Before Starting

The Why

The What

The How

The Problem Statement

Learning in the age of AI

Our own newsletter

Time Constraints

Step 1 - The Structure

Introduction

Big Story of the Week

3–5 Quick Updates

Top Research Papers

Top Github Repos

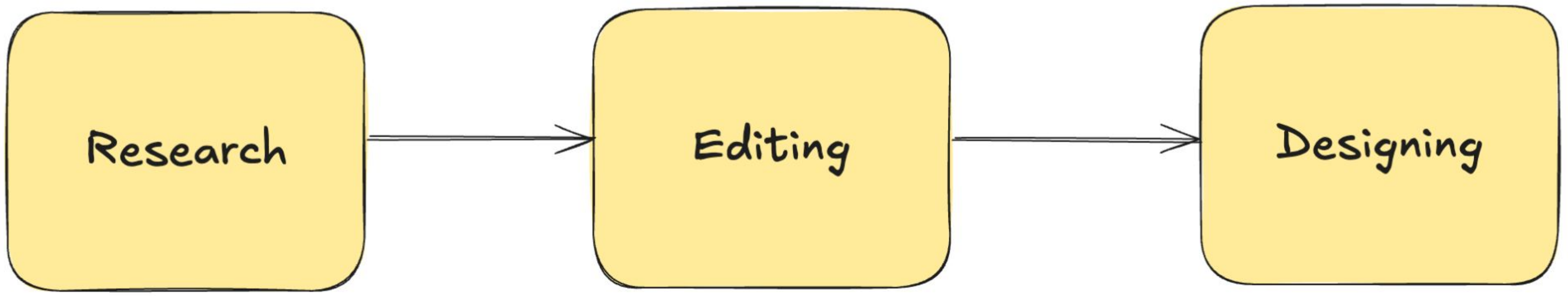
1 Quick Tutorial

Top AI products

Top X posts

Closing notes

Step 2 - Defining the Process



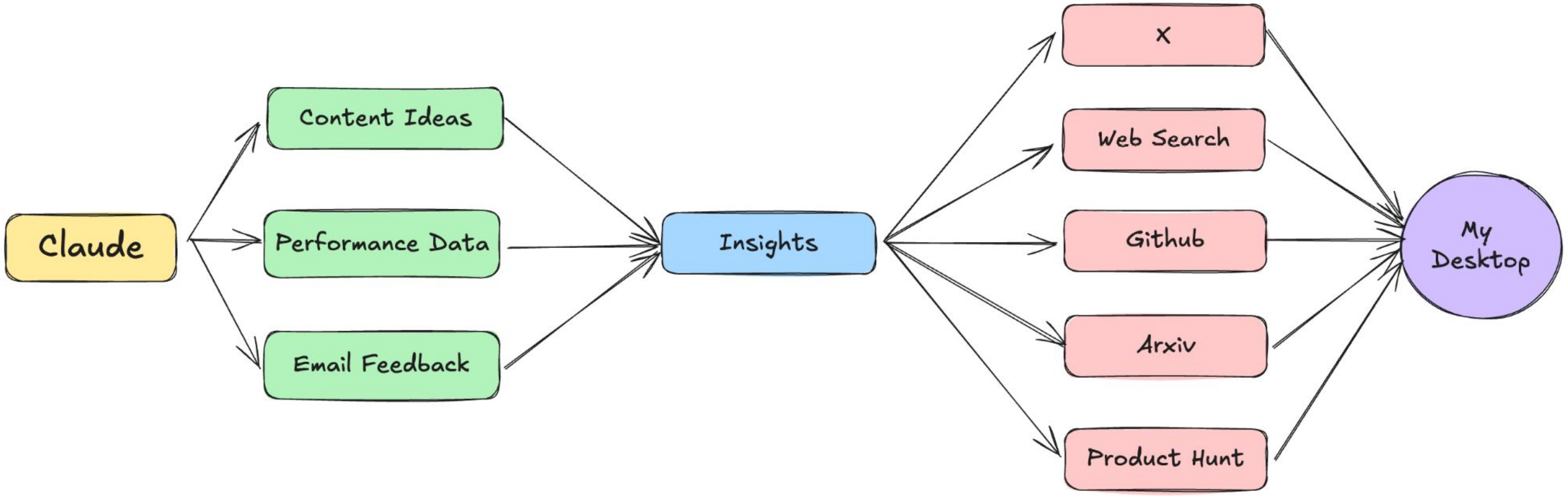
Step 3 - Deciding the Tools

The Main AI

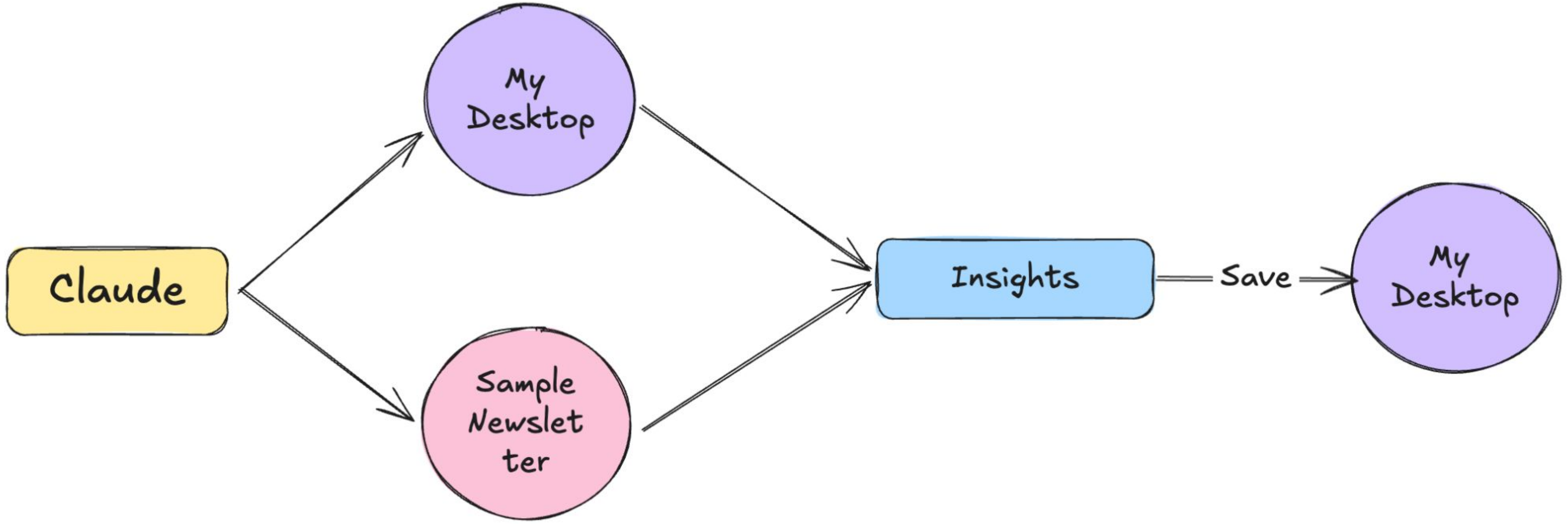
The Tools

Github	Web Search	Google Drive
Arxiv	Gmail	Product Hunt

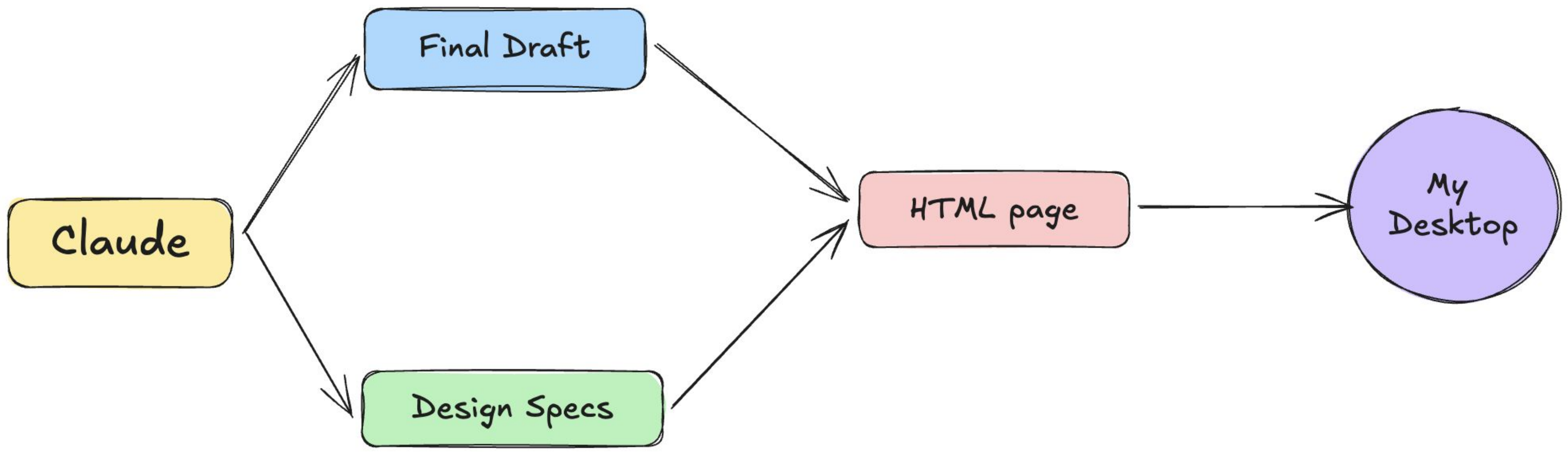
Step 4 - The Research Phase



Step 5 - Editing Phase



Step 6 - Designing Phase



PART 1

THE WHY

The Arrival of LLMs

ChatGPT was launched on Nov 30 2022

It crossed 1 Million users in 5 days

Then it crossed 100 Million users in 2 months

Not another software launch.

Waves of Adoption

Wave 1 - Pure Wonder

Explain Quantum Physics from a cat’s perspective

What would happen if gravity worked backwards

Write a song about pizza in the style of Shakespeare

Impact - Social Media Exploded

Waves of Adoption

Wave 2 - Professional Adoption

Lawyers: "Summarize this 50-page contract."

Developers: "Debug this Python function."

Teachers: "Create a lesson plan about photosynthesis."

Impact - Individual productivity boom

Waves of Adoption

Wave 3 – The API Revolution

Copilot across Word, Excel, PowerPoint, and Outlook

AI in Gmail, Docs, Sheets, and Drive

New AI-first tools like Cursor, Perplexity emerged

Impact - AI became more accessible

The Problem of Fragmentation

AI in notion couldn't talk to AI in Slack

VS Code coding assistant knew nothing about discussions in MS teams

People found themselves living in multiple AI worlds

Users were juggling between multiple AI assistants

The Vision vs The Reality

They wanted one unified AI partner that can understand their work

A unified tool that can solve any problem related to their work

Users never wanted 5 different AI tools

But there was a big problem in building a unified AI

The problem of Context

What is Context

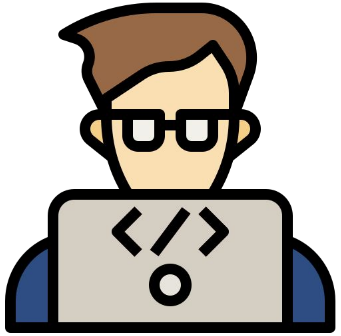
Context is everything an AI can "see" when it generates a response.

More formally, Context refers to the information(conversation history, external docs etc) that the LLM uses to generate a response.

For e.g. while chatting with ChatGPT, the past messages forms the context.

Example - Software Engg.

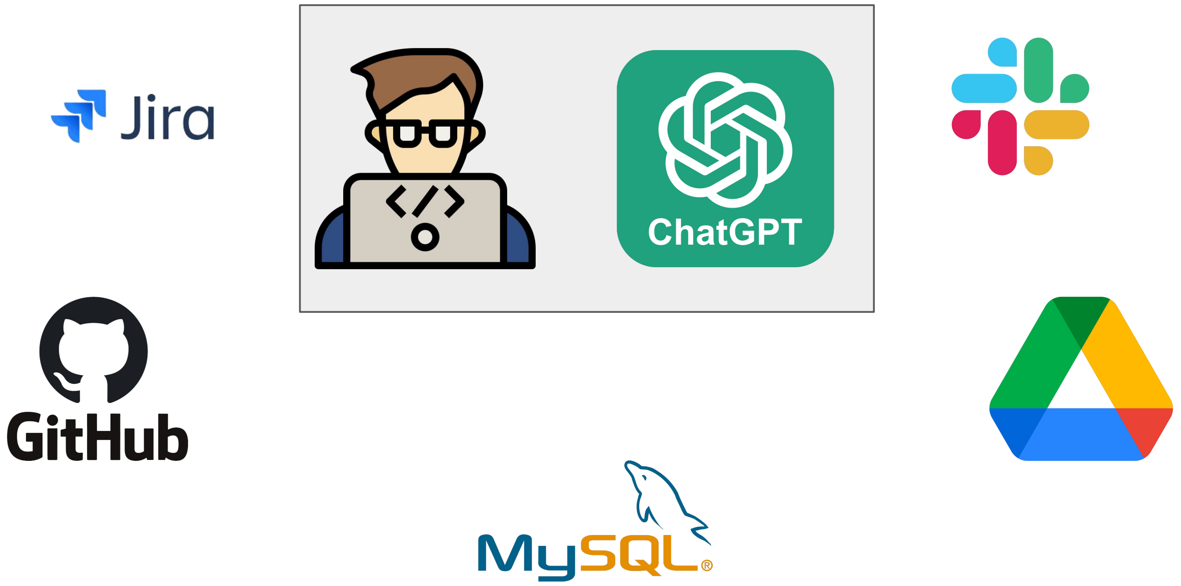
Jira



GitHub

MySQL

Software Engg. workflow with AI



Context is scattered

The Copy-Paste Hell

Need to paste thousands of line to ask one simple question

Developers have become Human APIs

Context Assembly Time > Development Time

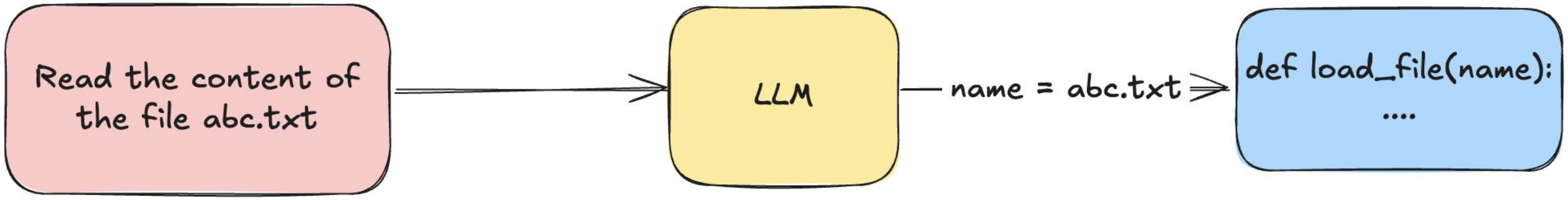
Managing what the AI remembers

Scaling problems

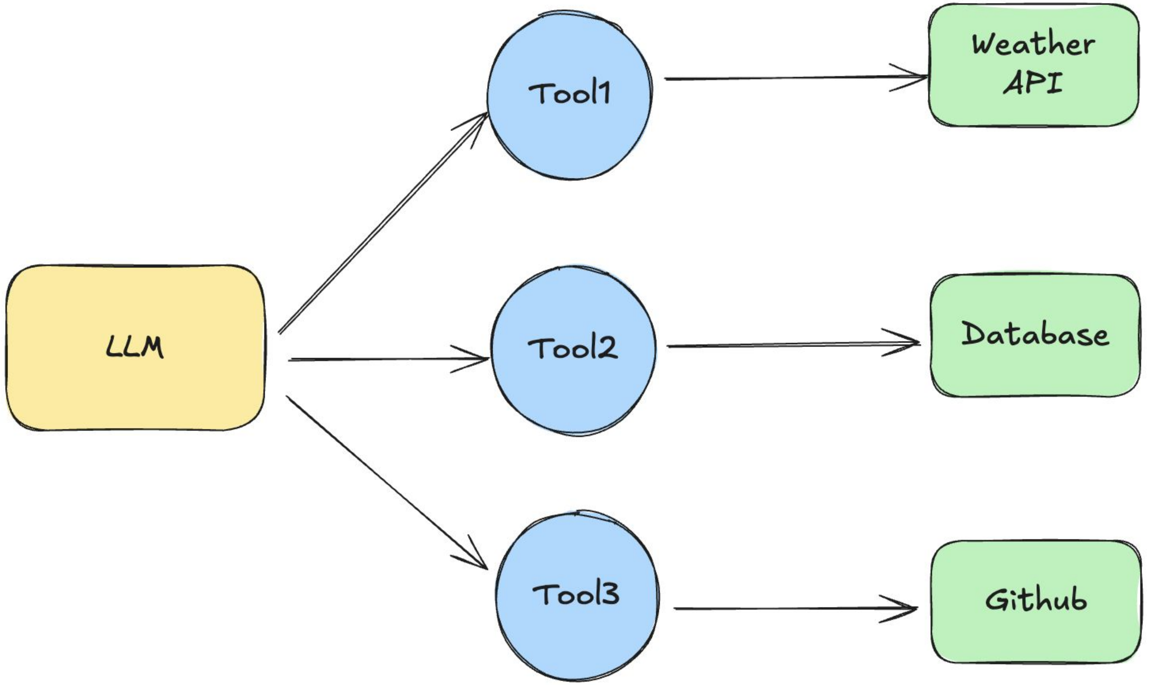
The Solution - Function Calling

OpenAI introduced function calling in mid 2023

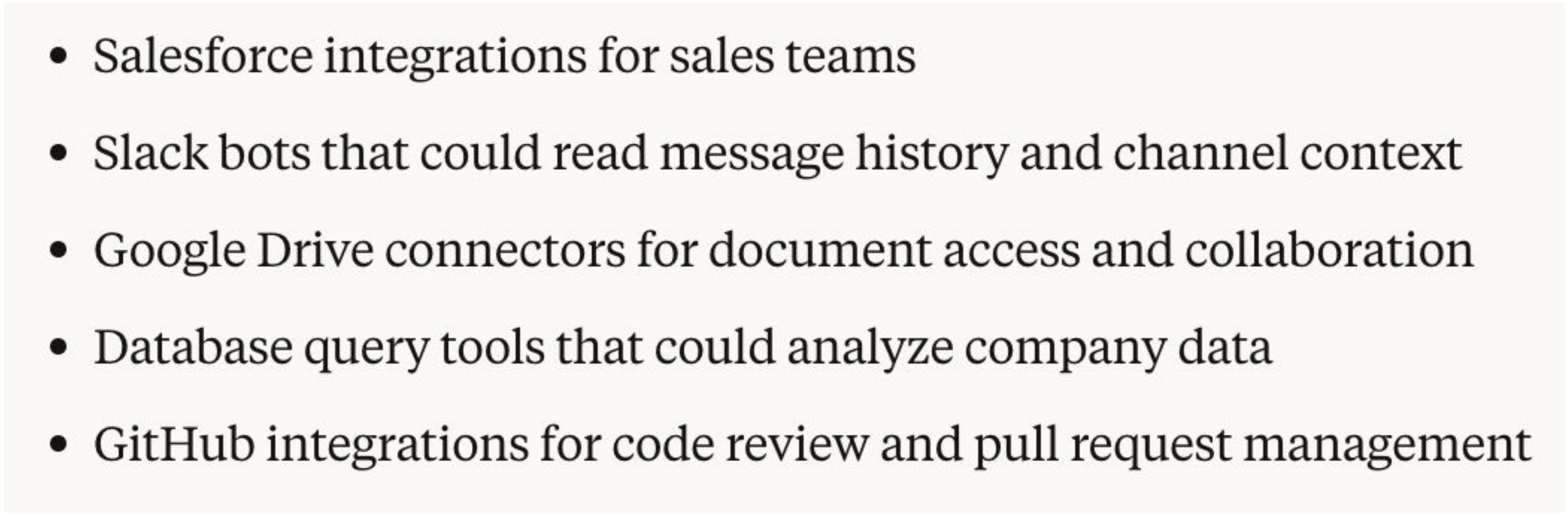
Function Calling is a way using which LLMs can call ext functions



The Rise of Tools

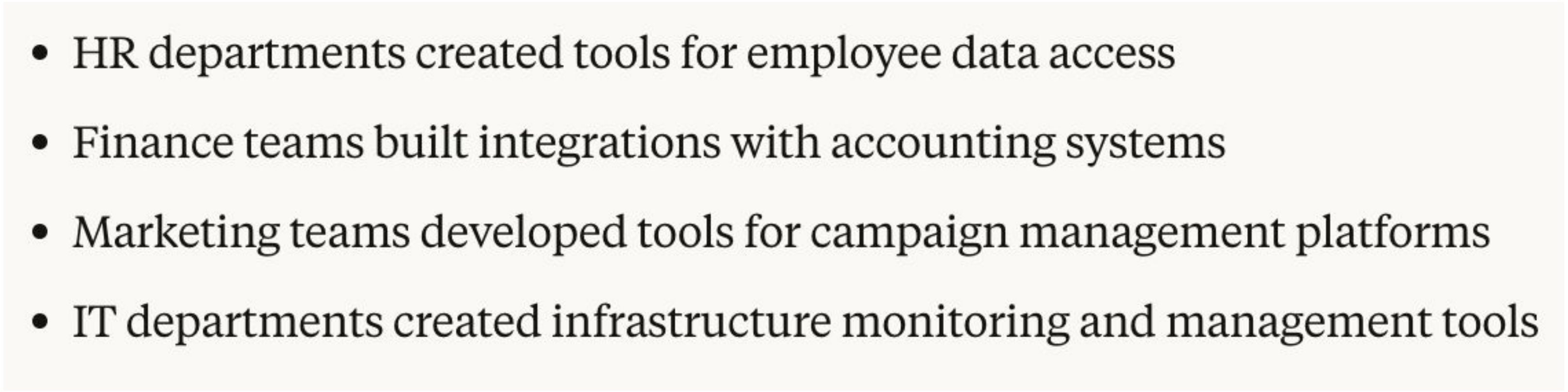


Implication of Tools

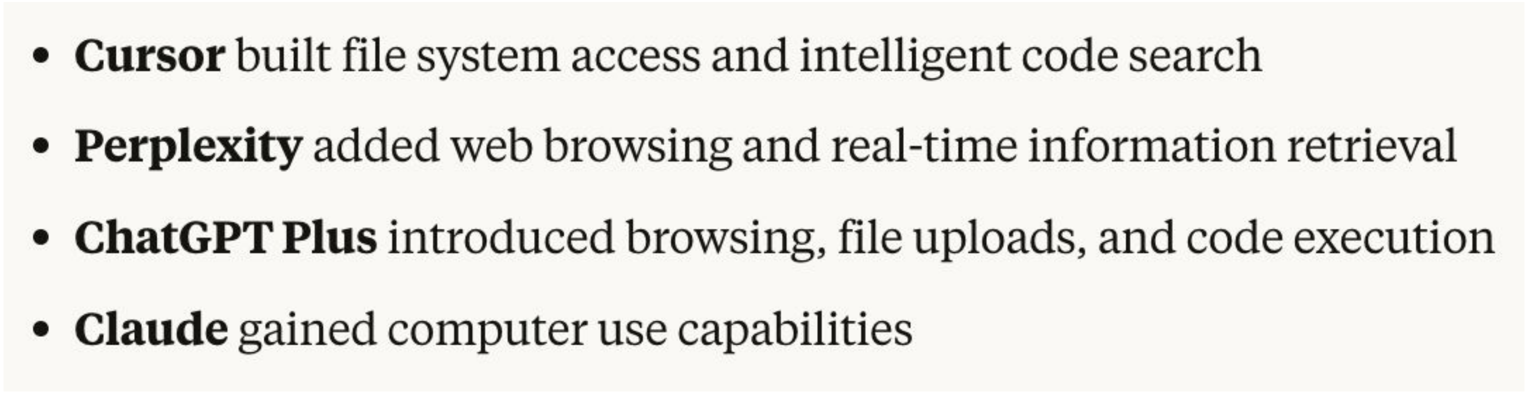


- Salesforce integrations for sales teams
- Slack bots that could read message history and channel context
- Google Drive connectors for document access and collaboration
- Database query tools that could analyze company data
- GitHub integrations for code review and pull request management

Implication of Tools

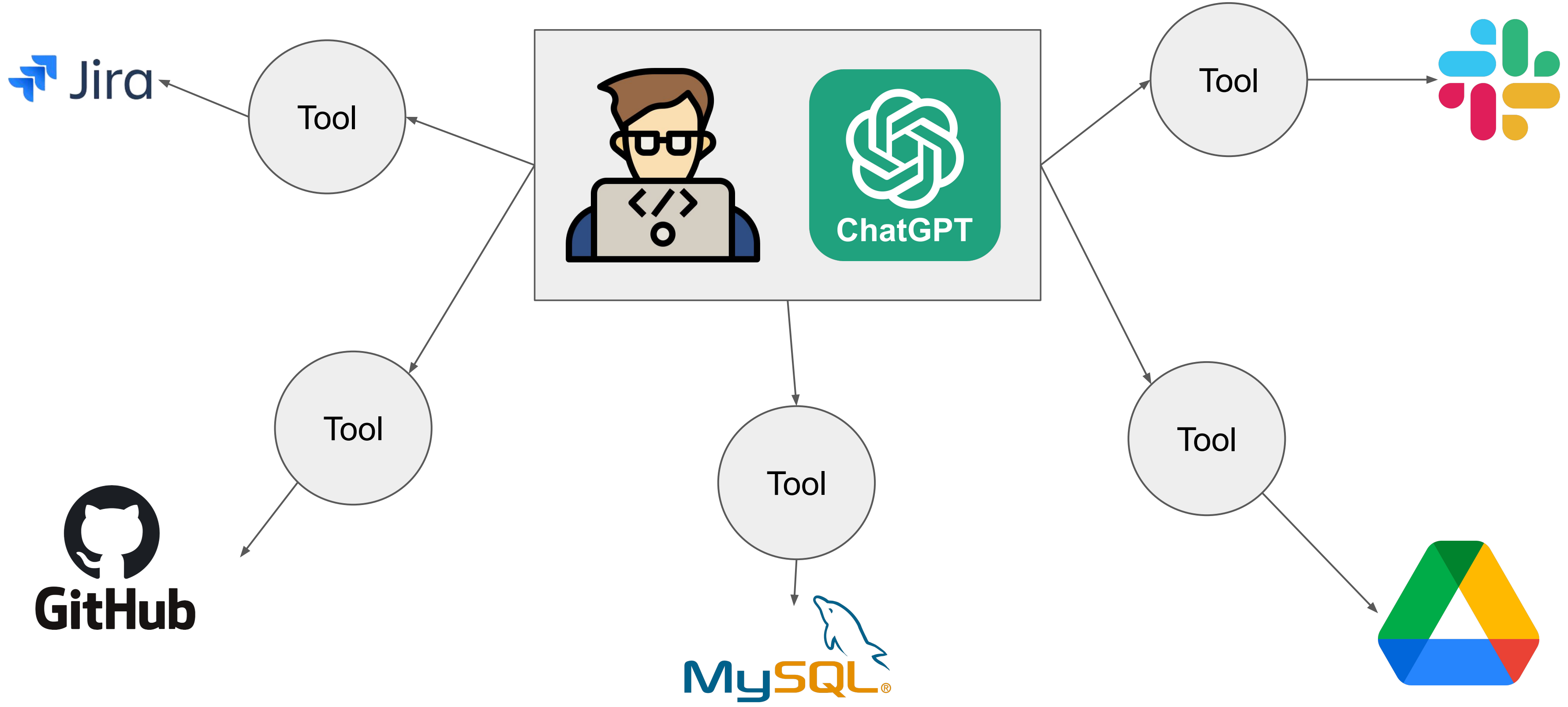


- HR departments created tools for employee data access
- Finance teams built integrations with accounting systems
- Marketing teams developed tools for campaign management platforms
- IT departments created infrastructure monitoring and management tools



- **Cursor** built file system access and intelligent code search
- **Perplexity** added web browsing and real-time information retrieval
- **ChatGPT Plus** introduced browsing, file uploads, and code execution
- **Claude** gained computer use capabilities

The New Scenario



Problem with Tools

Integration Problem

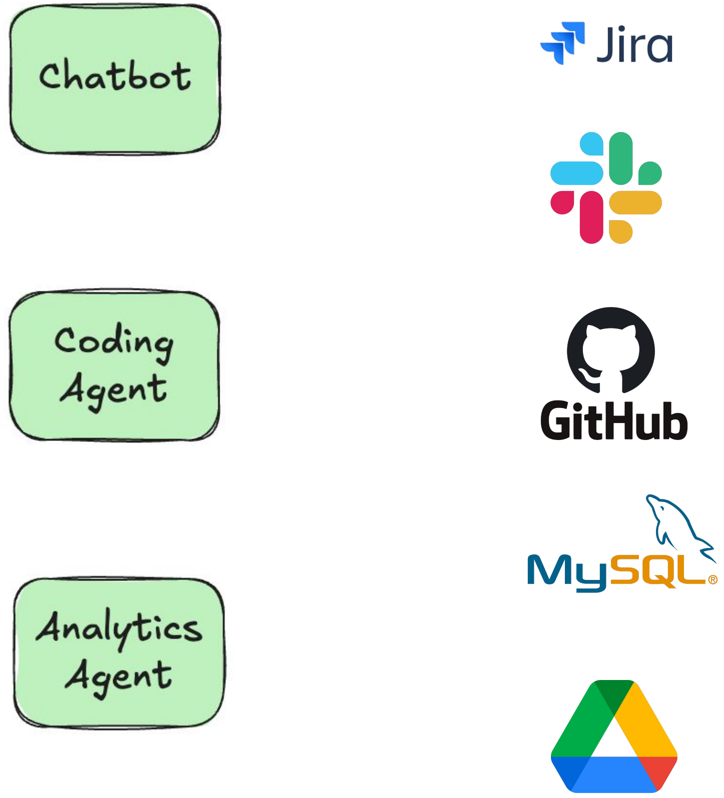
N * M

Development Nightmare

Diff authentication methods

Diff data formats and API patterns

Diff error handling



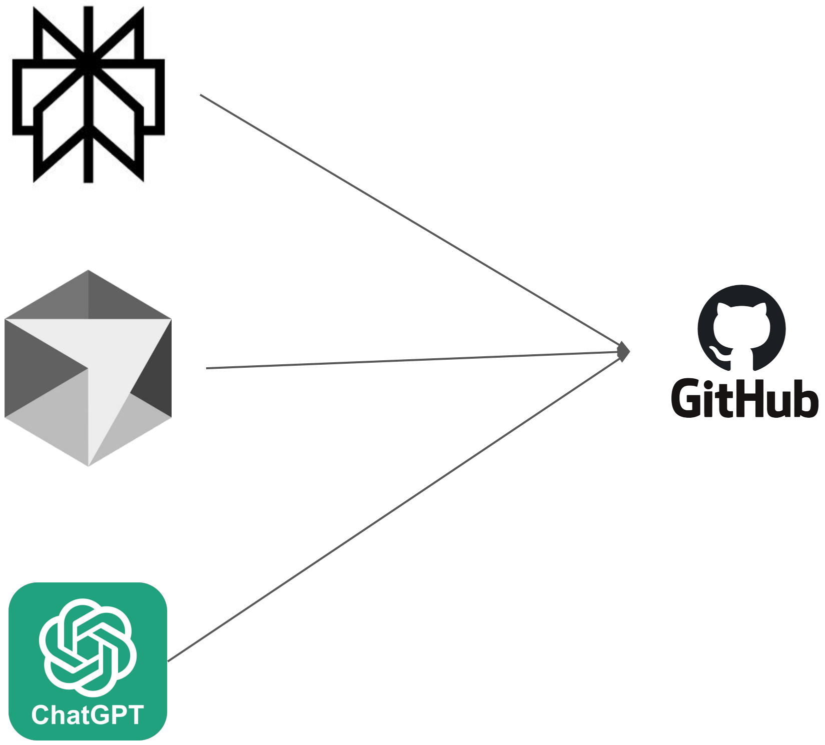
Problem with Tools

Maintenance Problem

Security fragmentation

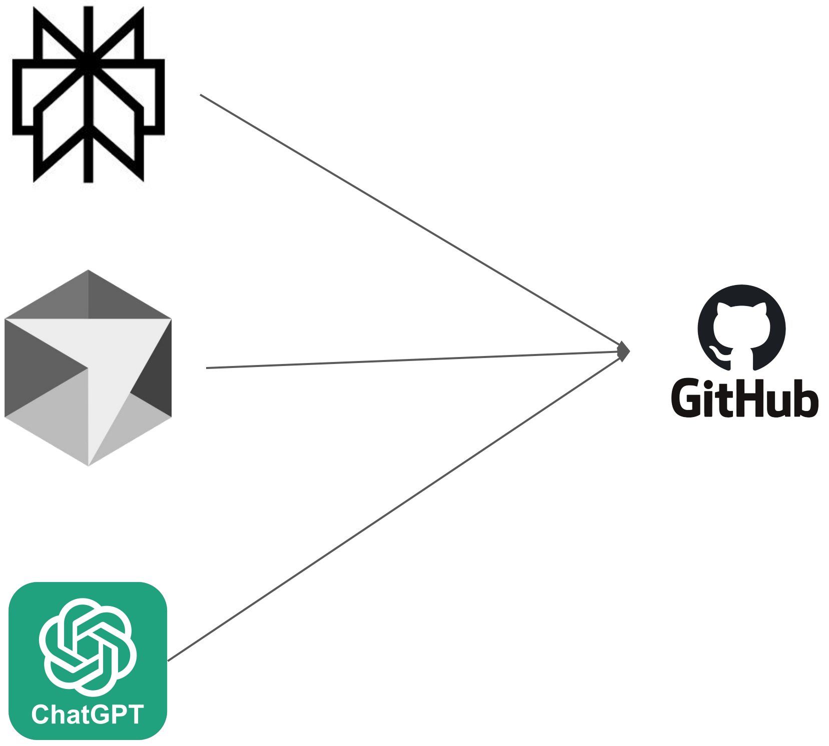
Cost and time wastage

Overview of the Problem



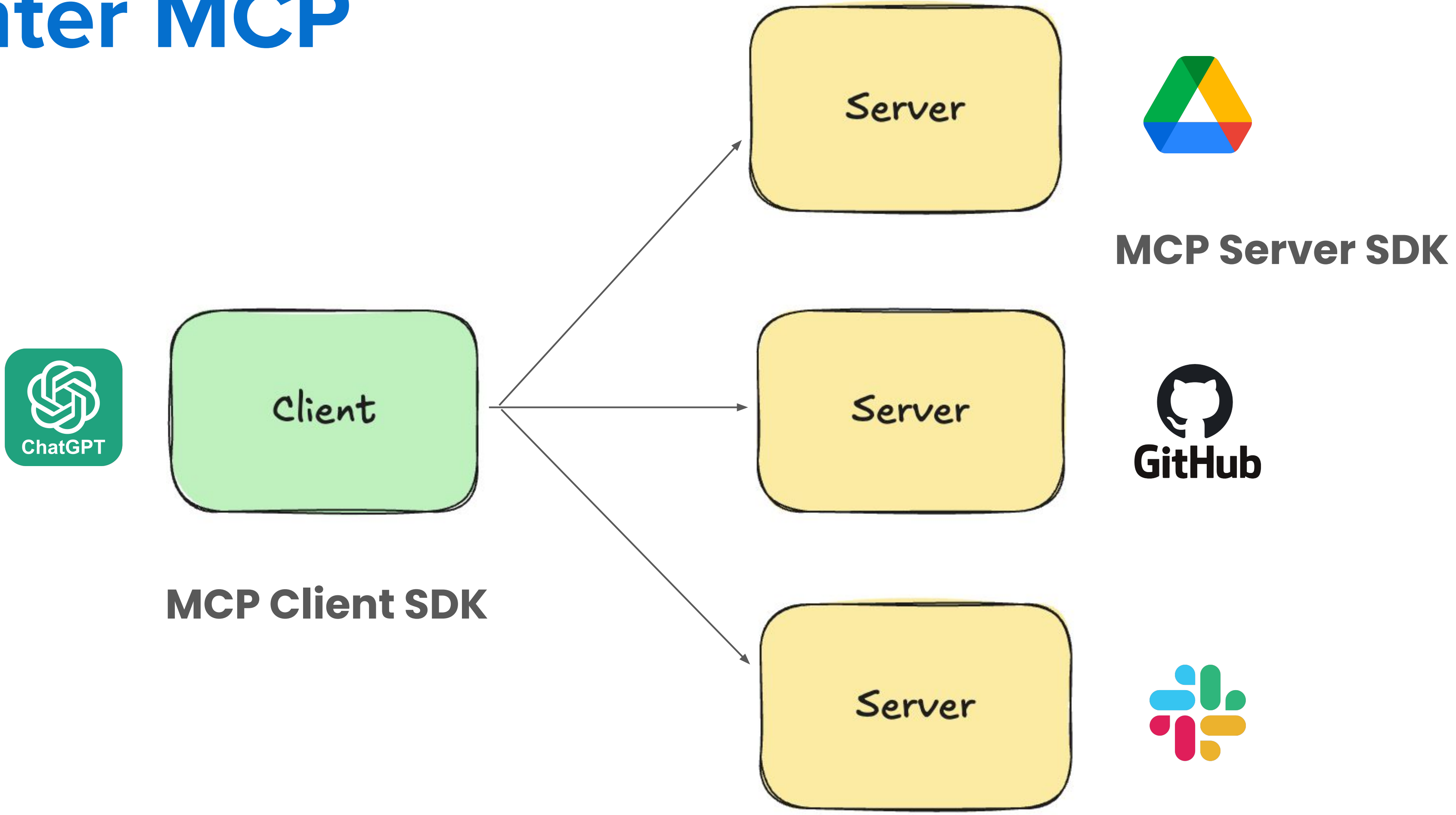
Every AI tool was building its own way to call every API.

The Solution



Github builds an integration that can be used by any AI Tool

Enter MCP



MCP Server SDK

MCP Client SDK

MCP vs Function/Tool Calling

```
def get_weather(city: str):
    url = f"https://myweatherapi.com/weather?city={city}&key=API_KEY"
    response = http_get(url)
    data = json_parse(response)
    return f"{city}: {data['temp_c']}°C, {data['condition']}"
```

Client

```
@app.route("/weather")
def weather_endpoint():
    city = request.args.get("city")
    # Query internal weather database
    result = lookup_weather(city)    #
    return jsonify(result)
```

Server

MCP vs API

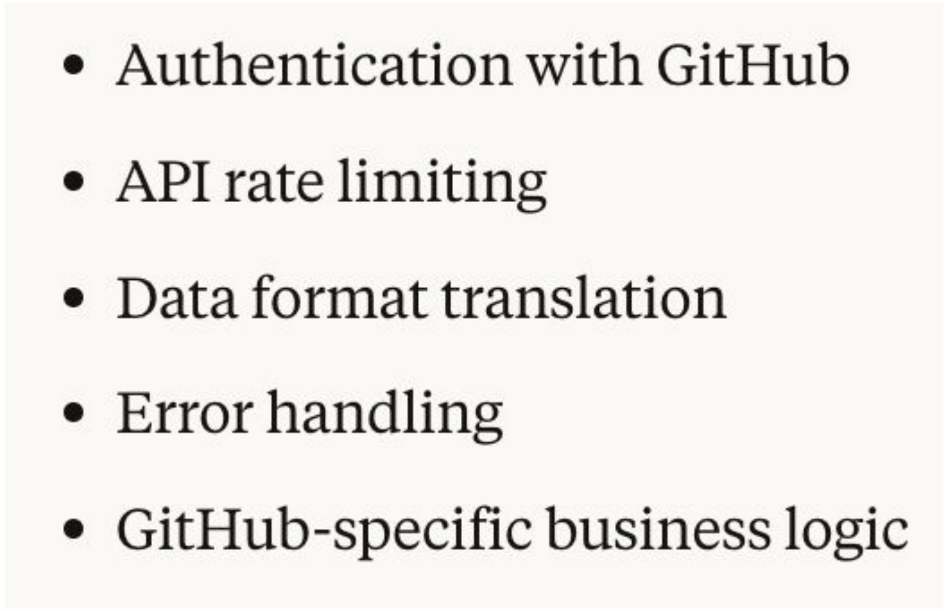
Server

```python
from mcp.server import Server

server = Server("WeatherServer")

@server.tool("get_weather")
def get_weather(city: str):
    data = lookup_weather(city)  # e.g.,
    return {
        "temperature": data["temp_c"],
        "condition": data["condition"]
    }

server.start()
```

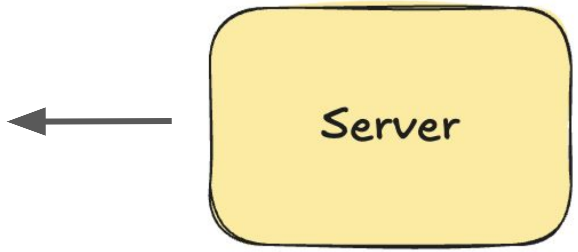
Client

```python
# AI client just calls tools via MCP
result = call_tool("get_weather", {"city": "London"})
```

Server does the Heavy Lifting



- Authentication with GitHub
- API rate limiting
- Data format translation
- Error handling
- GitHub-specific business logic



The Client has to just connect to the server using the same language

Benefits

N clients and M servers = M + N integrations

No maintenance overhead

Reduced cost and time

Better security

MCP Ecosystem

More AI Chatbots supporting MCP → More valuable for services to build MCP servers

More MCP servers available → More valuable for AI tools to support MCP

More adoption → More standardization → More ecosystem value

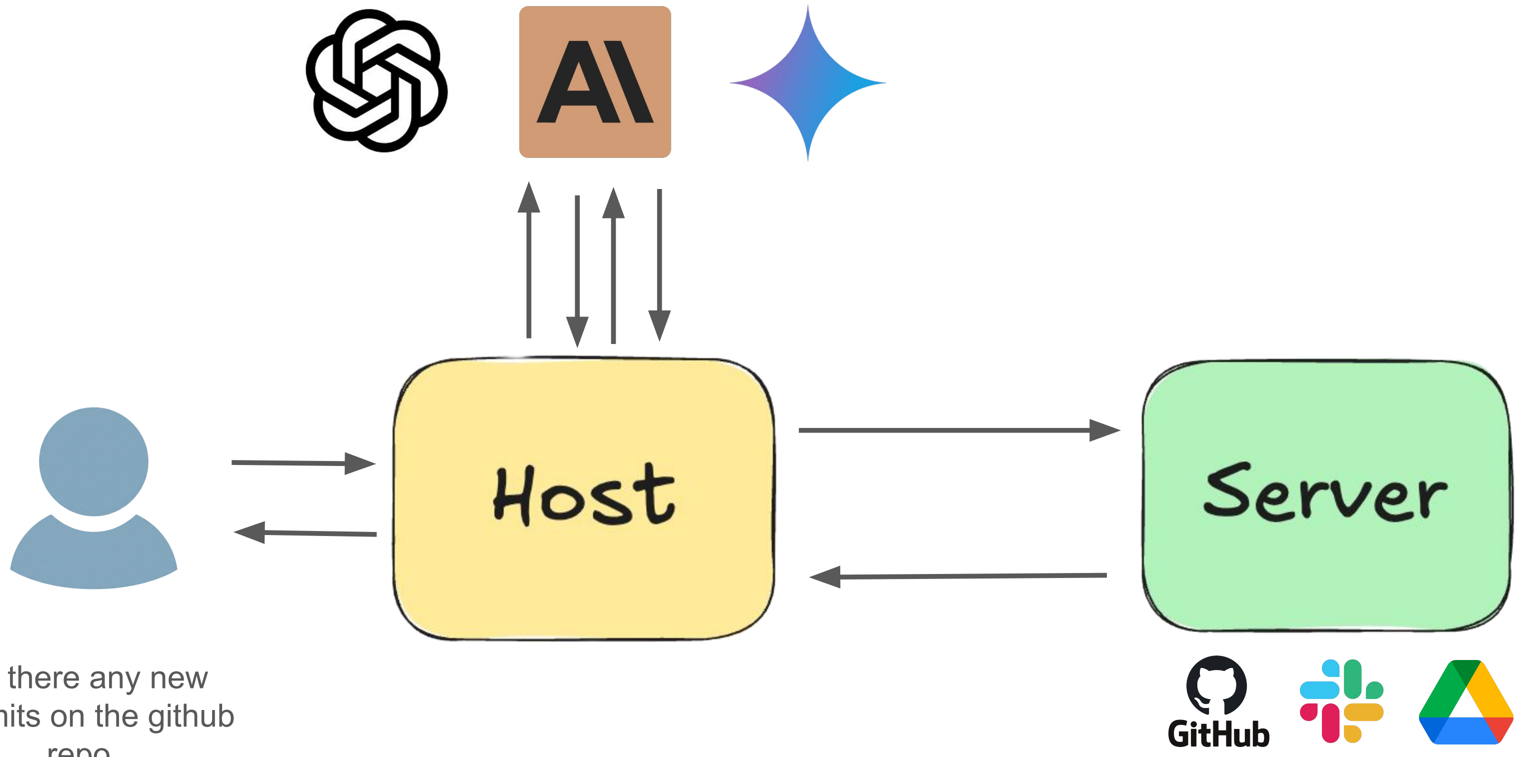
Not supporting MCP meant being cut off from the rapidly growing ecosystem

PART 2

THE WHAT

Architecture

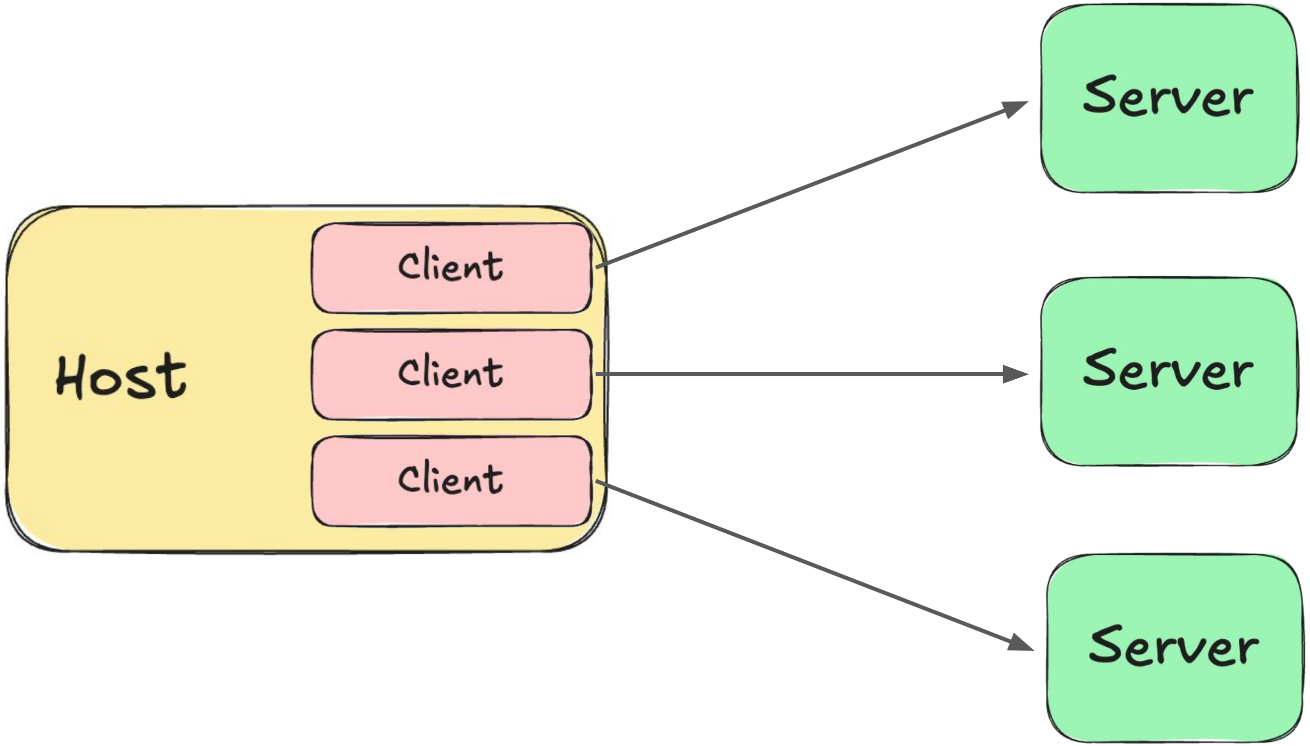
Simplest Version



Are there any new commits on the github repo

Architecture

Adding a Client



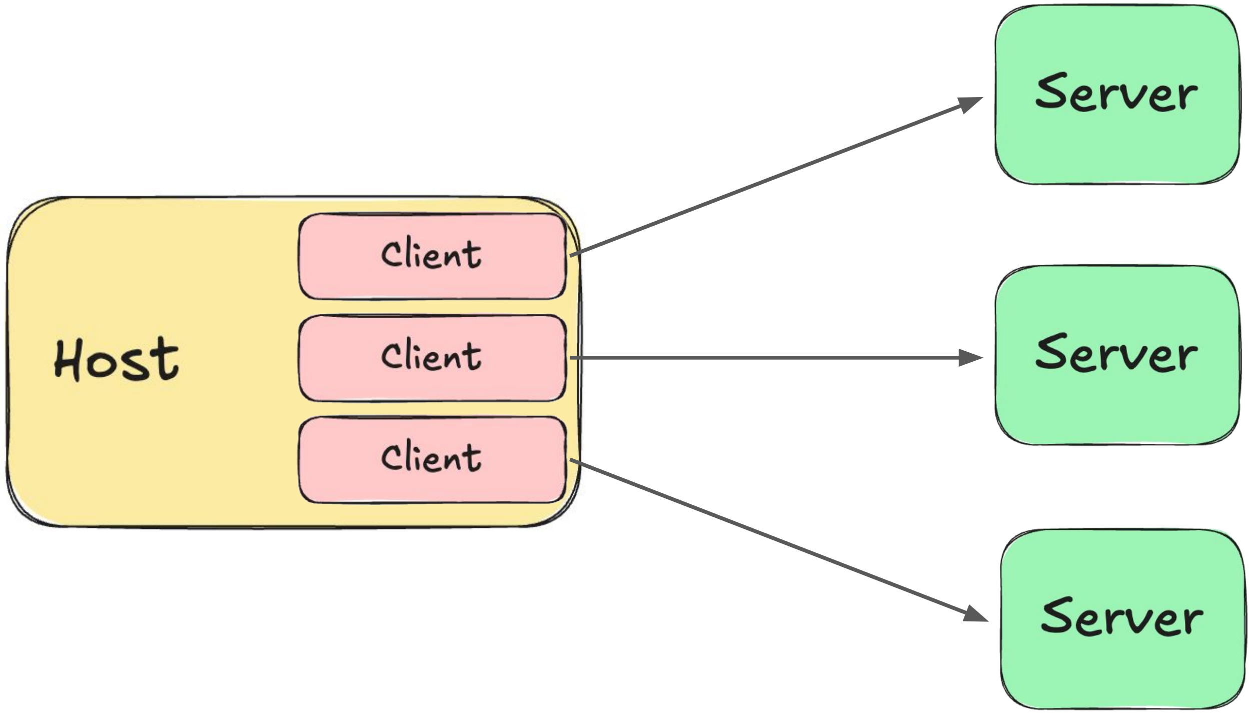
Benefits

Decoupling

- Safety
- Parallelism

Scalability

Architecture



Primitives

Things the server can offer to Host

Tools

Resources

Prompts

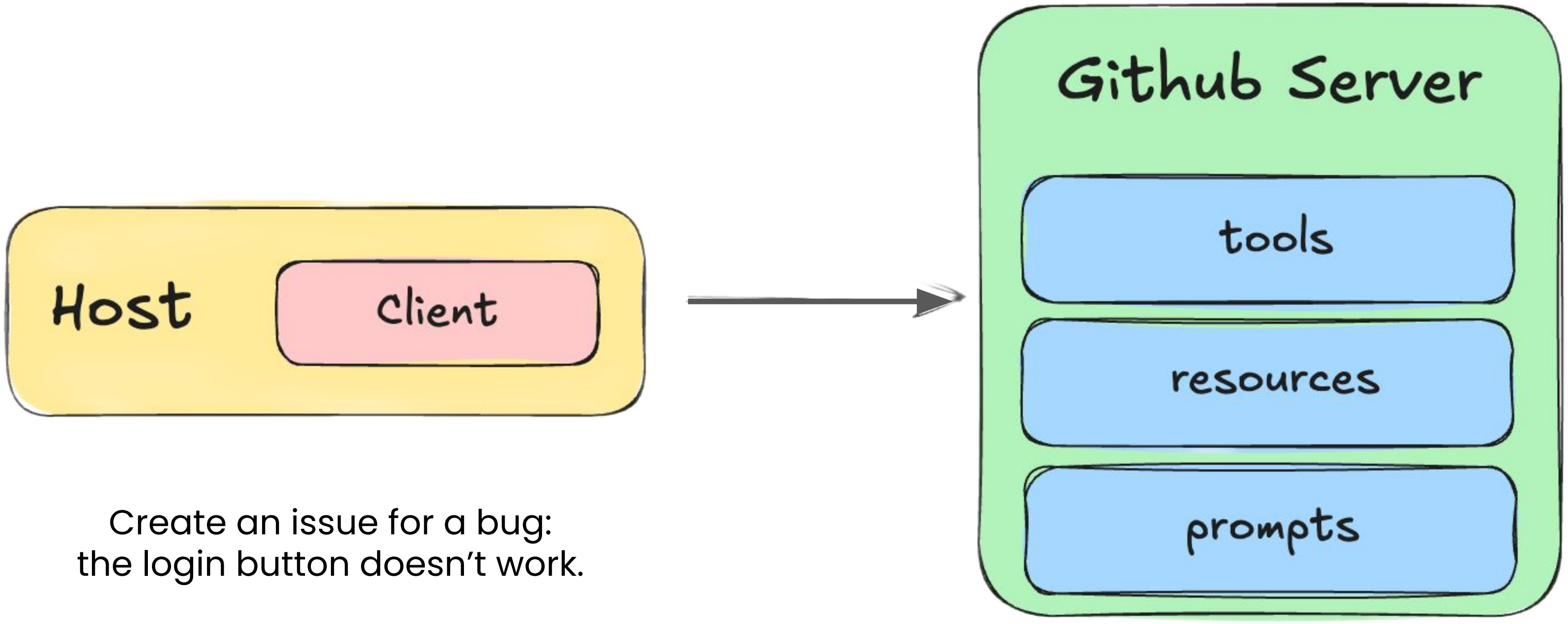
MCP Primitives

Tools - Actions the AI ask the server to perform.

Resources - Structured data sources that the AI can read

Prompts - Predefined prompt templates or instructions that the server offers to help shape the AI's behavior

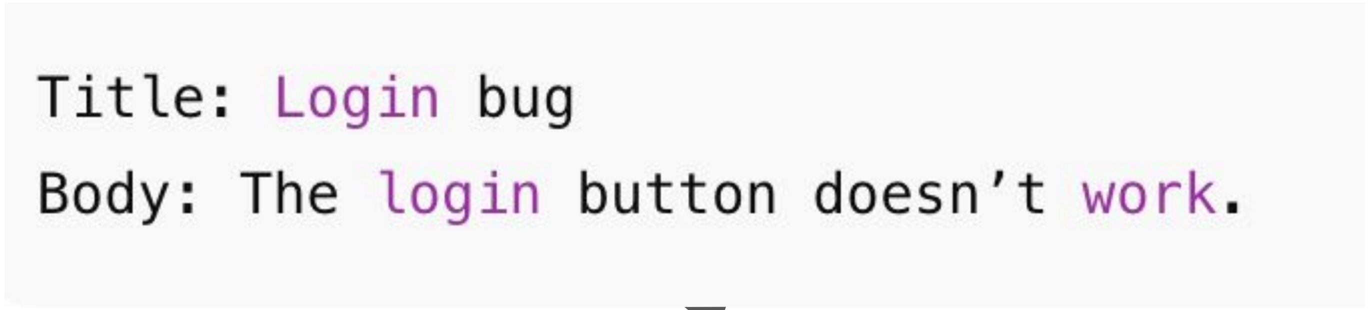
The Prompts Primitives



Create an issue for a bug:
the login button doesn't work.

/create-issue

The Prompts Primitives



```
Title: Login bug
Body: The login button doesn’t work.
```

Too vague!

The Prompts Primitives

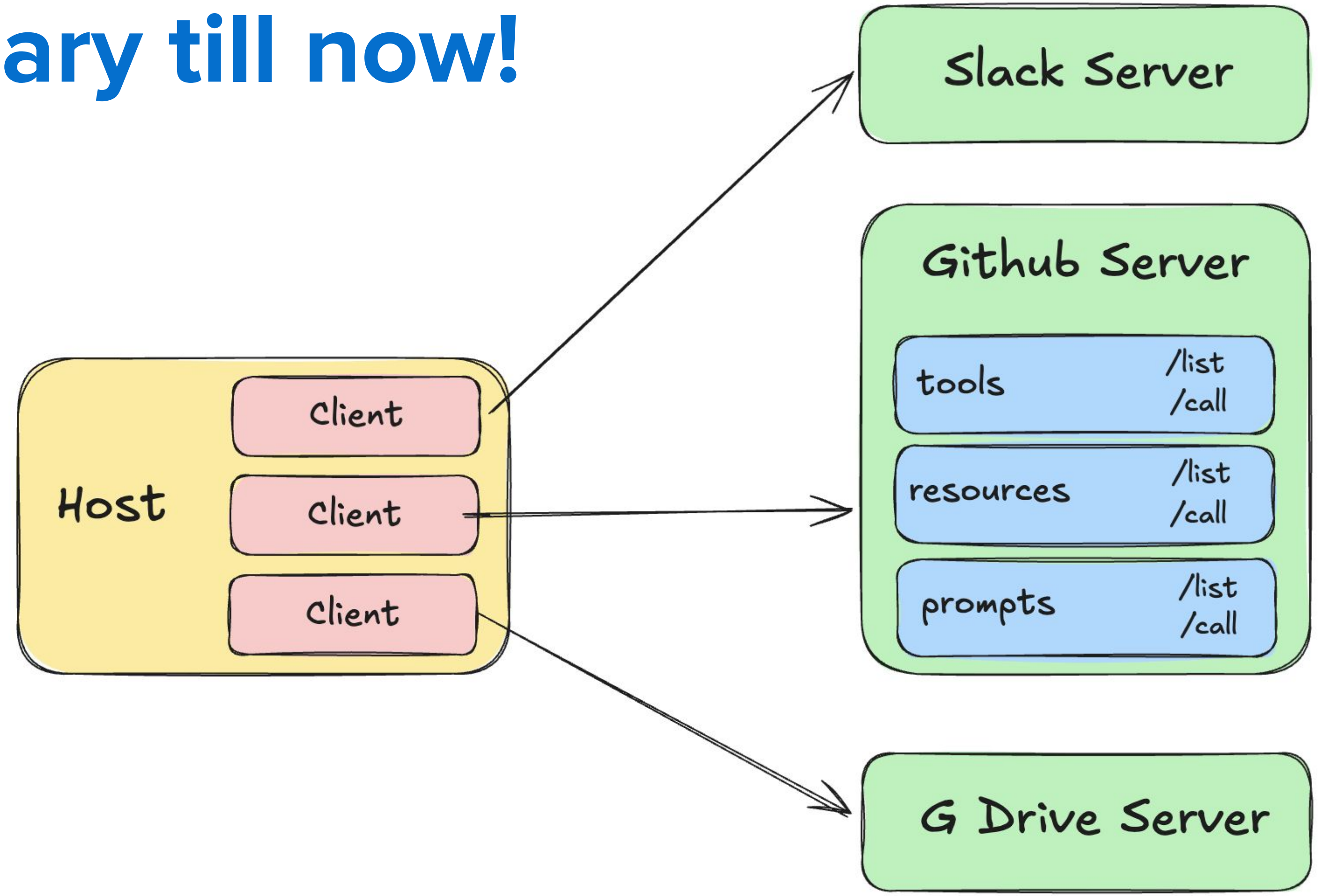
```
{
  "name": "issue_report_prompt",

  "description": "Write clear, detailed GitHub issues",

  "messages": [{
         "role": "system",

         "content": "Always include: Title, Steps to
Reproduce, Expected, Actual, Environment"}
  }]
}
```

```
Title: Bug – Login Button Not Working

**Steps to Reproduce:**
1. Open the login page
2. Enter valid credentials
3. Click the Login button

**Expected Behavior:**
User should be logged in and redirected to dashboard.

**Actual Behavior:**
Nothing happens when clicking the Login button.

**Environment:**
Chrome 121, macOS 14.2
```

Primitives - Standard Operations

- **Tools**
  - `tools/list` → Client asks the Server: *“What tools do you provide?”*
  - `tools/call` → Client tells the Server: *“Please run this tool with these arguments.”*
- **Resources**
  - `resources/list` → Client asks: *“What resources are available?”*
  - `resources/read` → Client says: *“Give me the content of this resource.”*
  - `resources/subscribe` / `unsubscribe` → Client subscribes or unsubscribes from updates.
- **Prompts**
  - `prompts/list` → Client asks: *“What prompt templates do you provide?”*
  - `prompts/get` → Client fetches a specific prompt template.

Summary till now!



MCP Data Layer

Data Layer - The data layer is the language and grammar of the MCP ecosystem that everyone agrees upon to communicate.

In MCP, **JSON RPC 2.0** serves as the foundation of the data layer

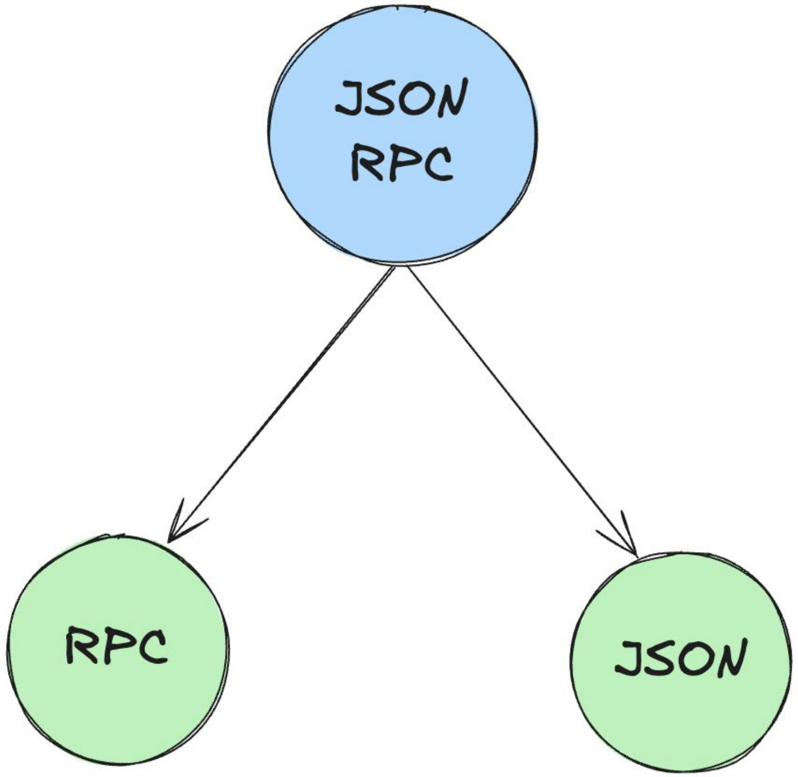
JSON RPC 2.0

JSON-RPC stands for **JavaScript Object Notation – Remote Procedure Call**.

A **Remote Procedure Call (RPC)** allows a program to execute a function on another computer as if it were local, hiding the details of network communication and data transfer. This abstraction makes it easier to build distributed applications.

Instead of writing add(2, 3) locally, you send a request to the server saying "please run add with parameters 2 and 3."

JSON RPC 2.0



JSON-RPC combines the concept of **Remote Procedure Calls** with the simplicity of **JSON**, allowing developers to structure RPC requests and responses in a standardized JSON format.

JSON RPC 2.0

Request Structure

```
{
  "jsonrpc": "2.0",
  "method": "add",
  "params": [2, 3],
  "id": 1
}
```

Response Structure

```
{
  "jsonrpc": "2.0",
  "result": 5,
  "id": 1
}
```

```
{
  "jsonrpc": "2.0",
  "error": { "code": -32601, "message": "Method not found" },
  "id": 1
}
```

Example Communication

Initial handshake

Calling a tool

Batching

Notification

Error Handling

Why JSON RPC for Data Layer?

It’s lightweight

Supports bi-directional communication

It is transport-agnostic

Supports batching

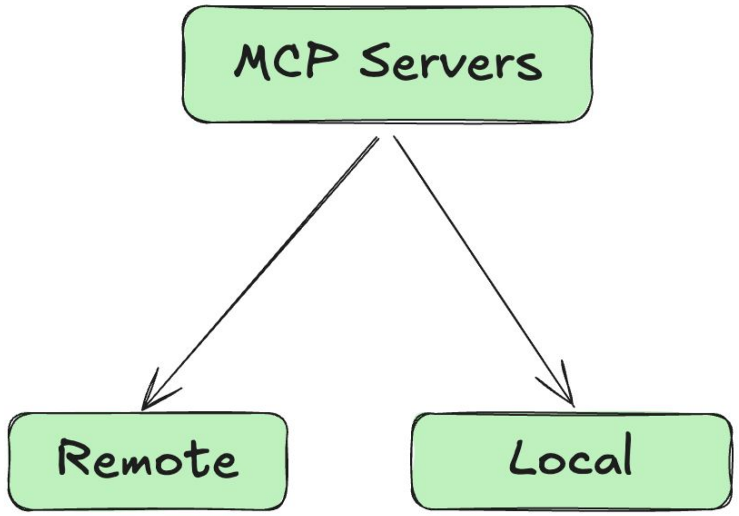
Supports notification

MCP Transport Layer

The **Transport Layer** is the mechanism that moves JSON-RPC messages between the Client and Server.

The choice of transport depends on the type of server.

MCP - Types of Server



A **local server** is a program running on your own computer.

-> STDIO

A **remote server** is a program running on another computer (somewhere else on the network or internet) that you connect to over a network.

-> HTTP/SSE

Local Servers - STDIO

STDIO refers to the built-in streams every program has.

stdin (input the program reads)

stdout (output the program writes)

In MCP, these streams are used as transport layer between the client and server.

How does STDIO work?

The host launches the server as a subprocess on the same machine.

The host(client) writes JSON-RPC messages into the server’s STDIN

The server reads those messages, processes them, and writes back responses to it’s STDOUT

Benefits of the STDIO transport

Fast - data is passed directly between processes.

Secure - no open network port that could be attacked; communication is only local.

Simple → every language/runtime supports reading/writing from stdin/stdout. No extra libraries required.

Remote Servers - HTTP + SSE

Using HTTP allows the host to reach Servers running anywhere

Host sends JSON RPC requests using POST requests with a JSON payload

The transport supports standard HTTP auth methods(like API keys)

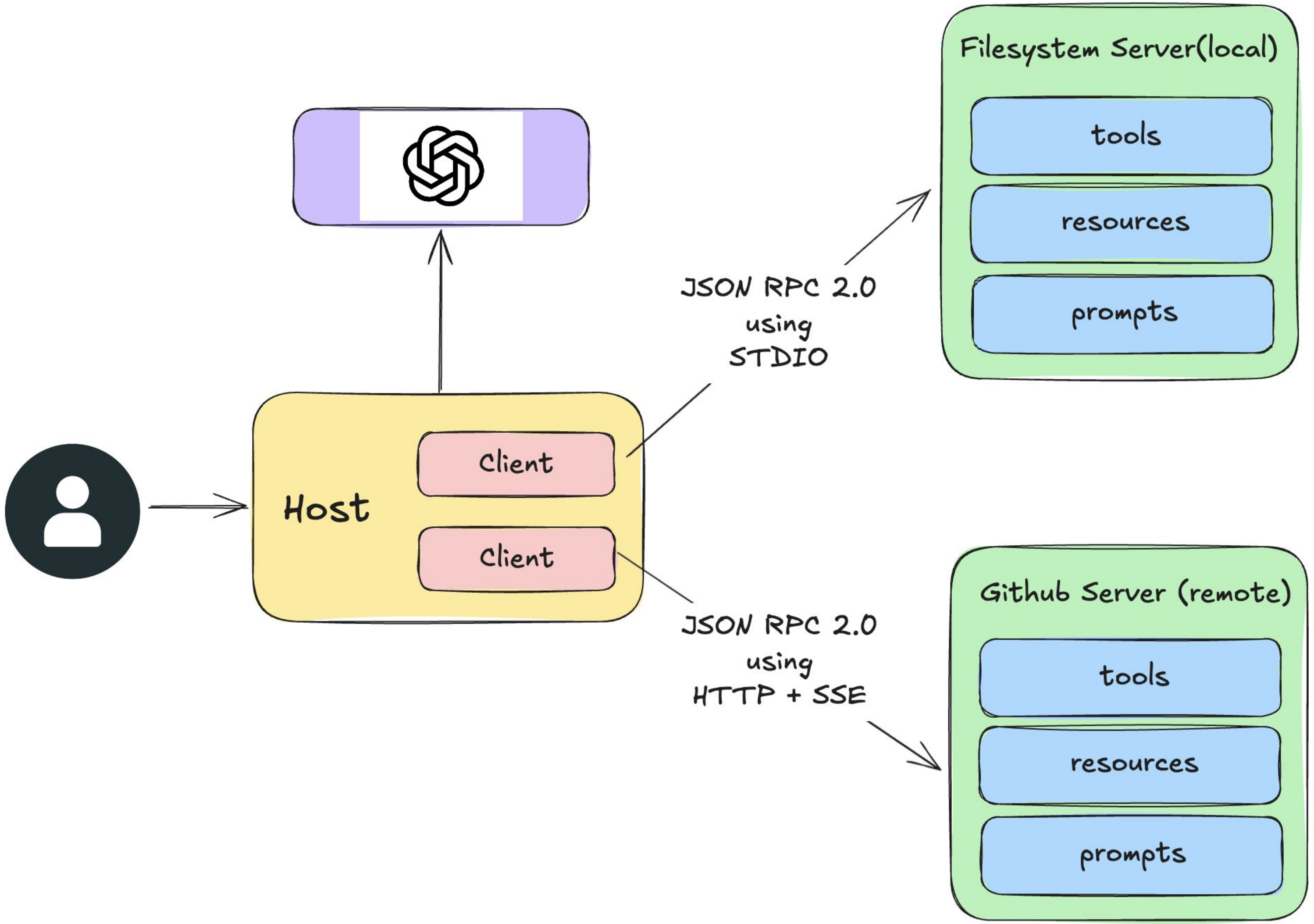
Remote Servers - HTTP + SSE

SSE stands for Server Sent Events, and it's an extension of HTTP

Using SSE the server sends multiple messages to client over a single open connection

Instead of sending one large JSON blob, the server can stream chunks of data as they are ready

Ideal for long running tasks or incremental updates


Filesystem Server(local)
tools
resources
prompts
JSON RPC 2.0
using
STDIO
Host
Client
Client
JSON RPC 2.0
using
HTTP + SSE
Github Server (remote)
tools
resources
prompts

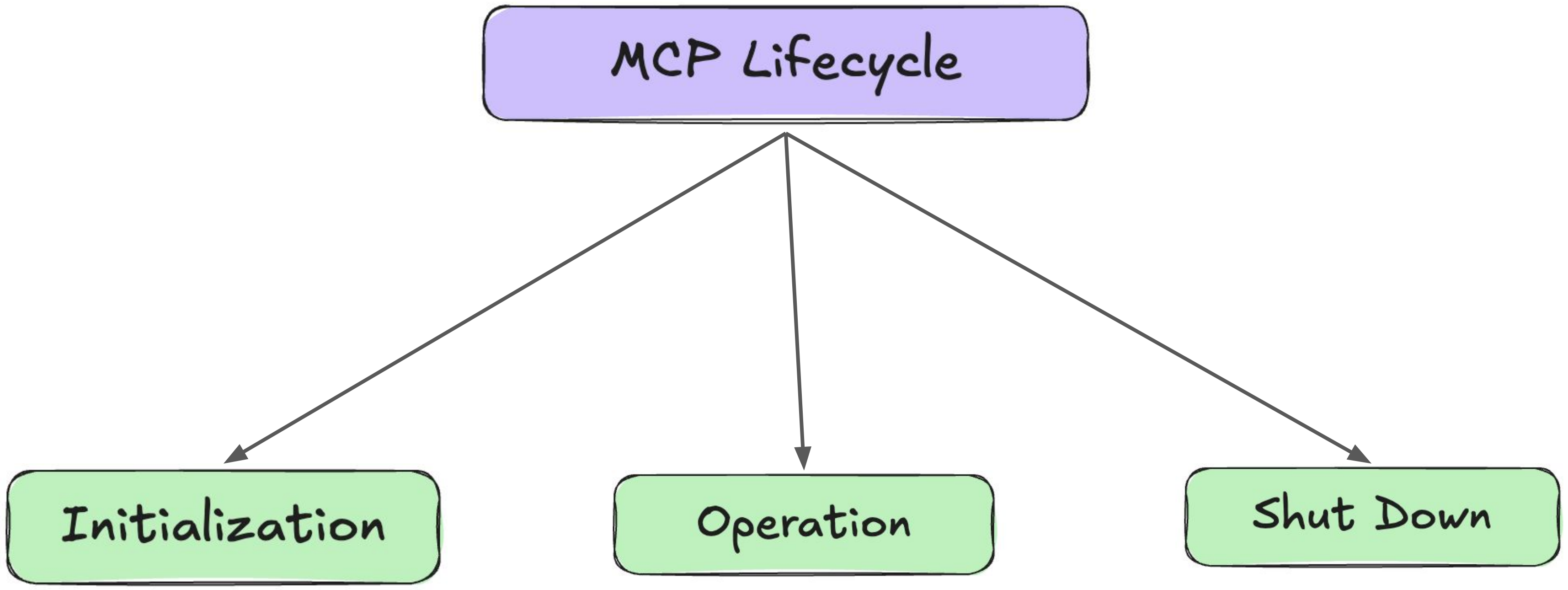
MCP

LIFECYCLE

What is MCP Lifecycle?

The **MCP Life Cycle** describes the **complete sequence of steps** that govern how a Host (client) and a Server establish, use, and end a connection during a session.

Stages of MCP Lifecycle



Initialization Phase

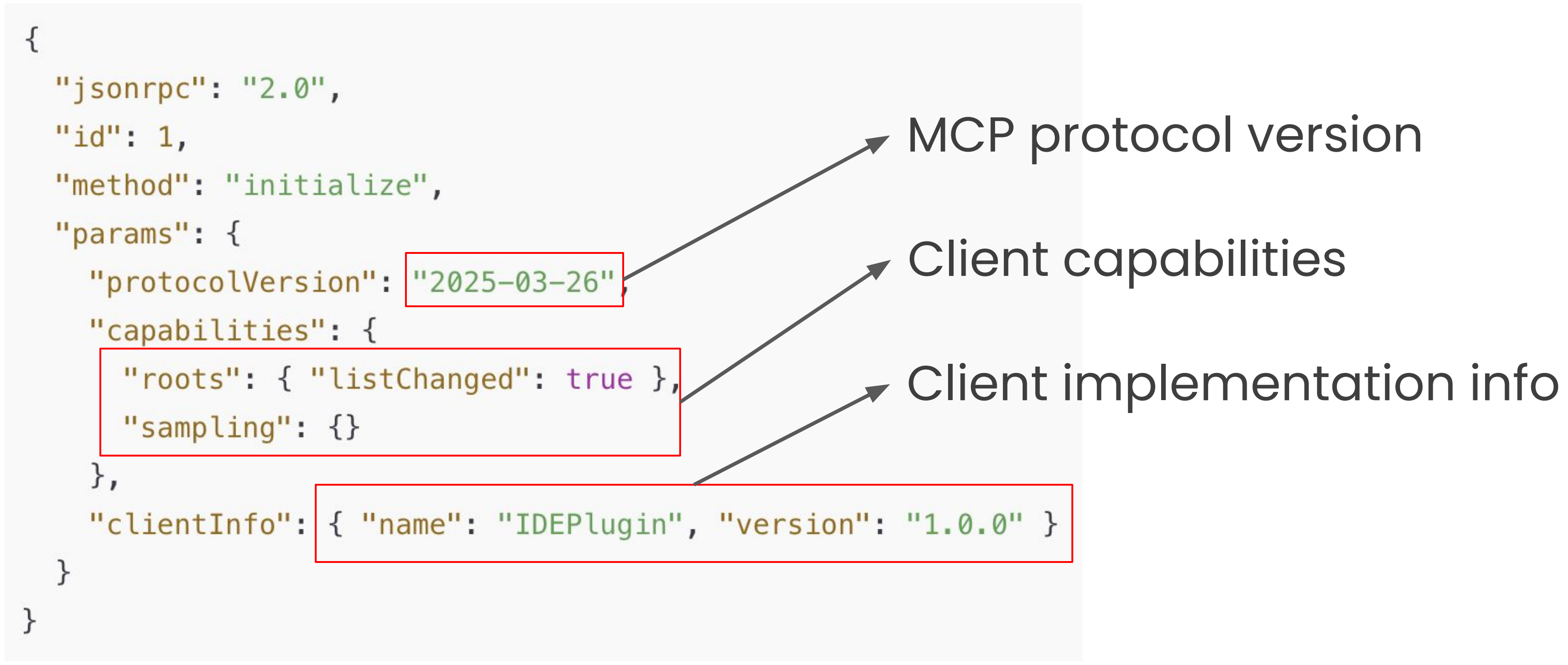
The initialization phase MUST be the first interaction between client and server.

Establish protocol version compatibility

Exchange and negotiate capabilities

Initialization Phase - Step 1

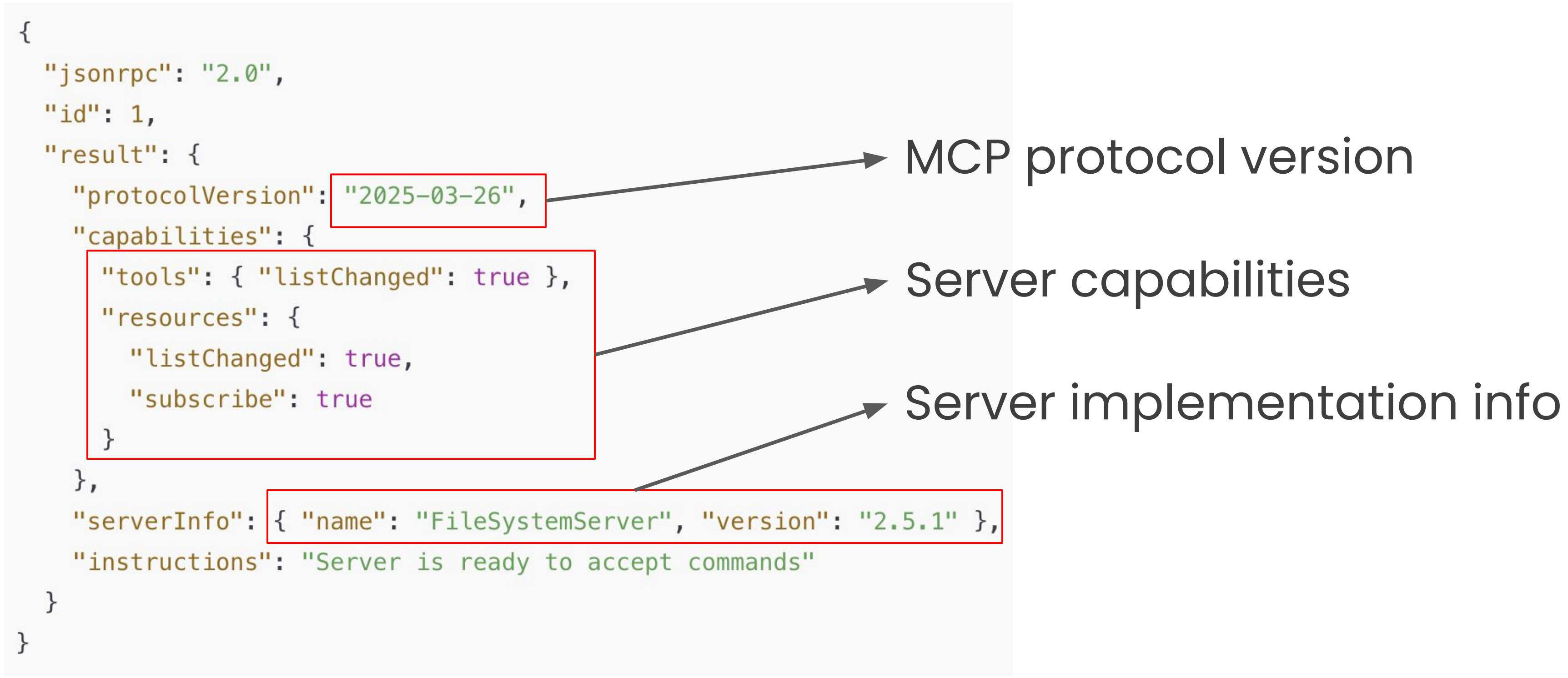
The Client sends a *initialize* request containing:



```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "roots": { "listChanged": true },
      "sampling": {}
    },
    "clientInfo": { "name": "IDEPlugin", "version": "1.0.0" }
  }
}
```

Initialization Phase - Step 2

The Server sends its own capabilities and info:



```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "tools": { "listChanged": true },
      "resources": {
        "listChanged": true,
        "subscribe": true
      }
    },
    "serverInfo": { "name": "FileSystemServer", "version": "2.5.1" },
    "instructions": "Server is ready to accept commands"
  }
}
```

Initialization Phase - Step 3

After successful initialization, the client MUST send an *initialized* notification to indicate it is ready to begin normal operations:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/initialized"
}
```

Now the Client and Server are connected

Important Rules

The client **SHOULD NOT** send requests other than **pings** before the server has responded to the *initialize* request.

The server **SHOULD NOT** send requests other than **pings** and **logging** before receiving the *initialized* notification.

Version Negotiation

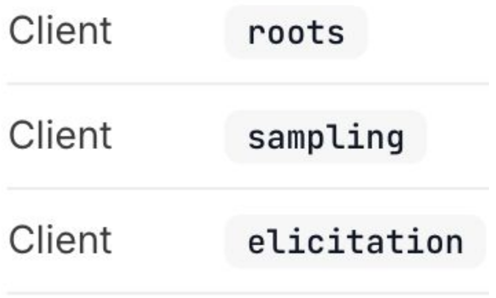
```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": {},
    "clientInfo": { "name": "IDEPlugin",
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2024-11-05",
    "capabilities": {},
    "serverInfo": { "name": "FileSystemServer"
  }
}
```

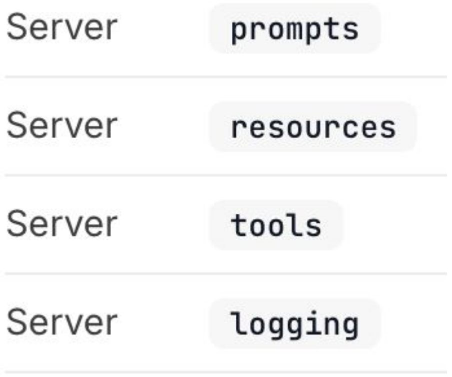
```
SUPPORTED_PROTOCOL_VERSIONS = ["2025-03-26", "2024-11-05"]
```

Capability Negotiation

Client and server capabilities establish which protocol features will be available during the session.

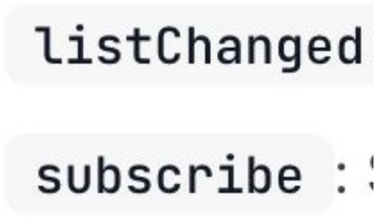


Client	`roots`
Client	`sampling`
Client	`elicitation`



Server	`prompts`
Server	`resources`
Server	`tools`
Server	`logging`

<u>Sub-capabilities</u>



```
listChanged
subscribe :
```

Operation Phase

During the operation phase, the client and server exchange messages according to the negotiated capabilities.

Respect the negotiated protocol version

Only use capabilities that were successfully negotiated

Capability Discovery

```
{ "jsonrpc": "2.0", "id": 2, "method": "tools/list" }
```

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "tools": [
      { "name": "listRepos", "description": "List user/org repositories" },
      { "name": "getFile", "description": "Read a file from a repo (path@ref)" },
      { "name": "searchCode", "description": "Search code across repos" },
      { "name": "createIssue", "description": "Open a GitHub issue" },
      { "name": "listPRs", "description": "List pull requests for a repo" }
    ]
  }
}
```

Tool Calling

```json
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "getFile",
    "arguments": {
      "owner": "campusx-official",
      "repo": "mcp-examples",
      "path": "README.md",
      "ref": "main"
    }
  }
}
```

```json
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "content": "# MCP Examples\nThis repo contains MCP examples…",
    "encoding": "utf-8",
    "sha": "f3c0…"
  }
}
```

Shutdown Phase

One side (typically the **client**) initiates shutdown

No special JSON RPC shutdown message is defined

Transport layer is responsible for signaling termination

Shutdown in STDIO

Client-initiated shutdown (SHOULD):

- Close input stream to the child process (server)
- Wait for server to exit
- Send **SIGTERM** if server does not exit in time
- Send **SIGKILL** if still unresponsive

Client-initiated shutdown (SHOULD):

- Close output stream to the client
- Exit process

Shutdown in HTTP

Client-initiated shutdown (common case):

The client (Host) **closes the HTTP connection(s)** it opened to the Server.

Server-initiated shutdown (possible):

The server may close the connection from its side

The client must be prepared to detect a dropped connection and handle it (e.g., reconnect if appropriate).

SPECIAL

CASES

Pings

Ping is a lightweight request/response method defined in MCP.

Purpose: to check whether the other side (Host or Server) is still alive and the connection is responsive.

```
{ "jsonrpc": "2.0", "id": 42, "method": "ping" }
```

```
{ "jsonrpc": "2.0", "id": 42, "result": {} }
```

When is Ping used?

Useful for checking if the other side is up before full initialize.

If there’s no activity for a while, a client may send periodic pings. Prevents the connection from being dropped silently by the OS, proxies, or firewalls.

Error Handling

Error handling in MCP is how the Host (client) and Server signal that **something went wrong** with a request

MCP inherits JSON RPC's standard error object format

Causes of Error

Unsupported or mismatched protocol version.

Calling a method for a capability that wasn't negotiated.

Invalid arguments to a tool

Internal server failure while processing a request.

Timeout exceeded → client cancels request.

Malformed JSON-RPC messages.

Error Object Structure

code → integer code that categorizes the error.

message → short human-readable explanation.

data → (optional) extra structured data for debugging or context.

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "error": {
    "code": -32601,
    "message": "Method not found",
    "data": { "extra": "optional info" }
  }
}
```

Common Error Codes

Error Code	Name	Meaning	Example
-32601	Method not found	Called a method that doesn't exist or wasn't advertised	Host calls `prompts/list` but server never advertised `prompts`.
-32602	Invalid params	Request sent with wrong or missing parameters	Tool expects `{ "path": "…" }`, but client sends `{ "file": "…" }`.
-32600	Invalid Request	Malformed JSON-RPC request structure	Missing required fields like `jsonrpc` or `method`.
-32700	Parse error	JSON could not be parsed	Request body is not valid JSON.
-32000 and above	Server-defined errors	Custom errors defined by the server (implementation-specific)	Authentication failure, rate limit exceeded, quota errors, internal issues.

Timeout

Timeout is about ensuring requests don’t hang forever

Purpose

Protects against unresponsive or overloaded servers.

Ensures resources (memory, CPU) aren’t held indefinitely.

Gives the user feedback instead of waiting forever.

How Timeout Works

SDKs let Client sets a per-request timeout (e.g., 30s).

If the deadline passes with no result → client triggers a timeout.

Client then sends a **cancellation notification** to tell the server to stop.

The server **must stop processing** that request and **not return a result**.

Cancellation

```json
{
  "jsonrpc": "2.0",
  "id": "7",
  "method": "tools/call",
  "params": {
    "name": "searchCode",
    "arguments": { "query": "MCP" },
    "_meta": { "progressToken": "tok-7" }
  }
}
```

```json
{
  "jsonrpc": "2.0",
  "method": "notifications/cancelled",
  "params": { "requestId": "7", "reason": "Timeout exceeded (30s)" }
}
```

Progress Notification

Purpose: Let the client know a long-running request is still making progress.

Client includes a progressToken in the request’s _meta.

Server can then send notifications/progress updates while working.

Progress Notification

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "tools/call",
  "params": {
    "name": "searchCode",
    "arguments": { "query": "MCP lifecycle" },
    "_meta": { "progressToken": "tok-7" }
  }
}
```

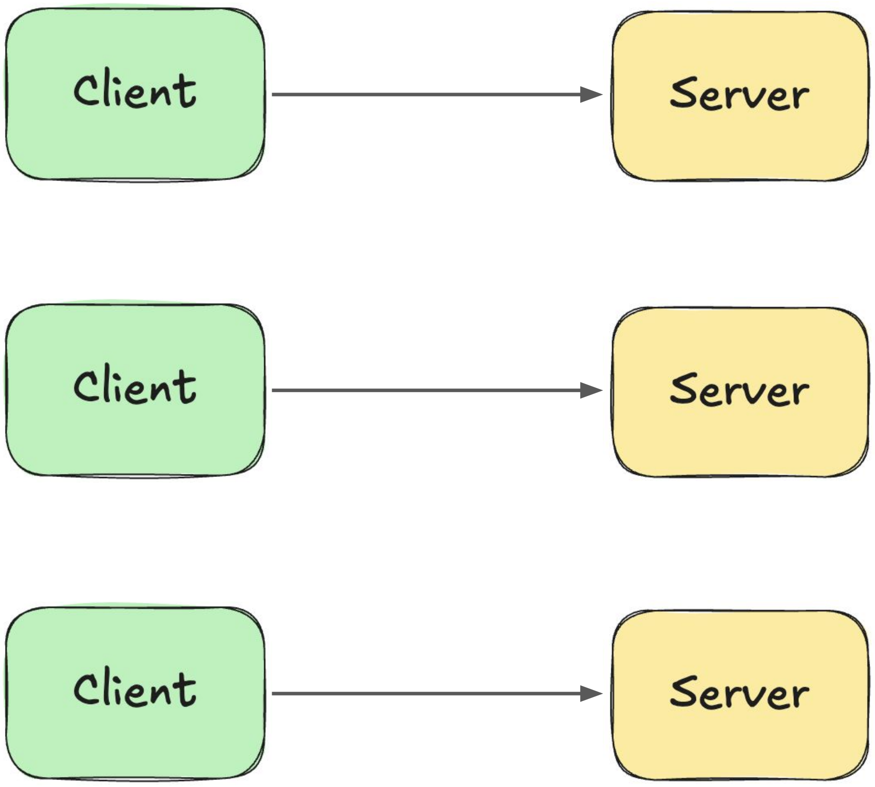
```
{
  "jsonrpc": "2.0",
  "method": "notifications/progress",
  "params": {
    "progressToken": "tok-7",
    "progress": 60,
    "total": 100,
    "message": "Searching 600 of 1000 files"
  }
}
```

THE

HOW

Strategy

Ready made client like **Claude Desktop**



Already Existing servers like **Google Drive**

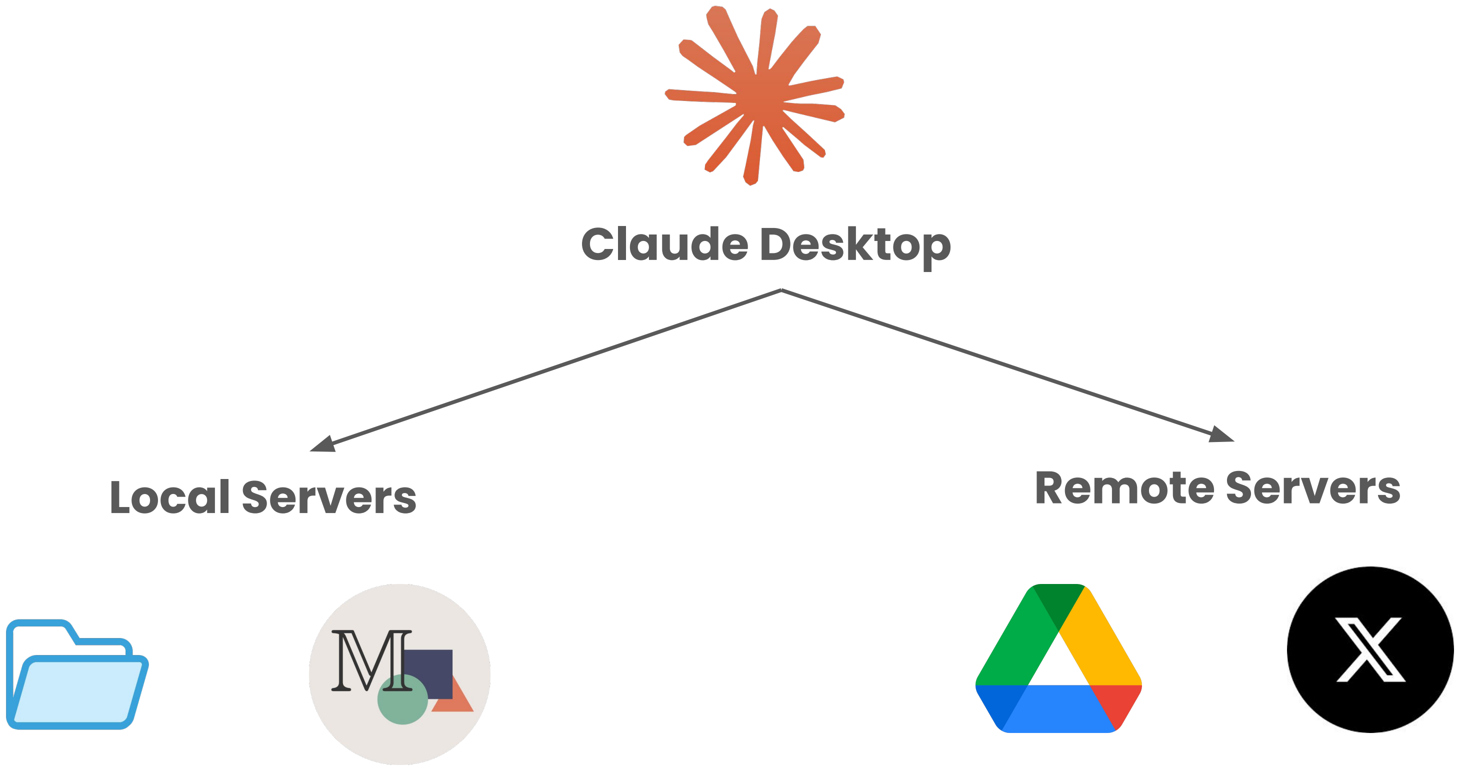
Claude Desktop

Our own **MCP Server**

Our own **MCP Client**

Our own **MCP Server**

Plan of Action



Type of Connections

Using Config File

Using Connectors

What are Connectors?

A **Connector** is a built-in feature that links Claude to MCP servers automatically, without the need for manual setup or configuration.

Most Claude Desktop users are **non-technical end-users** who just want Claude to “talk” to their apps (Notion, Google Drive, GitHub, Slack, etc.).

They don’t want to run servers, edit JSON, or worry about transports.

The **Connector system** wraps an MCP server behind the scenes and handles authentication via OAuth (sign-in with Google, GitHub, etc.).

This keeps things **easy, safe, and consistent**.

Think of Connectors as the “App Store” for MCP servers — user-friendly, click-based, pre-curated.

Why not use Connectors always?

Connectors Are **Curated & Managed**

Connectors are **officially built, hosted, and maintained** by Anthropic

They come with **OAuth login flows**, managed security, rate-limits, and guaranteed stability.

If every MCP server were required to be a Connector, it would mean Anthropic has to **review, host, and secure every possible server** — which doesn't scale

MCP is an **Open Standard**

MCP is designed so *anyone* can write a server

Forcing everything through Connectors would **close the ecosystem** and make you dependent on Anthropic to approve or publish servers.