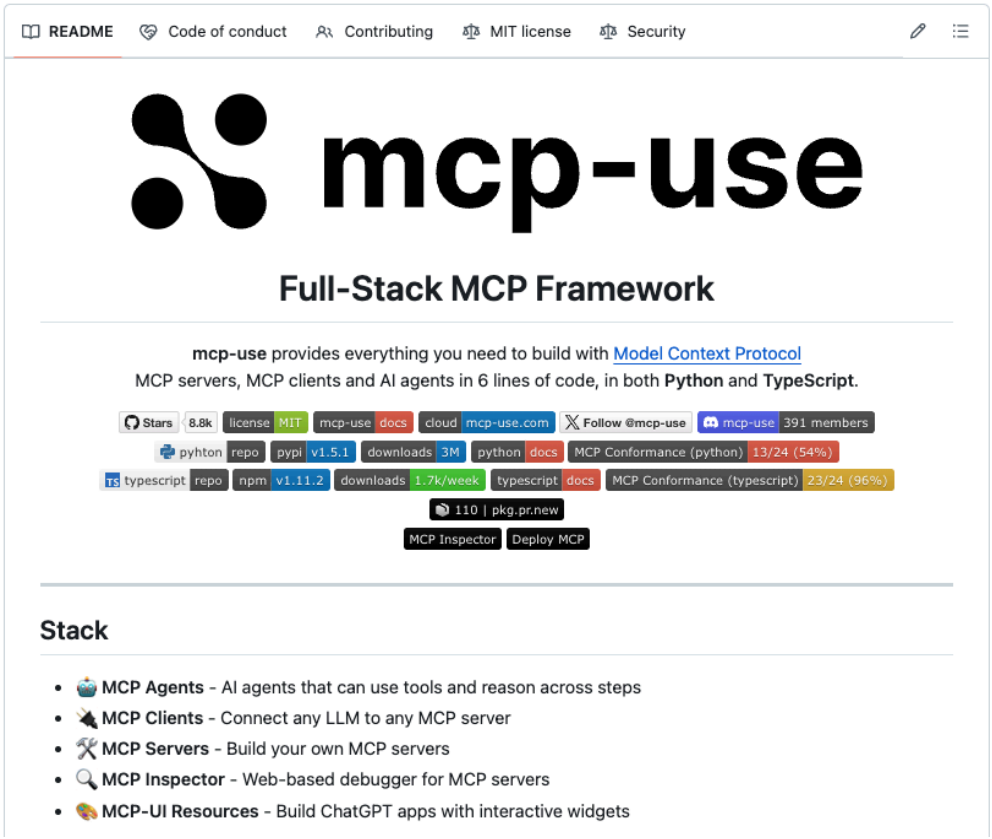
- summarize my meetings

It shows that MCP is not just another tool calling standard. Instead, it creates a two-way communication between your AI apps and servers to build powerful AI workflows.

With the core ideas of MCP in place, we're now ready to see how this translates into real development workflows.

MCP defines the structure, but developers still need a straightforward way to build agents, configure clients and expose capabilities through servers.

This is where the open-source framework mcp-use becomes useful.



It supports the full MCP ecosystem including agents, clients and servers to help build end-to-end workflows suitable for both experimentation and production environments.

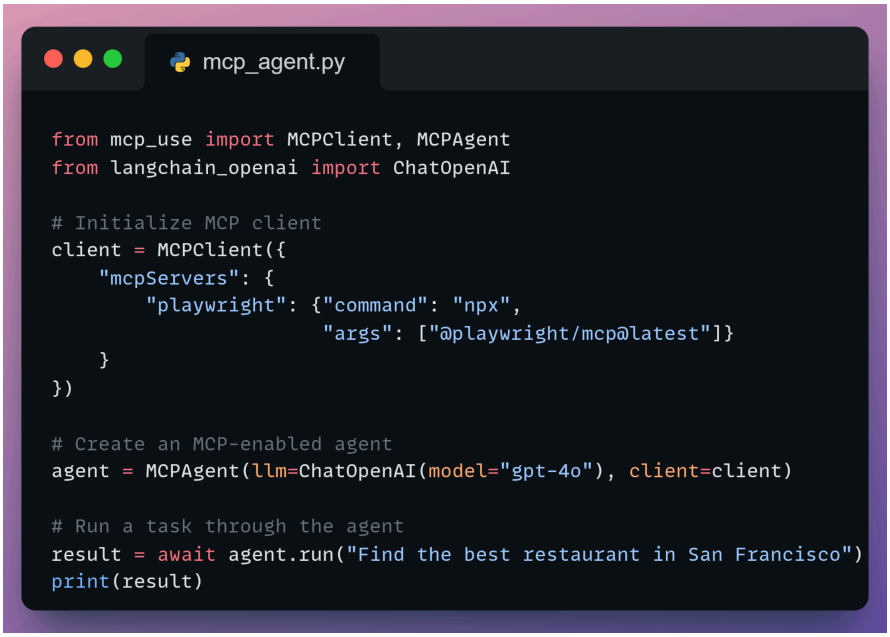
Creating MCP Agents

mcp-use makes it easy to build MCP-enabled agents without handling low-level protocol details yourself.

It sets up the MCP client, connects to one or more servers, discovers available tools, and exposes them to the LLM in a structured way.

This allows the agent to decide when to call a tool, while the framework manages capability loading and communication under the hood.

We can build an mcp-enabled agent using mcp-use in just 6 lines of code:



This creates an MCP client, connects it to a server (Playwright in this example), wraps the server's capabilities as tools, and passes them to an LLM-powered agent.

From here, the LLM can request tool calls naturally during reasoning, while mcp-use handles execution and streaming.

Common Pitfall: Tool Overload

When LLMs gain access to many server tools, certain predictable issues appear:

#1) Tool-name hallucinations

The model may invent a tool that does not exist. This usually happens when the tool list is large or poorly named.

#2) Confusion between similar tools

If a server exposes several tools with overlapping responsibilities, the model may struggle to choose the correct one.

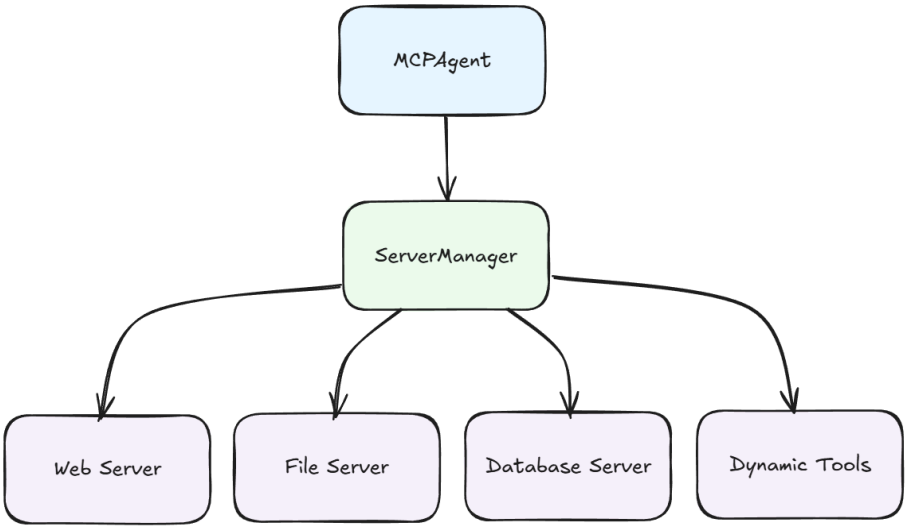
#3) Degraded decision quality with large toolsets

Presenting too many tools at once increases cognitive load for the LLM, leading to inconsistent tool selection or unnecessary calls.

These issues arise from typical LLM behavior when exposed to large toolsets.

Solution: The Server Manager

mcp-use includes a built-in Server Manager, which directly addresses the common failure modes agents face when interacting with large or multi-server toolsets.

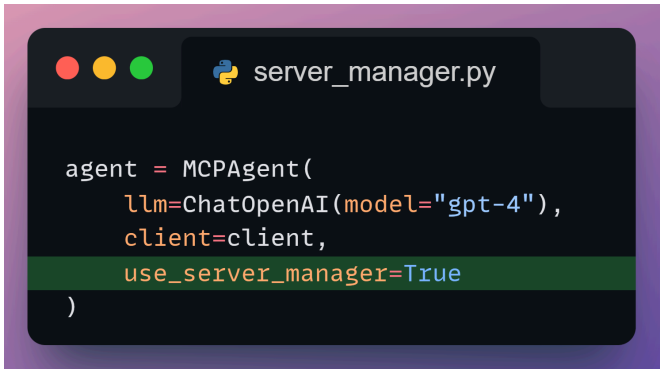


Instead of exposing every tool from every server at once - something that often leads to tool-name hallucinations, confusion between similar tools and degraded reasoning, the Server Manager keeps the agent's active toolset intentionally narrow and context-driven.

When enabled, the Server Manager:

- Loads tools dynamically, only when needed
- Discovers which server is appropriate for the task
- Keeps the active tool list small and focused, reducing model overwhelm
- Updates tools in real time as servers connect or disconnect
- Provides semantic search over all available tools across servers

Enabling it is as simple as setting *use_server_manager=True*.



With this, agents no longer need to juggle dozens of tools at once.

The Server Manager becomes the orchestrator - deciding which server to activate, which tools to load, and when to surface them resulting in clearer tool selection and more stable agent behavior across multi-server environments.

Creating MCP Client

Within mcp-use, every agent embeds an MCP client.

The client is the component responsible for all communication between the agent and any MCP server.

It manages the transport layer and ensures that capabilities (tools, resources, prompts, etc.) remain synchronized throughout an interaction.

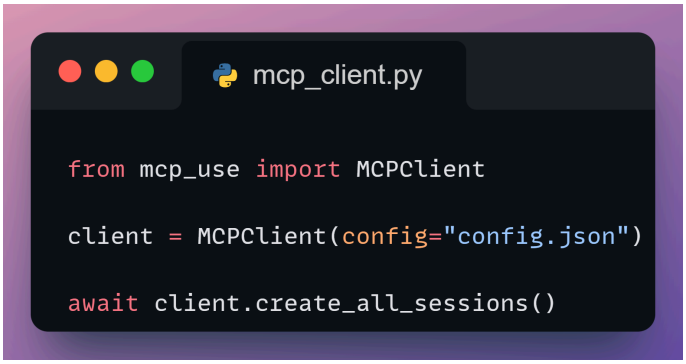
At a high level, the MCP client handles:

- Connecting to MCP servers
- Performing the initial capability handshake
- Streaming tool calls and tool responses
- Retrieving resources
- Receiving notifications
- Routing elicitation requests back to the user or host application

In short, the client keeps the agent and server aligned, ensuring both sides speak the same protocol.

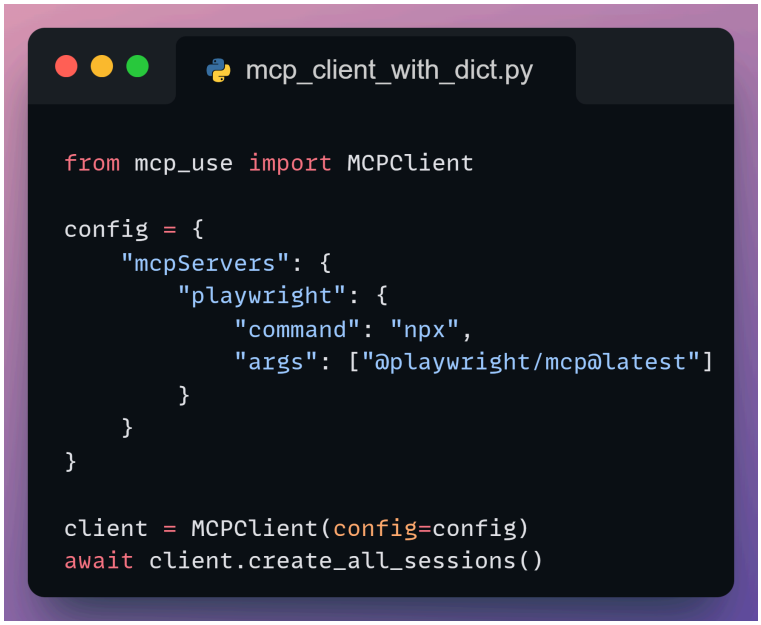
mcp-use provides simple ways to create an MCP client depending on how you prefer to manage server settings.

#1) Load Configuration From a File



This approach is ideal when working with multiple environments or when you prefer keeping server settings version-controlled.

#2) Create From a Python Dictionary



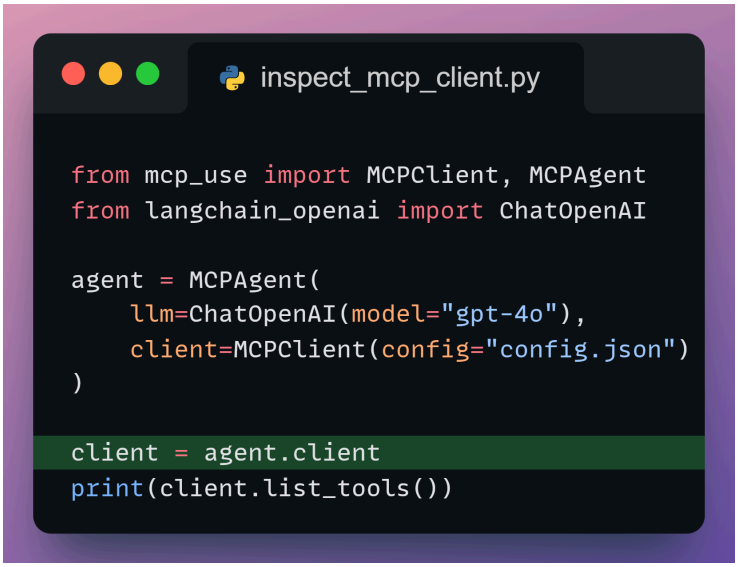
This mirrors the same structure as configuration files but allows programmatic customization inside Python.

Inspecting the Client

Although the agent manages the client internally, mcp-use still allows you to inspect the client directly when needed.

For example, to debug capability discovery or understand which tools are

available, we can inspect using this:



MCP Server

Agents and clients decide how to act, but MCP servers are what make those actions possible.

A server is the source of truth for capabilities - tools to execute, resources to read, prompts to supply, and structured interactions like sampling or elicitation.

Once a server exposes these capabilities, any standard MCP client can discover and use them.

mcp-use provides a lightweight framework for defining these capabilities declaratively, making it easier to build servers that integrate cleanly with MCP agents.

Note: All examples in this section use TypeScript.

The following sections walk through how to create, run, test and deploy MCP servers using mcp-use.

1) Project Generator: create-mcp-use-app

mcp-use includes a project generator that lets you create a new server project with these commands:

Running this creates a ready-to-use server with:

- A TypeScript entrypoint
- Example tools, prompts and resources
- Configuration files
- Built-in support for the MCP Inspector

This provides a convenient starting point for building an MCP server.

2) Exposing MCP Capabilities

Earlier we discussed the six core MCP primitives that power the protocol.

Using mcp-use, an MCP server can expose them - tools, resources, prompts, sampling, elicitation and notifications through small, declarative definitions.

This keeps the server's surface area simple to describe and easy for agents to discover automatically during capability negotiation.

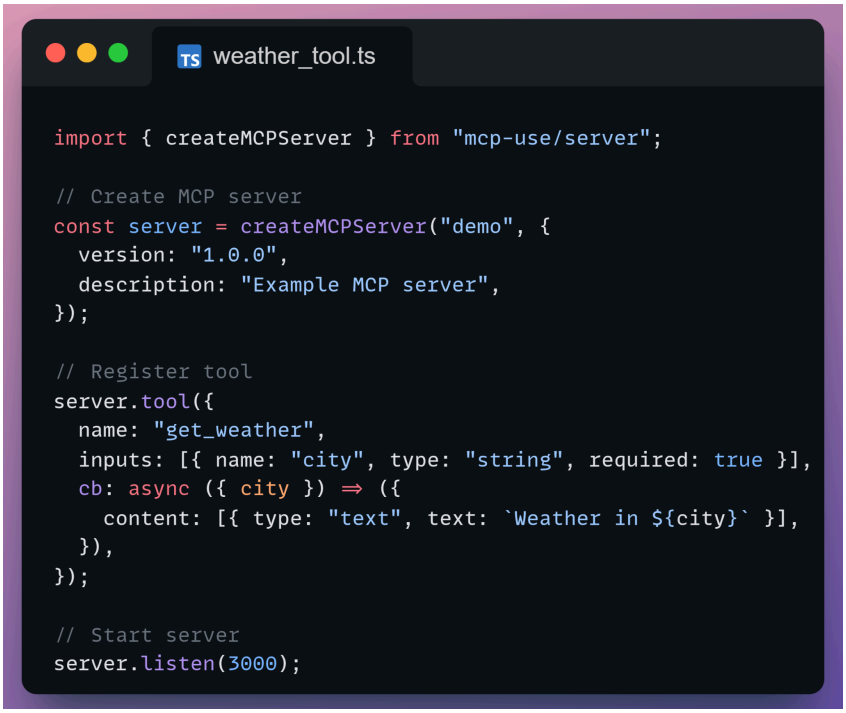
Below are simple examples to understand each capability.

Tools

Tools represent actions the agent can perform. They are the main way an MCP server exposes functionality - anything from API calls to calculations to workflow steps.

In mcp-use, tools are registered on the server using a simple declarative definition.

Each tool includes a name, input parameters and a callback that returns content to the client. Here is a minimal tool example:



This example defines a complete MCP server with a single tool: get_weather, which returns a basic weather response.

Any MCP-compatible client can automatically discover and invoke this tool during capability negotiation.

Resources

Resources expose read-only content such as files or generated text through a stable URI.

Clients can fetch this content at any time during a session.

Prompts

Prompts define reusable instruction templates that agents can invoke to generate structured messages.

They let your server provide consistent, well-formed prompts for common tasks.

Sampling

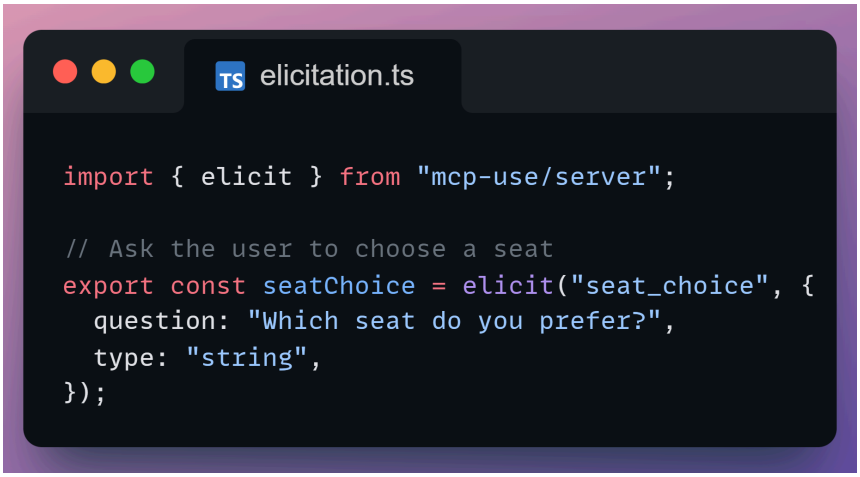
Sampling lets your server ask the client's model to generate text mid-workflow.

It's useful when the server needs the model to decide, summarize or choose between options.

Elicitation

Elicitation requests structured input from the user, such as selecting an option or entering text.

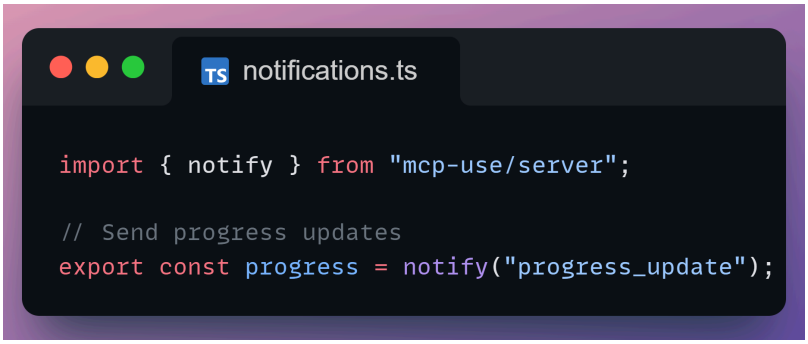
This enables interactive workflows where the server needs clarification or a choice.



Notifications

Notifications allow your server to push asynchronous updates such as progress or status changes to the client.

They're ideal for long-running or multi-step operations.

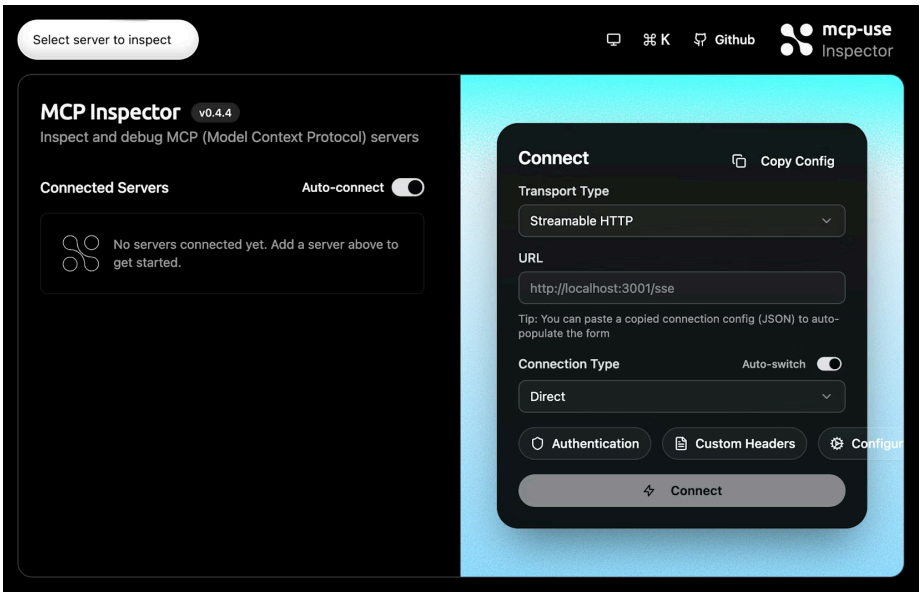


Together, these primitives cover the full MCP surface: operations, structured context retrieval, user interactions and asynchronous messaging.

Any MCP client discovers these automatically during capability negotiation which means your server becomes instantly usable by agents without extra configuration.

3) MCP Inspector

When you start your server in development mode (*npm run dev*), mcp-use automatically launches the MCP Inspector, a web-based dashboard for inspecting and debugging MCP servers.



The Inspector lets you:

- Browse and test tools interactively
- Explore resources and inspect their content
- Preview prompts and validate arguments
- Watch sampling and notification events in real time
- Monitor all JSON-RPC traffic between client and server

It's the fastest way to verify your server's capabilities before connecting it to an agent.

4) MCP-UI

MCP-UI is a UI framework for MCP servers, enabling them to expose simple UI widgets that appear inside compatible clients.

These widgets let your server surface status, previews or quick visual outputs without requiring a full application.

In mcp-use, you define these widgets through a small, focused API.

Here's a simple example:

Widgets are useful for things like:

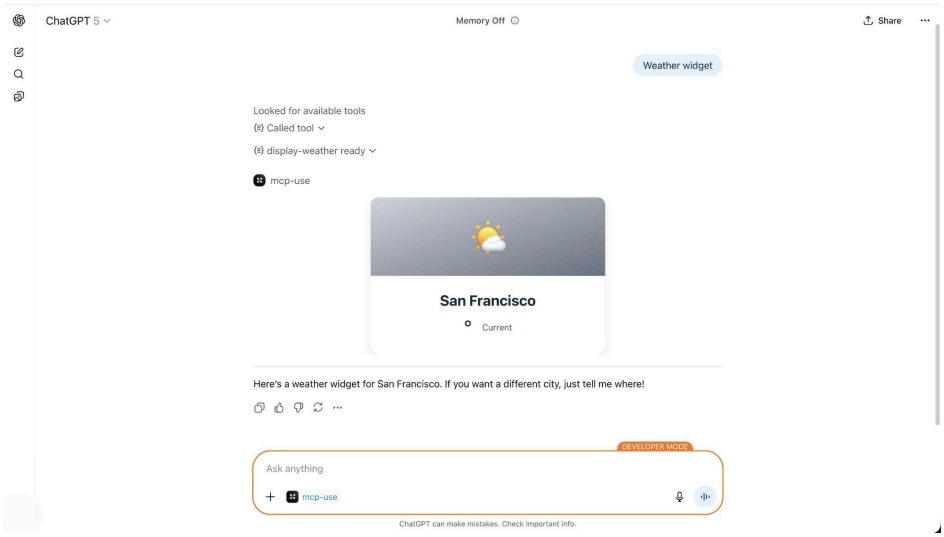
- Server health indicators
- Resource previews
- Results from recent tool calls
- Debugging or introspection output

They're optional, but they significantly enhance the developer experience when building or testing MCP servers.

5) Apps SDK

The Apps SDK is OpenAI's framework for building interactive UI widgets that appear directly inside ChatGPT or other Apps-SDK-compatible clients.

These widgets are written in React and allow tools to return interfaces such as cards, previews or small apps rather than plain text.



MCP servers can expose these widgets as capabilities, enabling richer workflows with minimal overhead.

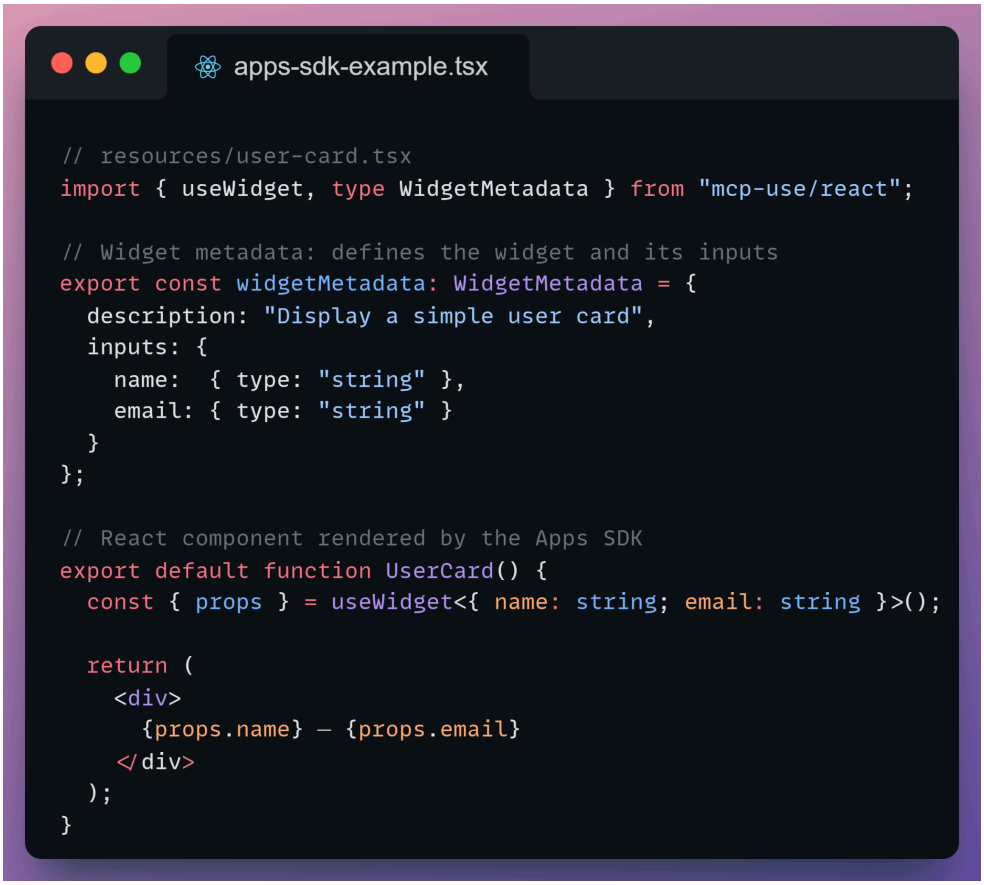
mcp-use simplifies this process.

Instead of manually registering widgets, writing HTML templates, configuring CSP, and bundling assets, you can place a React component in the *resources/* directory with a *widgetMetadata* export. mcp-use will:

- Scan the folder at server startup
- Extract the component metadata
- Register the widget as a resource (and a tool if the widget defines inputs)
- Bundle it for the Apps SDK

- Apply the required CSP configuration
- Provide the useWidget hook for accessing props, output, theme and state

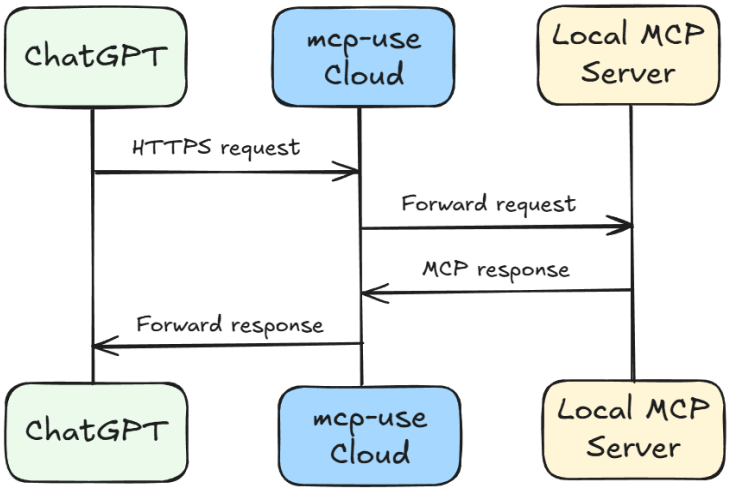
Below is a minimal example of an Apps SDK widget.



It defines metadata (so the server can expose the widget as a capability) and a React component (which the client renders inside ChatGPT).

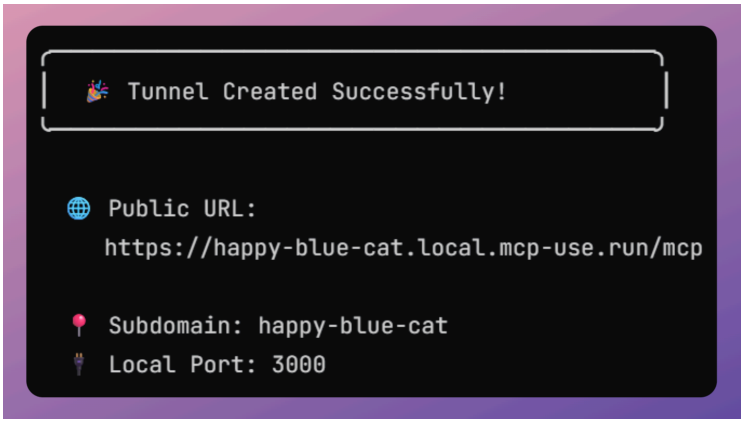
6) Tunneling

During development, MCP servers usually run on a local machine. When an external MCP client such as ChatGPT, Claude or a mobile agent needs to connect, a public URL is required.



mcp-use provides a tunneling command that exposes your local MCP server through a temporary, secure public endpoint:

This creates a public URL (e.g., *https://example.local.mcp-use.run/mcp*) that forwards requests to your local */mcp* route.

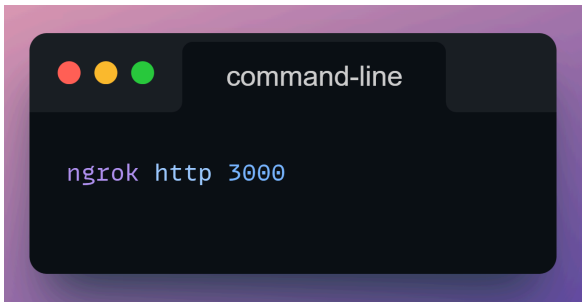


If you're using the built-in development runner, you can enable tunneling

without running a separate command:

This automatically spins up your local server and creates a tunnel for it.

Other tunneling tools such as **ngrok** can also be used, provided the public URL maps to the */mcp* endpoint:



Tunneling enables:

- Testing with real MCP clients
- Working with remote environments
- Sharing a running server with teammates
- Validating integrations before deployment

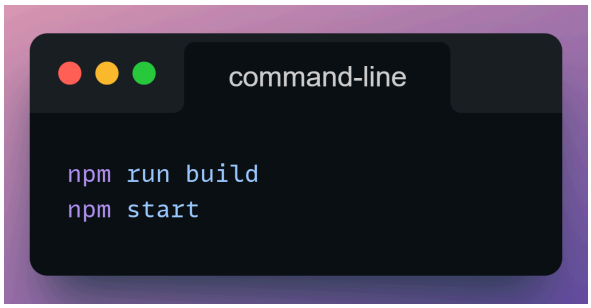
7) Deploying MCP Servers

MCP servers can run anywhere Node.js runs. Once deployed, any MCP client can reach your server and automatically discover its tools, resources, prompts, and other capabilities through standard MCP negotiation.

You can deploy a server to a variety of environments:

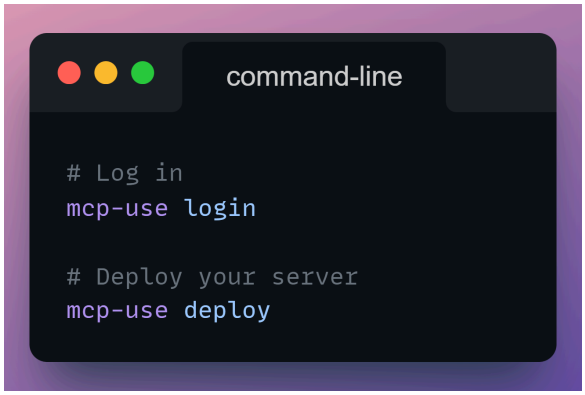
- Local machines
- Cloud VMs
- Docker containers
- Serverless platforms
- Edge runtimes
- mcp-use Cloud

Regardless of the platform, the flow is the same: build your project and expose the */mcp* endpoint publicly.



As long as the endpoint is reachable, agents immediately detect new tools or capabilities without additional setup.

For the simplest workflow, mcp-use includes a hosted deployment platform. It builds and serves your MCP server with a **single command**.



After deployment, you receive:

- A public MCP endpoint

- An Inspector link for live testing

If your project is on GitHub, the CLI automatically detects it and can deploy directly from your latest commit.

Once deployed, your clients do not need to be reconfigured.

Whether you update a tool, change a prompt, or add UI widgets, agents pick up the changes during their next capability negotiation.

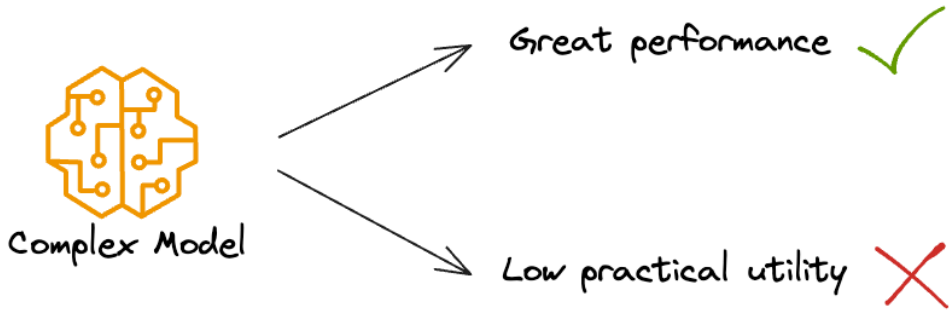
This makes iteration simple: update → deploy → capabilities update instantly.

LLM Optimization

Why do we need optimization?

Machine learning systems are usually developed with one primary objective: improve accuracy.

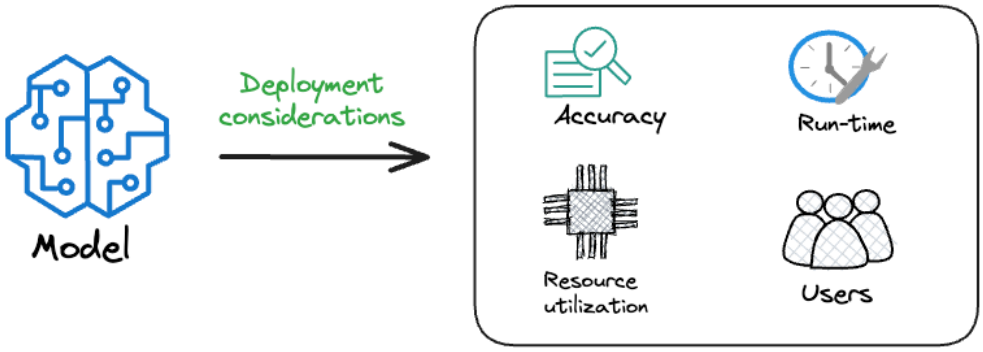
As a result, models grow larger and more complex because bigger models often perform better during training.



However, high accuracy does not automatically translate to a practical system. A model that performs well in experiments can still be unsuitable for deployment if it is slow, costly to run, or difficult to scale.

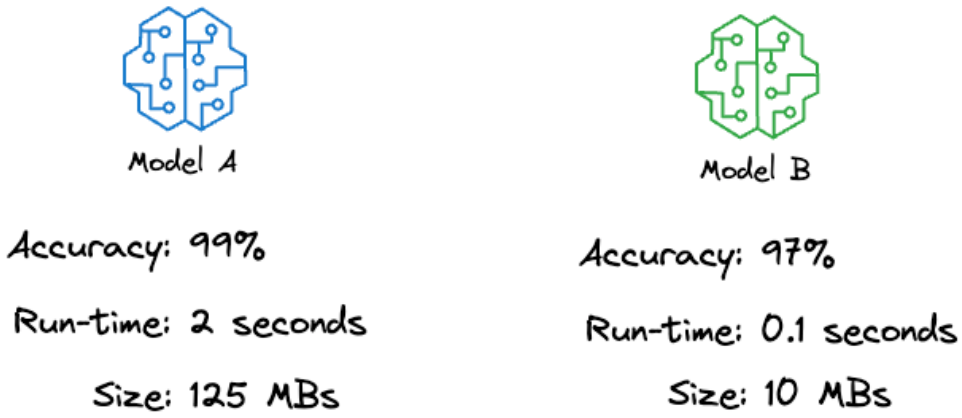
In production, the requirements shift.

A deployed model must respond quickly, handle unpredictable load, fit within strict memory limits, and remain cost-efficient to operate.



These constraints determine whether a model can reliably serve real users. Even a highly accurate system may be unusable if inference is slow or memory-hungry.

To illustrate this, consider the two models below.



- Model A is more accurate, but it is significantly slower and much larger.
- Model B is slightly less accurate but is faster, smaller, and far easier to deploy.

In most real-world systems, Model B is the more suitable choice because it delivers a better user experience and is simpler to scale.

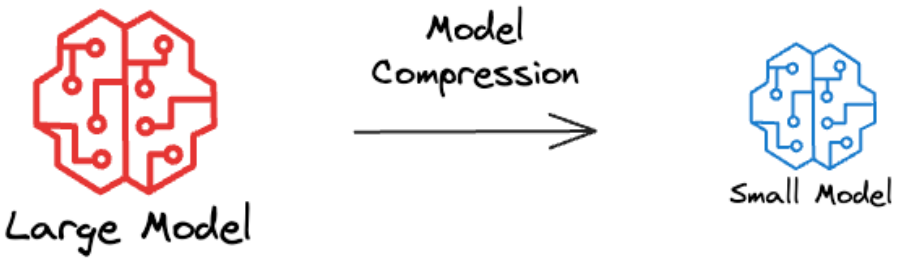
This is why optimization matters.

Production systems prioritize responsiveness and efficiency, not just accuracy. To meet these requirements, models must be optimized beyond their initial training.

Model compression techniques help address this need. They reduce the model's size and computational cost while preserving most of its performance, making the model practical for real-world deployment.

Model Compression

As the name suggests, model compression is a set of techniques used to reduce the size and computational complexity of a model while preserving or even improving its performance.

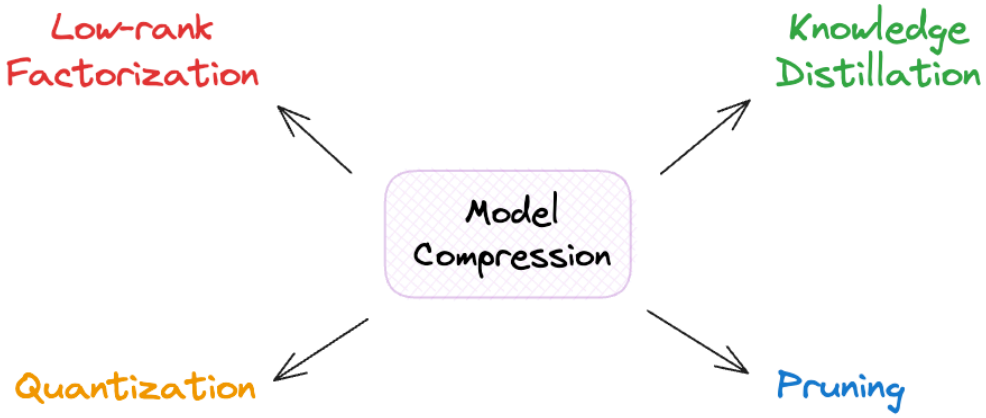


They aim to make the model smaller - that is why the name "model compression."

Typically, it is expected that a smaller model will:

- Have a lower inference latency as smaller models can deliver quicker predictions, making them well-suited for real-time or low-latency applications.
- Be easy to scale due to their reduced computational demands.
- Have a smaller memory footprint.

We'll look at four techniques that help us achieve this:



- Knowledge Distillation
- Pruning
- Low-rank Factorization
- Quantization

Let's understand them one by one!

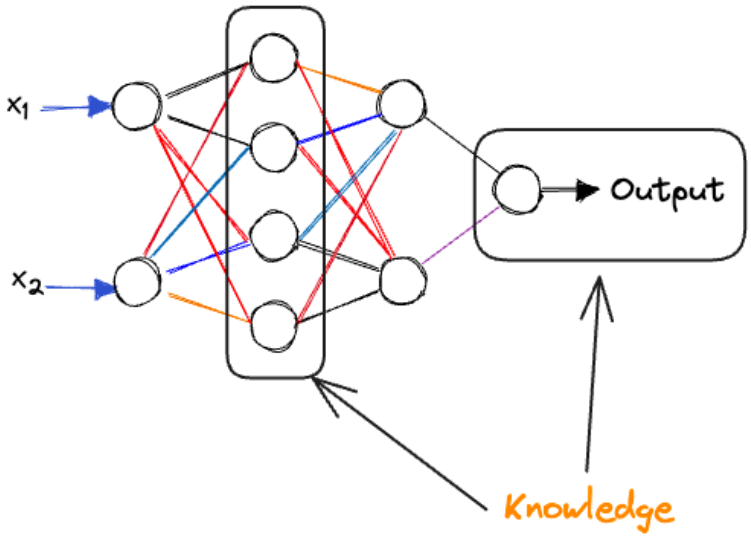
1) Knowledge Distillation

Knowledge distillation is one of the simplest and most effective ways to shrink a model without sacrificing much performance.

Essentially, knowledge distillation involves training a smaller, simpler model (referred to as the "student" model) to mimic the behavior of a larger, more complex model (known as the "teacher" model).

The term can be broken down as follows:

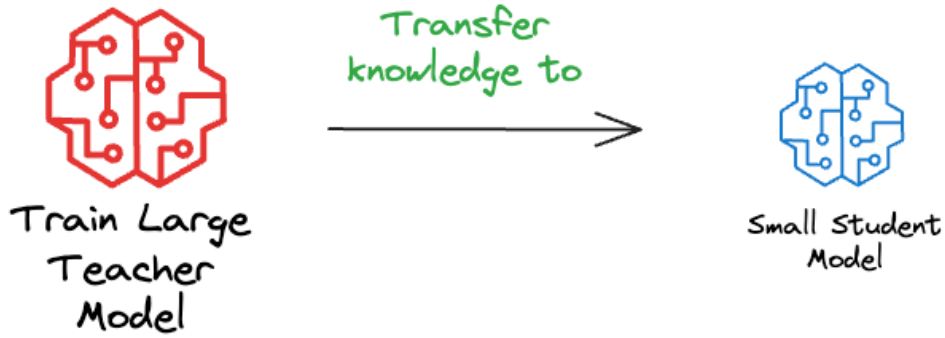
Knowledge: Refers to the understanding, insights, or information that a machine learning model has acquired during training. This "knowledge" can be typically represented by the model's parameters, learned patterns, and its ability to make predictions.



Distillation: In this context, distillation means transferring or condensing knowledge from one model to another. It involves training the student model to mimic the behavior of the teacher model, effectively transferring the teacher's knowledge.

This is a two-step process:

- Train the large model as you typically would. This is called the "teacher" model.
- Train a smaller model, which is intended to mimic the behavior of the larger model. This is also called the "student" model.



The primary objective of knowledge distillation is to transfer the knowledge, or the learned insights, from the teacher to the student model.

This allows the student model to achieve comparable performance with fewer parameters and reduced computational complexity.

The technique makes intuitive sense as well.

Of course, comparing it to a real-world teacher-student scenario in an academic setting, the student model may never perform as well as the teacher model.

But with consistent training, we can create a smaller model that is almost as good as the larger one.

A classic example of a model developed in this way is DistillBERT. It is a student model of BERT.

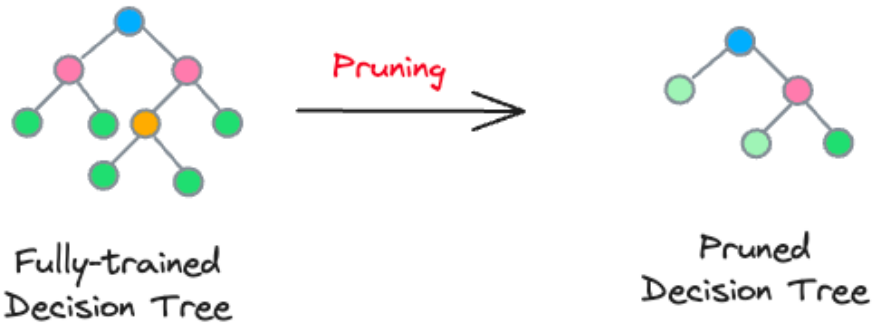
DistilBERT is approximately 40% smaller than BERT, which is a massive difference in size.

Still, it retains approximately 97% of the natural language understanding (NLU) capabilities of BERT.

What's more, DistilBERT is roughly 60% faster in inference.

2) Pruning

Pruning is commonly used in tree-based models, where it involves removing branches (or nodes) to simplify the model.

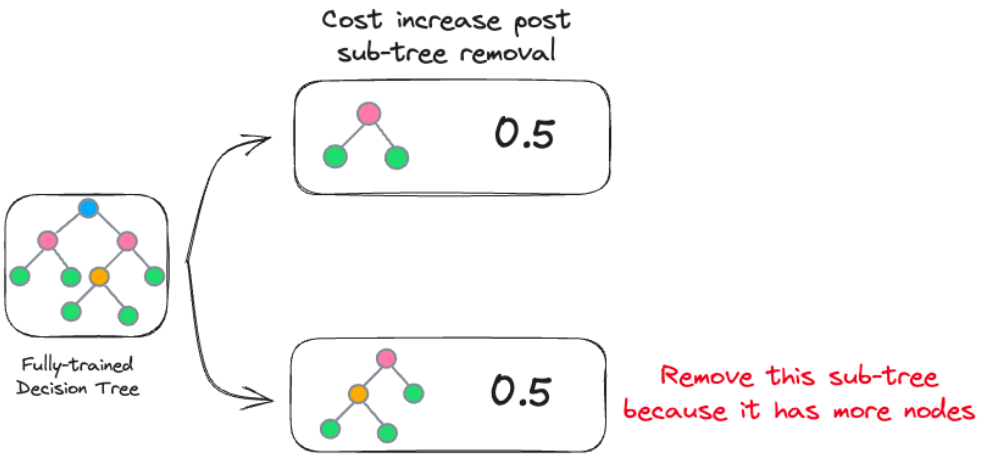


Of course, dropping nodes will result in a drop in the model's accuracy.

Thus, in the case of decision trees, the core idea is to iteratively drop sub-trees, which, after removal, leads to:

- a minimal increase in classification cost
- a maximum reduction of complexity (or nodes)

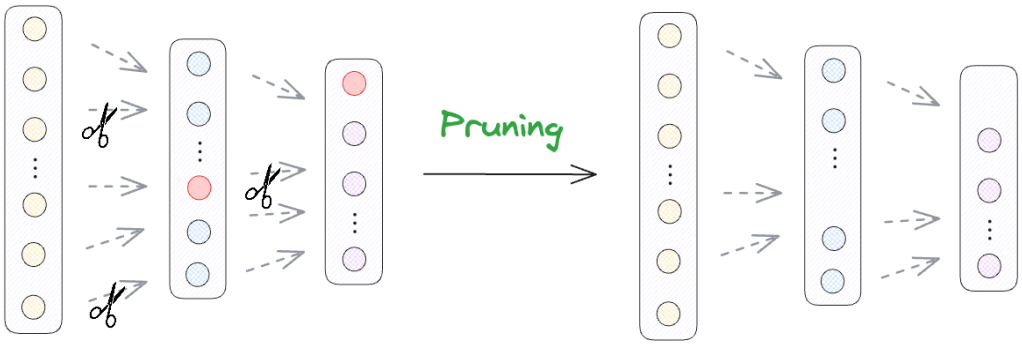
This is depicted below:



In the image above, both sub-trees result in the same increase in cost. However, it makes more sense to remove the sub-tree with more nodes to reduce computational complexity.

The same idea can be translated to neural networks as well.

As you may have guessed, pruning in neural networks involves identifying and eliminating specific connections or neurons that contribute minimally to the model's overall performance.



Removing an entire layer is another option. But we rarely practice it because it may result in misaligned weight matrices.

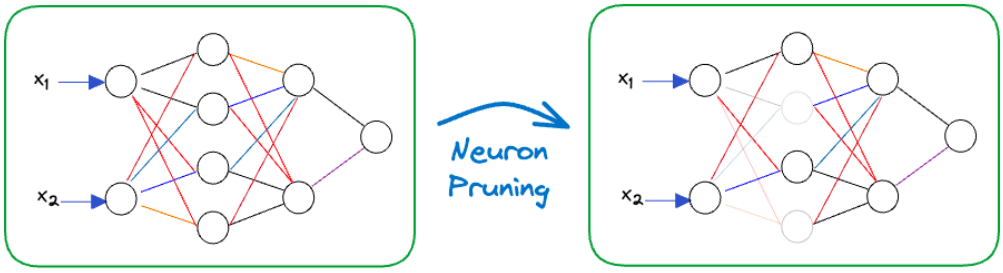
What's more, it isn't easy to quantify the contribution of a specific layer towards the final output.

Thus, instead of removing layers of a trained network, redefining the architecture without the layer we intend to remove can be a better approach.

With pruning, the goal is to create a more compact neural network while retaining as much predictive power as possible.

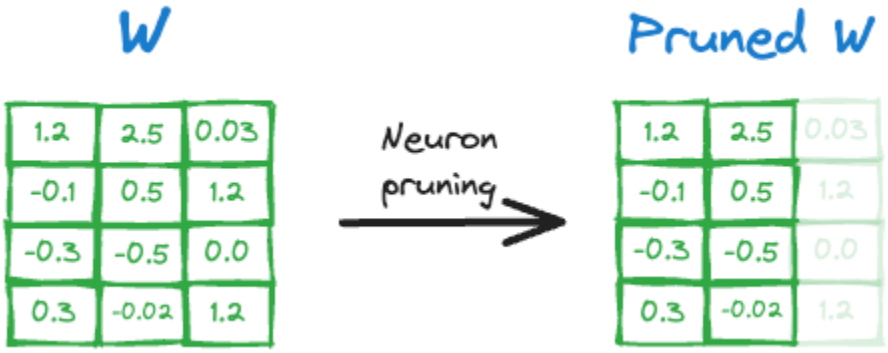
This is primarily done in two ways:

Neuron pruning:



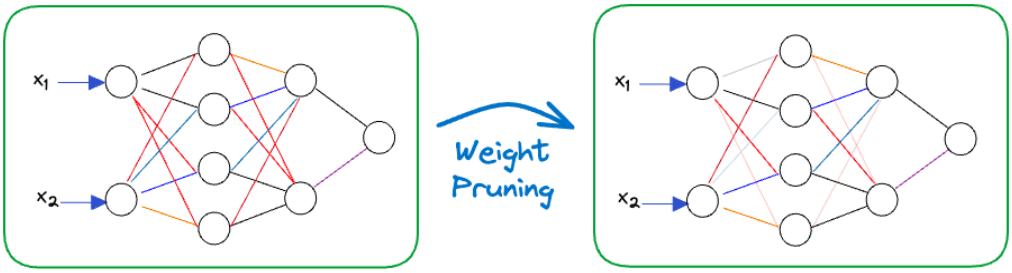
The idea is to eliminate entire nodes from the network.

As a result, the matrices representing the layers become small.

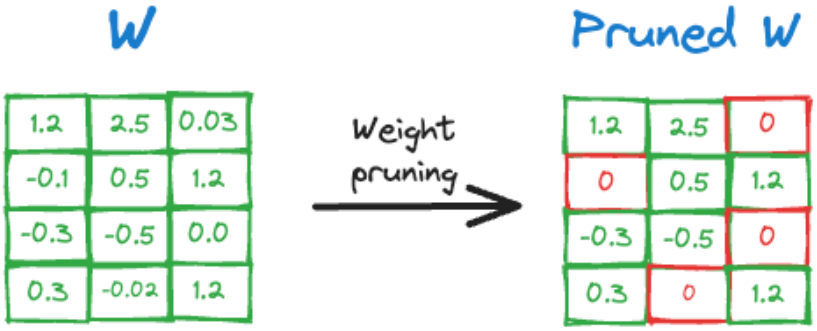


This results in faster inference and lower memory usage.

Weight pruning:



- This involves eliminating edges from the network.
- Weight pruning can be thought of as placing zeros in the matrices to represent the removed edges.



- However, in this case, the size of the matrices remains unaffected.
- Thus, the size of the matrices remains the same, but they become sparse.
- While eliminating edges may not necessarily result in a faster inference, it does help in optimizing memory usage. This is because sparse matrices typically occupy less space than dense matrices.

Note: We can eliminate both edges and nodes as well.

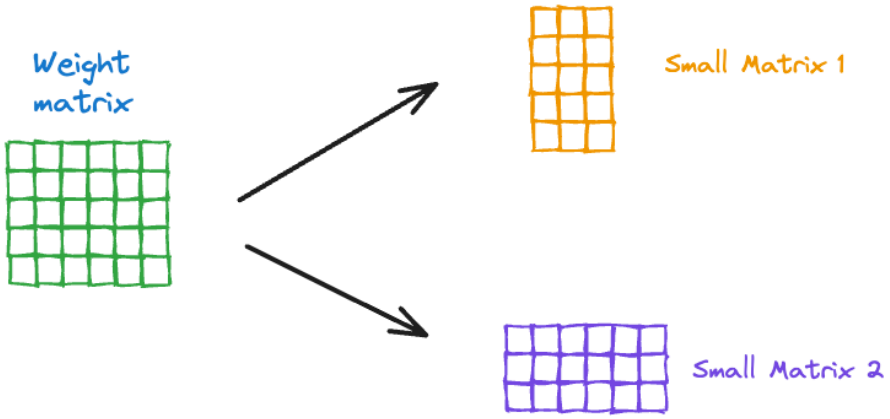
It is evident from the above ideas that by removing unimportant weights (or nodes) from a network, several improvements can be expected:

- Better generalization
- Improved speed of inference
- Reduced model size

3) Low-rank Factorization

At its core, Low-rank Factorization aims to approximate the weight matrices of neural networks using lower-rank matrices.

Essentially, the idea is to represent complex weight matrices as products of two or more simpler matrices.



The idea will become more clear if we understand these individual terms:

Low-rank:

- In linear algebra, the "rank" of a matrix refers to the maximum number of linearly independent rows (or columns) in that matrix.
- Thus, a matrix with a low rank has fewer linearly independent rows or columns.
- This implies that it can be approximated as a combination of a smaller number of basis vectors, helping us to reduce the dimensionality.

Factorization:

- In mathematics, "factorization" means expressing a complex mathematical object (such as a matrix) as the product of simpler mathematical objects.

Thus, Low-rank Factorization means breaking down a given weight matrix into the product of two or more matrices of lower dimensions.

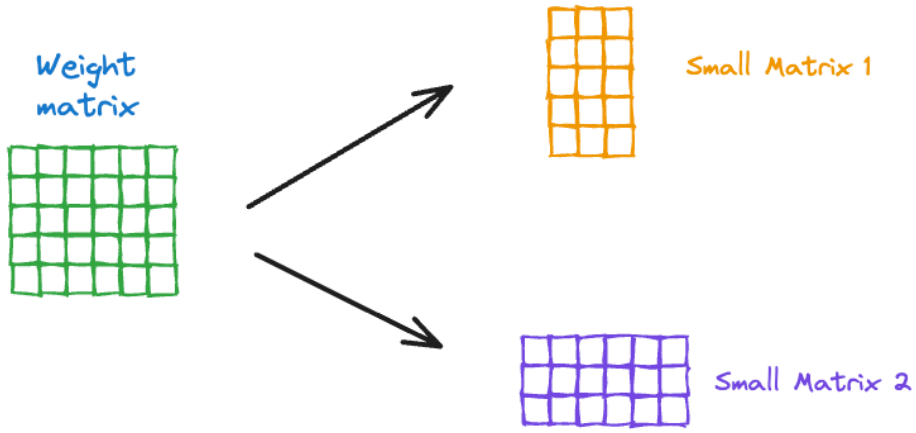
These lower-dimensional matrices are often called "factor matrices" or "basis matrices."

Let's understand how it works.

Step 1) Perform Matrix Factorization

In a neural network, every layer will have a weight matrix.

We can decompose these original weight matrices into lower-rank approximations.

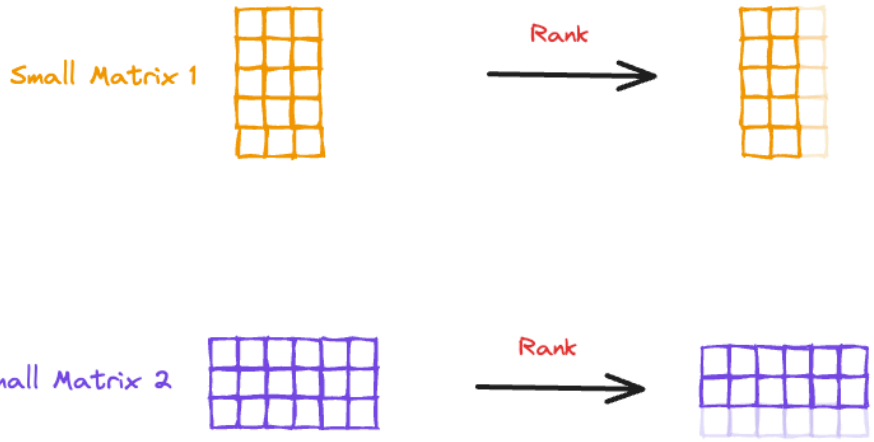


There are many different matrix factorization methods available, such as Singular Value Decomposition (SVD), Non-negative Matrix Factorization (NMF), or Truncated SVD.

Step 2) Specify Rank

In matrix factorization, you'll typically have to choose the rank k for the lower-rank approximation.

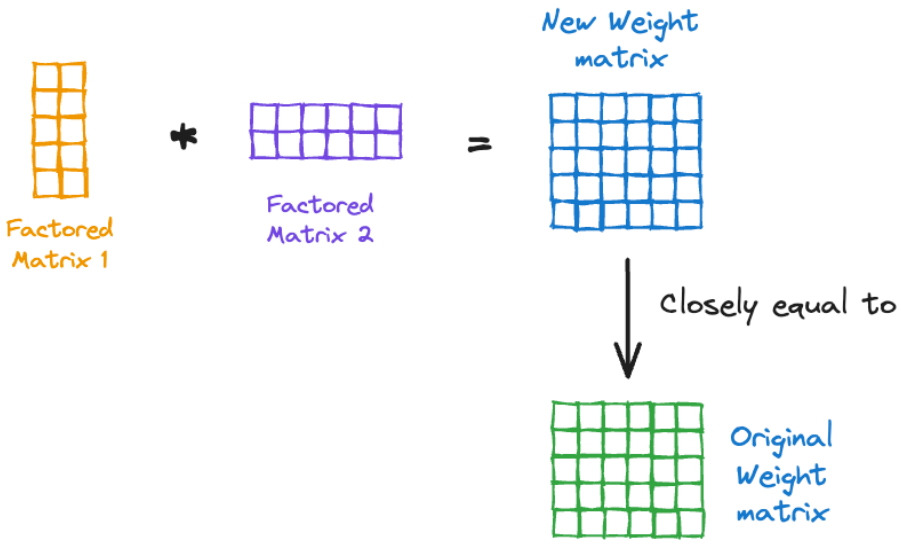
It determines the number of singular values (for SVD) or the number of factors (for factorization methods).



The choice of rank *k* is directly linked to the trade-off between model size reduction and preservation of information.

Step 3) Reconstruct the Weight Matrices

Once you've obtained the lower-rank matrices, you can use them to transform the input instead of the original weight matrices.



The benefit of doing this is that it reduces the computational complexity of the neural network while retaining important features learned during training.

By replacing the original weight matrices with their lower-rank approximations, we can effectively reduce the number of parameters in the model, which reduces

its size.

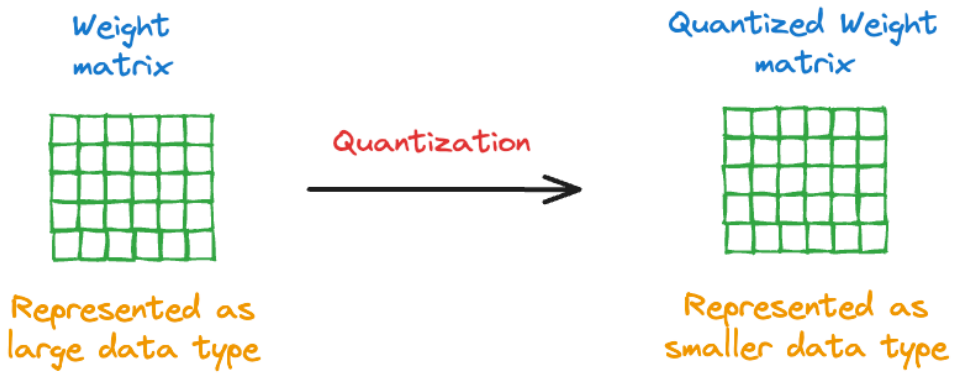
4) Quantization

Typically, the parameters of a neural network (layer weights) are represented using 32-bit floating-point numbers. This is useful because it offers a high level of precision.

Also, as parameters are typically not constrained to any specific range of values, all deep learning frameworks, by default, assign the biggest data type to parameters.

But using the biggest data type also means consuming more memory.

As you may have guessed, Quantization involves using lower-bit representations, such as 16-bit, 8-bit, 4-bit, or even 1-bit, to represent parameters.



This results in a significant decrease in the amount of memory required to store the model's parameters.

For instance, consider your model has over a million parameters, each represented with 32-bit floating-point numbers.

If possible, representing them with 8-bit numbers can result in a significant decrease (~75%) in memory usage while still allowing for a large range of values to be represented.

Of course, Quantization introduces a trade-off between model size and precision.

While reducing the bit-width of parameters makes the model smaller, it also leads to a loss of precision.

This means the model's predictions become more approximate than the original, full-precision model.

Thus, it is important to carefully evaluate the trade-off between model size/inference speed and accuracy when considering Quantization for deployment.

Despite this trade-off, Quantization can be particularly useful in scenarios where model size is a critical constraint, such as edge devices, mobile applications, or specialized hardware like smartphones.

Model compression makes models smaller and faster, but LLMs also introduce challenges that show up only during inference.

To handle those, we use techniques built for autoregressive decoding like KV caching.

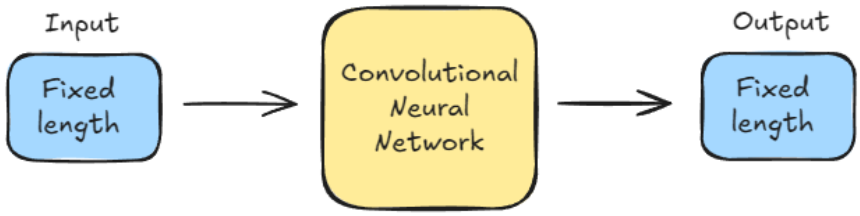
Regular ML Inference vs. LLM Inference

LLM Inference presents unique challenges over regular ML inference, due to which we have specialized, high-performance LLM inference engines, like vLLM, LMCache, SGLang, and TensorRT LLM.

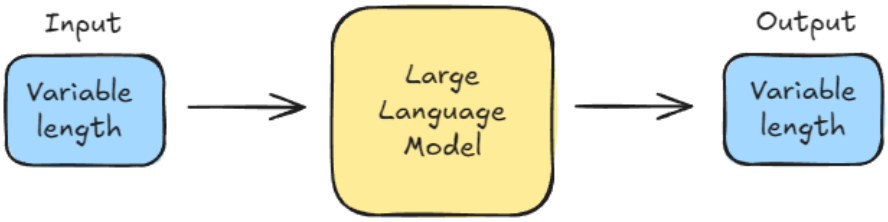
Let's understand these challenges today and how we solve them!

Continuous batching

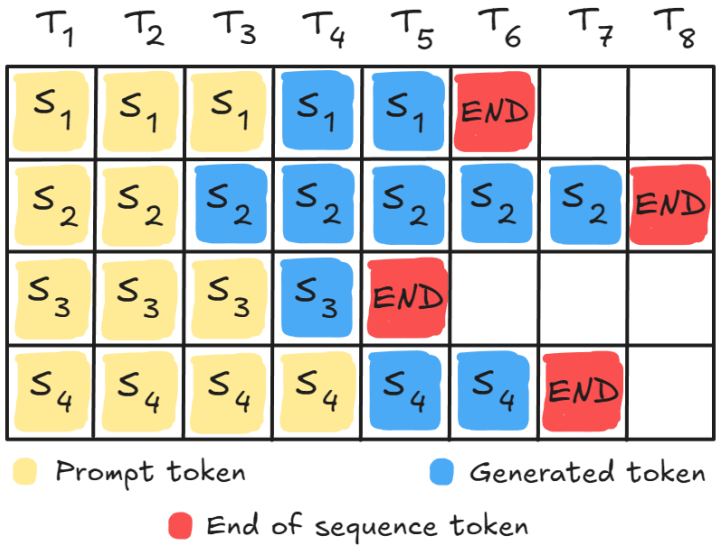
Traditional models, like CNNs, have a fixed-size image input and a fixed-length output (like a label). This makes batching easy.



LLMs, however, deal with variable-length inputs (the prompt) and generate variable-length outputs.



So if you batch some requests, all will finish at different times, and the GPU would have to wait for the longest request to finish before it can process new requests. This leads to idle time on the GPU:



Continuous Batching solves this.

Instead of waiting for the entire batch to finish, the system monitors all sequences and swaps completed ones (<EOS> token) with new queries: