Model Context Protocol

☰ Concepts › Tools

Concepts

Tools

Enable LLMs to perform actions through your server

Tools are a powerful primitive in the Model Context Protocol (MCP) that enable servers to expose executable functionality to clients. Through tools, LLMs can interact with external systems, perform computations, and take actions in the real world.

> ⓘ Tools are designed to be **model-controlled**, meaning that tools are exposed from servers to clients with the intention of the AI model being able to automatically invoke them (with a human in the loop to grant approval).

Overview

Tools in MCP allow servers to expose executable functions that can be invoked by clients and used by LLMs to perform actions. Key aspects of tools include:

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

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

Tool definition structure

Each tool is defined with the following structure:

```
{
  name: string;          // Unique identifier for the tool
  description?: string;  // Human-readable description
  inputSchema: {         // JSON Schema for the tool's parameters
    type: "object",
    properties: { ... }  // Tool-specific parameters
  }
}
```

Implementing tools

Here’s an example of implementing a basic tool in an MCP server:

TypeScript Python

```
const server = new Server({
  name: "example-server",
  version: "1.0.0"
}, {
  capabilities: {
    tools: {}
  }
});

// Define available tools
server.setRequestHandler(ListToolsRequestSchema, async () => {
  return {
    tools: [{
      name: "calculate_sum",
      description: "Add two numbers together",
      inputSchema: {
        type: "object",
        properties: {
          a: { type: "number" },
          b: { type: "number" }
        },
        required: ["a", "b"]
      }
    }]
  };
});

// Handle tool execution
server.setRequestHandler(CallToolRequestSchema, async (request) => {
  if (request.params.name === "calculate_sum") {
    const { a, b } = request.params.arguments;
    return {
      content: [
        {
          type: "text",
          text: String(a + b)
        }
      ]
    };
  }
  throw new Error("Tool not found");
});
```

Example tool patterns

Here are some examples of types of tools that a server could provide:

System operations

Tools that interact with the local system:

```
{
  name: "execute_command",
  description: "Run a shell command",
  inputSchema: {
    type: "object",
    properties: {
      command: { type: "string" },
      args: { type: "array", items: { type: "string" } }
    }
  }
}
```

API integrations

Tools that wrap external APIs:

```
{
  name: "github_create_issue",
  description: "Create a GitHub issue",
  inputSchema: {
    type: "object",
    properties: {
      title: { type: "string" },
      body: { type: "string" },
      labels: { type: "array", items: { type: "string" } }
    }
  }
}
```

Data processing

Tools that transform or analyze data:

```
{
  name: "analyze_csv",
  description: "Analyze a CSV file",
  inputSchema: {
    type: "object",
    properties: {
      filepath: { type: "string" },
      operations: {
        type: "array",
        items: {
          enum: ["sum", "average", "count"]
        }
      }
    }
  }
}
```

Best practices

When implementing tools:

1. Provide clear, descriptive names and descriptions
2. Use detailed JSON Schema definitions for parameters
3. Include examples in tool descriptions to demonstrate how the model should use them
4. Implement proper error handling and validation
5. Use progress reporting for long operations
6. Keep tool operations focused and atomic
7. Document expected return value structures
8. Implement proper timeouts
9. Consider rate limiting for resource-intensive operations
10. Log tool usage for debugging and monitoring

Security considerations

When exposing tools:

Input validation

Validate all parameters against the schema

- Sanitize file paths and system commands
- Validate URLs and external identifiers
- Check parameter sizes and ranges
- Prevent command injection

Access control

- Implement authentication where needed
- Use appropriate authorization checks
- Audit tool usage
- Rate limit requests
- Monitor for abuse

Error handling

- Don’t expose internal errors to clients
- Log security-relevant errors
- Handle timeouts appropriately
- Clean up resources after errors
- Validate return values

Tool discovery and updates

MCP supports dynamic tool discovery:

1. Clients can list available tools at any time
2. Servers can notify clients when tools change using `notifications/tools/list_changed`
3. Tools can be added or removed during runtime
4. Tool definitions can be updated (though this should be done carefully)

Error handling

Tool errors should be reported within the result object, not as MCP protocol-level errors. This allows the LLM to see and potentially handle the error. When a tool encounters an error:

1. Set `isError` to `true` in the result
2. Include error details in the `content` array

Here’s an example of proper error handling for tools:

TypeScript Python

```
try {
  // Tool operation
  const result = performOperation();
  return {
    content: [
      {
        type: "text",
        text: `Operation successful: ${result}`
      }
    ]
  };
} catch (error) {
  return {
    isError: true,
    content: [
      {
        type: "text",
        text: `Error: ${error.message}`
      }
    ]
  };
}
```

This approach allows the LLM to see that an error occurred and potentially take corrective action or request human intervention.

Testing tools

A comprehensive testing strategy for MCP tools should cover:

- **Functional testing**: Verify tools execute correctly with valid inputs and handle invalid inputs appropriately
- **Integration testing**: Test tool interaction with external systems using both real and mocked dependencies
- **Security testing**: Validate authentication, authorization, input sanitization, and rate limiting
- **Performance testing**: Check behavior under load, timeout handling, and resource cleanup
- **Error handling**: Ensure tools properly report errors through the MCP protocol and clean up resources

Was this page helpful? 👍 Yes 👎 No

‹ Prompts Sampling ›