On this page

MCP servers with the Gemini CLI

This document provides a guide to configuring and using Model Context Protocol (MCP) servers with the Gemini CLI.

What is an MCP server?

An MCP server is an application that exposes tools and resources to the Gemini CLI through the Model Context Protocol, allowing it to interact with external systems and data sources. MCP servers act as a bridge between the Gemini model and your local environment or other services like APIs.

An MCP server enables the Gemini CLI to:

- **Discover tools:** List available tools, their descriptions, and parameters through standardized schema definitions.
- **Execute tools:** Call specific tools with defined arguments and receive structured responses.
- **Access resources:** Read data from specific resources (though the Gemini CLI primarily focuses on tool execution).

With an MCP server, you can extend the Gemini CLI's capabilities to perform actions beyond its built-in features, such as interacting with databases, APIs, custom scripts, or specialized workflows.

Core Integration Architecture

The Gemini CLI integrates with MCP servers through a sophisticated discovery and execution system built into the core package (`packages/core/src/tools/`):

Discovery Layer (`mcp-client.ts`)

The discovery process is orchestrated by `discoverMcpTools()` , which:

1. **Iterates through configured servers** from your `settings.json` `mcpServers` configuration
2. **Establishes connections** using appropriate transport mechanisms (Stdio, SSE, or Streamable HTTP)
3. **Fetches tool definitions** from each server using the MCP protocol
4. **Sanitizes and validates** tool schemas for compatibility with the Gemini API
5. **Registers tools** in the global tool registry with conflict resolution

Execution Layer (`mcp-tool.ts`)

Each discovered MCP tool is wrapped in a `DiscoveredMCPTool` instance that:

- **Handles confirmation logic** based on server trust settings and user preferences
- **Manages tool execution** by calling the MCP server with proper parameters
- **Processes responses** for both the LLM context and user display
- **Maintains connection state** and handles timeouts

Transport Mechanisms

The Gemini CLI supports three MCP transport types:

- **Stdio Transport:** Spawns a subprocess and communicates via stdin/stdout
- **SSE Transport:** Connects to Server-Sent Events endpoints
- **Streamable HTTP Transport:** Uses HTTP streaming for communication

---

How to set up your MCP server

The Gemini CLI uses the `mcpServers` configuration in your `settings.json` file to locate and connect to MCP servers. This configuration supports multiple servers with different transport mechanisms.

Configure the MCP server in settings.json

You can configure MCP servers at the global level in the `~/.gemini/settings.json` file or in your project's root directory, create or open the `.gemini/settings.json` file. Within the file, add the `mcpServers` configuration block.

Configuration Structure

Add an `mcpServers` object to your `settings.json` file:

```json
{ ...file contains other config objects
  "mcpServers": {
    "serverName": {
      "command": "path/to/server",
      "args": ["--arg1", "value1"],
      "env": {
        "API_KEY": "$MY_API_TOKEN"
      },
      "cwd": "./server-directory",
      "timeout": 30000,
      "trust": false
    }
  }
}
```

Configuration Properties

Each server configuration supports the following properties:

Required (one of the following)

- **`command`** (string): Path to the executable for Stdio transport
- **`url`** (string): SSE endpoint URL (e.g., `"http://localhost:8080/sse"`)
- **`httpUrl`** (string): HTTP streaming endpoint URL

Optional

- **`args`** (string[]): Command-line arguments for Stdio transport
- **`headers`** (object): Custom HTTP headers when using `url` or `httpUrl`
- **`env`** (object): Environment variables for the server process. Values can reference environment variables using `$VAR_NAME` or `${VAR_NAME}` syntax

- `cwd` (string): Working directory for Stdio transport
- `timeout` (number): Request timeout in milliseconds (default: 600,000ms = 10 minutes)
- `trust` (boolean): When `true`, bypasses all tool call confirmations for this server (default: `false`)
- `includeTools` (string[]): List of tool names to include from this MCP server. When specified, only the tools listed here will be available from this server (whitelist behavior). If not specified, all tools from the server are enabled by default.
- `excludeTools` (string[]): List of tool names to exclude from this MCP server. Tools listed here will not be available to the model, even if they are exposed by the server. **Note:** `excludeTools` takes precedence over `includeTools` - if a tool is in both lists, it will be excluded.

OAuth Support for Remote MCP Servers

The Gemini CLI supports OAuth 2.0 authentication for remote MCP servers using SSE or HTTP transports. This enables secure access to MCP servers that require authentication.

Automatic OAuth Discovery

For servers that support OAuth discovery, you can omit the OAuth configuration and let the CLI discover it automatically:

```json
{
  "mcpServers": {
    "discoveredServer": {
      "url": "https://api.example.com/sse"
    }
  }
}
```

The CLI will automatically:

- Detect when a server requires OAuth authentication (401 responses)
- Discover OAuth endpoints from server metadata
- Perform dynamic client registration if supported
- Handle the OAuth flow and token management

Authentication Flow

When connecting to an OAuth-enabled server:

1. **Initial connection attempt** fails with 401 Unauthorized
2. **OAuth discovery** finds authorization and token endpoints
3. **Browser opens** for user authentication (requires local browser access)
4. **Authorization code** is exchanged for access tokens
5. **Tokens are stored** securely for future use
6. **Connection retry** succeeds with valid tokens

Browser Redirect Requirements

Important: OAuth authentication requires that your local machine can:

- Open a web browser for authentication
- Receive redirects on `http://localhost:7777/oauth/callback`

This feature will not work in:

- Headless environments without browser access
- Remote SSH sessions without X11 forwarding
- Containerized environments without browser support

Managing OAuth Authentication

Use the `/mcp auth` command to manage OAuth authentication:

```bash
# List servers requiring authentication
/mcp auth

# Authenticate with a specific server
/mcp auth serverName

# Re-authenticate if tokens expire
/mcp auth serverName
```

OAuth Configuration Properties

- `enabled` (boolean): Enable OAuth for this server
- `clientId` (string): OAuth client identifier (optional with dynamic registration)
- `clientSecret` (string): OAuth client secret (optional for public clients)

- `authorizationUrl` (string): OAuth authorization endpoint (auto-discovered if omitted)
- `tokenUrl` (string): OAuth token endpoint (auto-discovered if omitted)
- `scopes` (string[]): Required OAuth scopes
- `redirectUri` (string): Custom redirect URI (defaults to `http://localhost:7777/oauth/callback`)
- `tokenParamName` (string): Query parameter name for tokens in SSE URLs

Token Management

OAuth tokens are automatically:

- **Stored securely** in `~/.gemini/mcp-oauth-tokens.json`
- **Refreshed** when expired (if refresh tokens are available)
- **Validated** before each connection attempt
- **Cleaned up** when invalid or expired

Authentication Provider Type

You can specify the authentication provider type using the `authProviderType` property:

- `authProviderType` (string): Specifies the authentication provider. Can be one of the following:
  - `dynamic_discovery` (default): The CLI will automatically discover the OAuth configuration from the server.
  - `google_credentials`: The CLI will use the Google Application Default Credentials (ADC) to authenticate with the server. When using this provider, you must specify the required scopes.

```json
{
  "mcpServers": {
    "googleCloudServer": {
      "httpUrl": "https://my-gcp-service.run.app/mcp",
      "authProviderType": "google_credentials",
      "oauth": {
        "scopes": ["https://www.googleapis.com/auth/userinfo.email"]
      }
    }
  }
}
```

Example Configurations

Python MCP Server (Stdio)

```json
{
  "mcpServers": {
    "pythonTools": {
      "command": "python",
      "args": ["-m", "my_mcp_server", "--port", "8080"],
      "cwd": "./mcp-servers/python",
      "env": {
        "DATABASE_URL": "$DB_CONNECTION_STRING",
        "API_KEY": "${EXTERNAL_API_KEY}"
      },
      "timeout": 15000
    }
  }
}
```

Node.js MCP Server (Stdio)

```json
{
  "mcpServers": {
    "nodeServer": {
      "command": "node",
      "args": ["dist/server.js", "--verbose"],
      "cwd": "./mcp-servers/node",
      "trust": true
    }
  }
}
```

Docker-based MCP Server

```json
{
  "mcpServers": {
    "dockerizedServer": {
      "command": "docker",
      "args": [
        "run",
```

```
        "-i",
        "--rm",
        "-e",
        "API_KEY",
        "-v",
        "${PWD}:/workspace",
        "my-mcp-server:latest"
      ],
      "env": {
        "API_KEY": "$EXTERNAL_SERVICE_TOKEN"
      }
    }
  }
}
```

HTTP-based MCP Server

json

```
{
  "mcpServers": {
    "httpServer": {
      "httpUrl": "http://localhost:3000/mcp",
      "timeout": 5000
    }
  }
}
```

HTTP-based MCP Server with Custom Headers

json

```
{
  "mcpServers": {
    "httpServerWithAuth": {
      "httpUrl": "http://localhost:3000/mcp",
      "headers": {
        "Authorization": "Bearer your-api-token",
        "X-Custom-Header": "custom-value",
        "Content-Type": "application/json"
      },
      "timeout": 5000
    }


```

```
    }
  }
```

MCP Server with Tool Filtering

```json
{
  "mcpServers": {
    "filteredServer": {
      "command": "python",
      "args": ["-m", "my_mcp_server"],
      "includeTools": ["safe_tool", "file_reader", "data_processor"],
      // "excludeTools": ["dangerous_tool", "file_deleter"],
      "timeout": 30000
    }
  }
}
```

Discovery Process Deep Dive

When the Gemini CLI starts, it performs MCP server discovery through the following detailed process:

1. Server Iteration and Connection

For each configured server in `mcpServers`:

1. **Status tracking begins:** Server status is set to `CONNECTING`
2. **Transport selection:** Based on configuration properties:
   - `httpUrl` → `StreamableHTTPClientTransport`
   - `url` → `SSEClientTransport`
   - `command` → `StdioClientTransport`
3. **Connection establishment:** The MCP client attempts to connect with the configured timeout
4. **Error handling:** Connection failures are logged and the server status is set to `DISCONNECTED`

2. Tool Discovery

Upon successful connection:

1. **Tool listing:** The client calls the MCP server's tool listing endpoint
2. **Schema validation:** Each tool's function declaration is validated
3. **Tool filtering:** Tools are filtered based on `includeTools` and `excludeTools` configuration
4. **Name sanitization:** Tool names are cleaned to meet Gemini API requirements:
   - Invalid characters (non-alphanumeric, underscore, dot, hyphen) are replaced with underscores
   - Names longer than 63 characters are truncated with middle replacement (`___`)

3. Conflict Resolution

When multiple servers expose tools with the same name:

1. **First registration wins:** The first server to register a tool name gets the unprefixed name
2. **Automatic prefixing:** Subsequent servers get prefixed names: `serverName__toolName`
3. **Registry tracking:** The tool registry maintains mappings between server names and their tools

4. Schema Processing

Tool parameter schemas undergo sanitization for Gemini API compatibility:

- `$schema` **properties** are removed
- `additionalProperties` are stripped
- `anyOf` **with** `default` have their default values removed (Vertex AI compatibility)
- **Recursive processing** applies to nested schemas

5. Connection Management

After discovery:

- **Persistent connections:** Servers that successfully register tools maintain their connections
- **Cleanup:** Servers that provide no usable tools have their connections closed

- **Status updates:** Final server statuses are set to `CONNECTED` or `DISCONNECTED`

Tool Execution Flow

When the Gemini model decides to use an MCP tool, the following execution flow occurs:

1. Tool Invocation

The model generates a `FunctionCall` with:

- **Tool name:** The registered name (potentially prefixed)
- **Arguments:** JSON object matching the tool's parameter schema

2. Confirmation Process

Each `DiscoveredMCPTool` implements sophisticated confirmation logic:

Trust-based Bypass

```typescript
if (this.trust) {
  return false; // No confirmation needed
}
```

Dynamic Allow-listing

The system maintains internal allow-lists for:

- **Server-level:** `serverName` → All tools from this server are trusted
- **Tool-level:** `serverName.toolName` → This specific tool is trusted

User Choice Handling

When confirmation is required, users can choose:

- **Proceed once:** Execute this time only
- **Always allow this tool:** Add to tool-level allow-list
- **Always allow this server:** Add to server-level allow-list
- **Cancel:** Abort execution

3. Execution

Upon confirmation (or trust bypass):

1. **Parameter preparation:** Arguments are validated against the tool's schema
2. **MCP call:** The underlying `CallableTool` invokes the server with:

```typescript
const functionCalls = [
  {
    name: this.serverToolName, // Original server tool name
    args: params,
  },
];
```

3. **Response processing:** Results are formatted for both LLM context and user display

4. Response Handling

The execution result contains:

- `llmContent` : Raw response parts for the language model's context
- `returnDisplay` : Formatted output for user display (often JSON in markdown code blocks)

How to interact with your MCP server

Using the `/mcp` Command

The `/mcp` command provides comprehensive information about your MCP server setup:

```bash
/mcp
```

This displays:

- **Server list:** All configured MCP servers
- **Connection status:** `CONNECTED` , `CONNECTING` , or `DISCONNECTED`

- **Server details:** Configuration summary (excluding sensitive data)
- **Available tools:** List of tools from each server with descriptions
- **Discovery state:** Overall discovery process status

Example `/mcp` Output

```
MCP Servers Status:

📡 pythonTools (CONNECTED)
  Command: python -m my_mcp_server --port 8080
  Working Directory: ./mcp-servers/python
  Timeout: 15000ms
  Tools: calculate_sum, file_analyzer, data_processor

🔌 nodeServer (DISCONNECTED)
  Command: node dist/server.js --verbose
  Error: Connection refused

🐬 dockerizedServer (CONNECTED)
  Command: docker run -i --rm -e API_KEY my-mcp-server:latest
  Tools: docker__deploy, docker__status

Discovery State: COMPLETED
```

Tool Usage

Once discovered, MCP tools are available to the Gemini model like built-in tools. The model will automatically:

1. **Select appropriate tools** based on your requests
2. **Present confirmation dialogs** (unless the server is trusted)
3. **Execute tools** with proper parameters
4. **Display results** in a user-friendly format

Status Monitoring and Troubleshooting

Connection States

The MCP integration tracks several states:

Server Status (`MCPServerStatus`)

- `DISCONNECTED` : Server is not connected or has errors
- `CONNECTING` : Connection attempt in progress
- `CONNECTED` : Server is connected and ready

Discovery State (`MCPDiscoveryState`)

- `NOT_STARTED` : Discovery hasn't begun
- `IN_PROGRESS` : Currently discovering servers
- `COMPLETED` : Discovery finished (with or without errors)

Common Issues and Solutions

Server Won't Connect

Symptoms: Server shows `DISCONNECTED` status

Troubleshooting:

1. **Check configuration:** Verify `command` , `args` , and `cwd` are correct
2. **Test manually:** Run the server command directly to ensure it works
3. **Check dependencies:** Ensure all required packages are installed
4. **Review logs:** Look for error messages in the CLI output
5. **Verify permissions:** Ensure the CLI can execute the server command

No Tools Discovered

Symptoms: Server connects but no tools are available

Troubleshooting:

1. **Verify tool registration:** Ensure your server actually registers tools
2. **Check MCP protocol:** Confirm your server implements the MCP tool listing correctly
3. **Review server logs:** Check stderr output for server-side errors
4. **Test tool listing:** Manually test your server's tool discovery endpoint

Tools Not Executing

Symptoms: Tools are discovered but fail during execution

Troubleshooting:

1. **Parameter validation:** Ensure your tool accepts the expected parameters
2. **Schema compatibility:** Verify your input schemas are valid JSON Schema
3. **Error handling:** Check if your tool is throwing unhandled exceptions
4. **Timeout issues:** Consider increasing the `timeout` setting

Sandbox Compatibility

Symptoms: MCP servers fail when sandboxing is enabled

Solutions:

1. **Docker-based servers:** Use Docker containers that include all dependencies
2. **Path accessibility:** Ensure server executables are available in the sandbox
3. **Network access:** Configure sandbox to allow necessary network connections
4. **Environment variables:** Verify required environment variables are passed through

Debugging Tips

1. **Enable debug mode:** Run the CLI with `--debug` for verbose output
2. **Check stderr:** MCP server stderr is captured and logged (INFO messages filtered)
3. **Test isolation:** Test your MCP server independently before integrating
4. **Incremental setup:** Start with simple tools before adding complex functionality
5. **Use `/mcp` frequently:** Monitor server status during development

---

Important Notes

Security Considerations

- **Trust settings:** The `trust` option bypasses all confirmation dialogs. Use cautiously and only for servers you completely control
- **Access tokens:** Be security-aware when configuring environment variables containing API keys or tokens

- **Sandbox compatibility:** When using sandboxing, ensure MCP servers are available within the sandbox environment
- **Private data:** Using broadly scoped personal access tokens can lead to information leakage between repositories

Performance and Resource Management

- **Connection persistence:** The CLI maintains persistent connections to servers that successfully register tools
- **Automatic cleanup:** Connections to servers providing no tools are automatically closed
- **Timeout management:** Configure appropriate timeouts based on your server's response characteristics
- **Resource monitoring:** MCP servers run as separate processes and consume system resources

Schema Compatibility

- **Property stripping:** The system automatically removes certain schema properties (`$schema` , `additionalProperties`) for Gemini API compatibility
- **Name sanitization:** Tool names are automatically sanitized to meet API requirements
- **Conflict resolution:** Tool name conflicts between servers are resolved through automatic prefixing

This comprehensive integration makes MCP servers a powerful way to extend the Gemini CLI's capabilities while maintaining security, reliability, and ease of use.