# Client Concepts

Understanding MCP client concepts

MCP clients are instantiated by host applications to communicate with particular MCP servers. The host application, like Claude.ai or an IDE, manages the overall user experience and coordinates multiple clients. Each client handles one direct communication with one server.

Understanding the distinction is important: the *host* is the application users interact with, while *clients* are the protocol-level components that enable server connections.

## Core Client Features

In addition to making use of context provided by servers, clients may provide several features to servers. These client features allow server authors to build richer interactions. For example, clients can allow MCP servers to request additional information from the user via elicitations. Clients can offer the following capabilities:

### Sampling

Sampling allows servers to request language model completions through the client, enabling agentic behaviors while maintaining security and user control.

#### Overview

Sampling enables servers to perform AI-dependent tasks without directly integrating with or paying for AI models. Instead, servers can request that the client—which already has AI model access—handle these tasks on their behalf. This approach puts the client in complete control of user permissions and security measures. Because sampling requests occur within the context of other operations—like a tool analyzing data—and are processed as separate model calls, they maintain clear boundaries between different contexts, allowing for more efficient use of the context window.

**Sampling flow:**

sequenceDiagram

participant LLM

participant User

participant Client

participant Server

Note over Server,Client: Server initiates sampling

Server->>Client: sampling/createMessage

Note over Client,User: Human-in-the-loop review

Client->>User: Present request for approval

User-->>Client: Review and approve/modify

Note over Client,LLM: Model interaction

Client->>LLM: Forward approved request

LLM-->>Client: Return generation

Note over Client,User: Response review

Client->>User: Present response for approval

User-->>Client: Review and approve/modify

Note over Server,Client: Complete request

Client-->>Server: Return approved response

The flow ensures security through multiple human-in-the-loop checkpoints. Users review and can modify both the initial request and the generated response before it returns to the server.

**Request parameters example:**

{

messages: [

{

role: "user",

content: "Analyze these flight options and recommend the best choice:\n" +

"[47 flights with prices, times, airlines, and layovers]\n" +

"User preferences: morning departure, max 1 layover"

}

],

modelPreferences: {

hints: [{

name: "claude-3-5-sonnet" // Suggested model

}],

costPriority: 0.3, // Less concerned about API cost

speedPriority: 0.2, // Can wait for thorough analysis

intelligencePriority: 0.9 // Need complex trade-off evaluation

},

systemPrompt: "You are a travel expert helping users find the best flights based on their preferences",

maxTokens: 1500

}

#### Example: Flight Analysis Tool

Consider a travel booking server with a tool called findBestFlight that uses sampling to analyze available flights and recommend the optimal choice. When a user asks "Book me the best flight to Barcelona next month," the tool needs AI assistance to evaluate complex trade-offs.

The tool queries airline APIs and gathers 47 flight options. It then requests AI assistance to analyze these options: "Analyze these flight options and recommend the best choice: [47 flights with prices, times, airlines, and layovers] User preferences: morning departure, max 1 layover."

The client asks the user: "Allow sampling request?" Upon approval, the AI evaluates trade-offs—like cheaper red-eye flights versus convenient morning departures. The tool uses this analysis to present the top three recommendations.

#### User Interaction Model

Sampling is designed with human-in-the-loop control as a fundamental principle. Users maintain oversight through several mechanisms:

**Approval controls**: Every sampling request needs explicit user consent. Clients show what the server wants to analyze and why. Users can approve, deny, or modify requests.

**Transparency features**: Clients display the exact prompt, model selection, and token limits. Users review AI responses before they return to the server.

**Configuration options**: Users can set model preferences, configure auto-approval for trusted operations, or require approval for everything. Clients may provide options to redact sensitive information. Users decide how much conversation context may be included in sampling requests through the includeContext parameter.

**Isolation**: Sampling requests are isolated from the main conversation context by default. Servers cannot access user conversations.

**Security considerations**: Both clients and servers must handle sensitive data appropriately during sampling. Clients should implement rate limiting and validate all message content. The human-in-the-loop design ensures that server-initiated AI interactions cannot compromise security or access sensitive data without explicit user consent.

### Roots

Roots define filesystem boundaries for server operations, allowing clients to specify which directories servers should focus on.

#### Overview

Roots are a mechanism for clients to communicate filesystem access boundaries to servers. They consist of file URIs that indicate directories where servers can operate, helping servers understand the scope of available files and folders. Rather than giving servers unrestricted filesystem access, roots guide them to relevant working directories while maintaining security boundaries.

**Root structure:**

{

"uri": "file:///Users/agent/travel-planning",

"name": "Travel Planning Workspace"

}

Roots are exclusively filesystem paths and always use the file:// URI scheme. They help servers understand project boundaries, workspace organization, and accessible directories. The roots list can be updated dynamically as users work with different projects or folders, with servers receiving notifications through roots/list\_changed when boundaries change.

It's important to note that while roots provide guidance to servers about where to operate, the client is always in full control of file access. Roots simply communicate intended boundaries—actual file access is always mediated by the client's security policies.

#### Example: Travel Planning Workspace

A travel agent working with multiple client trips benefits from roots to organize filesystem access. Consider a workspace with different directories for various aspects of travel planning.

The client provides filesystem roots to the travel planning server:

* file:///Users/agent/travel-planning - Main workspace containing all travel files
* file:///Users/agent/travel-templates - Reusable itinerary templates and resources
* file:///Users/agent/client-documents - Client passports and travel documents

When the agent creates a Barcelona itinerary, the server works within these boundaries—accessing templates, saving the new itinerary, and referencing client documents. It cannot access files outside these roots. Servers typically access files within roots by using relative paths from the root directories or by utilizing file search tools that respect the root boundaries.

If the agent opens an archive folder like file:///Users/agent/archive/2023-trips, the client updates the roots list via roots/list\_changed.

#### User Interaction Model

Roots are typically managed automatically by host applications based on user actions, though some applications may expose manual root management:

**Automatic root detection**: When users open folders, clients automatically expose them as roots. Opening a travel workspace gives servers access to itineraries and documents within that directory.

**Manual root configuration**: Advanced users can specify roots through configuration. For example, adding /travel-templates for reusable resources while excluding directories with financial records.

### Elicitation

Elicitation enables servers to request specific information from users during interactions, creating more dynamic and responsive workflows.

#### Overview

Elicitation provides a structured way for servers to gather necessary information on demand. Instead of requiring all information up front or failing when data is missing, servers can pause their operations to request specific inputs from users. This creates more flexible interactions where servers adapt to user needs rather than following rigid patterns.

**Elicitation flow:**

sequenceDiagram

participant User

participant Client

participant Server

Note over Server,Client: Server initiates elicitation

Server->>Client: elicitation/create

Note over Client,User: Human interaction

Client->>User: Present elicitation UI

User-->>Client: Provide requested information

Note over Server,Client: Complete request

Client-->>Server: Return user response

Note over Server: Continue processing with new information

The flow enables dynamic information gathering. Servers can request specific data when needed, users provide information through appropriate UI, and servers continue processing with the newly acquired context.

**Elicitation components example:**

{

method: "elicitation/requestInput",

params: {

message: "Please confirm your Barcelona vacation booking details:",

schema: {

type: "object",

properties: {

confirmBooking: {

type: "boolean",

description: "Confirm the booking (Flights + Hotel = $3,000)"

},

seatPreference: {

type: "string",

enum: ["window", "aisle", "no preference"],

description: "Preferred seat type for flights"

},

roomType: {

type: "string",

enum: ["sea view", "city view", "garden view"],

description: "Preferred room type at hotel"

},

travelInsurance: {

type: "boolean",

default: false,

description: "Add travel insurance ($150)"

}

},

required: ["confirmBooking"]

}

}

}

#### Example: Holiday Booking Approval

A travel booking server demonstrates elicitation's power through the final booking confirmation process. When a user has selected their ideal vacation package to Barcelona, the server needs to gather final approval and any missing details before proceeding.

The server elicits booking confirmation with a structured request that includes the trip summary (Barcelona flights June 15-22, beachfront hotel, total $3,000) and fields for any additional preferences—such as seat selection, room type, or travel insurance options.

As the booking progresses, the server elicits contact information needed to complete the reservation. It might ask for traveler details for flight bookings, special requests for the hotel, or emergency contact information.

#### User Interaction Model

Elicitation interactions are designed to be clear, contextual, and respectful of user autonomy:

**Request presentation**: Clients display elicitation requests with clear context about which server is asking, why the information is needed, and how it will be used. The request message explains the purpose while the schema provides structure and validation.

**Response options**: Users can provide the requested information through appropriate UI controls (text fields, dropdowns, checkboxes), decline to provide information with optional explanation, or cancel the entire operation. Clients validate responses against the provided schema before returning them to servers.

**Privacy considerations**: Elicitation never requests passwords or API keys. Clients warn about suspicious requests and let users review data before sending.