Introduction to MCP

What is MCP?

As AI systems evolve from simple chatbots to agents capable of complex tasks, they need a robust way to access tools and data. The Model Context Protocol (MCP) is an open standard that defines how AI agents (clients) talk to external systems (servers) in a consistent, auditable way.

MCP standardizes:

- Tool discovery and invocation (tools/list, tools/call)
- Prompt management (prompts/list, prompts/get)
- Resource access (resources/list, resources/read)
- Optional flows like sampling, elicitation, logging, and roots

For ColdFusion, MCP provides a way to:

- Call MCP servers from your CF app using a single, structured client API.
- Expose ColdFusion business logic, prompts, and resources as an MCP server that any MCP host can connect to.

Key components

Conceptually, MCP is built around three roles. It's worth keeping these straight, because ColdFusion can act as more than one of them:

MCP Host is the outer application that orchestrates the conversation and user experience. It embeds an MCP client and usually an LLM. Examples include Claude Desktop, Cursor, and ColdFusion web apps that maintain user sessions and call MCP tools as needed.

MCP Client is the protocol adapter that understands how to talk to MCP servers over HTTP or STDIO and how to translate between MCP JSON-RPC requests/responses and CF structures. In this guide, MCPClient(configData) is the main entry point for creating MCP clients.

MCP Server is the provider of capabilities. A server might wrap Jira APIs, a file system, or a set of CFML CFCs. It receives MCP methods such as tools/list, tools/call, prompts/get, etc., and returns standardized responses.

Client vs server in ColdFusion

In ColdFusion, the distinction between client and server is straightforward:

- Use the MCP Client portion of this guide when your CF app needs to call MCP servers (e.g., Jira MCP, Wiki MCP, CF-based MCP, third-party MCP servers).
- Use the MCP Server portion when your CF app should act as an MCP server, so agents and tools can call your CF logic and data.

Most teams will eventually do both: they'll integrate existing MCP servers (client side) and expose their own CF capabilities via MCP (server side).

MCP Client

What is an MCP Client?

An MCP client in ColdFusion is an object created by MCPClient(configData) that:

- Manages a connection to one or more MCP servers.
- Performs the initial handshake to exchange capabilities and server info.
- Exposes CF-friendly methods like listTools(), callTool(), listResources(), readResource(), listPrompts(), and getPrompt().
- Optionally forwards sampling requests to your LLM integration, and elicitation requests to your UI code.

You configure its behavior using a struct, so it integrates naturally with Application.cfc and environment-driven config patterns.

Create an MCP Client

To create a client, you build a configData struct and pass it to MCPClient(configData). The function returns an array of clients. For single-server configs, this array contains a single client; for multi-server configs, it can contain one client per server.

Key fields in configData:

- transport (or configFile for multi-server)
- clientInfo
- capabilities
- initializationTimeout, requestTimeout
- Optional consumer callbacks (sampling, elicitation, logging, root changes)

For example,

```
<cfscript>
    configData = {
        transport: {
            type: "http",
            url: "http://localhost:8500/mcp/server.cfm"
        },
        clientInfo: {
            name: "my-coldfusion-client",
            version: "1.0.0"
        },
        capabilities: {
            sampling: true,
```

```
            roots: true,
            elicitation: true
        },
        initializationTimeout: 10,
        requestTimeout: 30
    };

    mcpClient = MCPClient(configData);
</cfscript>
```

Transport configuration

Transports define how the client talks to the server. MCP supports several transports at the spec level; ColdFusion MCP focuses on:

- HTTP (remote or local over HTTP)
- STDIO (local subprocess)

Each has different trade-offs around deployment, security, and performance.

HTTP transport

HTTP transports are best when:

- The MCP server is remote and shared (e.g., an MCP in an internal environment or a public MCP).
- You want standard observability and network-level controls.
- You're exposing a CF MCP server to multiple clients.

For example,

```
<cfscript>
    configData = {
        transport: {
            type: "http",
            url: "https://remote.mcpservers.org/fetch/mcp"
        },
        clientInfo: {
            name: "http-client",
            version: "1.0.0"
        },
        requestTimeout: 30
    };

    mcpClient = MCPClient(configData)
```

```
</cfscript>
```

Parameters

Parameter	Type	Required	Description
type	String	Yes	Must be "http"
url	String	Yes	MCP endpoint URL (HTTP/HTTPS)

STDIO transport

STDIO transports works when:

- The MCP server is a local Node/Python/Java.
- You don't want to expose an HTTP endpoint.
- You want to ship a self-contained utility that runs next to CF.

For example,

```
<cfscript>
    configData = {
        transport: {
            type: "stdio",
            command: "/usr/local/bin/node",
            args: ["/Users/user/mcp-servers/server.js"],
            env: {
                apiKey: "sk-aaa",
                NODE_ENV: "production"
            }
        },
        clientInfo: {
            name: "stdio-client",
            version: "1.0.0"
        },
        requestTimeout: 60
    };

    mcpClient = MCPClient(configData)
</cfscript>
```

Parameters

Parameter	Type	Required	Description
type	String	Yes	Must be "stdio"

command	String	Yes	Executable path (e.g., node, python)
args	Array	No	Command-line arguments
env	Struct	No	Environment variables

Client capabilities

Client capabilities allow which optional features your MCP client supports. This influences how the server interacts with you.

Common flags:

- sampling – whether the client can handle sampling requests (LLM calls).
- roots – whether the client will manage roots for resource scoping.
- elicitation – whether the client will handle elicitation flows.

For example,

```
<cfscript>
    configData = {
        transport: { /* transport config */ },
        capabilities: {
            sampling: true,     // Enable sampling requests
            roots: true,        // Enable root resource management
            elicitation: true   // Enable user input requests
        }
    };
</cfscript>
```

You can gradually enable these features: for example, start with sampling = false, roots = false, elicitation = false and enable as you implement the corresponding callbacks.

Work with tools

Tools are the primary way your client triggers actions on the server side (e.g., fetch data, update records, transform documents).

List available tools – listTools()

```
<cfscript>
    tools = mcpClient.listTools();
    writeDump(tools);
</cfscript>
```

Each tool record typically includes:

- name – the tool's identifier.
- description – a human-readable explanation.
- inputSchema – a JSON-schema-like structure describing required and optional parameters.

Call a tool – callTool(toolName, params)

Tools are the primary way your client triggers actions on the server side (e.g., fetch data, update records, transform documents).

```
<cfscript>
    toolName   = "get_weather";
    toolParams = { location = "New York" };

    result = mcpClient.callTool(toolName, toolParams);
    writeDump(result);
</cfscript>
```

Work with resources

Resources are read-only objects like files, logs, or domain-specific data that the MCP server exposes.

List resources- listResources()

```
<cfscript>
    resources = mcpClient.listResources();
    writeDump(resources);
</cfscript>
```

Read a resource – readResource(uri)

```
<cfscript>
    uri = "file:///logs/app.log";

    if (mcpClient.isResourceReadable(uri)) {
        resData = mcpClient.readResource(uri);

        logText = "";
        if (arrayLen(resData.contents) GT 0 &&
            structKeyExists(resData.contents[1], "text")) {
            logText = resData.contents[1].text;
        }
```

```
        writeOutput("<pre>" & encodeForHtml(logText) & "</pre>");
    } else {
        writeOutput("Resource not readable: " & uri);
    }
</cfscript>
```

Work with prompts

Prompts are server-defined templates that describe reusable workflows, often with arguments. They are essential when multiple clients or teams must reuse the same high-quality prompt.

List prompts – listPrompts()

```
<cfscript>
    prompts = mcpClient.listPrompts();
    writeDump(prompts);
</cfscript>
```

Get a prompt- getPrompt({ name, arguments })

```
<cfscript>
    promptParams = {
        name      = "explain-code",
        arguments = {
            code     = "<cfquery ...>",
            language = "coldfusion"
        }
    };

    promptDef = mcpClient.getPrompt(promptParams);
    writeDump(promptDef);
</cfscript>
```

The server returns a definition that can include fully structured messages you can send to a chat model. This ensures consistent, centrally maintained prompts.

Sampling

Sampling is used when the MCP server wants you to call your own LLM and then return the completion back to the server.

Configuration

```
<cfscript>
    function mySamplingHandler(request) {
        var messages = request.messages;
        var userMessage = messages[1].content.text;

        // Call your LLM here (OpenAI, Bedrock, etc.)
        var llmResponse = callMyLLM(userMessage);

        // Return as struct with model name
        return {
            modelname: "gpt-4o",
            result: llmResponse
        };

        // Or return plain string (model name will be "UNKNOWN")
        // return llmResponse;
    }

    configData = {
        transport: { /* transport config */ },
        capabilities: {
            sampling: true
        },
        samplingConsumer: mySamplingHandler
    };

    mcpClient = MCPClient(configData);
</cfscript>
```

Sampling request format (from server)

```
{
    messages: [
        {
            role: "user",
            content: {
                type: "text",
                text: "What is the weather in New York?"
            }
        }
    ]
}
```

Struct response (with model name)

```
return {
    modelname: "gpt-4o",
    result: "The weather in New York is sunny, 75°F"
};
```

Servers send sampling requests with a messages array; your handler decides how to respond and which model to use.

Elicitation

Elicitation is the inverse of sampling: the server asks the client to collect additional user input according to a schema.

Configuration

```
<cfscript>
    function myElicitationHandler(request) {
        var message = request.message;
        var requestedSchema = request.requestedSchema;

        // Show UI to user to collect input
        // Return based on user action

        // User accepts:
        return {
            action: "accept",
            content: {
                checkAlternatives: "true",
                flexibleDates: "next_day"
            }
        };

        // User declines:
        // return { action: "decline" };

        // User cancels:
        // return { action: "cancel" };
    }

    configData = {
```

```
        transport: { /* transport config */ },
        capabilities: {
            elicitation: true
        },
        elicitationConsumer: myElicitationHandler
    };

    mcpClient = MCPClient(configData)
</cfscript>
```

Elicitation request format (from server)

```
{
    message: "Would you like to check alternative flights?",
    requestedSchema: {
        type: "object",
        properties: {
            checkAlternatives: { type: "string" },
            flexibleDates: { type: "string" }
        }
    }
}
```

Response format

Field	Type	Required	Values	Description
`action`	String	Yes	`"accept"`, `"decline"`, `"cancel"`	User's response
`content`	Struct	Conditional	Key-value pairs	Required if action is `"accept"`

Manage roots

Roots inform servers about relevant resource locations the client can access. Roots are the client's way of telling the server:

"Here are the URIs that define my resource boundaries."

They are critical for:

- Security: limiting which directories, tenants, or systems are considered in-scope.

- Performance: preventing the server from scanning irrelevant or massive resource spaces.
- Context switching: switching tenant or project context by swapping roots.

Conceptually, a root is a struct with at least name and uri:

Add a root

```
mcpClient.addRoot("file:///home/user/testcases", "Test Cases");
```

Get a root

```
root = mcpClient.getRoot("file:///home/user/testcases");

// Returns: { uri: "file:///...", name: "Test Cases" }
```

List roots

```
roots = mcpClient.listRoots();

// Returns: { roots: [{ uri: "...", name: "..." }] }
```

Remove a root

```
mcpClient.removeRoot("file:///home/user/testcases");
```

Notify root changes

```
mcpClient.rootsListChangedNotification();
```

Logging

Logging allows the MCP server to ship structured log events back to the client. This is extremely useful when:

- The MCP server is remote and you do not have direct log access.
- You want to surface MCP-related logs in your existing logging system.
- You need to diagnose failures in tool calls or resource reads.

You attach a loggingConsumer to your client config; the client calls it whenever the server emits logs.

For example,

```
<cfscript>
    function myLoggingHandler(loggingMessage) {
        var level = loggingMessage.level;  // e.g., "info", "error", "debug"
        var data = loggingMessage.data;

        writeLog(type=level, text=data);
```

```
        return "Log received";
    }

    configData = {
        transport: { /* transport config */ },
        loggingConsumer: myLoggingHandler
    };

    mcpClient = MCPClient(configData);
</cfscript>
```

Configure bulk servers

Load multiple servers from a JSON configuration file. When you have multiple MCP servers to talk to (e.g., Jira, Wiki, GitHub, internal MCPs), it's easier to externalize them into a JSON file than to hard-code them in CFML. This also decouples environment changes (e.g., dev vs prod) from your CF code.

You can supply a configFile key in configData, pointing to a JSON file that lists MCP servers and their transports.

For example, **mcpServers.json**

```
{
    "mcpServers": {
        "weather-server": {
            "url": "http://localhost:8500/mcp/weather.cfm",
            "disabled": false
        },
        "database-server": {
            "url": "http://localhost:8500/mcp/database.cfm",
            "disabled": false
        },
        "external-api": {
            "url": "https://api.example.com/mcp",
            "disabled": true
        }
    }
}
```

Load multiple servers

```
<cfscript>
    configData = {
        configFile: "/path/to/mcpServers.json",
        clientInfo: {
            name: "bulk-client",
            version: "1.0.0"
        },
        capabilities: {
            sampling: true,
            roots: true
        },
        requestTimeout: 30
    };

    // Returns array of MCP clients (one per enabled server)
    mcpClients = MCPClient(configData);
</cfscript>
```

Complete example

```
<cfscript>
    function samplingHandler(request) {
    // Your LLM integration
        return {
            modelname: "gpt-4o",
            result: callOpenAI(request.messages)
        };
    }

    function elicitationHandler(request) {
        // Your UI for user input
        return {
            action: "accept",
            content: collectUserInput(request.message)
        };
    }

    function loggingHandler(loggingMessage) {
        writeLog(type=loggingMessage.level, text=loggingMessage.data);
        return "Logged";
    }
```

```
    configData = {
        transport: {
            type: "http",
            url: "http://localhost:8500/mcp/server.cfm"
        },
        clientInfo: {
            name: "my-ai-client",
            version: "1.0.0"
        },
        capabilities: {
            sampling: true,
            roots: true,
            elicitation: true
        },
        initializationTimeout: 10,
        requestTimeout: 30,
        samplingConsumer: samplingHandler,
        elicitationConsumer: elicitationHandler,
        loggingConsumer: loggingHandler
    };

    mcpClient = MCPClient(configData);

    // Use the client
    tools = mcpClient.listTools();
    result = mcpClient.callTool({
        name: "get_weather",
        arguments: { location: "New York" }
    });
</cfscript>
```

MCP Server

What is an MCP Server?

An MCP server is the “provider side” of MCP: it receives JSON-RPC requests and responds with tools, prompts, resource information, and results. In ColdFusion, an MCP server wraps your CFCs, prompts, and resources so that external MCP clients can use them in their own workflows. An MCP Server exposes ColdFusion components (CFCs) as tools, provides prompt templates, and serves resources to MCP clients.

From an architecture standpoint, a CF MCP server lets you define a clear boundary:

- MCP clients see only your MCP server surface (tools, prompts, resources).
- Internally, you can map those to any CF logic, services, and data sources you want.

Create an MCP Server

You create an MCP server by building a configData struct and passing it to MCPServer(configData). This is usually done once in onApplicationStart and stored in application scope.

For example,

```
<cfscript>
    configData = {
        serverInfo: {
            name: "Healthcare MCP Server",
            version: "1.0.0"
        },
        capabilities: {
            tools: true,
            prompts: true,
            resources: true
        },
        tools: [
            { cfc: "mcp.tools.healthcareTools" },
            { cfc: "mcp.tools.emailTool" }
        ],
        prompts: [
            {
                name: "generate_summary",
                title: "Generate Patient Summary",
                description: "Generate a comprehensive patient summary",
                arguments: [
                    {
                        name: "patientName",
                        description: "Name of the patient",
                        required: true
                    }
                ],
                template: "Generate a summary for patient {patientName}"
            }
        ],
        resources: resourcesArray,
        cfcCaching: true,
```

```
        corsEnabled: true
    };

    application.mcpServer = MCPServer(configData);

    // Handle incoming requests
    application.mcpServer.handleRequest();
</cfscript>
```

Server configuration parameters

Server info

serverInfo tells clients what this server is called and which version they're interacting with.

```
<cfscript>
    serverInfo: {
        name: "My MCP Server",
        version: "1.0.0"
    }
</cfscript>
```

Parameter	Type	Required	Description
`name`	String	Yes	Server identifier
`version`	String	Yes	Server version

This information appears in initialize responses, which helps with debugging, logging, and compatibility checks.

Capabilities

capabilities declare which MCP features your server implements.

```
<cfscript>
    capabilities: {
        tools: true,        // Enable tool execution
        prompts: true,      // Enable prompt templates
        resources: true     // Enable resource access
    }
</cfscript>
```

Capability	Description
`tools`	Server can execute CFC methods as tools

prompts	Server provides prompt templates
resources	Server provides access to resources

A simple server might only set tools = true. A more feature-rich one sets all three to true.

Configure tools

Tools are where your CF business logic gets exposed to MCP. For each CFC listed in tools, the MCP framework can treat public or remote methods as callable tools.

```
<cfscript>
    tools: [
        { cfc: "mcp.tools.healthcareTools" },
        { cfc: "mcp.tools.emailTool" },
        { cfc: "mcp.tools.pdfTools" }
    ]
</cfscript>
```

CFC example

component displayname="healthcareTools" hint="Healthcare tools" {

```
/**
 * Fetch users by search term
 * @searchTerm The search term to find users
 * @return Struct with user data
 */
remote struct function fetchUsers(required string searchTerm) {
    // Implementation
    return {
        success: true,
        users: queryResults,
        totalFound: queryResults.recordCount
    };
}

/**
 * Get patient details
 * @patientId Patient identifier
 * @return Patient information struct
 */
```

```
remote struct function getPatientDetails(required string patientId) {
    // Implementation
    return {
        id: patientId,
        name: "John Doe",
        age: 45
    };
}


}
```

Rules

1. Only functions with `remote` access modifier are exposed
2. Function parameters become tool arguments
3. Return values must be structs or simple types
4. Tool name format: `componentName.functionName` (e.g., `healthcareTools.fetchUsers`)

Configure prompts

Prompts defined on the server allow you to codify workflows in one place and share them with many clients.

For example,

```
<cfscript>
    prompts: [
        {
            name: "generate_discharge_summary",
            title: "Generate Discharge Summary",
            description: "Generate a comprehensive discharge summary for a
patient",
            arguments: [
                {
                    name: "patientName",
                    description: "Name of the patient",
                    required: true
                },
                {
                    name: "includeLabResults",
                    description: "Include lab results in summary",
                    required: false
```

```
                }
            ],
            template: "Generate a comprehensive discharge summary for patient
{patientName}. Include diagnosis, treatment, medications, and follow-up
instructions."
        }
    ]
</cfscript>
```

Prompt parameters

Parameter	Type	Required	Description
`name`	String	Yes	Unique prompt identifier
`title`	String	Yes	Human-readable display name
`description`	String	Yes	What the prompt does
`arguments`	Array	No	Expected parameters for the template
`template`	String	Yes	Prompt text with `{placeholders}`

Argument parameters

Parameter	Type	Required	Description
`name`	String	Yes	Parameter name (used in placeholders)
`description`	String	Yes	Parameter description
`required`	Boolean	Yes	Whether parameter is mandatory

Clients can then retrieve and use these templates by name, ensuring consistency across applications.

Configure resources

Resources provide access to files and data. Resources expose documents and other data sources over MCP. In CF, you typically delegate reading to a helper such as resourceReader.

For example,

```
<cfscript>
    // Create resource reader helper
    resourceReader = createObject("component", "mcp.tools.ResourceReader");

    resources: [
        {
```

```
            uri: "healthcare://patient-data/P001/ct-scan",
            name: "ct_scan_p001",
            title: "Latest Chest CT Scan - John Miller",
            description: "Chest CT scan results for patient P001",
            mimeType: "application/pdf",
            readResourceHandler: function(readResourceReq) {
                return resourceReader.readResource("ct-scan-p001.pdf");
            }
        },
        {
            uri: "healthcare://templates/discharge-summary",
            name: "discharge_template",
            title: "Discharge Summary Template",
            description: "Standard discharge summary template",
            mimeType: "text/plain",
            readResourceHandler: function(readResourceReq) {
                return resourceReader.readResource("discharge-template.txt");
            }
        }
    ]
</cfscript>
```

Resource parameters

Parameter	Type	Required	Description
`uri`	String	Yes	Unique resource identifier (custom protocol)
`name`	String	Yes	Short identifier
`title`	String	Yes	Human-readable display name
`description`	String	Yes	Resource description
`mimeType`	String	Yes	Content type (e.g., `"application/pdf"`, `"text/plain"`)
`readResourceHandler`	Function	Yes	Function that returns resource content

Common MIME types

MIME Type	Description	Use Case
`application/pdf`	PDF documents	Medical records, reports
`text/plain`	Plain text	Notes, templates
`application/json`	JSON data	Structured data
`text/html`	HTML documents	Formatted reports
`image/jpeg`	JPEG images	X-rays, photos

CFC caching

Tool CFCs can be expensive to instantiate repeatedly. The cfcCaching flag lets you cache them:

```
<cfscript>
    cfcCaching = true;
</cfscript>
```

With caching:

- The first time a tool is used, its CFC is instantiated and cached.
- Subsequent calls reuse that instance.

This is almost always what you want in production, unless your CFCs have side effects that require fresh instances.

CORS configuration

Enables browser-based access to the MCP server.

`corsEnabled: true`

Request handling (HTTP entry point)

After you have a configured MCP server object, you still need a way for external clients to reach it. The simplest pattern is:

1. Create the server in Application.cfc and store it in application.mcpServer.
2. Implement a CFC with a remote method that acts as the HTTP “front door” for MCP messages.

For example,

```
<cfscript>
    // Resource reader helper
    resourceReader = createObject("component", "mcp.tools.ResourceReader");

    // Define resources
    resourcesArray = [
        {
            uri: "healthcare://patient-data/P001/ct-scan",
            name: "ct_scan_p001",
            title: "CT Scan - Patient P001",
```

```
            description: "Latest chest CT scan results",
            mimeType: "application/pdf",
            readResourceHandler: function(req) {
                return resourceReader.readResource("ct-scan-p001.pdf");
            }
        }
    ];

    // Define prompts
    promptsArray = [
        {
            name: "generate_discharge_summary",
            title: "Generate Discharge Summary",
            description: "Generate comprehensive discharge summary",
            arguments: [
                {
                    name: "patientName",
                    description: "Name of the patient",
                    required: true
                }
            ],
            template: "Generate discharge summary for patient {patientName}
including diagnosis, treatment, and follow-up."
        }
    ];

    // Create server configuration
    configData = {
        serverInfo: {
            name: "Healthcare MCP Server",
            version: "1.0.0"
        },
        capabilities: {
            tools: true,
            prompts: true,
            resources: true
        },
        tools: [
            { cfc: "mcp.tools.healthcareTools" },
            { cfc: "mcp.tools.emailTool" },
            { cfc: "mcp.tools.pdfTools" }
        ],
        prompts: promptsArray,
        resources: resourcesArray,
```

```
        cfcCaching: true,
        corsEnabled: true
    };

    // Create and register server
    if (!structKeyExists(application, "mcpServer")) {
        application.mcpServer = MCPServer(configData);
    }

    // Handle incoming requests
    application.mcpServer.handleRequest();
</cfscript>
```

Registering server with a client

Once the server is deployed, register it with an MCP client.

For example,

```
<cfscript>
    // Client configuration pointing to your server
    clientConfig = {
        transport: {
            type: "http",
            url: "http://localhost:8500/mcp/healthcareServer.cfm"
        },
        clientInfo: {
            name: "healthcare-client",
            version: "1.0.0"
        }
    };

    mcpClient = MCPClient(clientConfig);
</cfscript>
```

Error handling and timeouts

MCP adds a layer of structured protocol on top of your logic. When things go wrong, it's important to distinguish between:

1. Transport errors: timeouts, network failures, HTTP 503s, STDIO crashes.
2. JSON-RPC errors: invalid method, bad params, internal server errors, represented by a top-level error object.

3. Tool-level errors: business logic failures, flagged via result.isError and possibly custom fields.

Timeouts

In configData, you define:

- initializationTimeout – how long the client waits for the initialize handshake to complete.
- requestTimeout – how long the client waits for each MCP request (tool calls, resource reads, etc).

For example,

```
<cfscript>
    configData = {
        transport: { /* ... */ },
        clientInfo: { name : "stable-client", version : "1.0.0" },
        capabilities: { sampling : false, roots : false, elicitation : false },
        initializationTimeout : 10,
        requestTimeout        : 10
    };
</cfscript>
```

If a server is slow or unresponsive, the client should raise an exception when a timeout is reached.

Security and best practices

MCP often sits right next to sensitive systems: ticketing, documentation, source control, customer data, and logs. That makes security and operational hygiene non-negotiable.

Protect secrets

Many MCP servers expect tokens or API keys via environment variables (env) or HTTP headers. In CF:

- Never log token values. Mask them if you must log their presence.
- Keep secrets out of source control; load them from secure storage or environment.
- Make sure any configFile with secrets or endpoints has restricted permissions.

Use roots to enforce scope

- Roots are not just hints. They should be part of your security design:
- Align roots with tenants, projects, or security domains.
- Change roots when a user's context changes.
- Avoid giving "one big root" that covers everything unless that is truly necessary.

This reduces the risk of returning or summarizing resources the current user shouldn't see.

Allow list trusted MCP Servers

Avoid letting users specify arbitrary MCP endpoints:

- Maintain an allow-list of MCP servers (in mcpServers.json or CF config) and validate that any url or command you use belongs to that list.
- If you accept user input that influences MCP behavior, never let it override the actual url or command.

This prevents CF from calling malicious or untrusted MCP servers.

Be careful with STDIO servers

STDIO MCP servers run under the same OS user as ColdFusion and therefore inherit its access:

- Ensure the script you run (e.g., server.js) is from a trusted source and maintained securely.
- Lock down the directories where these scripts reside (no world-writable directories).
- Keep environment minimal: only the tokens and settings the server absolutely needs.

Harden HTTP endpoints

For CF MCP servers exposed via HTTP:

- Use HTTPS in production environments.
- Put them behind auth, firewall rules, or internal networks if necessary.
- Log enough information to trace misuse but not so much that you leak sensitive data.

By combining capabilities, roots, and careful transport configuration, you can keep your MCP integration aligned with your broader security posture.

Frequently asked questions

Can I use multiple MCP servers at once?
Yes. Define multiple server configurations (for example in `Application.cfc`), create one MCP client per server at startup, and retrieve the appropriate client (like `application.mcpClients["jira"]` or `["wiki"]`) whenever you need to call a tool.

What happens if an MCP server returns an error?
Wrap every MCP call in `try/catch`, inspect the returned struct for JSON-RPC errors (`error` object) and tool-level flags (`result.isError`), log the details, and throw a clear application-level exception so your code can handle the failure gracefully.

Is MCP secure?
MCP itself supports secure patterns, but security depends on your implementation: keep API tokens out of logs and source, only connect to trusted/vetted MCP servers, lock down local stdio servers in production, and keep all MCP components patched and current.

Can MCP access my database directly?
Not automatically. MCP servers can only access databases if you explicitly implement tools that connect to them. When you do, enforce least privilege on DB credentials and validate all inputs before executing queries.

Do I need an AI model to use MCP from ColdFusion?
No. You can call MCP tools directly from CFML as structured API calls. AI models are optional and mainly used when you want natural-language orchestration or agentic behavior.

Can I use MCP in development but disable it in production?
Yes. Guard MCP initialization with environment checks or feature flags so that certain MCP clients (especially experimental or local ones) are only created in dev/stage and not in production.

How do I know which tools a server exposes?
After building the client, call `mcpClient.listTools()`. This returns a list of tool definitions (names, descriptions, and possibly schemas) that you can inspect, log, or display on an admin page.

Can I restrict which tools my CF app is allowed to call?
Yes. Route all tool invocations through a small dispatcher that only permits a curated set of tool names, and reject or ignore anything outside that allow-list.

What happens if a local MCP server hangs or crashes?
Your CF calls will time out or throw exceptions. Use `initializationTimeout` and `requestTimeout` in the MCP client config, and always wrap calls in `try/catch` with proper logging so you can detect and recover from these conditions.

Is it safe to point CF at public/open MCP servers?
Only if they're explicitly approved. Treat public MCP servers like any third-party service: review their security posture, avoid sending sensitive data, and don't expose internal tokens or systems to untrusted endpoints.

Can MCP servers be versioned and upgraded without breaking CF?
Yes, but you should track their versions and rerun your MCP integration tests after each upgrade. If a server changes tool names or schemas, you may need to update your ColdFusion code accordingly.

Can I share one MCP client across multiple requests?
Yes. It is common to build MCP clients once (in `onApplicationStart`) and store them in `application` scope, reusing them across requests, assuming you follow the standard thread-safe usage patterns.

Does MCP support streaming responses?
The MCP spec supports streaming (for example, streamable HTTP), but whether you can use it from ColdFusion depends on the current CF MCP client implementation. Unless explicitly documented, design your integration assuming non-streaming responses.