MCP Server Implementation Guide for ServiceNow Integration

Part 3: MCP Protocol Implementation and Tools

Document Series

This implementation guide is organized into five parts:

1. Part 1: Introduction and Overview
2. Part 2: Server Foundation - Core Infrastructure Setup
3. **Part 3: MCP Protocol Implementation and Tools** (This Document)
4. Part 4: OAuth 2.1 Implementation
5. Part 5: Appendices and Recommendations

Introduction

This document covers the MCP protocol implementation and tool patterns that form the functional core of your server. Using the house building metaphor, this is where we construct the "walls" (MCP protocol structure) and the "interior" (tool capabilities).

What You'll Build:

- MCP protocol endpoint with JSON-RPC 2.0 routing
- Initialize handshake for protocol negotiation
- Notifications handler for client communication
- Tools discovery mechanism (tools/list)
- Tool execution framework (tools/call)
- Custom tool implementation patterns

Building on the Foundation:

Part 2 established the server infrastructure (HTTP server, middleware, configuration, storage). Part 3 builds on that foundation by implementing:

- The MCP protocol layer that ServiceNow communicates with
- The tool framework that provides functional capabilities to AI agents

Looking Ahead:

Part 4 will add the OAuth 2.1 security layer (the "roof") that protects the MCP protocol and tools implemented in this part.

MCP Protocol Implementation

Overview

The MCP (Model Context Protocol) implementation provides the core communication layer that enables ServiceNow AI agents to discover and execute tools on your server. This section covers the protocol handlers that form the structural foundation of your MCP server.

MCP Endpoint Structure

Single Endpoint Pattern

The MCP protocol uses a single HTTP POST endpoint that handles all protocol methods through JSON-RPC 2.0 message routing:

Endpoint: `POST /mcp`

Supported Methods:

- `initialize` - Protocol handshake and capability negotiation (public - no auth)
- `notifications/initialized` - Client initialization notification (no response)
- `tools/list` - Retrieve available tools (requires authentication)
- `tools/call` - Execute a specific tool (requires authentication)

Authentication Requirements:

- `initialize`: Public (no authentication required)
- All other methods: Require valid OAuth access token

Note: Authentication implementation (Bearer token validation, JWT verification) is covered in **Part 4: OAuth 2.1 Implementation, Section 10 (Authentication Middleware)**. The `authenticate_request()` function referenced in this part's code examples is detailed there.

Implementation Approach

The MCP protocol uses a single HTTP POST endpoint that handles all protocol methods through JSON-RPC 2.0 message routing. This unified endpoint pattern simplifies ServiceNow configuration and follows modern API design principles. The server examines the `method` field in each request to determine which handler to invoke, similar to how GraphQL uses a single endpoint with query-based routing.

Key considerations:

- All MCP operations go through one endpoint: `POST /mcp`
- Method differentiation via JSON-RPC `method` field in request body
- Authentication applies selectively (initialize is public, others require auth)
- Consistent JSON-RPC 2.0 response structure for all methods
- Protocol version must match ServiceNow's supported version (2025-06-18 or 2025-03-26)

Pseudocode:

```
// Define the single MCP endpoint
FUNCTION handle_mcp_endpoint(request, response):
    // Parse JSON-RPC request
    jsonrpc_version = request.body.jsonrpc
    method = request.body.method
    params = request.body.params
    request_id = request.body.id

    // Validate JSON-RPC version
    IF jsonrpc_version != "2.0":
        RETURN json_rpc_error(response, -32600, "Invalid JSON-RPC version", request_id)
    END IF

    // Route based on method
    IF method == "initialize":
        // Public method - no authentication required
        RETURN handle_initialize(request, response)

    ELSE IF method == "notifications/initialized":
        // Notification - acknowledge with 200 OK, no response body
        response.STATUS(200)
        RETURN response.END()

    ELSE IF method == "tools/list":
        // Requires authentication
        IF NOT authenticate_request(request):
            RETURN unauthorized_error(response)
        END IF
        RETURN handle_tools_list(request, response)

    ELSE IF method == "tools/call":
        // Requires authentication
        IF NOT authenticate_request(request):
            RETURN unauthorized_error(response)
        END IF
        RETURN handle_tools_call(request, response)

    ELSE:
        // Unknown method
        RETURN json_rpc_error(response, -32601, "Method not found: " + method, request_id)
    END IF
END FUNCTION
```

```
// Register the endpoint
app.POST('/mcp', handle_mcp_endpoint)
```

JavaScript:

```
javascript
```

```
const express = require('express');
const app = express();

// Main MCP endpoint - single POST endpoint for all protocol methods
app.post('/mcp', async (req, res) => {
  const { jsonrpc, method, params, id } = req.body;

  // Validate JSON-RPC version
  if (jsonrpc !== '2.0') {
    return res.status(200).json({
      jsonrpc: '2.0',
      error: {
        code: -32600,
        message: 'Invalid JSON-RPC version'
      },
      id: id
    });
  }

  try {
    // Route based on method
    if (method === 'initialize') {
      // Public method - no authentication required
      return handleInitialize(req, res);
    }

    if (method === 'notifications/initialized') {
      // Notification - acknowledge with 200 OK, no response body
      console.log('[MCP] Client sent initialized notification');
      return res.status(200).send();
    }

    // All other methods require authentication
    authenticateRequest(req, res, async () => {
      if (method === 'tools/list') {
        return handleToolsList(req, res);
      } else if (method === 'tools/call') {
        return handleToolsCall(req, res);
      } else {
        // Unknown method
        return res.status(200).json({
          jsonrpc: '2.0',
          error: {
```

```
          code: -32601,
          message: `Method not found: ${method}`
        },
        id: id
      });
    }
  });

} catch (error) {
  console.error('[MCP] Error handling request:', error);
  return res.status(500).json({
    jsonrpc: '2.0',
    error: {
      code: -32603,
      message: 'Internal server error',
      data: { details: error.message }
    },
    id: id
  });
}
});
```

TypeScript:

typescript

```typescript
import express, { Application, Request, Response } from 'express';

const app: Application = express();

// JSON-RPC request structure
interface JsonRpcRequest {
  jsonrpc: string;
  method: string;
  params?: any;
  id?: number | string;
}

// Main MCP endpoint - single POST endpoint for all protocol methods
app.post('/mcp', async (req: Request, res: Response) => {
  const { jsonrpc, method, params, id }: JsonRpcRequest = req.body;

  // Validate JSON-RPC version
  if (jsonrpc !== '2.0') {
    return res.status(200).json({
      jsonrpc: '2.0',
      error: {
        code: -32600,
        message: 'Invalid JSON-RPC version'
      },
      id: id
    });
  }

  try {
    // Route based on method
    if (method === 'initialize') {
      // Public method - no authentication required
      return handleInitialize(req, res);
    }

    if (method === 'notifications/initialized') {
      // Notification - acknowledge with 200 OK, no response body
      console.log('[MCP] Client sent initialized notification');
      return res.status(200).send();
    }

    // All other methods require authentication
    authenticateRequest(req, res, async () => {
```

```
    if (method === 'tools/list') {
      return handleToolsList(req, res);
    } else if (method === 'tools/call') {
      return handleToolsCall(req, res);
    } else {
      // Unknown method
      return res.status(200).json({
        jsonrpc: '2.0',
        error: {
          code: -32601,
          message: `Method not found: ${method}`
        },
        id: id
      });
    }
  });

} catch (error) {
  console.error('[MCP] Error handling request:', error);
  return res.status(500).json({
    jsonrpc: '2.0',
    error: {
      code: -32603,
      message: 'Internal server error',
      data: { details: (error as Error).message }
    },
    id: id
  });
}
});
```

Initialize Method Handler

Protocol Handshake

The initialize method establishes the connection between ServiceNow and your MCP server:

Request Format:

```
json
```

```
{
  "jsonrpc": "2.0",
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": {},
    "clientInfo": {
      "name": "ServiceNow MCP Client",
      "version": "1.0"
    }
  },
  "id": 1
}
```

Response Requirements:

- Protocol version matching ServiceNow support (2025-06-18 or 2025-03-26)
- Capabilities object listing supported features
- Server information (name, version)

Protocol Version Note:

This guide uses `2025-03-26` in code examples, which was validated with ServiceNow Zurich Patch 4+. ServiceNow's official documentation states support for `2025-06-18`. Both versions work in practice with current ServiceNow releases.

Version Selection Guidance:

- Use `2025-03-26` if deploying to Zurich Patch 4+ environments
- Use `2025-06-18` for maximum compatibility across ServiceNow versions
- Differences between these versions are minimal and do not affect ServiceNow integration

Implementation Approach

The initialize method is the first interaction between ServiceNow and your MCP server, establishing protocol version compatibility and advertising server capabilities. This handshake occurs before authentication and determines what features ServiceNow can use. The response must declare your protocol version (matching ServiceNow's support) and list capabilities (tools, resources, prompts). ServiceNow uses this information to enable appropriate UI features and workflow options.

Key considerations:

- Always responds with protocol version ServiceNow supports (2025-06-18 or 2025-03-26)

- Public endpoint—no authentication required (enables ServiceNow to test connectivity)
- Advertise only capabilities you actually implement (tools are required, resources/prompts optional)
- Include server name and version for debugging and monitoring
- Response format is strictly defined by MCP specification

Pseudocode:

```
FUNCTION handle_initialize(request, response):
    // Extract client information from params
    client_info = request.body.params.clientInfo
    client_protocol_version = request.body.params.protocolVersion

    LOG("Initialize request received from: " + client_info.name)
    LOG("Client protocol version: " + client_protocol_version)

    // Build initialize response
    initialize_response = {
        jsonrpc: "2.0",
        result: {
            protocolVersion: "2025-03-26",  // Match ServiceNow support
            capabilities: {
                tools: {
                    listChanged: false  // Tool list is static
                },
                logging: {}  // Support logging utility
                // Note: Don't advertise resources or prompts if not implemented
            },
            serverInfo: {
                name: "Your MCP Server Name",
                version: "1.0.0"
            }
        },
        id: request.body.id
    }

    LOG("Initialize response sent - connection established")
    response.JSON(initialize_response)
END FUNCTION
```

JavaScript:

```
javascript
```

```
function handleInitialize(req, res) {
  const { params, id } = req.body;

  // Extract client information
  const clientInfo = params?.clientInfo;
  const clientProtocolVersion = params?.protocolVersion;

  console.log('[MCP] Initialize request received');
  console.log(`[MCP] Client: ${clientInfo?.name} v${clientInfo?.version}`);
  console.log(`[MCP] Client protocol version: ${clientProtocolVersion}`);

  // Build initialize response
  const response = {
    jsonrpc: '2.0',
    result: {
      protocolVersion: '2025-03-26',  // Match ServiceNow support
      capabilities: {
        tools: {
          listChanged: false  // Tool list is static
        },
        logging: {}  // Support logging utility
        // Note: Don't advertise resources or prompts if not implemented
      },
      serverInfo: {
        name: 'MCP Gateway Server',
        version: '1.0.0'
      }
    },
    id: id
  };

  console.log('[MCP] Initialize response sent - connection established');
  res.json(response);
}
```

TypeScript:

```
typescript
```

```typescript
import { Request, Response } from 'express';

// Initialize params structure
interface InitializeParams {
  protocolVersion: string;
  capabilities: Record<string, any>;
  clientInfo: {
    name: string;
    version: string;
  };
}

function handleInitialize(req: Request, res: Response): void {
  const { params, id } = req.body;

  // Extract client information
  const initParams = params as InitializeParams;
  const clientInfo = initParams?.clientInfo;
  const clientProtocolVersion = initParams?.protocolVersion;

  console.log('[MCP] Initialize request received');
  console.log(`[MCP] Client: ${clientInfo?.name} v${clientInfo?.version}`);
  console.log(`[MCP] Client protocol version: ${clientProtocolVersion}`);

  // Build initialize response
  const response = {
    jsonrpc: '2.0',
    result: {
      protocolVersion: '2025-03-26',  // Match ServiceNow support
      capabilities: {
        tools: {
          listChanged: false  // Tool list is static
        },
        logging: {}  // Support logging utility
        // Note: Don't advertise resources or prompts if not implemented
      },
      serverInfo: {
        name: 'MCP Gateway Server',
        version: '1.0.0'
      }
    },
    id: id
  };
```

```
  console.log('[MCP] Initialize response sent - connection established');
  res.json(response);
}
```

Notifications Handler

[This section will be populated as we build out the implementation guidance sections]

Notifications Handler

One-Way Notification

The `notifications/initialized` method is sent by ServiceNow after completing its initialization:

Request Format:

```
json

{
  "jsonrpc": "2.0",
  "method": "notifications/initialized",
  "params": {}
}
```

Response Requirements:

- HTTP 200 OK status
- No response body required

Implementation Approach

The notifications/initialized method is a one-way message from ServiceNow indicating it has completed its initialization process and is ready to use the MCP server. Unlike other MCP methods, notifications do not require a response body—simply acknowledge with HTTP 200 OK. This lightweight pattern follows the fire-and-forget notification model common in event-driven architectures.

Key considerations:

- No response body required (just HTTP 200 OK)
- Notification indicates ServiceNow is ready to make tool calls
- Optional logging for debugging and monitoring

- Does not require authentication (though ServiceNow may send tokens anyway)
- Failure to handle this gracefully may cause ServiceNow client errors

Pseudocode:

```
FUNCTION handle_notifications_initialized(request, response):
    LOG("Client sent initialized notification - ServiceNow is ready")

    // Acknowledge with 200 OK, no response body
    response.STATUS(200)
    response.END()
END FUNCTION
```

JavaScript:

```javascript
function handleNotificationsInitialized(req, res) {
  console.log('[MCP] Client sent initialized notification - ServiceNow is ready');

  // Acknowledge with 200 OK, no response body
  res.status(200).send();
}

// This handler is called from the main /mcp endpoint router
// when method === 'notifications/initialized'
```

TypeScript:

```typescript
import { Request, Response } from 'express';

function handleNotificationsInitialized(req: Request, res: Response): void {
  console.log('[MCP] Client sent initialized notification - ServiceNow is ready');

  // Acknowledge with 200 OK, no response body
  res.status(200).send();
}

// This handler is called from the main /mcp endpoint router
// when method === 'notifications/initialized'
```

Tool Implementation Patterns

Overview

Tools are the primary mechanism through which ServiceNow AI agents interact with your MCP server. This section covers how to implement the tool discovery and execution handlers, as well as patterns for creating custom tools that extend your server's capabilities.

Authentication Note: Both `tools/list` and `tools/call` require authentication via OAuth access tokens. The authentication mechanism is implemented in **Part 4: OAuth 2.1 Implementation**. This part focuses on the tool handlers themselves, assuming authentication is already in place.

Tools List Handler

Tool Discovery

The `tools/list` method returns the catalog of available tools to ServiceNow:

Request Format:

```json
{
  "jsonrpc": "2.0",
  "method": "tools/list",
  "params": {},
  "id": 2
}
```

Authentication: Required (must include valid OAuth access token in Authorization header)

Response Requirements:

- Array of tool definitions
- Each tool must include: name, description, inputSchema (JSON Schema)

Implementation Approach

The tools/list handler returns the catalog of available tools to ServiceNow, enabling AI agents to discover what capabilities your server provides. This method requires authentication and responds with an array of tool definitions, each including a name, description, and JSON Schema defining expected inputs. ServiceNow uses this information to present tool options to users and validate parameters before calling tools/call.

Key considerations:

- Requires authentication (ServiceNow must have valid OAuth token)
- Returns array of tool definitions in the response
- Each tool must have valid JSON Schema for inputSchema
- Tool names should be descriptive and unique
- Response is typically static (unless tools are dynamically registered)
- Called once during ServiceNow connection initialization, then cached

Pseudocode:

```
FUNCTION handle_tools_list(request, response):
    LOG("Tools list request received")

    // Define available tools
    tools = [
        {
            name: "llm_generate",
            description: "Generate text using local LLM",
            inputSchema: {
                type: "object",
                properties: {
                    prompt: {
                        type: "string",
                        description: "The prompt to send to the LLM"
                    },
                    model: {
                        type: "string",
                        description: "Model to use",
                        enum: ["tinyllama", "phi3"],
                        default: "tinyllama"
                    }
                },
                required: ["prompt"]
            }
        },
        {
            name: "file_read",
            description: "Read contents of a file",
            inputSchema: {
                type: "object",
                properties: {
                    filename: {
                        type: "string",
                        description: "Name of file to read"
                    }
                },
                required: ["filename"]
            }
        }
        // ... additional tools
    ]

    // Build JSON-RPC response
```

```
    tools_response = {
        jsonrpc: "2.0",
        result: {
            tools: tools
        },
        id: request.body.id
    }

    LOG("Returned " + tools.LENGTH + " tool(s) to client")
    response.JSON(tools_response)
END FUNCTION
```

JavaScript:

```
javascript
```

```
async function handleToolsList(req, res) {
  console.log('[MCP] Tools list request received');

  // Define available tools
  const tools = [
    {
      name: 'llm_generate',
      description: 'Generate text using local LLM',
      inputSchema: {
        type: 'object',
        properties: {
          prompt: {
            type: 'string',
            description: 'The prompt to send to the LLM'
          },
          model: {
            type: 'string',
            description: 'Model to use',
            enum: ['tinyllama', 'phi3'],
            default: 'tinyllama'
          }
        },
        required: ['prompt']
      }
    },
    {
      name: 'file_read',
      description: 'Read contents of a file',
      inputSchema: {
        type: 'object',
        properties: {
          filename: {
            type: 'string',
            description: 'Name of file to read'
          }
        },
        required: ['filename']
      }
    }
    // ... additional tools
  ];

  // Build JSON-RPC response
```

```
  const response = {
    jsonrpc: '2.0',
    result: {
      tools: tools
    },
    id: req.body.id
  };

  console.log(`[MCP] Returned ${tools.length} tool(s) to client`);
  res.json(response);
}
```

TypeScript:

```typescript
```

```typescript
import { Request, Response } from 'express';

// Tool definition structure
interface ToolDefinition {
  name: string;
  description: string;
  inputSchema: {
    type: string;
    properties: Record<string, any>;
    required: string[];
  };
}

async function handleToolsList(req: Request, res: Response): Promise<void> {
  console.log('[MCP] Tools list request received');

  // Define available tools
  const tools: ToolDefinition[] = [
    {
      name: 'llm_generate',
      description: 'Generate text using local LLM',
      inputSchema: {
        type: 'object',
        properties: {
          prompt: {
            type: 'string',
            description: 'The prompt to send to the LLM'
          },
          model: {
            type: 'string',
            description: 'Model to use',
            enum: ['tinyllama', 'phi3'],
            default: 'tinyllama'
          }
        },
        required: ['prompt']
      }
    },
    {
      name: 'file_read',
      description: 'Read contents of a file',
      inputSchema: {
        type: 'object',
```

```
        properties: {
          filename: {
            type: 'string',
            description: 'Name of file to read'
          }
        },
        required: ['filename']
      }
    }
    // ... additional tools
  ];

  // Build JSON-RPC response
  const response = {
    jsonrpc: '2.0',
    result: {
      tools: tools
    },
    id: req.body.id
  };

  console.log(`[MCP] Returned ${tools.length} tool(s) to client`);
  res.json(response);
}
```

Tools Call Handler

Tool Execution

The `tools/call` method executes a specific tool requested by ServiceNow:

Request Format:

```
json
```

```
{
 "jsonrpc": "2.0",
 "method": "tools/call",
 "params": {
  "name": "llm_generate",
  "arguments": {
   "prompt": "What is 5 + 3?",
   "model": "tinyllama"
  }
 },
 "id": 3
}
```

Authentication: Required (must include valid OAuth access token in Authorization header)

Response Requirements:

- Success: Return result with content array
- Failure: Return error with code and message

Implementation Approach

The tools/call handler executes a specific tool requested by ServiceNow, passing provided arguments to the tool implementation and returning the result. This handler acts as a router, validating the tool name, extracting arguments, invoking the appropriate tool function, and formatting the response. Proper error handling is critical—tool execution failures should return structured errors rather than crashing the server.

Key considerations:

- Requires authentication (ServiceNow must have valid OAuth token)
- Extract tool name and arguments from params
- Validate tool name exists before attempting execution
- Pass arguments to tool implementation function
- Catch and format tool execution errors gracefully
- Return results in MCP content array format
- Log tool calls for debugging and audit purposes

Pseudocode:

```
FUNCTION handle_tools_call(request, response):
    // Extract tool name and arguments
    tool_name = request.body.params.name
    tool_arguments = request.body.params.arguments
    request_id = request.body.id

    LOG("Tool call request: " + tool_name)

    TRY:
        // Route to appropriate tool implementation
        IF tool_name == "llm_generate":
            result = execute_llm_generate(tool_arguments)
        ELSE IF tool_name == "file_read":
            result = execute_file_read(tool_arguments)
        ELSE IF tool_name == "file_write":
            result = execute_file_write(tool_arguments)
        ELSE IF tool_name == "file_list":
            result = execute_file_list(tool_arguments)
        ELSE:
            THROW_ERROR("Unknown tool: " + tool_name)
        END IF

        // Build success response
        success_response = {
            jsonrpc: "2.0",
            result: result,
            id: request_id
        }

        LOG("Tool execution successful: " + tool_name)
        response.JSON(success_response)

    CATCH error:
        LOG_ERROR("Tool execution failed (" + tool_name + "): " + error.message)

        // Build error response
        error_response = {
            jsonrpc: "2.0",
            error: {
                code: -32603,
                message: "Tool execution failed: " + error.message
            },
            id: request_id
```

```
    }

    response.JSON(error_response)
  END TRY
END FUNCTION
```

JavaScript:

```
javascript
```

```
async function handleToolsCall(req, res) {
  const { params, id } = req.body;
  const { name: toolName, arguments: toolArguments } = params;

  console.log(`[MCP] Tool call request: ${toolName}`);

  try {
    let result;

    // Route to appropriate tool implementation
    switch (toolName) {
      case 'llm_generate':
        result = await executeLLMGenerate(toolArguments);
        break;

      case 'file_read':
        result = await executeFileRead(toolArguments);
        break;

      case 'file_write':
        result = await executeFileWrite(toolArguments);
        break;

      case 'file_list':
        result = await executeFileList(toolArguments);
        break;

      default:
        throw new Error(`Unknown tool: ${toolName}`);
    }

    // Build success response
    const response = {
      jsonrpc: '2.0',
      result: result,
      id: id
    };

    console.log(`[MCP] Tool execution successful: ${toolName}`);
    res.json(response);

  } catch (error) {
    console.error(`[MCP] Tool execution failed (${toolName}):`, error.message);
```

```
    // Build error response
    res.json({
      jsonrpc: '2.0',
      error: {
        code: -32603,
        message: `Tool execution failed: ${error.message}`
      },
      id: id
    });
  }
}
```

TypeScript:

typescript

```typescript
import { Request, Response } from 'express';

// Tool call params structure
interface ToolCallParams {
  name: string;
  arguments: Record<string, any>;
}

async function handleToolsCall(req: Request, res: Response): Promise<void> {
  const { params, id } = req.body;
  const { name: toolName, arguments: toolArguments } = params as ToolCallParams;

  console.log(`[MCP] Tool call request: ${toolName}`);

  try {
    let result: any;

    // Route to appropriate tool implementation
    switch (toolName) {
      case 'llm_generate':
        result = await executeLLMGenerate(toolArguments);
        break;

      case 'file_read':
        result = await executeFileRead(toolArguments);
        break;

      case 'file_write':
        result = await executeFileWrite(toolArguments);
        break;

      case 'file_list':
        result = await executeFileList(toolArguments);
        break;

      default:
        throw new Error(`Unknown tool: ${toolName}`);
    }

    // Build success response
    const response = {
      jsonrpc: '2.0',
      result: result,
```

```
      id: id
    };

    console.log(`[MCP] Tool execution successful: ${toolName}`);
    res.json(response);

  } catch (error) {
    console.error(`[MCP] Tool execution failed (${toolName}):`, (error as Error).message);

    // Build error response
    res.json({
      jsonrpc: '2.0',
      error: {
        code: -32603,
        message: `Tool execution failed: ${(error as Error).message}`
      },
      id: id
    });
  }
}
```

Creating Tools

Creating Tools

Tool Structure and Best Practices

Tools are the functional capabilities your MCP server exposes to ServiceNow AI agents. Each tool consists of three essential components: a unique name, a human-readable description, and a JSON Schema defining expected inputs. Well-designed tools balance flexibility with clear constraints, providing ServiceNow with enough information to validate inputs before execution while remaining simple enough for AI agents to understand and use effectively.

Tool Definition Components

Every tool must include:

1. **Name** (string, required)
    - Unique identifier for the tool
    - Use descriptive, action-oriented names (e.g., "search_documents", "create_ticket")

- Snake_case or camelCase convention
- Must match the name used in tools/call routing

2. **Description** (string, required)
    - Human-readable explanation of what the tool does
    - Used by ServiceNow to help users understand tool purpose
    - Keep concise but informative (1-2 sentences)
    - Include expected behavior and constraints
3. **Input Schema** (object, required)
    - JSON Schema (Draft 7 or later) defining expected parameters
    - Must include `type: "object"` at root level
    - Define each parameter in `properties` object
    - List required parameters in `required` array
    - Provide descriptions for each property

JSON Schema Essentials

ServiceNow validates tool arguments against your inputSchema before calling tools/call. Use these JSON Schema patterns:

Basic Types:

```
String: { type: "string", description: "..." }
Number: { type: "number", description: "..." }
Boolean: { type: "boolean", description: "..." }
Array: { type: "array", items: { type: "string" }, description: "..." }
Object: { type: "object", properties: {...}, description: "..." }
```

Common Constraints:

```
Enum (limited choices): { type: "string", enum: ["option1", "option2"], description: "..." }
Default value: { type: "string", default: "value", description: "..." }
Min/Max (numbers): { type: "number", minimum: 0, maximum: 100, description: "..." }
Pattern (regex): { type: "string", pattern: "^[A-Z]+$", description: "..." }
String length: { type: "string", minLength: 1, maxLength: 500, description: "..." }
```

Required Parameters:

```
inputSchema: {
  type: "object",
  properties: {
    param1: { type: "string" },
    param2: { type: "number" }
  },
  required: ["param1"]  // param2 is optional
}
```

Response Format

Tool execution must return results in the MCP content array format:

Success Response Structure:

```
{
  content: [
    {
      type: "text",
      text: "Your result data here"
    }
  ]
}
```

Content Types Supported:

- `text`: Plain text or JSON string
- `image`: Base64-encoded image data (with media_type)
- `resource`: Reference to MCP resource (if resources implemented)

Multiple Content Blocks: You can return multiple content items in the array:

```
{
  content: [
    { type: "text", text: "Analysis results:" },
    { type: "text", text: "Data point 1" },
    { type: "text", text: "Data point 2" }
  ]
}
```

Error Handling in Tools

Tool implementations should throw descriptive errors when execution fails. The tools/call handler catches these errors and formats them appropriately:

Error Throwing Pattern:

```
FUNCTION execute_your_tool(arguments):
  // Validate required parameters
  IF arguments.required_param IS NULL:
    THROW_ERROR("required_param is missing")
  END IF

  // Validate parameter format
  IF NOT VALID_FORMAT(arguments.required_param):
    THROW_ERROR("required_param must be a valid format")
  END IF

  // Attempt operation
  TRY:
    result = PERFORM_OPERATION(arguments)
    RETURN { content: [{ type: "text", text: result }] }
  CATCH operation_error:
    THROW_ERROR("Operation failed: " + operation_error.message)
  END TRY
END FUNCTION
```

Best Practices for Tool Implementation

1. **Parameter Validation:**
   - Validate all required parameters exist
   - Check parameter types and formats
   - Provide clear error messages for validation failures
2. **Error Messages:**
   - Be specific about what went wrong
   - Include relevant context (filename, parameter name, etc.)
   - Avoid exposing internal system details in production
3. **Response Consistency:**
   - Always return content array format
   - Use text type for most responses
   - Keep response sizes reasonable (avoid multi-MB responses)

4. **Async Operations:**
    - Use async/await for I/O operations (file access, API calls)
    - Set reasonable timeouts for external API calls
    - Handle network errors gracefully
5. **Logging:**
    - Log tool invocations for debugging
    - Log execution time for performance monitoring
    - Log errors with full context for troubleshooting
6. **Security:**
    - Validate and sanitize file paths (prevent directory traversal)
    - Limit file sizes for read/write operations
    - Restrict access to sensitive resources
    - Never expose secrets or credentials in responses

Tool Implementation Example

Pseudocode:

```
// Example: File read tool implementation
FUNCTION execute_file_read(arguments):
    filename = arguments.filename

    // Validate parameter
    IF filename IS NULL:
        THROW_ERROR("filename parameter is required")
    END IF

    // Sanitize filename (prevent directory traversal)
    IF filename.CONTAINS("..") OR filename.CONTAINS("/"):
        THROW_ERROR("Invalid filename - path traversal not allowed")
    END IF

    // Construct safe file path
    workspace_dir = GET_WORKSPACE_DIRECTORY()
    file_path = JOIN_PATH(workspace_dir, filename)

    // Attempt to read file
    TRY:
        content = READ_FILE(file_path)

        // Return in MCP format
        RETURN {
            content: [
                {
                    type: "text",
                    text: content
                }
            ]
        }
    CATCH file_error:
        THROW_ERROR("Failed to read file: " + file_error.message)
    END TRY
END FUNCTION
```

JavaScript:

```
javascript
```

```
const fs = require('fs');
const path = require('path');

async function executeFileRead(args) {
  const { filename } = args;

  // Validate parameter
  if (!filename) {
    throw new Error('filename parameter is required');
  }

  // Sanitize filename (prevent directory traversal)
  if (filename.includes('..') || filename.includes('/')) {
    throw new Error('Invalid filename - path traversal not allowed');
  }

  // Construct safe file path
  const workspaceDir = path.join(process.env.HOME, 'mcp-workspace');
  const filepath = path.join(workspaceDir, filename);

  // Attempt to read file
  try {
    const content = fs.readFileSync(filepath, 'utf8');

    // Return in MCP format
    return {
      content: [
        {
          type: 'text',
          text: content
        }
      ]
    };
  } catch (error) {
    throw new Error(`Failed to read file: ${error.message}`);
  }
}
```

TypeScript:

```
typescript
```

```typescript
import fs from 'fs';
import path from 'path';

// Tool arguments interface
interface FileReadArgs {
  filename: string;
}

// MCP content response structure
interface McpContent {
  type: string;
  text?: string;
  data?: string;
  mimeType?: string;
}

interface McpToolResponse {
  content: McpContent[];
}

async function executeFileRead(args: FileReadArgs): Promise<McpToolResponse> {
  const { filename } = args;

  // Validate parameter
  if (!filename) {
    throw new Error('filename parameter is required');
  }

  // Sanitize filename (prevent directory traversal)
  if (filename.includes('..') || filename.includes('/')) {
    throw new Error('Invalid filename - path traversal not allowed');
  }

  // Construct safe file path
  const workspaceDir: string = path.join(process.env.HOME || '', 'mcp-workspace');
  const filepath: string = path.join(workspaceDir, filename);

  // Attempt to read file
  try {
    const content: string = fs.readFileSync(filepath, 'utf8');

    // Return in MCP format
    return {
```

```
      content: [
        {
          type: 'text',
          text: content
        }
      ]
    };
  } catch (error) {
    throw new Error(`Failed to read file: ${(error as Error).message}`);
  }
}
```

Next Steps

Proceed to **Part 4: OAuth 2.1 Implementation** to add the security layer that protects your MCP server.

Document Status

- **Part:** 3 of 5
- **Version:** 1.0
- **Last Updated:** January 29, 2026
- **Status:** In Progress - Missing 'Creating Tools' section