

MCP

MODEL CONTEXT PROTOCOL

1. Introducción al MCP.

El *Model Context Protocol (MCP)* se presentó públicamente el **25 de noviembre de 2024**, fue desarrollado e introducido por **Anthropic**, la empresa de IA que hace Claude. El Protocolo de Contexto de Modelos (MCP) es un estándar emergente diseñado para enriquecer a los Modelos de Lenguaje Grandes (LLM) con contexto externo, permitiéndoles interactuar con datos y servicios del mundo real. Su función principal es normalizar la forma en que las integraciones (servidores MCP) proveen información y capacidades a las aplicaciones cliente (como ChatGPT o Visual Studio Code) que utilizan LLMs.

El protocolo se estructura en torno a tres componentes fundamentales:

Herramientas (Tools), que son acciones que el LLM puede decidir ejecutar de forma autónoma; **Recursos (Resources)**, que son fuentes de datos controladas por el usuario o el cliente para ser injectadas en el contexto; y **Prompts**, que son plantillas de consulta predefinidas y parametrizables que el servidor genera para guiar al LLM en tareas complejas.

La interacción clave se produce entre el **cliente, el servidor MCP y el LLM**. El **cliente** media la comunicación, proveyendo al **LLM** el **contexto** necesario a través de su **system prompt** y ejecutando las llamadas que el modelo solicita. Aunque el protocolo está en una fase "embrionaria" y presenta algunas inconsistencias, evoluciona rápidamente y está siendo adoptado por clientes clave, abriendo la puerta a interacciones significativamente más potentes y sofisticadas con la IA, como la comunicación bidireccional y la delegación de tareas.

2. La Evolución de los LLM y el Desafío de la Integración

El desarrollo de los Modelos de Lenguaje Grandes (LLM) se puede dividir en dos fases, donde la segunda fase reveló una debilidad fundamental en el paradigma de conexión basado en API.

- **Fase 1: Interacción Directa:** Los LLM funcionaban como sistemas de entrada y salida de texto. Un usuario hacía una pregunta (input) y el modelo generaba una respuesta en texto (output).
- **Fase 2: Integración con Herramientas:** Los desarrolladores comenzaron a conectar los LLM a aplicaciones externas como Gmail, Discord o Slack,

permitiendo a los modelos realizar acciones en el mundo digital (enviar correos, mandar mensajes, etc.).

El problema surgió en esta segunda fase. La conexión se realizaba a través de las API de cada aplicación. Aunque muchas APIs siguen una estructura similar, cada empresa implementa sus propios detalles y particularidades. Al intentar conectar un agente de IA a múltiples aplicaciones para que funcionen de manera cohesiva, se generaba un "caos" interno en el código. Mantener este ecosistema se volvía "muy tormentoso", ya que cada aplicación hablaba su propio "lenguaje" y cualquier actualización o fallo en una API podía afectar a todo el sistema.

3. API vs MCP

La diferencia fundamental entre ambos patrones de diseño radica en su público y propósito objetivo.

- **API (Application Programming Interface):** Diseñadas para que **programadores humanos** conecten una aplicación con otra. Son el puente para la integración de software a nivel de desarrollo.
- **MCP (Model-Context Protocol):** Diseñadas para que **agentes de IA** se conecten a las aplicaciones. Es un "lenguaje" optimizado para que los modelos de lenguaje puedan entender y utilizar recursos externos de forma fácil y escalable.

3. Conceptos sobre LLMs

Ventana de contexto

La **ventana de contexto** de un LLM es su “**memoria a corto plazo**”: es la cantidad máxima de texto (tokens) que el modelo puede tener en cuenta a la vez para responder. Dentro de esa ventana se incluyen las instrucciones, el historial reciente de la conversación y el mensaje actual. Si la conversación o los documentos son más largos que ese límite, se recortan o resumen las partes más antiguas, por eso el modelo puede “olvidar” cosas que se dijeron hace mucho.

System prompt

El **system prompt** es un mensaje especial de instrucciones que se envía al modelo antes que los mensajes del usuario y que tiene prioridad sobre ellos.

Define el rol y el comportamiento del modelo (por ejemplo: “responde en español”, “actúa como profesor”, “sé conciso”, “no des consejos médicos”).

Normalmente el usuario no lo ve, pero siempre forma parte del contexto que el modelo usa para decidir cómo contestar.

Roles

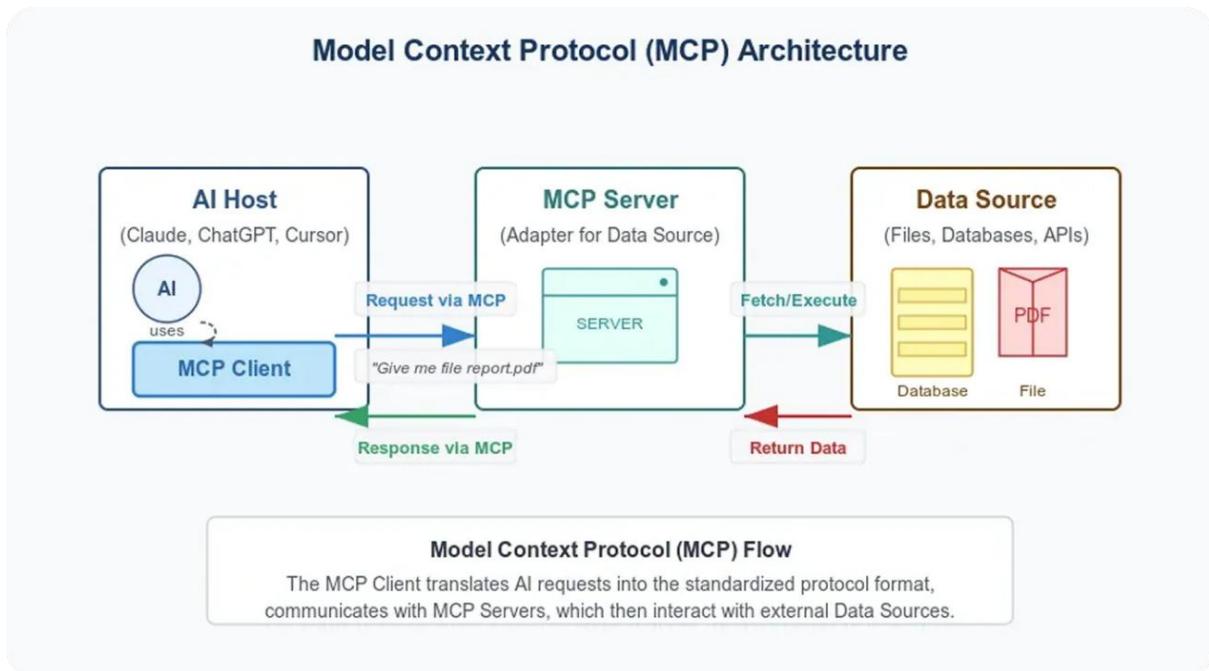
Indica **desde qué “voz” o “papel” se envía ese mensaje al LLM**.

El **cliente MCP**, cuando recibe esto en prompts/get, normalmente lo va a traducir 1:1 a mensajes para el modelo.

- "**system**" → reglas del juego (quién eres, cómo responder, límites, estilo).
- "**user**" → lo que escribe la persona (tú).
- "**assistant**" → lo que responde el modelo.

4. Arquitectura del MCP

El MCP es, en su esencia, "**un protocolo para añadir contexto a nuestros modelos**". Su objetivo es superar las limitaciones inherentes de los LLMs, que por defecto no tienen acceso a información en tiempo real o a sistemas privados como calendarios, bases de datos o sistemas de gestión de tareas. El MCP presenta los siguientes elementos como parte de su arquitectura.

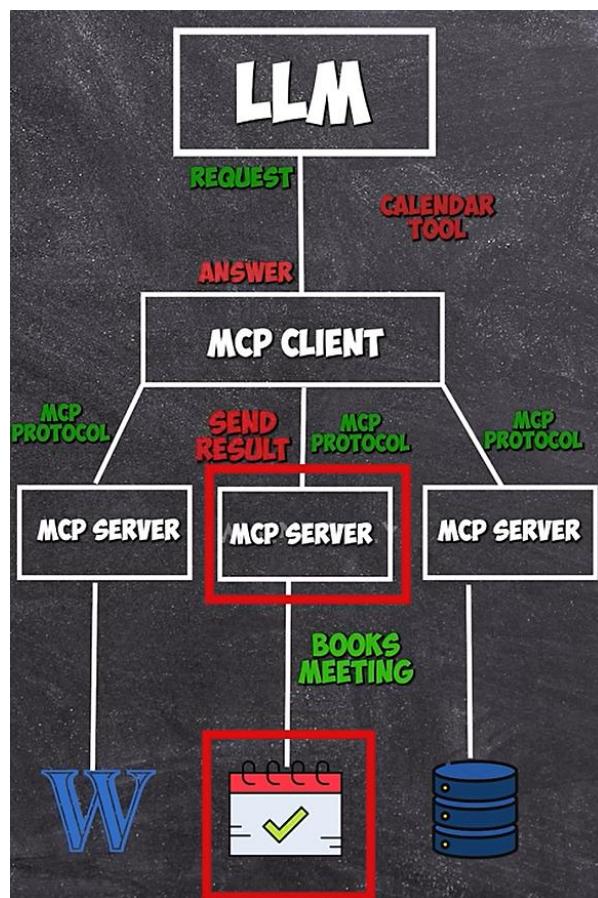


El ecosistema del MCP se compone de tres actores principales:

- **Apps Clientes:** Son las aplicaciones con las que el usuario final interactúa. Incluyen tanto asistentes de chat (ChatGPT, Claude) como editores de código y IDEs (Visual Studio Code, Cursor, IntelliJ).

- **Servidores MCP:** Son las integraciones que exponen datos y funcionalidades. Por ejemplo, podría haber un servidor MCP para Google Calendar, Jira, o una base de datos PostgreSQL.
- **LLM (Large Language Model):** Es el modelo de inteligencia artificial que procesa el lenguaje natural y, gracias al MCP, puede solicitar el uso de herramientas o analizar recursos para formular sus respuestas.

Un ejemplo práctico ilustra su potencial: si un usuario pregunta ¿Qué eventos tengo hoy en mi calendario?, un LLM estándar respondería Lo siento, no tengo acceso a tu calendario. Sin embargo, al integrar un servidor MCP de un calendario, el LLM puede identificar y utilizar una herramienta disponible como read_events para consultar el calendario y proporcionar una respuesta precisa.



1. LLM (arriba del todo)

- Es el modelo de lenguaje.
- Envía una **REQUEST** (petición) al MCP client y recibe la **ANSWER** (respuesta).

2. MCP CLIENT (caja central grande)

- Es el “cerebro intermedio” entre el LLM y los servidores.
- Recibe las peticiones del LLM (por ejemplo usar la *CALENDAR TOOL*).
- Habla con los MCP servers usando el **MCP PROTOCOL**.
- Recoge los resultados y se los devuelve al LLM.

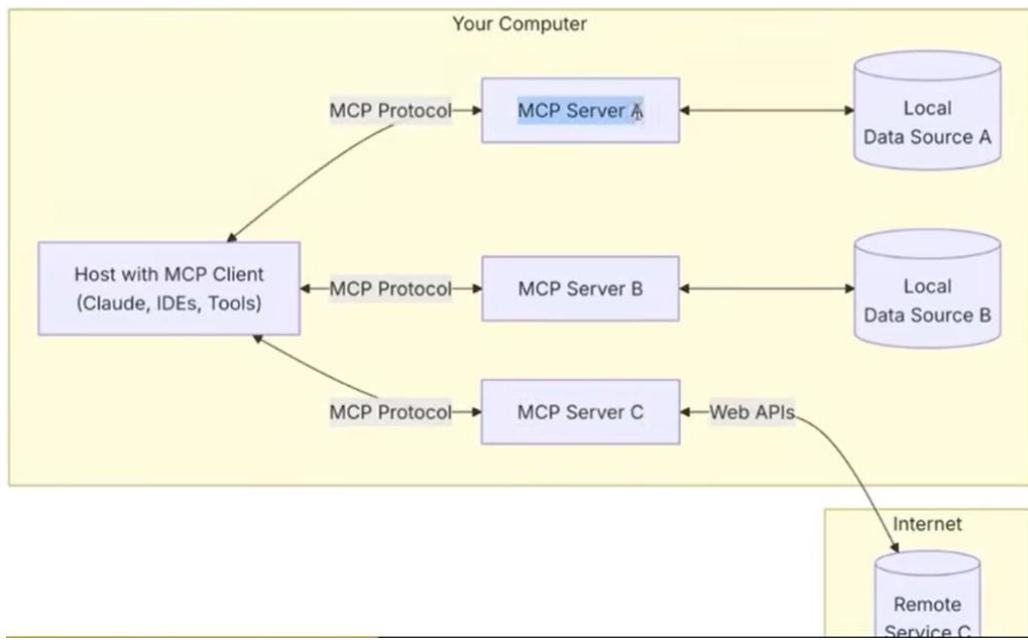
3. **MCP SERVERS** (tres cajas en la fila de abajo)

- Cada uno expone diferentes tools y recursos:
 - Izquierda: icono de **W** (tipo Word / documentos).
 - Centro (enmarcado en rojo): servidor que tiene la tool de **calendario**.
 - Derecha: icono de **base de datos** (BD).
- Entre el MCP client y cada server aparece el texto **MCP PROTOCOL**, indicando cómo se comunican.

4. Tool concreta: “Books meeting”

- El MCP server central ofrece una tool que **reserva reuniones** (*BOOKS MEETING*).
- La parte de abajo en rojo con el calendario marcado representa esa acción concreta.
- Encima se ve el texto **SEND RESULT**, indicando que ese servidor envía el resultado de la reserva al MCP client, que luego lo pasa al LLM.

En resumen: el LLM pide algo → el MCP client decide a qué MCP server hablar → ese servidor ejecuta su tool (por ejemplo reservar una reunión) → devuelve el resultado al MCP client → éste se lo pasa al LLM.



5. El Protocolo JSON-RPC

Se usa para la comunicación entre el cliente MCP y el servidor MCP. JSON-RPC es un protocolo ligero para hacer llamadas a procedimientos remotos usando JSON como formato de mensaje. No define el transporte (puede ir por HTTP, WebSocket, stdio, etc.).

Claves (v2.0)

- **jsonrpc:** versión del protocolo, normalmente "2.0".
- **method:** nombre del método remoto a invocar.
- **params:** argumentos (objeto o array). Opcional.
- **id:** identifica la petición para emparejar la respuesta.
 - Si no hay id es una **notification** (no se espera respuesta).
- **Respuesta éxito:** { "jsonrpc":"2.0", "id": <mismo>, "result": ... }
- **Respuesta error:** { "jsonrpc":"2.0", "id": <mismo>, "error": { "code": ..., "message": "...", "data": ...? } }

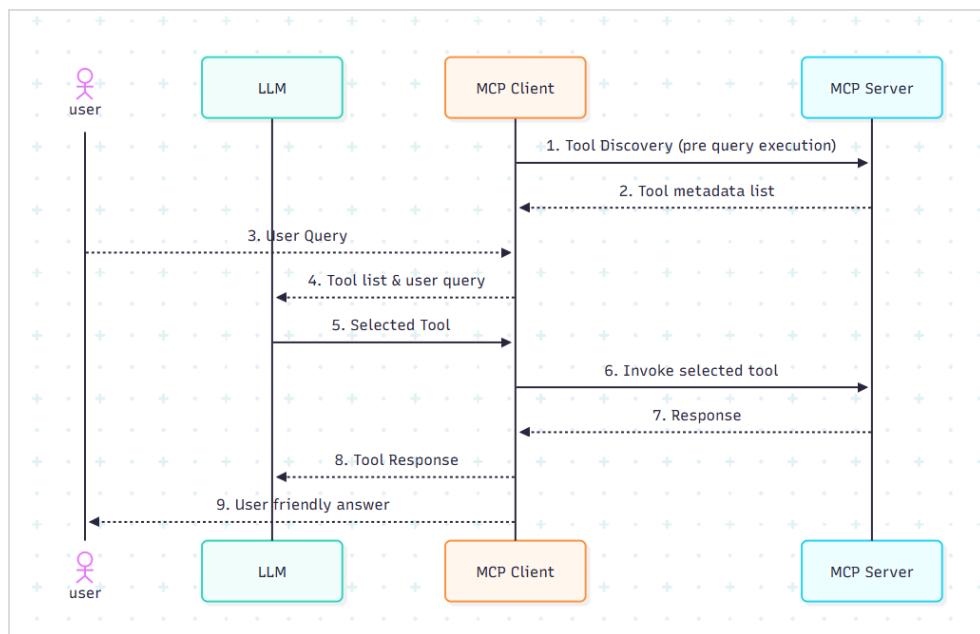
6. Componentes Fundamentales del MCP

El protocolo se define a través de una especificación **JSON-RPC** que articula tres tipos principales de componentes, cada uno con un propósito y un mecanismo de control distintos.

6.1. Herramientas (Tools)

Las **herramientas** representan acciones o capacidades que el LLM puede decidir invocar de forma autónoma para responder a una consulta. Son el componente más conocido y utilizado del MCP.

Mecanismo de Funcionamiento:



- Descubrimiento de herramientas:** Durante la configuración del servidor MCP, el cliente MCP descubre las herramientas disponibles en el servidor MCP. Esto es importante para que la aplicación conozca las capacidades disponibles en el servidor. Estas herramientas pueden asignarse luego a los agentes para la toma de decisiones dinámica.
- Detección de intención e invocación de herramientas:** Cuando un usuario envía una consulta, la lista de herramientas (**se inyecta**) y la consulta se envían al LLM para la detección de la intención y la invocación de herramientas. El LLM identifica la herramienta que debe usarse y construye una llamada estructurada a dicha herramienta.
- El cliente MCP envía la invocación al servidor MCP:** El cliente MCP envía una solicitud estructurada al servidor MCP con el nombre de la herramienta y los parámetros requeridos.
- Ejecución:** El servidor MCP ejecuta la lógica de la herramienta y devuelve los resultados al cliente MCP.

5. Generación de la respuesta: El agente utiliza la salida de la herramienta para formular, con ayuda del LLM, una respuesta en lenguaje natural para el usuario.

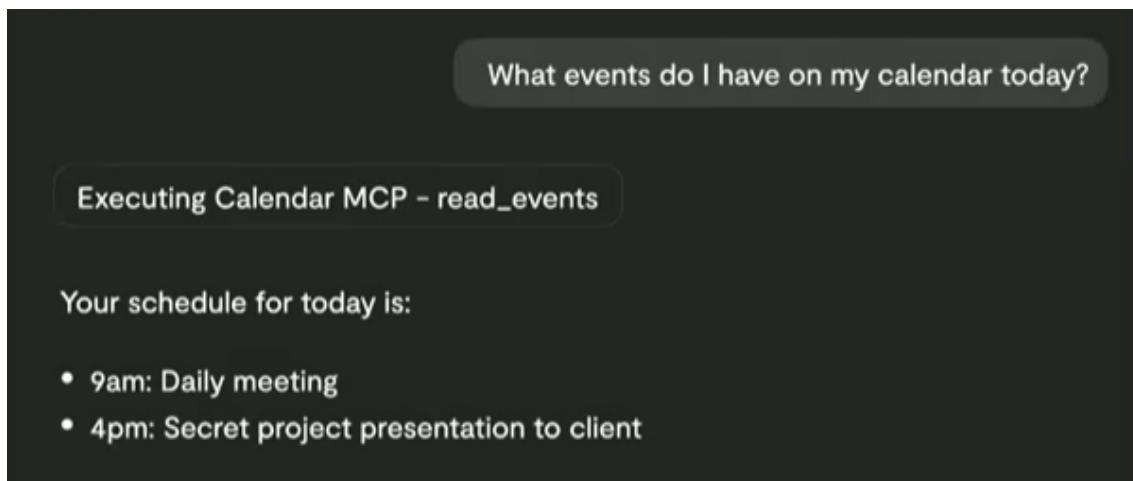
Estructura y Parámetros: Las herramientas se definen con un **nombre, título y descripción** para que el LLM sepa cuándo utilizarlas. Pueden aceptar parámetros definidos a través de un **inputSchema** en formato JSON Schema, permitiendo al LLM inferir los valores necesarios a partir del lenguaje natural del usuario (por ejemplo, traducir "hoy" a un formato de fecha **YYYY-MM-DD**).

6.1.1 Traza de mensajes entre cliente y servidor

Vamos a ver cómo es una **traza completa** de lo que pasa, paso a paso:

- **Fase de inicialización**
- **Fase de descubrimiento de tools**
- **Fase de invocación de una tool**

Ejemplo de solicitud del usuario



6.1.2. Fase de inicialización. Handshake

Exacto, **una parte importante** del handshake es ponerse de acuerdo en la versión del protocolo... pero no solo eso.

En la fase de **initialize** en MCP se hace básicamente esto:

1. **Negociar la versión del protocolo**

- El cliente dice: “hablo MCP X.Y”.
- El servidor responde con la versión que soporta.
- Si son incompatibles, se corta aquí.

2. Intercambiar información del cliente y del servidor

- Nombre de la app/cliente, versión, etc.
- Datos del servidor (nombre, versión).

3. Anunciar capacidades

- El cliente dice qué cosas sabe manejar (tools, resources, prompts, mensajes de log, etc.).
- El servidor dice qué funcionalidades ofrece (qué tipos de tools, recursos, notificaciones soporta...).

Cliente → Servidor

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "method": "initialize",
  "params": {
    "protocolVersion": "2024-11-05",
    "capabilities": {},
    "clientInfo": {
      "name": "ClienteMCP_Didactico",
      "version": "1.0"
    }
  }
}
```

Servidor → Cliente

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": {
    "protocolVersion": "2024-11-05",
    "serverInfo": {
      "name": "MiServidorMCP",
      "version": "1.0.0"
    },
    "capabilities": {
      "tools": {
        "listChanged": true
      },
      "prompts": {
        "listChanged": false
      },
      "resources": {
        "listChanged": false
      }
    }
  }
}
```

```

        "resources": {
            "listChanged": false
        }
    }
}

```

Con esto el “**handshake**” está hecho y el cliente ya puede pedir herramientas.

6.1.3. Fase de descubrimiento de tools

Cliente → Servidor

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

params puede ir vacío o directamente omitirse, según la implementación.

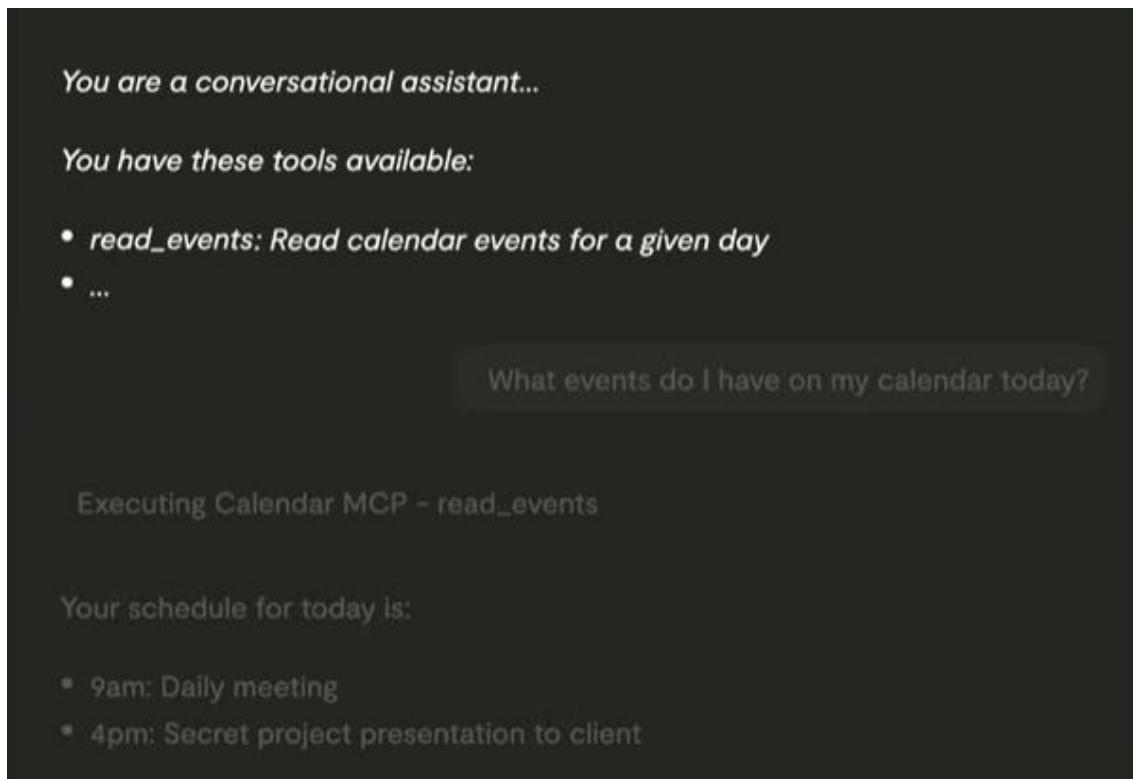
Servidor → Cliente

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "tools": [
            {
                "name": "read_events",
                "title": "Read calendar events",
                "description": "Read calendar events for a given day",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "date": {
                            "type": "string",
                            "description": "Date to read events.\nFormat: YYYY-MM-DD"
                        }
                    },
                    "required": ["date"]
                }
            }
        ]
    }
}
```

A partir de aquí, el cliente (o el LLM) sabe:

- Que existe la herramienta `read_events`.
- Que necesita un argumento `string date` con formato `YYYY-MM-DD`.

En el System Prompt del LLM se añaden las tools disponibles



6.1.4 Fase de invocación de tool

Ahora el cliente quiere leer los eventos del **2025-03-21**.

Cliente → Servidor

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "method": "tools/call",  
    "params": {  
        "name": "read_events",  
        "arguments": {  
            "date": "2025-03-21"  
        }  
    }  
}
```

Servidor → Cliente

Respuesta de ejemplo (inventando el contenido):

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "result": {  
        "content": [  
            {  
                "type": "text",  
                "text": "Your schedule for today is:  
                • 9am: Daily meeting  
                • 4pm: Secret project presentation to client"  
            }  
        ]  
    }  
}
```

```
        "text": "Events for 2025-03-21:\n- 09:00 Doctor appointment\n- 12:30 Team  
meeting\n- 18:00 Gym session"  
    }  
}  
}
```

En MCP, el resultado de un tools/call viene en:

- **result.content** → array de **bloques de contenido** (text, image, etc.).
En tu servidor típico de ejemplo casi siempre usarás type: "text".

Para **evitar colisiones** entre herramientas con el mismo nombre de diferentes MCPs, el cliente suele **añadir un prefijo** al nombre (ej. **calendar/read_events**, **notion/read_events**, **calendar-mcp-read_events**).

Resumen:

1. Handshake:

Cliente manda initialize → servidor responde con protocolVersion, serverInfo, capabilities.

2. Descubrir herramientas:

Cliente manda tools/list → servidor responde con el catálogo (tools) y sus inputSchema.

3. Usar una herramienta:

Cliente manda tools/call con name + arguments → servidor responde con result.content[...].

6.2. Recursos (Resources)

6.2.1. Contexto rápido: MCP, tools y resources

En el protocolo MCP (Model Context Protocol) un servidor se conecta a un cliente (por ejemplo, un IDE o una UI tipo ChatGPT) para darle superpoderes al LLM. Esos superpoderes se ofrecen principalmente de dos formas: **tools** (herramientas que el **modelo** puede llamar) y **resources** (recursos de información que el **cliente** puede inyectar como contexto).

6.2.2. Analogía mental “GET = resource” y “POST = tool”

A primera vista, muchos piensan que “**GET = resource**” y “**POST = tool**”, pero esto **no es verdad**. El criterio importante no es el verbo HTTP, sino **quién decide cuándo se usa y con qué intención**: llamar a una API (tool) o aportar contexto (resource).

La asociación con GET/POST viene de:

- La **spec oficial** y, sobre todo, de la **librería TypeScript oficial**, que dicen algo así como: “piensa en resources como llamadas tipo GET a APIs, y en tools como operaciones tipo POST”.
- ¿Por qué lo explican así? Porque **muchos servidores MCP envuelven APIs HTTP ya existentes**:
 - Un *resource* suele encajar muy bien con un “endpoint de solo lectura” (lo típico de un GET).
 - Una *tool* suele encajar muy bien con una operación que “hace cosas” (crear, modificar, disparar lógica), lo típico de un POST/PUT/DELETE.

Pero eso es **una guía de diseño, no una regla técnica**. MCP no sabe ni le importa si tú, por debajo, llamas a un GET o a un POST o lees un fichero local.

6.2.3 Qué es un resource en MCP

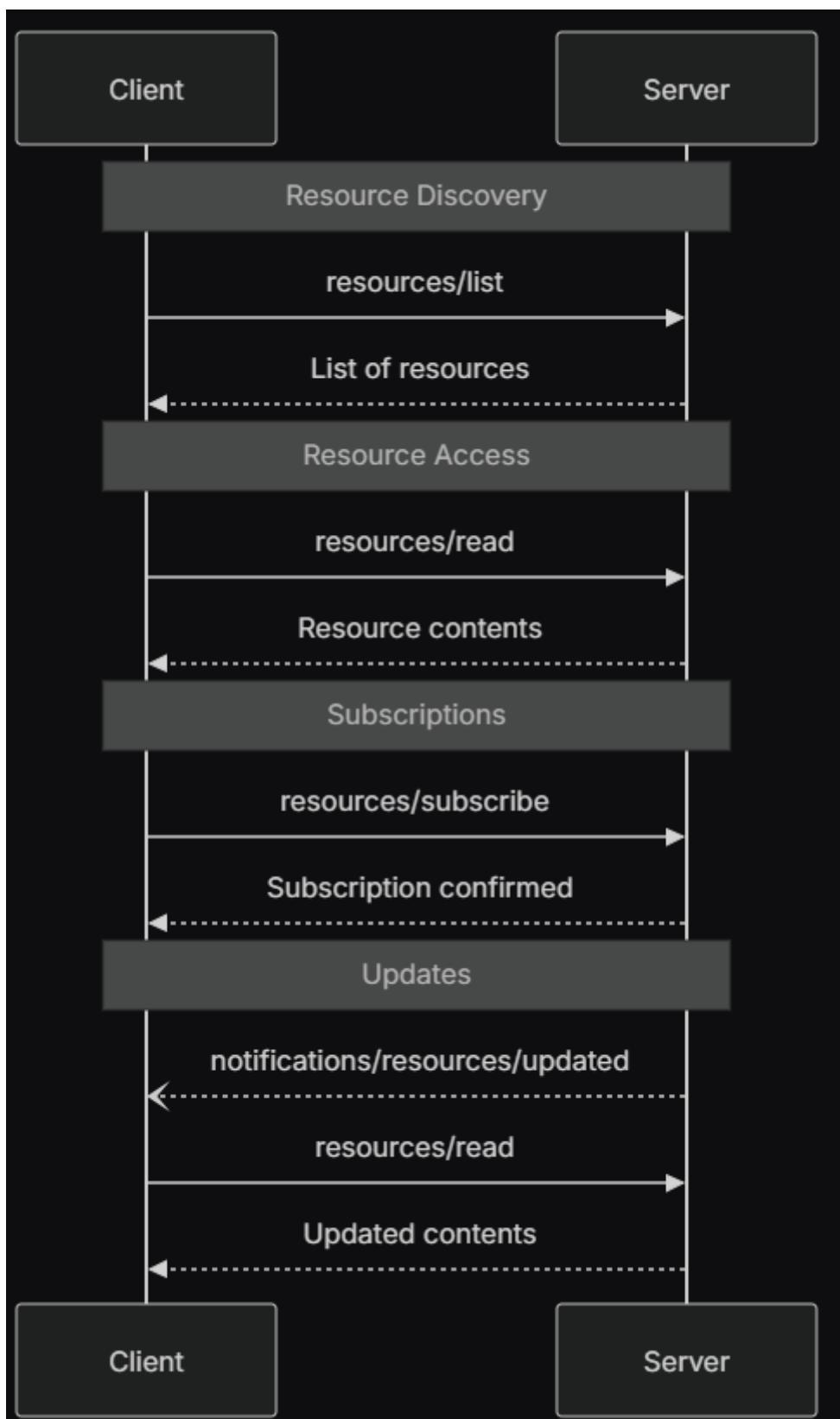
Un **resource** es, básicamente, un trozo de información accesible a través del servidor MCP, que **el cliente puede poner en el contexto del LLM cuando le convenga**. Podemos imaginarlo como si, en lugar de arrastrar un fichero JSON a ChatGPT, el propio servidor MCP dijera: “tengo este fichero JSON de cursos, si quieras te lo paso para que el modelo lo use como contexto”.

La idea clave es que el **resource no es una acción**, sino **información disponible**. Esa información puede ser un fichero JSON, un texto plano, un PDF, un audio transrito, una imagen, etc. **El servidor MCP declara qué recursos existen y cómo acceder a ellos**, pero no es el modelo quien los descubre solo, sino **el cliente (o el usuario) quien decide cuándo añadirlos a la conversación**.

¿Como los añade el cliente?

Lo que cambia de un cliente a otro es cómo decide:

- Puede ser un **if** tonto de palabras clave (como en tu ejemplo).
- Puede ser un “**router**” un poco más fino (intentar clasificar la intención con otro LLM).
- Puede ser algo explícito en la **UI** (el usuario marca “usar recurso de cursos” en una lista).



6.2.4 Diferencia de filosofía: tools vs resources

Una **tool** es algo que el **LLM** **puede decidir llamar de forma autónoma**. En el system prompt, el modelo sabe que tiene ciertas herramientas, sus nombres, sus descripciones y los parámetros que deben pasarse. Entonces, el modelo puede “razonar” y decir: “creo que ahora debo llamar a esta tool para obtener datos frescos” y el cliente ejecuta esa tool siguiendo las instrucciones del modelo.

Un **resource** funciona de forma distinta: el **LLM no sabe** por defecto que existen esos recursos ni puede decidir “por sí mismo” cuándo traerlos. No se le anuncian en el system prompt como herramientas utilizables. En su lugar, es **el cliente o el usuario quien decide**: “ahora voy a añadir este recurso al contexto del modelo”. Cuando el cliente inyecta un resource, lo hace como si pegara su contenido en el contexto, de forma similar a cómo un usuario copia y pega un JSON o arrastra un archivo a la ventana de chat.

Por eso, podemos tener una **misma información** disponible tanto como:

- **tool**: para que el modelo pueda llamar a “getCursos” cuando lo considere necesario.
- **resource**: para que el cliente, en un momento concreto, decida “inyectar cursos.json” como contexto y después hacer una pregunta al modelo.

La diferencia es **quién controla el momento de uso**:

- En tools, **decide el LLM** (dentro de las reglas del cliente).
- En resources, **decide el cliente/usuario**, el modelo no tiene iniciativa.

6.2.5 Analogía: arrastrar un JSON vs resource en MCP

Si arrastramos un fichero **cursos.json** a ChatGPT y preguntamos: “¿Qué cursos de arquitectura de software hay en este JSON?”, el LLM responde porque ese JSON se ha añadido a su contexto. Ese proceso se parece mucho a un **resource**:

- El archivo está disponible.
- El usuario (no el modelo) decide cuándo añadirlo.
- El modelo simplemente ve texto/JSON y razona sobre él.

En MCP con resources hacemos algo equivalente, pero de forma **más estructurada y programática**. El servidor MCP declara que existe, por ejemplo, un resource cursos/ que devuelve una lista de cursos en JSON. El cliente, cuando quiera, hace una llamada para obtener ese resource, y luego pasa el contenido al modelo como contexto antes de lanzar la pregunta sobre esos cursos.

La clave didáctica: **un resource es “información que se mete en el contexto bajo demanda”, igual que pegar un fichero, pero gestionado por el servidor MCP.**

6.2.6 Formato URI y el esquema file:

Los resources en MCP se identifican mediante **URIs**. En los ejemplos se usa un formato tipo **URL/URI** que puede incluir esquemas como **file:**. Esta elección no es casual: muchos de los **primeros clientes MCP eran IDEs** (como Cursor) y ya usaban **URIs** tipo **file://** para identificar ficheros de un proyecto. Esa experiencia influyó mucho en el diseño de la spec.

Gracias al esquema **file:**, en teoría podríamos usar **recursos locales** (ficheros de tu propio disco) como resources. Esto solo es posible si el servidor MCP tiene acceso a esos ficheros (por ejemplo, si el servidor corre en la misma máquina que el usuario y conoce el path). En ese caso, el servidor podría exponer, por ejemplo, **file:///Users/david/proyecto/cursos.json** como un resource que el cliente puede pedir y luego injectar al modelo.

Lo importante es que la URI actúa como un **identificador único del recurso**. El **cliente no necesita saber la implementación interna** (si viene de un fichero, una API, una base de datos); solo sabe que esa URI representa “ese trozo de información” que puede obtener cuando lo necesite.

6.2.7 El tipo de fichero (MIME type) y cómo lo interpreta el LLM

Al declarar un resource, el servidor MCP no solo indica su nombre y descripción, sino también el **tipo de contenido**: por ejemplo, **application/json**, **text/plain**, **image/png**, etc. Este tipo es esencial para que el cliente y el modelo sepan **cómo tratar esa información**.

Si el tipo es JSON, el cliente puede decidir mostrarlo estructurado o incluso hacer preprocesado antes de injectarlo. El modelo también puede recibir pistas de que el contenido es una estructura de datos sobre la que se puede razonar. Si el tipo es texto plano, se tratará simplemente como texto. Para imágenes, audio o vídeo, el cliente puede combinarlo con capacidades multimodales del modelo.

En resumen, el tipo de fichero ayuda a que el ecosistema MCP entienda qué clase de recurso se está proporcionando y cómo conviene manipularlo antes de dárselo al LLM.

6.2.8. Resource templates: recursos parametrizables

Además de resources “fijos” (siempre el mismo contenido), MCP introduce los **resource templates**. Un resource template es un recurso cuya URI lleva **parámetros**. Esos parámetros aparecen en la URI como variables entre llaves, por ejemplo:

```
mcp://cursos/{id}
```

En este caso, **{id}** es un parámetro que permite pedir “el recurso asociado a un curso concreto”. Cuando el cliente quiera usarlo, sustituirá **{id}** por un valor real, por ejemplo **mcp://cursos/arquitectura-software**, y el servidor devolverá solo la información de ese curso específico.

La gracia es que **el protocolo infiere los parámetros directamente de la URI template**. No hace falta declarar un esquema de parámetros como en tools. El parámetro id será siempre recibido como string, y el servidor decide qué hacer con él (consultar una base de datos, leer un fichero concreto, etc.).

Así, un resource template permite cosas como:

- Listar todos los cursos con un resource general (por ejemplo, **mcp://cursos/**).
- Pedir los detalles de un curso concreto con un resource template (**mcp://cursos/{id}**).

El flujo típico sería: **el cliente primero inyecta el resource general** con todos los cursos, el modelo responde, y si luego el usuario quiere detalles de **un curso concreto, el cliente usa el resource template**, le pide al usuario (o a otra capa lógica) el id adecuado y vuelve a inyectar ese recurso específico.

Genial, vamos a verlo todo seguido y bien ordenado, con **resource templates** incluidos.

Te enseño las **3 fases** y, en cada una, la **petición del cliente y la respuesta del servidor**.

Nota: Los nombres de método (resources/list, resources/read, etc.) son los de la spec actual. En algunos SDK pueden envolverlos, pero la idea es esta.

6.2.9 Cómo encaja todo en una conversación real

En una conversación real con un cliente MCP, **el modelo no ve “resources”** como tal. **Lo que ve es contexto:** bloques de texto, JSON, etc. Es el cliente quien, en segundo plano, decide:

1. Qué resource o resource template **pedir al servidor MCP**.
2. **Cuándo pedirlo** (antes de la pregunta del usuario, como respuesta a una acción en la UI, etc.).
3. Cómo **transformar** el contenido del resource y cómo insertarlo en el mensaje que se envía al LLM.

Por eso, cuando el usuario pregunta “¿Qué cursos de arquitectura de software hay?”, el flujo suele ser:

- El usuario hace la pregunta.
- El cliente detecta (por reglas suyas o por una acción del usuario) que debe injectar el resource cursos/.
- El cliente pide el resource al servidor MCP, recibe el JSON y lo añade al contexto.
- El LLM ve la pregunta + el JSON y responde usando esa información.

En resumen, los **resources en MCP son una forma estructurada de hacer RAG controlado por el cliente**: son datos que viven en el servidor MCP (o a los que el servidor puede acceder) y que el cliente decide cuándo injectar en el contexto del modelo. No son “acciones” que el modelo elige ejecutar, sino **contexto que el cliente decide proporcionar**.

6.2.10. Traza Resources

1. Fase de información de protocolo (capabilities / initialize)

Aquí el cliente se conecta y pregunta: “¿qué sabes hacer?”

El servidor responde: “soporto tools, resources, etc.”

Todavía no se habla de **qué** resources, solo de que **existen**.

Servidor - Cliente (respuesta simplificada de initialize del servidor):

```
{  
  "jsonrpc": "2.0",  
  "id": 0,  
  "result": {  
    "protocolVersion": "2024-11-05",  
    "capabilities": {  
      "tools": {},  
      "resources": {}  
    }  
  }  
} // El servidor dice: sé trabajar con resources
```

```

        "resourceTemplates": {},
    },
    "serverInfo": {
        "name": "courses-server",
        "version": "1.0.0"
    }
}

```

El cliente solo sabe: “este servidor soporta resources”, pero aún no sabe que existe courses://all.

2. Fase de descubrimiento (resources/list)

Ahora el cliente quiere saber **qué recursos concretos** hay.

Petición del cliente (descubrir resources):

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

Respuesta del servidor (metadatos de resources):

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "resources": [
            {
                "uri": "courses://all",
                "name": "courses",
                "title": "All Courses",
                "description": "Complete list of courses available in the catalog",
                "mimeType": "application/json"
            },
            {
                "uri": "users://all",
                "name": "users",
                "title": "All Users",
                "description": "Complete list of users in the platform",
                "mimeType": "application/json"
            }
        ]
    }
}
```

Aquí el cliente aprende:

- Que existe el recurso courses://all (lista de cursos).
 - Que existe users://all (lista de usuarios).
- Todavía **no** tiene el contenido, solo los **metadatos**.

3. Fase de uso / invocación (resources/read)

Cuando el cliente quiere **usar** de verdad un recurso (leer su contenido), llama a resources/read para una URI concreta.

3.1. Leer la lista de cursos

Petición del cliente (leer el resource):

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "method": "resources/read",  
    "params": {  
        "uri": "courses://all"  
    }  
}
```

◆ **Respuesta del servidor (contenido del resource):**

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "result": {  
        "contents": [  
            {  
                "uri": "courses://all",  
                "mimeType": "application/json",  
                "text": "[{\\"id\\":1,\\"name\\\":\\"Software Architecture\\"},{\\"id\\":2,\\"name\\\":\\"Clean  
Code\\"}]"  
            }  
        ]  
    }  
}
```

Ahora sí: aquí viene el **JSON real** con los cursos.

Este JSON es lo que el **cliente MCP** luego inyecta en el prompt del LLM (como contexto) antes de mandarle la pregunta del usuario.

Resumen

- **Fase 1 — capabilities:** “Sé trabajar con resources”.
- **Fase 2 — resources/list:** “Tengo estos resources: courses://all, users://all, etc.”
- **Fase 3 — resources/read:** “Dame el contenido de courses://all... aquí lo tienes en JSON”.

6.2.11 Traza Resource Templates

Fase de información de protocolo (initialize / capabilities)

Petición del cliente → servidor

```
{  
    "jsonrpc": "2.0",  
    "id": 0,  
    "method": "initialize",  
    "params": {  
        "protocolVersion": "2024-11-05",  
        "clientInfo": {  
            "name": "my-mcp-client",  
            "version": "1.0.0"  
        },  
        "capabilities": {}  
    }  
}
```

Respuesta del servidor → cliente

```
{  
    "jsonrpc": "2.0",  
    "id": 0,  
    "result": {  
        "protocolVersion": "2024-11-05",  
        "serverInfo": {  
            "name": "courses-server",  
            "version": "1.0.0"  
        },  
        "capabilities": {  
            "tools": {},  
            "resources": {},           // Soporto resources  
            "prompts": {},  
            "sampling": {},  
            "roots": {}  
        }  
    }  
}
```

“Este servidor soporta **resources** (y, por extensión, también resource templates).”

Fase de descubrimiento (resources/list y templates)

Descubrir resources directos

Petición cliente → servidor

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

Respuesta servidor → cliente

```
{  
    "jsonrpc": "2.0",  
    "id": 1,
```

```

"result": {
  "resources": [
    {
      "uri": "courses://all",
      "name": "courses",
      "title": "All Courses",
      "description": "Complete list of courses available in the catalog",
      "mimeType": "application/json"
    }
  ]
}

```

Aquí se anuncian **recursos concretos**, con URI fija (no plantilla).

Descubrir resource templates

Ahora el cliente quiere saber qué **URI templates** hay, para poder construir URLs dinámicas (por ejemplo courses://AI101).

En la práctica, algunos SDK tienen un método propio (p.ej. resources/listTemplates).

Petición cliente → servidor

```

{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "resources/listTemplates",
  "params": {}
}

```

Respuesta servidor → cliente

```

{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resourceTemplates": [
      {
        "uriTemplate": "courses:///{id}",
        "name": "course-details",
        "title": "Course Detail",
        "description": "Get detailed information for a course by id",
        "mimeType": "application/json"
      }
    ]
}

```

Aquí el cliente aprende: “Si construyo una URI como courses://AI101, el servidor sabe generar un recurso con los detalles de ese curso.”

Fase de uso / invocación del template (resources/read con URI concreta)

Petición del cliente → servidor (leer un recurso generado desde el template)

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "resources/read",
  "params": {
    "uri": "courses://AI101"
  }
}
```

Respuesta del servidor → cliente (contenido del recurso)

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "contents": [
      {
        "uri": "courses://AI101",
        "mimeType": "application/json",
        "text": "{ \"id\": \"AI101\", \"name\": \"Intro to AI\", \"level\": \"Beginner\", \"hours\": 40 }"
      }
    ]
  }
}
```

Ese text (el JSON con los datos del curso) es lo que el **cliente** meterá después como contexto en el prompt del LLM.

Resumen

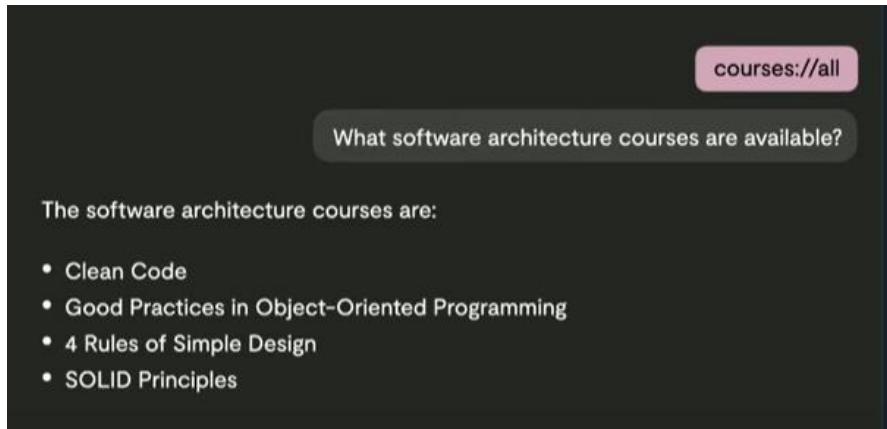
- **Fase 1 (initialize/capabilities):** el servidor dice “soporto resources”.
- **Fase 2 (list + listTemplates):** el servidor dice “tengo courses://all y una plantilla courses://{id}”.
- **Fase 3 (read):** el cliente lee courses://AI101 con resources/read y recibe el JSON del curso AI101.

6.2.12. Ejemplos

Resources

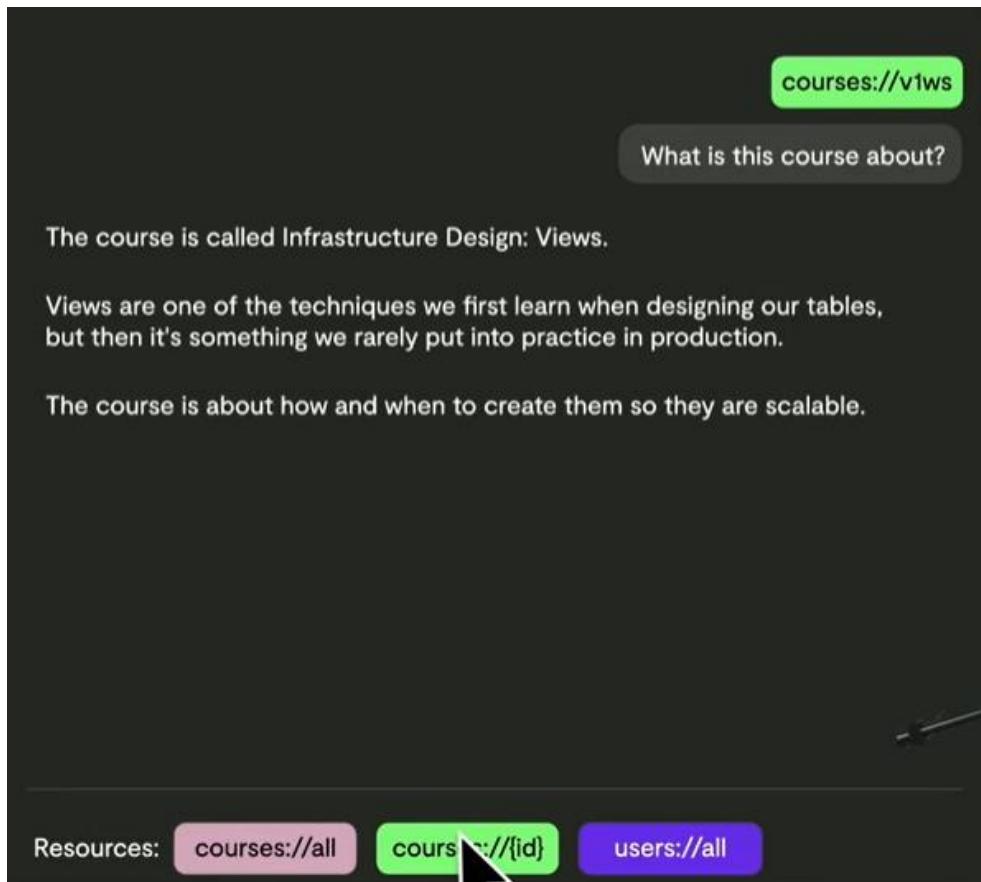
El resource lo incluye el cliente programáticamente

- Puede ser un **if** tonto de palabras clave (como en tu ejemplo).
- Puede ser un “**router**” un poco más fino (intentar clasificar la intención con otro LLM).
- Puede ser algo explícito en la **UI** (el usuario marca “usar recurso de cursos” en una lista).



Resource templates

Cuando el usuario quiera consultar un curso concreto, el cliente MCP utilizará el *resource template* y necesitará saber el valor del parámetro id. Para ello, el cliente nos pedirá ese identificador (por ejemplo, seleccionándolo en la UI o preguntándonoslo explícitamente) y, con ese id, construirá la URI correspondiente (por ejemplo, courses://AI101) y llamará a resources/read. El servidor devolverá entonces los datos de ese curso en concreto, y el LLM podrá responder usando solo el contexto de ese curso específico.



2.3. Prompts

Los prompts son el componente más avanzado y menos soportado actualmente, pero ofrecen un gran potencial. **Son plantillas de consulta predefinidas y parametrizables que el servidor MCP provee al cliente.** Permiten abstraer interacciones complejas en una acción simple para el usuario.

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "prompts": [  
            {  
                "name": "courses-search_similar_by_names",  
                "title": "Search Courses with Similar Names",  
                "description": "Generate a prompt to search for courses with similar names.",  
                "arguments": [  
                    {  
                        "name": "names",  
                        "description": "Comma-separated list of course names",  
                        "required": true  
                    }  
                ]  
            }  
        ]  
    }  
}
```

Flujo de Interacción:

1. El usuario selecciona un prompt disponible en la interfaz del cliente (ej. **"Buscar Cursos Similares por Nombre"**).
2. El cliente solicita al usuario los argumentos necesarios para ese prompt (ej. una **lista de nombres de cursos**).
3. **Los argumentos se envían al servidor MCP.** El servidor **ejecuta una lógica interna**, como traducir los nombres de los cursos a sus IDs correspondientes.
4. **El servidor construye y devuelve una cadena de texto completa (un prompt) al cliente.** Este prompt puede contener instrucciones complejas, como llamar a una herramienta específica con los IDs correctos.
5. **El cliente inserta este prompt generado en la ventana de chat**, como si lo hubiera escrito el usuario, y lo envía al LLM para su ejecución.

Este mecanismo es descrito como "una especie de RAG superinteresante, superpotente", ya que permite guiar al LLM de manera precisa, **superando la brecha entre la entrada intuitiva del usuario (nombres) y los requisitos técnicos de la API (IDs)**.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "prompts": [
      {
        "name": "courses-search_similar_by_names",
        "title": "Search Courses with Similar Names",
        "description": "Generate a prompt to search for courses with similar names.",
        "arguments": [
          {
            "name": "names",
            "description": "Comma-separated list of course names",
            "required": true
          }
        ]
      }
    ]
  }
}
```

3. Funcionalidades Avanzadas del Lado del Cliente

Además de los tres componentes principales, la especificación del MCP contempla funcionalidades avanzadas que dependen en gran medida de la implementación en el cliente:

- **Raíces (Roots):** Permiten a un servidor MCP definir a qué ficheros o directorios raíz tiene permiso de acceso un cliente, una funcionalidad especialmente relevante para IDEs como Visual Studio Code.
- **Sampling:** Es la capacidad de un servidor MCP para delegar una tarea de procesamiento al LLM del cliente. Por ejemplo, el servidor puede enviar un XML al cliente con la instrucción "traduce esto a JSON" y pedir que se le llame de vuelta con el resultado, externalizando así la carga de trabajo.
- **Sonsacamientos (Elicitation):** Habilita una comunicación bidireccional o "full duplex". El servidor MCP puede hacer preguntas directamente al usuario a través de la interfaz del cliente para recabar información necesaria (ej. ¿Cuál es tu usuario de GitHub?), permitiendo flujos de trabajo interactivos y conversacionales.

4. Estado Actual y Evolución del Protocolo

El MCP es un estándar en desarrollo, caracterizado por una rápida evolución y un estado actual que presenta tanto fortalezas como debilidades.

- **Adopción:** La adopción está creciendo. Clientes como **Visual Studio Code** han integrado recientemente el "Full Spect MCPS", mientras que otros clientes populares aún no soportan las funcionalidades más avanzadas como los prompts o la elicitation.

- **Evolución Rápida:** Se describe como un protocolo en "fase embrionaria". Las primeras versiones tenían deficiencias en áreas críticas como la autenticación y autorización, aspectos que han sido mejorados en iteraciones posteriores.
- **Críticas e Inconsistencias:** La especificación presenta inconsistencias. Un ejemplo claro es la definición de parámetros requeridos: en las herramientas, se usa un array required dentro de un JSON Schema, mientras que en los prompts se utiliza una propiedad booleana required en cada argumento, mostrando formatos diferentes para un mismo concepto.
- **Comparación con APIs REST:** Existe un debate sobre por qué no se utilizó el estándar REST. La justificación radica en que MCP fue diseñado con un enfoque nativo en el contexto y las descripciones para los LLMs. Su origen en clientes de terminal que luego evolucionaron a HTTP también influyó en su diseño actual.

6.3. *Prompts* en MCP

En MCP, además de **tools** y **resources**, existe un tercer bloque conceptual: los **prompts**. Un *prompt* en MCP es como una “plantilla de mensaje” que vive en el **servidor MCP**. El servidor define una serie de prompts con nombre, título, descripción y, opcionalmente, argumentos. El **cliente MCP** sabe que esos prompts existen y puede pedir al servidor: “genérame el texto de este prompt con estos argumentos”, y el servidor devuelve un texto listo para enviar al LLM, como si lo hubiera escrito el usuario.

6.3.1. Problema que resuelven: el usuario no sabe IDs, solo nombres

En el ejemplo del vídeo, el usuario quiere “buscar cursos relacionados con el curso de vistas y caché”. Si el LLM solo tiene una tool que trabaja con **IDs de curso** (no con nombres), el modelo, por sí solo, no sabe cómo pasar de “vistas y caché” a los IDs internos. Además, el usuario final nunca va a conocer esos identificadores internos, solo sabe que quiere cursos sobre “inteligencia artificial”, “transacciones”, “caché”, etc. Los prompts cubren justo ese hueco: permiten que el servidor MCP transforme entradas “amigables” (nombres) en algo más técnico (IDs) sin que el usuario tenga que preocuparse de la lógica interna.

Search for courses related to views and cache

I'm sorry, I don't have access to any tools that can search for courses by their names.

6.3.2. Ejemplo de prompt parametrizado

El servidor MCP define un prompt del tipo “**buscar cursos similares por nombre**”. Ese prompt tiene un argumento (por ejemplo, nombres) que el cliente pedirá al usuario.

Cuando el usuario introduce “**caché y vistas**”, el cliente envía ese valor al servidor junto con el nombre del prompt.

El servidor, por detrás, busca cada nombre de curso en su sistema, obtiene los **IDs reales** y construye un prompt más elaborado, del estilo: “Usa la herramienta X para obtener cursos similares a los cursos con ID 12 y 34...”.

El servidor devuelve ese texto y **el cliente lo pega en la conversación como si lo hubiera escrito el usuario**.

El LLM ya ve una instrucción bien formada, con los IDs correctos, y puede decidir llamar a la tool adecuada con esos parámetros.

Find similar courses using the tool
courses-search_similar_by_ids to these IDs: v1ws, c4ch

Executing Codely MCP - courses-search_similar_by_ids("v1ws,c4ch")

I have found 10 courses similar to the courses with IDs "v1ws" and "c4ch".

Here are the results:

Infrastructure Design: Transactions (ID: tx4n)

Learn how ...

Prompts: courses-search_similar_by_names

Esto no lo ha escrito el usuario, esto lo ha escrito el servidor que devuelve el texto y el cliente lo ha pegado como si fuese el usuario. El servidor por detrás ha realizado una búsqueda de los cursos por nombre y ha devuelto los id, para que el cliente los ponga en el prompt, de esta forma el LLM puede responder.

6.3.3. Importante: quién genera qué

Un punto clave es que **el texto del prompt no lo escribe el usuario ni el cliente, lo genera el servidor MCP**. El cliente solo hace de “pegamento”: recoge el argumento nombres, lo manda al servidor, recibe el texto ya montado y lo inserta en el chat del LLM. Desde la perspectiva del modelo, ese mensaje parece un mensaje normal del usuario, pero en realidad es una plantilla inteligente construida por el servidor para guiar al LLM hacia el uso correcto de tools y recursos. Esto permite que el usuario hable en lenguaje natural y el servidor se encargue de traducirlo a “lenguaje operativo” (IDs, llamadas a tools, etc.).

6.3.4 Cómo se definen los prompts en la spec

En la especificación MCP, los prompts se definen con metadatos similares a tools y resources: tienen un **nombre**, un **título**, una **descripción** y una lista de **argumentos**. Cada argumento suele incluir su nombre, su descripción (para que el cliente sepa qué pedir al usuario) y si es requerido o no. Aquí se ve una cierta inconsistencia de la spec: en tools los parámetros se describen con JSON Schema y el campo required es un array de nombres, mientras que en prompts el formato de argumentos es distinto. Aun así, conceptualmente, la idea está muy bien: los prompts son otro “bloque de construcción” junto con tools y resources para orquestar cómo se guía al modelo.

1. Descubrimiento de prompts (prompts/list)

Petición: Cliente → Servidor

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

Respuesta: Servidor → Cliente

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "prompts": [  
            {  
                "name": "course-similar-by-name",  
                "title": "Find similar courses by name",  
                "description": "Given one or more course names, build a prompt that asks the LLM to search similar courses using tools and IDs.",  
                "arguments": [  
                    {  
                        "name": "names",  
                        "description": "Comma-separated list of course names to use as reference",  
                        "required": true  
                    }  
                ]  
            }  
        ]  
    }  
}
```

Ahí el cliente aprende que existe un prompt llamado course-similar-by-name y qué argumentos necesita.

2. Uso de un prompt (prompts/get) sin argumentos

Por ejemplo, un prompt simple que no recibe parámetros, tipo “plantilla de resumen general”.

Petición: Cliente → Servidor

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "method": "prompts/get",  
    "params": {  
        "name": "general-summary",  
        "arguments": {}  
    }  
}
```

Respuesta: Servidor → Cliente

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "result": {  
        "messages": [  
            {  
                "role": "user",  
                "content": [  
                    {  
                        "type": "text",  
                        "text": "Summarize the following content in a concise way, highlighting key  
concepts and main sections."  
                    }  
                ]  
            }  
        ]  
    }  
}
```

El servidor devuelve **los mensajes** que el cliente debe pegar en el chat (en este caso, uno tipo user con texto).

Roles

Indica desde qué “voz” o “papel” se envía ese mensaje al LLM.

El **cliente MCP**, cuando recibe esto en prompts/get, normalmente lo va a traducir 1:1 a mensajes para el modelo.

- **"system"** → reglas del juego (quién eres, cómo responder, límites, estilo).
- **"user"** → lo que escribe la persona (tú).
- **"assistant"** → lo que responde el modelo.

3. Uso de un prompt parametrizado (prompts/get con argumentos)

Ahora sí, el ejemplo interesante: course-similar-by-name con argumento names = "caché, vistas".

Petición: Cliente → Servidor

```
{  
    "jsonrpc": "2.0",  
    "id": 3,  
    "method": "prompts/get",  
    "params": {  
        "name": "course-similar-by-name",  
        "arguments": {  
            "names": "caché, vistas"  
        }  
    }  
}
```

Respuesta: Servidor → Cliente

```
{  
    "jsonrpc": "2.0",  
    "id": 3,  
    "result": {  
        "messages": [  
            {  
                "role": "user",  
                "content": [  
                    {  
                        "type": "text",  
                        "text": "Given the following course names: \"caché\", \"vistas\", look up  
                        their internal IDs, then use the tool `findSimilarCourses` to retrieve courses that are  
                        related to those IDs. Finally, answer the user with a short description of the most  
                        relevant similar courses."  
                    }  
                ]  
            }  
        ]  
    }  
}
```

Aquí pasan varias cosas importantes:

- El **servidor MCP** ha cogido el argumento names y ha construido una “super-prompt” lista para el LLM.
- El **cliente** lo único que hace es:
 1. Llamar a prompts/get con los argumentos.
 2. Recibir estos messages.
 3. Pegarlos tal cual en la conversación con el modelo (como si el usuario los hubiera escrito).

4. Resumen rápido

- prompts/list → **descubre** qué prompts hay y qué argumentos esperan.
- prompts/get → **genera** los mensajes (trama de prompt) en función del nombre del prompt y los argumentos.
- El texto que ves en messages es la “trama” de prompt que luego irá al LLM.

6.3.5 Prompts como capa de RAG “inteligente”

Combinando **resources** y **prompts**, se consigue una especie de RAG más sofisticado. El usuario dice lo que quiere en términos de contenidos (“quiero cursos de X”), el cliente usa un prompt parametrizado, el servidor traduce esos nombres a IDs y, a partir de ahí, puede leer resources concretos o instruir al modelo para usar tools. Es una manera elegante de evitar que el usuario tenga que conocer detalles internos (identificadores, endpoints, etc.) y centralizar esa lógica de traducción en el servidor MCP.

6.5 Otras capacidades del lado cliente: roots, sampling y “preguntas inversas”

Además de prompts, la spec MCP menciona otras funcionalidades que caen más del lado del **cliente** que del servidor.

Los **roots** son las “raíces” de ficheros a las que el cliente tiene acceso (por ejemplo, qué carpetas del proyecto puede ver Visual Studio Code o Cursor).

El **sampling** permite que el servidor delegue una tarea al cliente usando un prompt, por ejemplo, “traduce este XML a JSON” y que el cliente lo resuelva con su propio LLM y devuelva el resultado al servidor.

Por último, existe la capacidad de que el **MCP haga preguntas al cliente** (por ejemplo, “¿cuál es tu usuario de GitHub?”), de modo que la comunicación pasa a ser más bidireccional: no solo el usuario alimenta al MCP, sino que el MCP también puede iniciar solicitudes de información al lado cliente.

Servidor MCP → Cliente (software): “Necesito que me des el usuario de GitHub”.

El **cliente (software)** decide cómo obtener eso:

- Puede sacar la info de configuración local.
- O puede abrir un cuadro de diálogo / prompt en la UI y preguntárselo al **usuario humano**.

El cliente responde al servidor MCP con el dato.

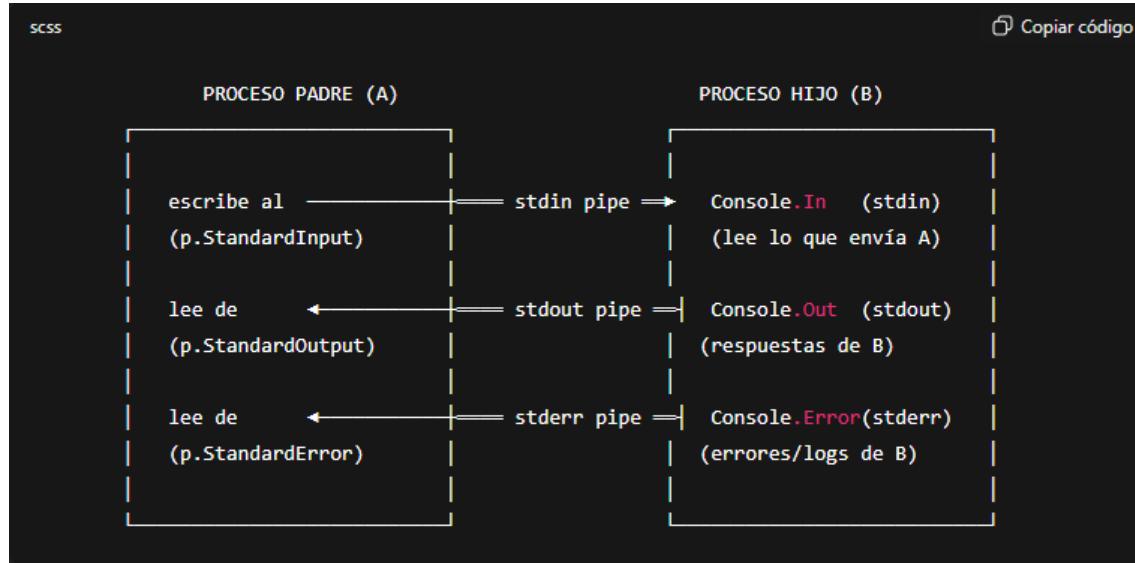
6.6 Tools vs Resources vs Prompts

Aspecto	Tool	Resource	Prompt
¿Qué es?	Una acción que el servidor expone para que el LLM la llame.	Un bloque de datos (información) que el servidor puede entregar.	Una plantilla de mensaje que el servidor rellena y devuelve como texto.
Quién decide usarlo	El LLM (mediante tool_calls).	El cliente (o la UI), nunca el LLM directamente.	El cliente (elige qué prompt usar y con qué argumentos).
Cómo se invoca en MCP	tools/call	resources/read	prompts/get (o similar según la librería).
Para qué se usa	Hacer cosas: consultar APIs, escribir en BD, crear tickets...	Dar contexto: listas, documentos, ficheros, etc.	Guiar al modelo, traducir inputs “humanos” a instrucciones técnicas.
Tipo de operación mental	“ Haz esto ”	“ Ten en cuenta estos datos ”	“ Genera este mensaje base a partir de estos parámetros ”
Ejemplo típico	getCourseById(id)	courses://all (lista de cursos)	course-similar-by-name(nombres)
Quién genera el contenido	El servidor ejecuta la acción y devuelve el resultado (datos).	El servidor devuelve datos ya existentes (contenido del recurso).	El servidor genera el texto del prompt (el cliente solo lo pega al chat).

7. Comunicación entre cliente y servidor

7.1 STDIO.

¿Cómo se comunican dos procesos por stdio?



Cuando un proceso A(padre) lanza a otro proceso B(hijo), el SO puede **redirigir** los flujos estándar de B:

- **stdin** (entrada) → lo que B lee.
- **stdout** (salida) → lo que B escribe “normal”.
- **stderr** (errores) → mensajes de error de B.

A nivel de SO, stdin/stdout/stderr son **pipes** (tuberías) con buffers.

A, como “padre”, escribe en el **stdin** de B y lee del **stdout/stderr** de B.

Flujo típico:

1. A crea B con redirección activada.
2. A **escribe** bytes (p. ej., texto UTF-8) → stdin de B.
3. B **lee** esos bytes, los procesa y **escribe** su respuesta en stdout.
4. A **lee** stdout y actúa.

Para “delimitar” mensajes se usa un **encuadre (framing)**. Dos comunes:

- **NDJSON:** un JSON por línea (\n). Simple y muy usado (MCP).
- **Content-Length + \r\n\r\n** (como el LSP). Robustez para binarios y streams largos.

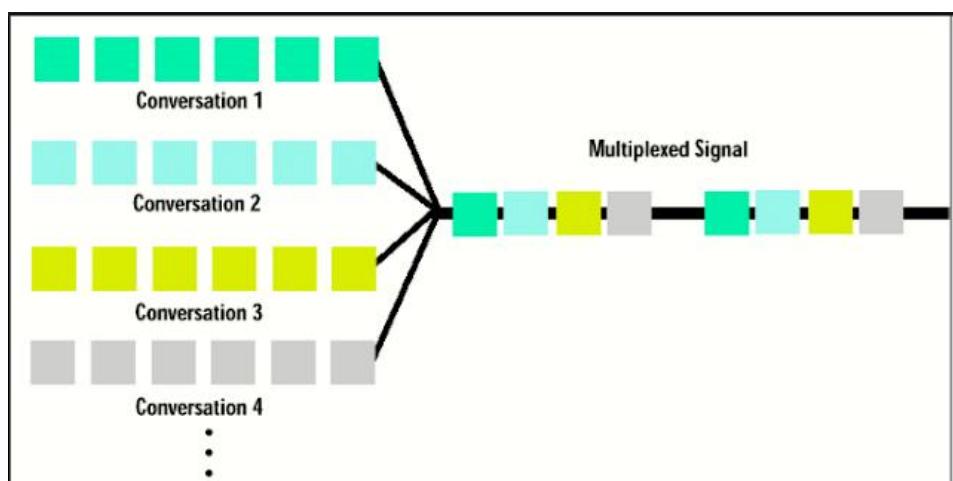
¿Cómo lo hace MCP?

MCP usa **JSON-RPC 2.0 sobre stdio** (normalmente **NDJSON**):

- El **cliente** suele lanzar al **servidor MCP** como proceso hijo.
- Cada mensaje es un JSON en **una línea** (o con framing equivalente).
- Ejemplo:

```
{"jsonrpc": "2.0", "id": 1, "method": "initialize", "params": {...}}\n{"jsonrpc": "2.0", "id": 2, "method": "tools/list", "params": {}}\n{"jsonrpc": "2.0", "id": 2, "result": {"tools": [...]}}\n
```

- JSON-RPC aporta id para **multiplexar** peticiones/resuestas por el mismo canal.



7.1.1 Ventajas e inconvenientes

¿Por qué MCP suele usar stdio?

Ventajas

1. **Simplicidad y portabilidad:** No hay sockets, ni puertos, ni CORS; funciona igual en Windows, macOS, Linux.
2. **Aislamiento y seguridad:** El servidor es un binario aparte (sandboxable). Al no abrir puertos, menor superficie de ataque.
3. **Cero configuración de red:** Nada de firewalls/iptables. Ideal para plugins locales y entornos corporativos restrictivos.
4. **Agnóstico de lenguaje/runtime:** Cualquier lenguaje que lea/escriba JSON por stdio sirve.
5. **Backpressure natural:** Los pipes tienen buffer; si A no lee, B se bloquea al escribir (previene “flooding”).

6. **Supervisor fácil:** El cliente puede arrancar, parar y reiniciar al servidor con el ciclo de vida del proceso.

Inconvenientes

- **Throughput** moderado (pero suficiente para control/órdenes). **Throughput** es la cantidad de datos u operaciones que un sistema consigue procesar por unidad de tiempo.
 - En red: “esta conexión tiene un throughput de 10 MB/s” → realmente está enviando/recibiendo 10 megabytes por segundo.
 - En un sistema de colas: “el servidor procesa 500 peticiones por segundo” → throughput = 500 req/s.
- **Reconexión** manual (si se cae el proceso, hay que relanzarlo).
- **Framing**: debes definirlo bien (NDJSON o Content-Length).
- **Bloqueos** si ambos esperan leer/escribir sin drenar (siempre leer stdout y stderr en hilos separados).

7.1.2 Mini-esquema de implementación (C#)

```
var p = new Process {
    StartInfo = new ProcessStartInfo {
        FileName = "dotnet",
        ArgumentList = { "run", "--project", "Servidor.csproj" },
        UseShellExecute = false,
        RedirectStandardInput = true, // escribir al stdin del servidor
        RedirectStandardOutput = true, // leer stdout del servidor
        RedirectStandardError = true // leer stderr del servidor
    }
};
p.Start();

// Lectores async (evita deadlocks)
_ = Task.Run(async () => { while ((line = await p.StandardOutput.ReadLineAsync()) != null) Handle(line); });
_ = Task.Run(async () => { while ((line = await p.StandardError.ReadLineAsync()) != null) LogErr(line); });

// Envío NDJSON (una línea = un mensaje JSON-RPC)
await p.StandardInput.WriteLineAsync(jsonOneLine);
await p.StandardInput.FlushAsync();
```

Librería StdioClientTransport

```
// Transport MCP por stdio usando el SDK
var clientTransport = new StdioClientTransport(new StdioClientTransportOptions
{
    Name = "3_ServidorMCP_LLM",
    Command = "dotnet",
    Arguments = new[] { "run", "--project", serverProjectPath },
});

// Creamos el cliente MCP (hace initialize internamente)
var client = await McpClient.CreateAsync(clientTransport);
```

```
var tools = await client.ListToolsAsync();

var result = await client.CallToolAsync(
    routed.Name,
    argsForSdk,
    cancellationToken: CancellationToken.None);
```

7.2 Websockets

Usaremos websocket cuando queramos comunicar de forma remota el cliente MCP y el servidor MCP.

Un **WebSocket** es un tipo de conexión de red que permite que cliente y servidor mantengan un canal abierto, persistente y bidireccional. Normalmente empieza como una petición HTTP que luego hace *upgrade* a WebSocket. A partir de ahí, ambas partes pueden enviarse mensajes en cualquier momento, sin seguir el patrón rígido de “request → response → cerrar conexión” típico de HTTP clásico. Además, el propio protocolo WebSocket ya incluye su mecanismo de *framing* de mensajes, así que tú solo te tienes que preocupar de mandar JSON (en el caso de MCP) y no de cómo separar un mensaje de otro en el stream.

En MCP puedes usar dos tipos de transporte principales: **stdio** o **WebSocket**. Con stdio, el cliente lanza el servidor MCP como un proceso local y se comunica con él mediante la entrada estándar (stdin) y la salida estándar (stdout), enviando y recibiendo mensajes JSON-RPC. Es un enfoque muy parecido al de LSP (Language Server Protocol) y encaja muy bien cuando tu MCP es simplemente otro binario que vive en la misma máquina que el cliente, por ejemplo un plugin que un IDE ejecuta junto al proyecto. Con WebSocket, en cambio, el servidor MCP vive en una dirección de red (wss://...) y el cliente se conecta a él a través de una conexión persistente de red, también hablando JSON-RPC por encima. En ambos casos, lo que viaja es lo mismo: mensajes JSON-RPC siguiendo la spec de MCP; solo cambia el “cable”.

La elección entre stdio y WebSocket depende sobre todo de **dónde vive tu servidor MCP y cómo quieres consumirlo**. Tiene sentido usar **WebSockets** cuando tu servidor MCP está desplegado en la nube o en otra máquina distinta al cliente, cuando quieres que varios clientes diferentes (un IDE, una aplicación web, otro servicio backend) se conecten al mismo MCP compartido o cuando estás en un navegador y no puedes lanzar procesos locales ni acceder a stdio. En ese contexto, WebSocket es el medio natural: conexión de red full-duplex, persistente, fácil de integrar en arquitecturas distribuidas. Por el contrario, **stdio** es ideal cuando el servidor MCP se ejecuta como proceso local que el propio cliente arranca. Es simple, no requiere puertos ni infraestructura de red adicional, y funciona muy bien para herramientas de desarrollo: el cliente (por ejemplo, un

IDE) lanza el ejecutable del MCP, le abre pipes de stdin/stdout y habla JSON-RPC por ahí.

¿Cuándo usar sockets/WebSocket en vez de stdio?

- Servicios **remotos** o **multi-cliente**.
- **Alto volumen** o **streaming binario** intensivo.
- Necesidad de **TLS**, balanceo, etc.

Para un **plugin local**, aislado, multiplataforma y simple como MCP, **stdio** es ideal.

```
// WebSocket
using var ws = new ClientWebSocket();
ws.Options.Proxy = null; // sin proxy de SO
ws.Options.UseDefaultCredentials = false;
// ws.Options.AddSubProtocol("jsonrpc"); // si tu server lo requiere

Console.WriteLine("[CLIENT] Conectando...");
await ws.ConnectAsync(uri, cts.Token);
Console.WriteLine("[CLIENT] Conectado. (enviendo initialize + tools/list)\n");
```

7.3 Por qué no usar API REST

Respecto a por qué **no se ha elegido una API REST** como mecanismo de comunicación MCP cliente ↔ servidor, hay varios motivos de diseño.

Primero, REST está pensado para un modelo **request/response unidireccional**: el cliente pide y el servidor responde, y normalmente la conexión se cierra. MCP, en cambio, necesita una comunicación más **interactiva y bidireccional**, donde el servidor pueda también iniciar mensajes (por ejemplo, sampling, “licitations”, preguntas al cliente, logs, streaming de resultados parciales, etc.). Eso encaja mucho mejor con un canal tipo WebSocket o stdio con JSON-RPC que con un API REST clásico.

Segundo, una API REST implicaría **HTTP + toda su sobrecarga** (cabeceras, rutas, verbos, CORS, infraestructura de servidor web) incluso para el caso más simple, que es un servidor MCP local lanzado como proceso. Uno de los objetivos de MCP es que puedas escribir un servidor que se ejecute como binario local, sin tener que montar un servidor HTTP, gestionar puertos, certificados, CORS, proxies, etc. Con stdio basta con leer y escribir JSON en stdin/stdout y listo. JSON-RPC encima de stdio o WebSocket proporciona justo lo que se necesita: mensajes estructurados, IDs para correlacionar peticiones y respuestas, y soporte natural para llamadas del servidor al cliente, sin todo el “equipaje” adicional de REST.

Por último, MCP no solo consiste en “exponer endpoints”, sino en un modelo más rico de **tools, resources, prompts, sampling, roots**, etc., con interacción

continua entre LLM, cliente y servidor. Ese modelo se ajusta mejor a un protocolo de mensajes (JSON-RPC) sobre un canal persistente que a una colección de endpoints REST independientes. Podrías intentar “meter MCP dentro de REST”, pero estarías reinventando algo muy parecido a JSON-RPC encima de HTTP, con más fricción y menos simetría. Por eso la spec opta por stdio/WebSocket + JSON-RPC como base, y deja REST para otras capas (por ejemplo, el propio servidor MCP puede internamente llamar a APIs REST, pero eso ya es detalle de implementación).