Model Context Protocol

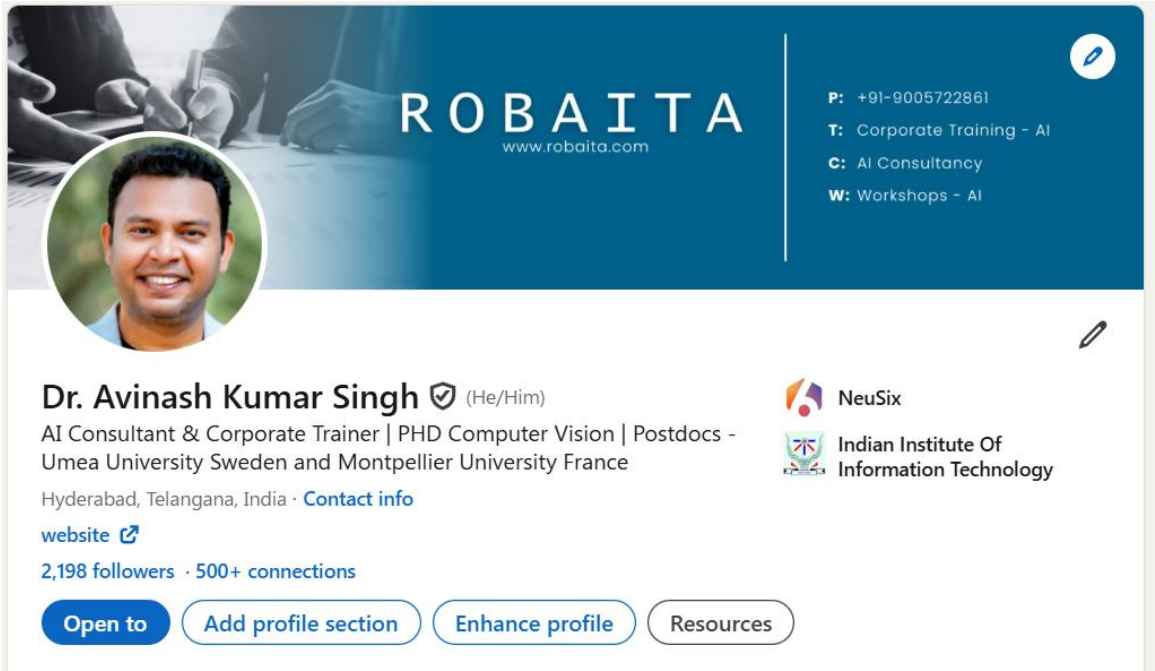
USB-C for AI applications

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Dr. Avinash Kumar Singh

- **Possess** 15+ years of **hands-on expertise** in Machine Learning, Computer Vision, NLP, IoT, Robotics, and Generative AI.
- **Founded** Robaita—an initiative **empowering** individuals and organizations to **build, educate, and implement** AI solutions.
- **Earned** a Ph.D. in Human-Robot Interaction from IIIT Allahabad in 2016.
- **Received** postdoctoral fellowships at Umeå University, Sweden (2020) and Montpellier University, France (2021).
- **Authored** 30+ research papers in **high-impact** SCI journals and international conferences.
- Unlearning, learning, making mistakes …



https://www.linkedin.com/in/dr-avinash-kumar-singh-2a570a31/

Foundations of Model Context Protocol

https://modelcontextprotocol.io/introduction

Why MCP? – The "M × N Integration Problem"

Problem: M × N Integration Overhead

In multi-agent LLM systems:

- M = # of tools or APIs
- N = # of agents (models, modules)
- Without standardization, each agent must learn to interact with every tool ⇒
- M × N custom integrations

- This leads to:
  - Redundant code
  - Fragile system architecture
  - High maintenance overhead
- Example: A system with
  - 4 tools (search, summarizer, calculator, database)
  - 3 agents (planner, reasoner, retriever)
  - → 3 × 4 = 12 unique tool-agent interfaces

MCP as the Solution

Model Context Protocol (MCP) standardizes:

- How tools expose capabilities
- How agents invoke tools
- Shared context, state, memory

This reduces integration from M × N to M + N.

✅ **Benefits:**

- Simplified scaling
- Modularity and reuse
- Easier debugging & logging
- Production-grade orchestration

What is MCP? – The USB-C for AI Context

Model Context Protocol (MCP) is a specification and communication standard that allows AI agents, tools, and orchestrators to share and understand context, state, memory, and intent consistently—just like USB-C standardizes hardware connectivity.

Why "USB-C for AI"?

- Just as USB-C unifies chargers and data cables, MCP unifies how agents exchange context.
- Promotes interoperability across different LLMs, tools, and agent frameworks (e.g., CrewAI, LangGraph).

Core Components

- Context: User goal, history, metadata
- Memory: Past interactions or retrieved docs
- State: Current task progress
- Intent: Agent goal, next action

MCP - Introduction

- The **Model Context Protocol (MCP)** was introduced in **November 2024** by **Anthropic** as an **open standard** to help AI assistants connect seamlessly with tools like content systems, business apps, and developer platforms.
- Earlier solutions like **OpenAI's function calling** and **ChatGPT plugins** solved similar problems but were tied to specific vendors and required custom logic.
- To solve this, **MCP was designed as a universal connector**—just like USB-C.
  - Uses proven ideas from the **Language Server Protocol (LSP)**
  - Communicates over **JSON-RPC 2.0**
  - Comes with **SDKs** in popular languages like Python, TypeScript, Java, and C#

MCP helps eliminate information silos and makes AI systems easier to build, maintain, and extend—no matter which tools or platforms you're working with.

Source: https://modelcontextprotocol.io/introduction

MCP - Introduction

ChatGPT Plugins (launched in 2023 by OpenAI) allow ChatGPT to **access external tools and APIs in real time**. These plugins extend the model's capabilities beyond its training data—letting it retrieve up-to-date information, perform actions like booking a flight, or fetch data from third-party services.

Example: Using the Expedia Plugin to Book a Hotel

User Prompt: "Find me a 4-star hotel in Goa from June 20 to June 23 for under ₹7,000 per night."

What ChatGPT Does (with Plugin Enabled):

- Uses the Expedia plugin to access real-time hotel listings.
- Sends the query to Expedia's API securely.
- Parses the response and shows options.

```
Here are 3 hotel options in Goa:
1. Hotel Serenity – ₹6,800/night – Near Baga Beach
2. La Calypso Resort – ₹6,200/night – Includes breakfast
3. Acron Waterfront – ₹6,950/night – River view

Would you like to book one of these?
```

Why Plugins Matter

- **Sandboxed:** Only enabled plugins can be called, ensuring safety.
- **Pre-registered:** Plugins expose metadata and OpenAPI specs so the model knows how to use them.
- **Interactive:** Allow dynamic decision-making by the model with real-world data.

MCP - Introduction

OpenAI Function calling is a feature introduced in 2023, that lets GPT models call the backend functions by returning structured data (like JSON), instead of just plain text. This allows the model to interact with real-world tools, fetch data, perform actions, or trigger workflows.

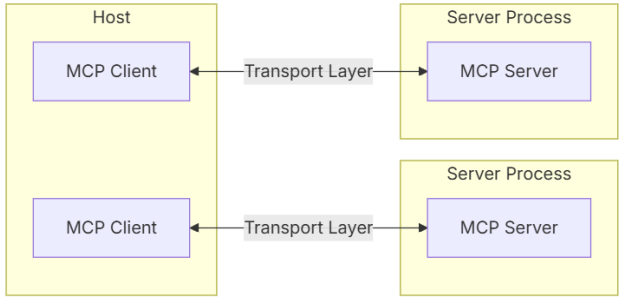
How it works?

- Define a function schema (name, description, and parameters).
- User asks a question, e.g., “What’s the weather in Delhi?”
- GPT model detects that a function should be called.
- It returns a JSON object with the function name and arguments.
- Systems run that function on the backend (e.g., call a weather API).
- Send the result back to GPT, and it generates the final response.

Core Architecture – Host, Clients, Servers

MCP follows a client-server architecture where:

- Hosts are LLM applications (like Claude Desktop or IDEs) that initiate connections
- Clients maintain 1:1 connections with servers, inside the host application
- Servers provide context, tools, and prompts to clients



Example: A chat app needs to fetch document content:

- Host starts MCP session.
- Client sends resources/list → Server returns available document URIs.
- Client selects a resource, fetches via resources/get, and passes content to LLM.
- LLM responds enriched with real data.

Core Architecture – Host, Clients, Servers

Protocol Layer

Defines the **rules of communication** between clients and servers using the **JSON-RPC 2.0** standard.

Key Responsibilities:

- Specifies how requests and responses are structured
- Manages method names, parameters, and result formatting
- Ensures consistency across different clients and servers

A request to list resources

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

Response

```
{
  "jsonrpc": "2.0",
  "result": [
    { "uri": "doc://manual.pdf", "title": "User Manual" }
  ],
  "id": 1
}
```

Core Architecture – Host, Clients, Servers

Transport Layer

Specifies how messages are delivered between clients and servers.

Supported Options:

- HTTP(S) (most common in production)
- WebSocket (for streaming use cases)
- stdin/stdout (local CLI apps)

Type	Purpose	Example Use Case
Request	Initiates an action	resources/list, tools/call
Response	Returns result of a request	{ "result": [...] }
Notification	Fire-and-forget (no response expected)	Log updates or state change alerts

Example:

- A desktop LLM app (Host) sends a resources/get request over HTTP POST to a remote document server.
- Another use case might involve a CLI tool sending messages over stdin/stdout locally.

```
curl -X POST http://server/api \
     -H "Content-Type: application/json" \
     -d '{ "jsonrpc": "2.0", "method": "tools/list", "id": 2 }'
```

Core Architecture – Host, Clients, Servers

Message Types

MCP supports **3 types of JSON-RPC messages**:

Type	Purpose	Example Use Case
Request	Initiates an action	resources/list, tools/call
Response	Returns result of a request	{ "result": [...] }
Notification	Fire-and-forget (no response expected)	Log updates or state change alerts

Request and Response Pair

```
# Request
{ "jsonrpc": "2.0", "method": "tools/call", "params": { "name": "weather", "args": {"city": "Delhi"} }, "id": 5 }

# Response
{ "jsonrpc": "2.0", "result": "Temperature in Delhi is 38°C", "id": 5 }
```

Notification(No Reply Expected)

```
# Notification
{ "jsonrpc": "2.0", "method": "log/event", "params": { "message": "Tool called" } }
```

MCP Primitives: Resources, Prompts, Tools, Sampling, Roots, Transports

MCP defines a shared language for AI agents to coordinate. The core primitives are:

Resources

- Objects passed between agents (e.g., text, code, JSON).
- **Example**: A PDF document or a function definition.

```
resource = Resource(id="doc123", type="text/plain",
                    content="Summarize this contract.")
```

Prompts

- Structured messages or instructions given to agents.
- **Example**: “Translate this paragraph to French.”

```
prompt = Prompt(role="user",
    content="Translate this into French: Hello world!")
```

Tools

- External functions or APIs that agents can call.
- **Example**: Weather API or SQL query tool.

```
tool = Tool(name="getWeather", description="Returns current weather",
                                parameters=["city"])
```

MCP Primitives: Resources, Prompts, Tools, Sampling, Roots, Transports

MCP defines a shared language for AI agents to coordinate. The core primitives are:

Sampling

- Strategy to control variability in LLM responses (e.g., top-k, temperature).
- **Example:** Use temperature=0.7 for creative writing, 0.0 for factual QA.

```
sampling = Sampling(temperature=0.7, top_k=40)
```

Roots

- Starting point or anchor for context in a multi-agent thread.
- **Example**: A user request like “Generate marketing copy” as the root of a conversation tree.

```
root = Root(task="GenerateAdCopy", user_id="U456")
```

Transports

- Mechanisms for agents to communicate (HTTP, WebSocket, etc.).
- **Example**: Agent A sends JSON to Agent B over gRPC.

```
transport = Transport(type="http",
        endpoint="https://agent-b.com/receive")
```

MCP Primitives: Resources, Prompts, Tools, Sampling, Roots, Transports

Resources represent any kind of data that an MCP server wants to make available to clients.

This can include:

- File contents
- Database records
- API responses
- Live system data
- Screenshots and images
- Log files
- And more

Each resource is identified by a unique URI and can contain either text or binary data.

MCP Primitives: Resources, Prompts, Tools, Sampling, Roots, Transports

Resource types: Resources can contain two types of content:

Text resources

Text resources contain UTF-8 encoded text data. These are suitable for:

- Source code
- Configuration files
- Log files
- JSON/XML data
- Plain text

Binary resources

Binary resources contain raw binary data encoded in base64. These are suitable for:

- Images
- PDFs
- Audio files
- Video files
- Other non-text formats

MCP Primitives: Resources, Prompts, Tools, Sampling, Roots, Transports

Resource discovery

Clients can discover available resources through two main methods:

Direct resources

Servers expose a list of concrete resources via the resources/list endpoint. Each resource includes:

```
{
  uri: string;           // Unique identifier for the resource
  name: string;          // Human-readable name
  description?: string;  // Optional description
  mimeType?: string;     // Optional MIME type
  size?: number;         // Optional size in bytes
}
```

Resource templates

For dynamic resources, servers can expose URI templates that clients can use to construct valid resource URIs:

```
{
  uriTemplate: string;   // URI template following RFC 6570
  name: string;          // Human-readable name for this type
  description?: string;  // Optional description
  mimeType?: string;     // Optional MIME type for all matching resources
}
```

MCP Primitives: Resources, **Prompts**, Tools, Sampling, Roots, Transports

Prompts in MCP are predefined templates that can:

- Accept dynamic arguments
- Include context from resources
- Chain multiple interactions
- Guide specific workflows
- Surface as UI elements (like slash commands)

Prompt Structure

```
{
  name: string;              // Unique identifier for the prompt
  description?: string;      // Human-readable description
  arguments?: [              // Optional list of arguments
    {
      name: string;          // Argument identifier
      description?: string;  // Argument description
      required?: boolean;    // Whether argument is required
    }
  ]
}
```

MCP Primitives: Resources, Prompts, **Tools**, Sampling, Roots, Transports

Tools in MCP allow servers to expose executable functions that can be invoked by clients and used by LLMs to perform actions.

Key aspects of tools include:

- **Discovery**: Clients can list available tools through the tools/list endpoint
- **Invocation**: Tools are called using the tools/call endpoint, where servers perform the requested operation and return results
- **Flexibility**: Tools can range from simple calculations to complex API interactions

Like resources, tools are identified by unique names and can include descriptions to guide their usage. However, unlike resources, tools represent dynamic operations that can modify state or interact with external systems.

MCP Lifecycle: Initialize → Operate → Shutdown

The Model Context Protocol (MCP) lifecycle defines how an AI system initializes, operates, and shuts down with consistent context and coordination across agents and tools.

1 Initialize

Step 1: Client → initialize request

Includes protocol version & supported features

Example: Client: “I support JSON-RPC 2.0 and syntax highlighting”

Step 2: Server → Response

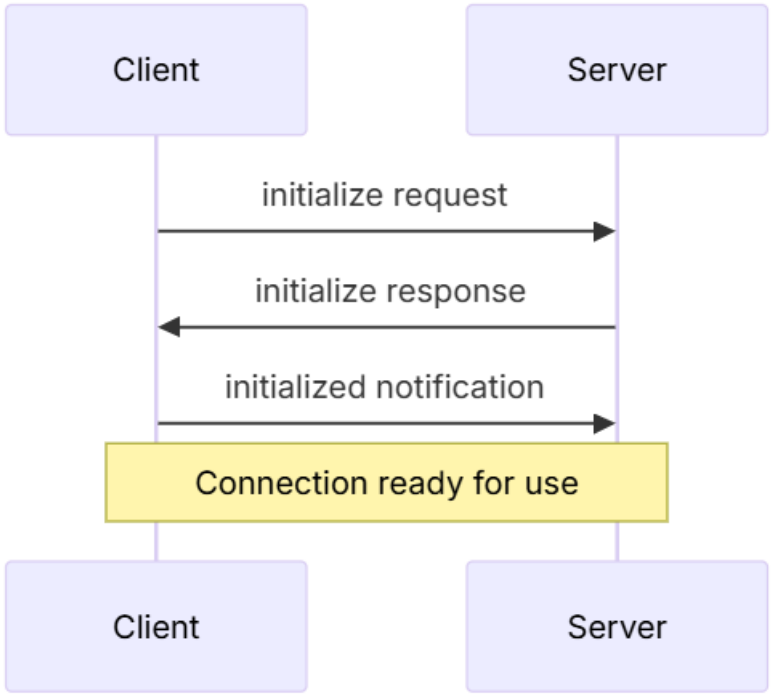
Confirms protocol version & its capabilities

Example: Server: “I support JSON-RPC 2.0 and hover info”

Step 3: Client → initialized notification

Acknowledges server response

Result: Connection is ready for message exchange

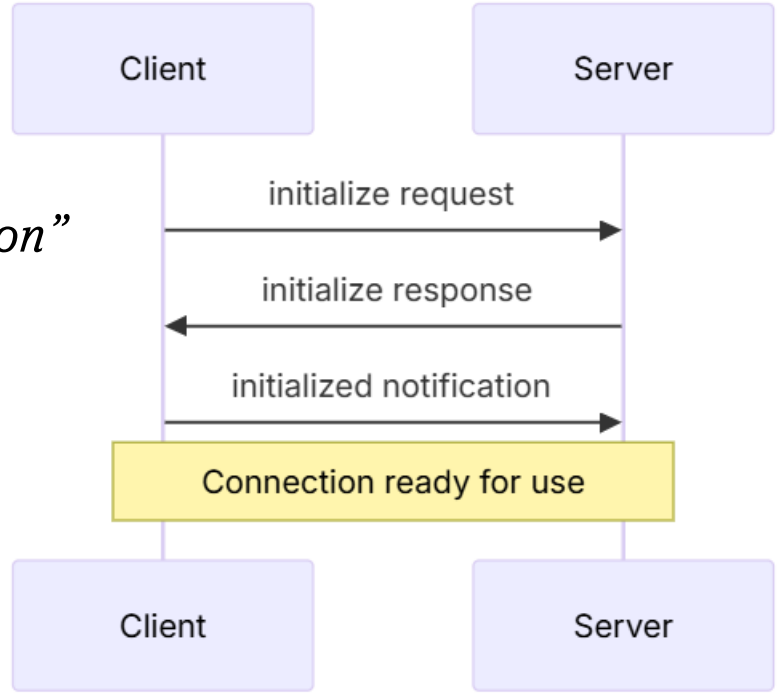


MCP Lifecycle: Initialize → Operate → Shutdown

The Model Context Protocol (MCP) lifecycle defines how an AI system initializes, operates, and shuts down with consistent context and coordination across agents and tools.

2 Operate: Message Exchange

- **Request-Response**
  ↔ Bi-directional messages expecting replies
  e.g., Client requests: "Get symbol info", Server responds: "It's a function"
- **Notifications**
  🔁 One-way messages without responses
  e.g., Client sends: "Document saved" (no reply expected)



MCP Lifecycle: Initialize → Operate → Shutdown

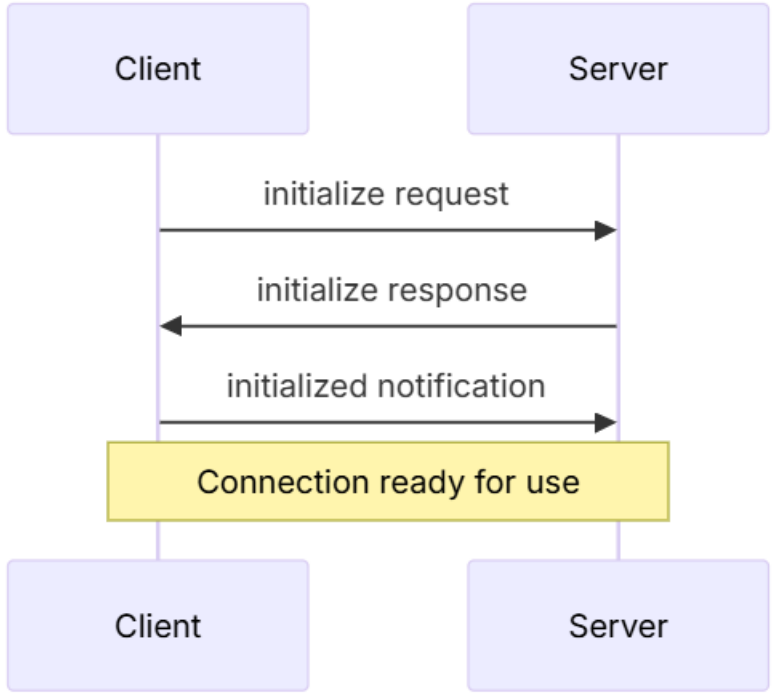
The Model Context Protocol (MCP) lifecycle defines how an AI system initializes, operates, and shuts down with consistent context and coordination across agents and tools.

3 Shutdown: Termination

- **Clean Shutdown:** Either party calls close()
- **Transport Disconnection:** Network or socket closes
- **Error Condition:** Unexpected crash or protocol error

Example: Client closes editor → triggers close()

→ server shuts down gracefully



MCP - Error Handling

Error handling in MCP ensures that unexpected failures are gracefully detected, communicated, and resolved between clients and servers.

```
enum ErrorCode {
  ParseError       = -32700, // Invalid JSON received
  InvalidRequest   = -32600, // Missing or malformed request
  MethodNotFound   = -32601, // Called method doesn't exist
  InvalidParams    = -32602, // Parameters are wrong or missing
  InternalError    = -32603, // Server-side exception
}
```

Benefits

- ✅ Prevents silent failures in agent communication
- 🧠 Helps diagnose integration issues
- 🔄 Enables reliable multi-agent orchestration
- 🔐 Ensures protocol robustness across tools and platforms

Error Propagation Mechanisms

- **Error Responses**
  ➤ Returned when a request fails (e.g., invalid method call)
- **Error Events**
  ➤ Emitted at the transport layer (e.g., connection timeout)
- **Protocol-Level Handlers**
  ➤ Catch and manage errors at the protocol level (e.g., fallback logic)

Thanks for your time