

Tutorial 16: Model Context Protocol (MCP) Integration - Standardized Tool Protocols

Difficulty: advanced

Reading Time: 2 hours

Tags: advanced, mcp, protocol, tools, standardization

Description: Integrate MCP servers for standardized tool access including filesystem, databases, and external services using the Model Context Protocol.

Tutorial 16: Model Context Protocol (MCP) Integration

Goal: Integrate external tools and services into your agents using the Model Context Protocol (MCP), expanding your agent's capabilities with community-built tool servers.



Quick Start

The easiest way to get started is with our **working implementation**:

```
cd tutorial_implementation/tutorial16
make setup
make dev
```

Then open `http://localhost:8000` in your browser and try the MCP filesystem agent!

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 02 (Function Tools)
- Node.js installed (for MCP servers)
- Basic understanding of protocols and APIs
- **ADK Version:** 1.15.0+ recommended (tool_name_prefix, OAuth2 features)

What You'll Learn:

- Understanding Model Context Protocol (MCP)
- Using `MCPToolset` to connect to MCP servers
- Configuring stdio-based MCP connections
- Building agents with filesystem access
- Creating custom MCP server integrations
- Session pooling and resource management
- Best practices for production MCP deployments

Time to Complete: 50-65 minutes

:::warning ADK 1.16.0+ Callback Signature Change

Critical Update: ADK 1.16.0 changed the `before_tool_callback` signature.

Old (< 1.16.0): `callback_context, tool_name, args`

New (1.16.0+): `tool, args, tool_context`

See **Section 7: Human-in-the-Loop (HITL) with MCP** for details.





:::

Why MCP Matters

Problem: Building custom tools for every external service is time-consuming and repetitive.

Solution: Model Context Protocol (MCP) is an open standard for connecting AI agents to external tools and data sources. Instead of writing custom integrations, use **pre-built MCP servers** from the community.

Benefits:

-  **Plug-and-Play:** Connect to existing MCP servers instantly
-  **Community Ecosystem:** Leverage community-built tools
- [TOOLS] **Standardized Interface:** Consistent API across all tools
-  **Rich Capabilities:** Filesystem, databases, APIs, and more
- [FLOW] **Reusable:** Same server works with multiple agents
-  **Extensible:** Build custom servers when needed

MCP Ecosystem:

- Official MCP servers: filesystem, GitHub, Slack, database, and more
 - Community servers: 100+ available servers covering databases, APIs, development tools, and specialized services
 - Custom servers: Build your own for proprietary systems
-

1. MCP Basics

| What is Model Context Protocol?

MCP defines a standard way for AI models to discover and use external tools.

An **MCP server** exposes:

- **Tools:** Functions the agent can call
- **Resources:** Data the agent can access
- **Prompts:** Predefined instruction templates

Architecture:

```

Agent (ADK)
  ↓
MCPToolset (ADK wrapper)
  ↓
MCP Client
  ↓
MCP Server (stdio/HTTP)
  ↓
External Service (filesystem, API, database, etc.)

```

Source: `google/adk/tools/mcp_tool/mcp_tool.py`, `mcp_toolset.py`

MCP Connection Types

Stdio (Standard Input/Output):

```

from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

# Connect via stdio (most common)
mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx', # Node package executor
        args=['-y', '@modelcontextprotocol/server-filesystem', '/path/to/directories']
    )
)

```

HTTP (coming soon):

```

# Future: HTTP-based connections
# mcp_tools = MCPToolset(
#     connection_params=HttpConnectionParams(
#         url='http://localhost:3000'
#     )
# )

```

SSE (Server-Sent Events) -  **Supported in ADK 1.16.0+**

SSE connections enable real-time, streaming communication with MCP servers:

```
from google.adk.tools.mcp_tool import MCPToolset, SseConnectionParams

# Connect via Server-Sent Events (SSE)
mcp_tools = MCPToolset(
    connection_params=SseConnectionParams(
        url='https://api.example.com/mcp/sse',
        headers={'Authorization': 'Bearer your-token'}, # Optional headers
        timeout=30.0, # Connection timeout
        sse_read_timeout=300.0 # SSE read timeout
    )
)
```

Streamable HTTP - Supported in ADK 1.16.0+

HTTP connections support bidirectional streaming communication:

```
from google.adk.tools.mcp_tool import MCPToolset, StreamableHTTPConnectionPara

# Connect via Streamable HTTP
mcp_tools = MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url='https://api.example.com/mcp/stream',
        headers={'Authorization': 'Bearer your-token'}, # Optional headers
        timeout=30.0, # Connection timeout
        sse_read_timeout=300.0 # Read timeout
    )
)
```

2. Using MCP Filesystem Server

The most common MCP server is the **filesystem server**, which gives agents controlled file access.

Basic Setup

```
from google.adk.agents import Agent, Runner
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

# Create MCP toolset for filesystem access
mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=[
            '-y', # Auto-install if needed
            '@modelcontextprotocol/server-filesystem',
            '/Users/username/documents' # Directory to access
        ]
    )
)

# Create agent with MCP tools
agent = Agent(
    model='gemini-2.0-flash',
    name='file_assistant',
    instruction='You can read and write files in the documents directory.',
    tools=[mcp_tools]
)

runner = Runner()
result = runner.run(
    "List all text files in the directory",
    agent=agent
)

print(result.content.parts[0].text)
```

Available Filesystem Operations

The filesystem MCP server provides these tools:

```
# Tools automatically available through MCPToolset:

# 1. read_file - Read file contents
"Read the contents of report.txt"

# 2. write_file - Write to file
"Create a new file called notes.md with content: Hello World"

# 3. list_directory - List directory contents
"Show me all files in the current directory"

# 4. create_directory - Create new directory
"Create a folder called 'projects'"

# 5. move_file - Move or rename file
"Rename old_report.txt to archived_report.txt"

# 6. search_files - Search for files
"Find all Python files containing 'TODO'"

# 7. get_file_info - Get file metadata
"What's the size and modification date of config.json?"
```

3. Real-World Example: Document Organizer

Let's build an agent that organizes documents using MCP filesystem access.

| Complete Implementation


```

"""
Document Organizer using MCP Filesystem Server
Automatically organizes documents by type, date, and content.
"""

import asyncio
import os
from google.adk.agents import Agent, Runner, Session
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

class DocumentOrganizer:
    """Intelligent document organizer using MCP."""

    def __init__(self, base_directory: str):
        """
        Initialize document organizer.

        Args:
            base_directory: Root directory to organize
        """

        self.base_directory = base_directory

        # Create MCP toolset for filesystem access
        self.mcp_tools = MCPToolset(
            connection_params=StdioConnectionParams(
                command='npx',
                args=[
                    '-y',
                    '@modelcontextprotocol/server-filesystem',
                    base_directory
                ]
            ),
            retry_on_closed_resource=True # Auto-retry on connection issues
        )

        # Create organizer agent
        self.agent = Agent(
            model='gemini-2.0-flash',
            name='document_organizer',

```

```

        description='Intelligent document organization agent',
        instruction="""
You are a document organization expert with filesystem access.

Your responsibilities:
1. Analyze files by name, type, and content
2. Create logical folder structures
3. Move files to appropriate locations
4. Rename files for clarity
5. Generate organization reports

Guidelines:
- Create folders by category (e.g., Documents, Images, Code, Archives)
- Use subcategories when helpful (e.g., Documents/2024/, Documents/Work/)
- Preserve original filenames unless unclear
- Never delete files
- Report all changes made

You have access to filesystem tools:
- read_file: Read file contents
- write_file: Create files
- list_directory: List directory contents
- create_directory: Create folders
- move_file: Move/rename files
- search_files: Search by pattern
- get_file_info: Get file metadata
    """
    ).strip(),
    tools=[self.mcp_tools],
    generate_content_config=types.GenerateContentConfig(
        temperature=0.2, # Deterministic for file operations
        max_output_tokens=2048
    )
)

self.runner = Runner()
self.session = Session()

async def organize(self):
    """Organize documents in base directory."""

    print(f"{'='*70}")
    print(f"ORGANIZING: {self.base_directory}")
    print(f"{'='*70}\n")

    result = await self.runner.run_async(
        """
Organize all files in the directory:

```

1. List all files and analyze their types
2. Create appropriate folder structure
3. Move files to their logical locations
4. Generate a summary report of changes

Start by listing the directory contents.

```

        """.strip(),
        agent=self.agent,
        session=self.session
    )

    print("\n📁 ORGANIZATION REPORT:\n")
    print(result.content.parts[0].text)
    print(f"\n{'='*70}\n")

    async def search_documents(self, query: str):
        """
        Search documents by content.

        Args:
            query: Search query
        """

        print(f"\n🔍 SEARCHING FOR: {query}\n")

        result = await self.runner.run_async(
            f"Search all files for content related to: {query}",
            agent=self.agent,
            session=self.session
        )

        print("RESULTS:\n")
        print(result.content.parts[0].text)
        print()

    async def summarize_directory(self):
        """Generate directory summary."""

        print("\n📁 DIRECTORY SUMMARY:\n")

        result = await self.runner.run_async(
            """
Generate a comprehensive directory summary:
1. Total number of files
2. Files by type (documents, images, code, etc.)
3. Total size

```

```
4. Largest files
5. Recommendations for further organization
    """.strip(),
    agent=self.agent,
    session=self.session
)

print(result.content.parts[0].text)
print()

async def main():
    """Main entry point."""

    # Set base directory
    base_dir = '/Users/username/Documents/ToOrganize'

    # Create organizer
    organizer = DocumentOrganizer(base_dir)

    # Organize documents
    await organizer.organize()

    # Search for specific content
    await organizer.search_documents('budget reports')

    # Get summary
    await organizer.summarize_directory()

if __name__ == '__main__':
    asyncio.run(main())
```

| Expected Output

```
=====
ORGANIZING: /Users/username/Documents/ToOrganize
=====
```

ORGANIZATION REPORT:

****Initial Analysis:****

Found 25 files in directory:

- 8 PDF documents
- 6 Word documents (.docx)
- 5 Images (.jpg, .png)
- 3 Spreadsheets (.xlsx)
- 2 Python scripts (.py)
- 1 Text file (.txt)

****Actions Taken:****

1. ****Created Folder Structure:****

- Documents/
 - 2024/
 - Work/
 - Personal/
- Images/
- Code/
- Spreadsheets/

2. ****File Movements:****


- Moved 8 PDFs to Documents/ (3 to Work/, 5 to Personal/)
- Moved 6 DOCX files to Documents/2024/
- Moved 5 images to Images/
- Moved 3 spreadsheets to Spreadsheets/
- Moved 2 Python scripts to Code/

3. ****Files Renamed:****

- IMG_1234.jpg → vacation_photo_2024.jpg
- document.docx → project_proposal_draft.docx
- script.py → data_processor.py

****Summary:****

- ✓ Organized 25 files into 6 folders
- ✓ Renamed 3 files for clarity
- ✓ Created logical structure for future files
- ✓ All files preserved (no deletions)

 SEARCHING FOR: budget reports

RESULTS:

Found 3 files matching "budget reports":

1. ****Documents/Work/Q3_Budget_Report.pdf****
 - Contains: Q3 financial summary, expense breakdown
 - Size: 2.4 MB
 - Modified: 2024-09-15
2. ****Spreadsheets/Budget_2024.xlsx****
 - Contains: Annual budget with quarterly projections
 - Size: 156 KB
 - Modified: 2024-10-01
3. ****Documents/Work/Budget_Meeting_Notes.docx****
 - Contains: Meeting notes from budget review
 - Size: 45 KB
 - Modified: 2024-09-20

DIRECTORY SUMMARY:

****Directory Statistics:****

- Total Files: 25
- Total Size: 47.3 MB
- Folders: 6

****Files by Type:****

- Documents (PDF/DOCX): 14 files (35.2 MB)
- Images (JPG/PNG): 5 files (8.1 MB)
- Spreadsheets (XLSX): 3 files (2.8 MB)
- Code (PY): 2 files (18 KB)
- Other: 1 file (1.2 MB)

****Largest Files:****

1. Documents/Personal/Family_Photos_Archive.pdf (12.5 MB)
2. Images/high_res_photo.jpg (3.8 MB)
3. Spreadsheets/Annual_Data.xlsx (2.8 MB)

****Recommendations:****

- Consider archiving files older than 1 year
- Large images could be compressed
- Create additional subfolder for monthly reports in Documents/Work/

4. Advanced MCP Features

| Session Pooling

MCPToolset maintains a pool of connections for efficiency:

```
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', '/path']
    ),

    # Session pooling configuration
    retry_on_closed_resource=True, # Auto-retry on connection loss

    # Pool automatically manages:
    # - Connection reuse
    # - Resource cleanup
    # - Error recovery
)
```

| Multiple MCP Servers

Use multiple MCP servers simultaneously:


```

from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

# Filesystem server
filesystem_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', '/documents']
    ),
    tool_name_prefix='fs_' # ADK 1.15.0+: Avoid name conflicts
)

# GitHub server (hypothetical)
github_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-github', '--token', 'YOUR_TOKEN']
    ),
    tool_name_prefix='gh_' # ADK 1.15.0+: Avoid name conflicts
)

# Agent with multiple MCP toolsets
agent = Agent(
    model='gemini-2.0-flash',
    name='multi_tool_agent',
    instruction='You have access to both filesystem (fs_*) and GitHub (gh_*) o',
    tools=[filesystem_tools, github_tools]
)

```

Tool Name Prefix (ADK 1.15.0+):

When using multiple MCP servers, tools from different servers might have conflicting names.

The `tool_name_prefix` parameter prefixes all tool names to avoid conflicts:

```

# Without prefix: Both servers might have a "read_file" tool
# With prefix: "fs_read_file" and "gh_read_file"

# Agent can distinguish: "Use fs_read_file to read local docs"
# vs "Use gh_read_file to read repository files"

```

Resource Access

MCP servers can expose **resources** (read-only data):

```
# Resources are automatically discovered
# Agent can access them like:
# "Read the README resource from the GitHub server"

# Resources appear as:
# - resource://server/path/to/resource
# - Automatically listed when agent queries available resources
```

5. MCP Limitations

| ❌ Sampling Not Supported (ADK 1.16.0)

Important Limitation: Google ADK's MCP implementation **does not support sampling** as of version 1.16.0.

What is MCP Sampling?

MCP sampling allows servers to request LLM completions/generations from the client:

```
# Server can request LLM generation (NOT supported by ADK):
{
  "method": "sampling/createMessage",
  "params": {
    "messages": [{"role": "user", "content": "Summarize this data"}],
    "modelPreferences": {"hints": [{"name": "gemini-2.0-flash"}]},
    "maxTokens": 100
  }
}
```

Why Sampling Matters

Sampling enables **agentic behaviors** in MCP servers:

- Dynamic content generation during tool execution
- LLM-powered analysis and summarization
- Conversational AI capabilities within server tools

- Nested AI interactions (LLM calls within MCP server logic)

ADK's Current Behavior

```
# ADK returns error for sampling requests:
{
  "error": {
    "code": -32600,
    "message": "Sampling not supported"
  }
}
```

Workarounds

For MCP Servers:

- Implement your own LLM integration (direct API calls to Gemini)
- Use pre-computed responses instead of dynamic generation
- Handle text generation outside the MCP protocol

For ADK Applications:

- Use ADK's native LLM capabilities instead of MCP sampling
- Implement sampling logic in your ADK agents directly
- Consider hybrid approaches (MCP for tools, ADK for LLM calls)

Future Support

Sampling support may be added in future ADK versions. Check the [ADK changelog](https://github.com/google/adk-python/blob/main/CHANGELOG.md) (<https://github.com/google/adk-python/blob/main/CHANGELOG.md>) for updates.

6. Building Custom MCP Servers

| Simple MCP Server (Node.js)

```
// custom-mcp-server.js

// Create server
const server = new Server(
  {
    name: "custom-calculator-server",
    version: "1.0.0",
  },
  {
    capabilities: {
      tools: {},
    },
  },
);

// Register tool
server.setRequestHandler("tools/list", async () => {
  return {
    tools: [
      {
        name: "calculate",
        description: "Perform mathematical calculations",
        inputSchema: {
          type: "object",
          properties: {
            expression: {
              type: "string",
              description: "Mathematical expression to evaluate",
            },
          },
          required: ["expression"],
        },
      },
    ],
  };
});

// Handle tool calls
server.setRequestHandler("tools/call", async (request) => {
  if (request.params.name === "calculate") {
    const expression = request.params.arguments.expression;
    try {
      const result = eval(expression); // In production, use safe math parser
      return {
        content: [
          {

```

```
        type: "text",
        text: `Result: ${result}`,
      },
    ],
  };
} catch (error) {
  return {
    content: [
      {
        type: "text",
        text: `Error: ${error.message}`,
      },
    ],
    isError: true,
  };
}
}
});

// Start server
const transport = new StdioServerTransport();
await server.connect(transport);
```

Using Custom MCP Server

```
from google.adk.agents import Agent, Runner
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

# Connect to custom server
custom_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='node',
        args=['custom-mcp-server.js']
    )
)

# Use in agent
agent = Agent(
    model='gemini-2.0-flash',
    name='calculator_agent',
    tools=[custom_tools]
)

runner = Runner()
result = runner.run("Calculate 25 * 4 + 10", agent=agent)

print(result.content.parts[0].text)
# Output: "The result is 110"
```

6. Popular MCP Servers

| Official MCP Servers


```
# 1. Filesystem Server
filesystem = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', '/path']
    )
)

# 2. GitHub Server
github = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=[
            '-y',
            '@modelcontextprotocol/server-github',
            '--token', 'YOUR_GITHUB_TOKEN'
        ]
    )
)

# 3. Slack Server
slack = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=[
            '-y',
            '@modelcontextprotocol/server-slack',
            '--token', 'YOUR_SLACK_TOKEN'
        ]
    )
)

# 4. Postgres Server
postgres = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=[
            '-y',
            '@modelcontextprotocol/server-postgres',
            'postgresql://user:pass@localhost:5432/dbname'
        ]
    )
)
```

Community MCP Servers

The MCP ecosystem includes 100+ community-built servers covering specialized use cases:

Development & DevOps:

- Git integrations (GitLab, Bitbucket, Azure DevOps)
- CI/CD tools (Jenkins, GitHub Actions, CircleCI)
- Container management (Docker, Kubernetes, Podman)
- Cloud platforms (AWS, Azure, GCP, DigitalOcean)

Databases & Data:

- MySQL, MongoDB, Redis, Elasticsearch
- Data warehouses (BigQuery, Snowflake, ClickHouse, Redshift)
- Vector databases (Pinecone, Weaviate, Chroma, Qdrant)
- Graph databases (Neo4j, ArangoDB)

APIs & Integrations:

- REST APIs (OpenAPI/Swagger auto-generation)
- GraphQL endpoints
- Web scraping and automation (Playwright, Puppeteer)
- Social media (Twitter/X, Discord, Bluesky, LinkedIn)

Productivity & Communication:

- Email servers (Gmail, Outlook, SendGrid)
- Calendar integrations (Google Calendar, Outlook)
- Task management (Linear, Jira, Asana, Monday.com)
- Document processing (PDF tools, Office files, Notion)

Specialized Tools:

- Code analysis and linting
- Testing frameworks (Jest, Pytest, Selenium)
- Security scanning and vulnerability assessment
- Financial data (stocks, crypto, banking APIs)
- Weather, location, and mapping services

- Media processing (images, video, audio)

Browse the complete list at the [MCP Server Registry](https://github.com/modelcontextprotocol/servers) (<https://github.com/modelcontextprotocol/servers>).

7. Human-in-the-Loop (HITL) with MCP

ADK 1.16.0+ Callback Signature: Implementing approval workflows for destructive operations.

| Why HITL Matters

MCP filesystem servers provide powerful file manipulation capabilities, but **destructive operations** (write, move, delete) need human approval in production to prevent:

- Accidental file overwrites
- Unintended file deletions
- Security breaches
- Data loss

| ADK 1.16.0 Callback Signature

Critical Discovery: ADK 1.16.0 changed the callback signature significantly.

Correct Signature (ADK 1.16.0+):

```

from typing import Dict, Any, Optional

def before_tool_callback(
    tool, # BaseTool object (NOT string!)
    args: Dict[str, Any],
    tool_context # Has .state attribute (NOT callback_context!)
) -> Optional[Dict[str, Any]]:
    """
    Callback invoked before tool execution.

    Args:
        tool: BaseTool object with .name attribute
        args: Arguments passed to the tool
        tool_context: Context with state access via .state

    Returns:
        None: Allow tool execution
        dict: Block tool execution, return this result instead
    """
    # Extract tool name from object
    tool_name = tool.name if hasattr(tool, 'name') else str(tool)

    # Access state via tool_context.state (NOT callback_context.state)
    count = tool_context.state.get('temp:tool_count', 0) or 0
    tool_context.state['temp:tool_count'] = count + 1

    # Your approval logic here
    return None # Allow execution

```

Key Changes from Older Versions:

Aspect	Old (< 1.16.0)	New (1.16.0+)
First parameter	<code>callback_context</code>	Removed
Tool parameter	<code>tool_name: str</code>	<code>tool</code> (object)
State access	<code>callback_context.state</code>	<code>tool_context.state</code>
Tool name	Direct string	Extract from <code>tool.name</code>

| Complete HITL Implementation

```

"""
MCP Agent with Human-in-the-Loop Approval Workflow
Demonstrates ADK 1.16.0 callback signature.
"""

import os
import logging
from typing import Dict, Any, Optional
from google.adk.agents import Agent
from google.adk.tools.mcp_tool import McpToolset, StdioConnectionParams

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def before_tool_callback(
    tool, # BaseTool object
    args: Dict[str, Any],
    tool_context # Has .state attribute
) -> Optional[Dict[str, Any]]:
    """
    Human-in-the-Loop callback for MCP filesystem operations.

    Implements approval workflow for destructive operations:
    - Write operations require confirmation
    - Move/delete operations require explicit approval
    - Read operations are allowed without confirmation

    ADK Best Practice: Use before_tool_callback for:
    1. Validation: Check arguments are safe
    2. Authorization: Require approval for sensitive operations
    3. Logging: Track tool usage for audit
    4. Rate limiting: Prevent abuse

    Args:
        tool: BaseTool object being called (has .name attribute)
        args: Arguments passed to the tool
        tool_context: ToolContext with state and invocation access

    Returns:
        None: Allow tool execution
        dict: Block tool execution and return this result instead
    """
    # Extract tool name from tool object
    tool_name = tool.name if hasattr(tool, 'name') else str(tool)

```

```

logger.info(f"[TOOL REQUEST] {tool_name} with args: {args}")

# Track tool usage in session state
tool_count = tool_context.state.get('temp:tool_count', 0) or 0 # Handle None
tool_context.state['temp:tool_count'] = tool_count + 1
tool_context.state['temp:last_tool'] = tool_name

# Define destructive operations that require approval
DESTRUCTIVE_OPERATIONS = {
    'write_file': 'Writing files modifies content',
    'write_text_file': 'Writing files modifies content',
    'move_file': 'Moving files changes file locations',
    'create_directory': 'Creating directories modifies filesystem structure'
}

# Check if this is a destructive operation
if tool_name in DESTRUCTIVE_OPERATIONS:
    reason = DESTRUCTIVE_OPERATIONS[tool_name]

    # Log the approval request
    logger.warning(f"[APPROVAL REQUIRED] {tool_name}: {reason}")
    logger.info(f"[APPROVAL REQUEST] Arguments: {args}")

    # Check approval flag in state
    auto_approve = tool_context.state.get('user:auto_approve_file_ops', False)

    if not auto_approve:
        # Return blocking response - tool won't execute
        return {
            'status': 'requires_approval',
            'message': (
                f"⚠️ APPROVAL REQUIRED\n\n"
                f"Operation: {tool_name}\n"
                f"Reason: {reason}\n"
                f"Arguments: {args}\n\n"
                f"To approve, set state['user:auto_approve_file_ops'] = True\n"
                f"Or use the ADK UI approval workflow.\n\n"
                f"This operation has been BLOCKED for safety."
            ),
            'tool_name': tool_name,
            'args': args,
            'requires_approval': True
        }
    else:
        logger.info(f"[APPROVED] {tool_name} approved via auto_approve flag")

# Allow non-destructive operations (read, list, search, get_info)

```

```

logger.info(f"[ALLOWED] {tool_name} approved automatically")
return None # None means allow tool execution

def create_mcp_filesystem_agent(
    base_directory: str = None,
    enable_hitl: bool = True
) -> Agent:
    """
    Create MCP filesystem agent with optional HITL.

    Args:
        base_directory: Directory to restrict access to (default: ./sample_files)
        enable_hitl: Enable Human-in-the-Loop approval workflow

    Returns:
        Agent with MCP filesystem tools and HITL callback
    """
    # Default to sample_files directory
    if base_directory is None:
        current_dir = os.getcwd()
        base_directory = os.path.join(current_dir, 'sample_files')
        if not os.path.exists(base_directory):
            os.makedirs(base_directory, exist_ok=True)

    base_directory = os.path.abspath(base_directory)
    logger.info(f"[SECURITY] MCP filesystem access restricted to: {base_directory}")

    # Create MCP toolset
    mcp_tools = McpToolset(
        connection_params=StdioConnectionParams(
            command='npx',
            args=[
                '-y',
                '@modelcontextprotocol/server-file-system',
                base_directory
            ],
            timeout=30.0 # 30 second timeout
        ),
        retry_on_closed_resource=True
    )

    # Create agent with HITL callback
    agent = Agent(
        model='gemini-2.0-flash-exp',
        name='mcp_filesystem_agent',
        description='MCP filesystem agent with HITL approval workflow',
        instruction=f"""

```


You are a filesystem assistant with access to: {base_directory}

IMPORTANT SECURITY BOUNDARIES:

- You can ONLY access files within: {base_directory}
- All destructive operations (write, move, create) require approval
- Read operations are allowed without approval

AVAILABLE TOOLS:

- read_file: Read file contents (APPROVED automatically)
- list_directory: List directory contents (APPROVED automatically)
- search_files: Search for files (APPROVED automatically)
- get_file_info: Get file metadata (APPROVED automatically)
- write_file: Write file contents (REQUIRES APPROVAL)
- move_file: Move/rename files (REQUIRES APPROVAL)
- create_directory: Create directories (REQUIRES APPROVAL)

APPROVAL WORKFLOW:

1. When you attempt a destructive operation, it will be BLOCKED
2. You'll receive an "APPROVAL REQUIRED" message
3. Explain to the user what was blocked and why
4. User must approve before operation proceeds
5. Once approved, you can proceed with the operation

Always explain what you're about to do before performing destructive operation

```

    """.strip(),
    tools=[mcp_tools],

    # Enable Human-in-the-Loop callback if requested
    before_tool_callback=before_tool_callback if enable_hitl else None
)

return agent

# Example usage
if __name__ == '__main__':
    from google.adk.agents import Runner
    import asyncio

    async def main():
        agent = create_mcp_filesystem_agent(
            base_directory='./sample_files',
            enable_hitl=True # Enable approval workflow
        )

        runner = Runner()

        # This will be approved automatically (read operation)
```

```
result1 = await runner.run_async(
    "List all files in the directory",
    agent=agent
)
print(result1.content.parts[0].text)

# This will be BLOCKED (write operation requires approval)
result2 = await runner.run_async(
    "Create a file called test.txt with content: Hello World",
    agent=agent
)
print(result2.content.parts[0].text)
# Expected: "⚠️ APPROVAL REQUIRED..." message

asyncio.run(main())
```

| Testing HITL Implementation

The tutorial includes **25 comprehensive tests** covering all aspects of the HITL workflow:

```
# tests/test_hitl.py - Comprehensive HITL test suite

import pytest
from unittest.mock import Mock
from mcp_agent.agent import before_tool_callback

class TestDestructiveOperationDetection:
    """Test detection of operations requiring approval."""

    @pytest.mark.parametrize("operation_name", [
        "write_file",
        "write_text_file",
        "move_file",
        "create_directory"
    ])
    def test_destructive_operations_require_approval(self, operation_name):
        """All destructive operations should require approval."""
        mock_tool = Mock()
        mock_tool.name = operation_name

        mock_context = Mock()
        mock_context.state = {} # No auto_approve flag

        result = before_tool_callback(
            tool=mock_tool,
            args={'path': '/test/file.txt'},
            tool_context=mock_context
        )

        # Should return approval required message
        assert result is not None
        assert result['status'] == 'requires_approval'
        assert 'APPROVAL REQUIRED' in result['message']

# Run tests
# pytest tests/test_hitl.py -v
# Expected: 25 passed
```

Test Coverage (25 tests):

1. **Tool Name Extraction** (2 tests) - Extract names from tool objects
2. **Destructive Operation Detection** (8 tests) - Block write/move/create
3. **Approval Workflow** (3 tests) - Auto-approve flag behavior
4. **State Management** (3 tests) - Tool counting and tracking

5. **Approval Message Content** (4 tests) - Message formatting
6. **Edge Cases** (3 tests) - None values, empty args, unknown tools
7. **Integration Scenarios** (2 tests) - Real workflow testing

| HITL Best Practices

DO:

- ✓ Use callbacks for all destructive operations
- ✓ Extract tool name: `tool_name = tool.name if hasattr(tool, 'name') else str(tool)`
- ✓ Access state via `tool_context.state` (not `callback_context.state`)
- ✓ Handle None values: `count = state.get('key', 0) or 0`
- ✓ Log approval requests for audit trail
- ✓ Provide clear approval messages with context
- ✓ Test with comprehensive test suite

DON'T:

- ✗ Use old callback signature (`callback_context` parameter removed in 1.16.0)
- ✗ Treat `tool` as string (it's a BaseTool object)
- ✗ Access `callback_context.state` (doesn't exist in 1.16.0)
- ✗ Forget to handle None in state values
- ✗ Block read operations (only destructive ones)
- ✗ Deploy without testing approval workflow

| Migration from Older ADK Versions

If migrating from ADK < 1.16.0, update your callbacks:

```
# OLD (< 1.16.0) - DON'T USE
def before_tool_callback(
    callback_context: CallbackContext, # REMOVED in 1.16.0
    tool_name: str, # Now an object, not string
    args: Dict[str, Any]
) -> Optional[Dict[str, Any]]:
    count = callback_context.state.get('count', 0) # Wrong state access
    if tool_name in DESTRUCTIVE_OPS:
        # Check approval
        pass
    return None

# NEW (1.16.0+) - CORRECT
def before_tool_callback(
    tool, # Object, not string!
    args: Dict[str, Any],
    tool_context # Replaces callback_context
) -> Optional[Dict[str, Any]]:
    tool_name = tool.name if hasattr(tool, 'name') else str(tool)
    count = tool_context.state.get('count', 0) or 0 # Handle None
    if tool_name in DESTRUCTIVE_OPS:
        # Check approval
        pass
    return None
```

Real-World HITL Logs

From actual ADK web server with HITL enabled:

```
2025-10-10 17:55:23,896 - INFO - [TOOL REQUEST] write_file with args: {'content': '...'}
2025-10-10 17:55:23,896 - WARNING - [APPROVAL REQUIRED] write_file: Writing file
2025-10-10 17:55:23,896 - INFO - [APPROVAL REQUEST] Arguments: {'content': '...'}
```

- ✓ Tool name extracted correctly (`write_file`)
- ✓ HITL blocking triggered
- ✓ Approval workflow operational

8. Best Practices

✓ DO: Use Retry on Closed Resource

```
# ✓ Good - Auto-retry on connection loss
mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(...),
    retry_on_closed_resource=True
)

# ✗ Bad - No retry (fails on connection loss)
mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(...)
)
```

✓ DO: Validate Directory Paths

```
import os

# ✓ Good - Validate path exists
directory = '/Users/username/documents'

if not os.path.exists(directory):
    raise ValueError(f"Directory does not exist: {directory}")

mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', directory]
    )
)

# ✗ Bad - No validation
mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', '/nonexistent']
    )
)
```

✓ DO: Provide Clear Instructions

```
# ✓ Good - Clear tool guidance
agent = Agent(
    model='gemini-2.0-flash',
    instruction="""
You have filesystem access via MCP tools:
- read_file: Read file contents
- write_file: Create/update files
- list_directory: List directory contents
- move_file: Move/rename files

Always explain what you're doing before file operations.
    """,
    tools=[mcp_tools]
)

# ✗ Bad - No guidance
agent = Agent(
    model='gemini-2.0-flash',
    instruction="You can access files",
    tools=[mcp_tools]
)
```

✓ DO: Handle MCP Errors

```
# ✓ Good - Error handling
try:
    mcp_tools = MCPToolset(
        connection_params=StdioConnectionParams(
            command='npx',
            args=['-y', '@modelcontextprotocol/server-filesystem', directory]
        )
    )

    result = runner.run(query, agent=agent)

except Exception as e:
    print(f"MCP Error: {e}")
    # Fallback behavior

# ✗ Bad - No error handling
mcp_tools = MCPToolset(...) # May fail silently
```

8. Troubleshooting

Error: "npx command not found"

Problem: Node.js not installed

Solution:

```
# Install Node.js
# macOS:
brew install node

# Ubuntu:
sudo apt install nodejs npm

# Verify
npx --version
```


Error: "MCP server connection failed"

Problem: Server not starting or wrong command

Solutions:

1. Test server manually:

```
# Run server directly to see errors
npx -y @modelcontextprotocol/server-filesystem /path/to/dir
```

1. Check path:

```
import os

directory = '/Users/username/documents'
print(f"Path exists: {os.path.exists(directory)}")
print(f"Absolute path: {os.path.abspath(directory)}")
```

1. Use correct command:

```
# ✓ Correct
StudioConnectionParams(
    command='npx', # Not 'npm' or 'node'
    args=['-y', '@modelcontextprotocol/server-filesystem', directory]
)
```

Issue: "Tools not appearing"

Problem: MCP server not exposing tools correctly

Solution: Check server logs and tool discovery:

```
# Enable debug logging
import logging
logging.basicConfig(level=logging.DEBUG)

# ADK will log MCP tool discovery
mcp_tools = MCPToolset(...)
```

9. Testing MCP Integrations

| Unit Tests

```

import pytest
import os
import tempfile
from google.adk.agents import Agent, Runner
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

@pytest.mark.asyncio
async def test_mcp_filesystem_read():
    """Test reading file via MCP."""

    # Create temp directory and file
    with tempfile.TemporaryDirectory() as tmpdir:
        test_file = os.path.join(tmpdir, 'test.txt')

        with open(test_file, 'w') as f:
            f.write('Hello MCP')

        # Create MCP toolset
        mcp_tools = MCPToolset(
            connection_params=StdioConnectionParams(
                command='npx',
                args=['-y', '@modelcontextprotocol/server-file-system', tmpdir]
            )
        )

        # Create agent
        agent = Agent(
            model='gemini-2.0-flash',
            tools=[mcp_tools]
        )

        runner = Runner()
        result = await runner.run_async(
            "Read the contents of test.txt",
            agent=agent
        )

        # Verify
        text = result.content.parts[0].text
        assert 'Hello MCP' in text

@pytest.mark.asyncio
async def test_mcp_filesystem_write():
    """Test writing file via MCP."""

    with tempfile.TemporaryDirectory() as tmpdir:

```

```

mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@modelcontextprotocol/server-filesystem', tmpdir]
    )
)

agent = Agent(
    model='gemini-2.0-flash',
    tools=[mcp_tools]
)

runner = Runner()
result = await runner.run_async(
    "Create a file called output.txt with content: Test content",
    agent=agent
)

# Verify file created
output_file = os.path.join(tmpdir, 'output.txt')
assert os.path.exists(output_file)

with open(output_file) as f:
    content = f.read()
    assert 'Test content' in content

```

7. MCP OAuth Authentication

Source: `google/adk/tools/mcp_tool/mcp_tool.py`, `contributing/samples/oauth2_client_credentials/`

MCP supports **multiple authentication methods** for securing access to MCP servers. This is critical for production deployments where MCP servers access sensitive resources.

Supported Authentication Methods

ADK's MCP implementation supports:

1. **OAuth2** (Client Credentials flow)

2. HTTP Bearer Token
3. HTTP Basic Authentication
4. API Key

| OAuth2 Authentication (Most Secure)

OAuth2 is the **recommended authentication method** for production MCP servers.

Use Case: Accessing protected APIs, enterprise data sources, cloud services.

Implementation:

```
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams
from google.adk.agents import Agent, Runner

# OAuth2 Client Credentials configuration
mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(
        command='npx',
        args=['-y', '@mycompany/secure-mcp-server']
    ),
    credential={
        'type': 'oauth2',
        'token_url': 'https://auth.example.com/oauth/token',
        'client_id': 'your-client-id',
        'client_secret': 'your-client-secret',
        'scopes': ['read', 'write'] # Optional
    }
)

agent = Agent(
    model='gemini-2.5-flash',
    name='secure_agent',
    instruction='You have authenticated access to secure resources.',
    tools=[mcp_tools]
)
```

How It Works:

1. ADK automatically requests access token from `token_url`
2. Token included in all MCP server requests
3. Token refreshed automatically when expired
4. Secure credential handling throughout

| HTTP Bearer Token (Simple)

For MCP servers that use bearer tokens.

```
mcp_tools = MCPToolset(  
    connection_params=StdioConnectionParams(  
        command='npx',  
        args=['-y', '@mycompany/api-server']  
    ),  
    credential={  
        'type': 'bearer',  
        'token': 'your-bearer-token-here'  
    }  
)
```

When to use: APIs with static bearer tokens, internal services.

| HTTP Basic Authentication

For MCP servers using username/password.

```
mcp_tools = MCPToolset(  
    connection_params=StdioConnectionParams(  
        command='npx',  
        args=['-y', '@mycompany/legacy-server']  
    ),  
    credential={  
        'type': 'basic',  
        'username': 'admin',  
        'password': 'secure-password'  
    }  
)
```

When to use: Legacy systems, simple internal tools.

| API Key Authentication

For MCP servers using API key headers.

```
mcp_tools = MCPToolset(  
    connection_params=StdioConnectionParams(  
        command='npx',  
        args=['-y', '@mycompany/api-gateway']  
    ),  
    credential={  
        'type': 'api_key',  
        'key': 'your-api-key',  
        'header': 'X-API-Key' # Optional, default: 'Authorization'  
    }  
)
```

When to use: Cloud services, third-party APIs.

Complete OAuth2 Example: Secure Document Server


```

"""
OAuth2-secured MCP server integration.
Source: contributing/samples/oauth2_client_credentials/oauth2_test_server.py
"""

import asyncio
import os
from google.adk.agents import Agent, Runner
from google.adk.tools.mcp_tool import MCPToolset, StdioConnectionParams

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

async def main():
    """Demonstrate OAuth2-secured MCP integration."""

    # OAuth2 configuration
    oauth2_credential = {
        'type': 'oauth2',
        'token_url': 'https://auth.company.com/oauth/token',
        'client_id': os.environ.get('OAUTH_CLIENT_ID'),
        'client_secret': os.environ.get('OAUTH_CLIENT_SECRET'),
        'scopes': ['documents.read', 'documents.write']
    }

    # Create MCP toolset with OAuth2
    secure_mcp_tools = MCPToolset(
        connection_params=StdioConnectionParams(
            command='npx',
            args=[
                '-y',
                '@company/secure-document-server',
                '--environment', 'production'
            ]
        ),
        credential=oauth2_credential,
        retry_on_closed_resource=True
    )

    # Create agent with authenticated MCP access
    agent = Agent(
        model='gemini-2.5-flash',
        name='secure_document_agent',
        description='Agent with OAuth2-secured document access',

```

```

        instruction="""
You have authenticated access to the company document server.
You can:
- Read confidential documents
- Create new documents with proper permissions
- Search across authorized document repositories
- Respect access control policies

Always handle sensitive information appropriately.
        """.strip(),
        tools=[secure_mcp_tools]
    )

    # Run queries with authentication
    runner = Runner()

    print("\n" + "="*60)
    print("SECURE MCP SERVER WITH OAUTH2")
    print("="*60 + "\n")

    # Query 1: Read secure document
    result1 = await runner.run_async(
        "Read the Q4 financial report from the secure archive.",
        agent=agent
    )
    print("📄 Q4 Report:\n")
    print(result1.content.parts[0].text)

    await asyncio.sleep(1)

    # Query 2: Create document
    result2 = await runner.run_async(
        "Create a summary document of key findings from the Q4 report.",
        agent=agent
    )
    print("\n\n📝 Summary Created:\n")
    print(result2.content.parts[0].text)

    print("\n" + "="*60 + "\n")

if __name__ == '__main__':
    asyncio.run(main())

```

How Authentication Works Internally

Source: `google/adk/tools/mcp_tool/mcp_tool.py` (simplified):

```
class McpTool:
    """Individual MCP tool with authentication."""

    def _get_headers(self, credential: dict) -> dict:
        """Generate authentication headers based on credential type."""

        if credential['type'] == 'oauth2':
            # Fetch OAuth2 access token
            token = self._fetch_oauth2_token(
                token_url=credential['token_url'],
                client_id=credential['client_id'],
                client_secret=credential['client_secret'],
                scopes=credential.get('scopes', [])
            )
            return {'Authorization': f'Bearer {token}'}

        elif credential['type'] == 'bearer':
            return {'Authorization': f"Bearer {credential['token']}"}

        elif credential['type'] == 'basic':
            # Base64 encode username:password
            import base64
            creds = f"{credential['username']}:{credential['password']}"
            encoded = base64.b64encode(creds.encode()).decode()
            return {'Authorization': f'Basic {encoded}'}

        elif credential['type'] == 'api_key':
            header_name = credential.get('header', 'Authorization')
            return {header_name: credential['key']}

        return {}
```

Best Practices for Authentication

DO:

- ✓ Use OAuth2 for production systems
- ✓ Store credentials in environment variables (not hardcoded!)
- ✓ Use least-privilege scopes (only necessary permissions)

- ✓ Rotate credentials regularly
- ✓ Monitor authentication failures
- ✓ Test with expired tokens

DON'T:

- ✗ Commit credentials to version control
- ✗ Use same credentials across environments (dev/prod)
- ✗ Share credentials between agents
- ✗ Ignore token expiration
- ✗ Use Basic auth for internet-facing services

Credential Management

Environment Variables (Recommended):

```
import os

# Load from environment
oauth2_credential = {
    'type': 'oauth2',
    'token_url': os.environ['OAUTH_TOKEN_URL'],
    'client_id': os.environ['OAUTH_CLIENT_ID'],
    'client_secret': os.environ['OAUTH_CLIENT_SECRET'],
    'scopes': os.environ.get('OAUTH_SCOPES', '').split(',')
}

mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(...),
    credential=oauth2_credential
)
```

Secret Manager (Production):

```
from google.cloud import secretmanager

def get_oauth_credentials():
    """Fetch OAuth2 credentials from Secret Manager."""
    client = secretmanager.SecretManagerServiceClient()

    # Fetch secrets
    client_id = client.access_secret_version(
        name="projects/PROJECT/secrets/oauth-client-id/versions/latest"
    ).payload.data.decode()

    client_secret = client.access_secret_version(
        name="projects/PROJECT/secrets/oauth-client-secret/versions/latest"
    ).payload.data.decode()

    return {
        'type': 'oauth2',
        'token_url': 'https://auth.company.com/oauth/token',
        'client_id': client_id,
        'client_secret': client_secret
    }

mcp_tools = MCPToolset(
    connection_params=StdioConnectionParams(...),
    credential=get_oauth_credentials()
)
```

| SSE/HTTP with OAuth2 Authentication

ADK 1.16.0+ supports OAuth2 authentication with SSE and HTTP connections for secure production deployments.

OAuth2 with SSE Connection

```
from google.adk.tools.mcp_tool import MCPToolset, SseConnectionParams
from google.adk.auth.auth_credential import (
    AuthCredential, AuthCredentialTypes, OAuth2Auth
)

# OAuth2 authentication for SSE
oauth2_credential = AuthCredential(
    auth_type=AuthCredentialTypes.OAUTH2,
    oauth2=OAuth2Auth(
        client_id='your-client-id',
        client_secret='your-client-secret',
        auth_uri='https://auth.example.com/oauth/authorize',
        token_uri='https://auth.example.com/oauth/token',
        scopes=['read', 'write']
    )
)

mcp_tools = MCPToolset(
    connection_params=SseConnectionParams(
        url='https://secure-api.example.com/mcp/sse',
        headers={'X-API-Version': '1.0'}, # Additional headers
        timeout=30.0,
        sse_read_timeout=300.0
    ),
    auth_credential=oauth2_credential
)
```

OAuth2 with HTTP Connection

```
from google.adk.tools.mcp_tool import MCPToolset, StreamableHTTPConnectionPara
from google.adk.auth.auth_credential import (
    AuthCredential, AuthCredentialTypes, OAuth2Auth
)

# OAuth2 authentication for HTTP streaming
oauth2_credential = AuthCredential(
    auth_type=AuthCredentialTypes.OAUTH2,
    oauth2=OAuth2Auth(
        client_id='your-client-id',
        client_secret='your-client-secret',
        auth_uri='https://auth.example.com/oauth/authorize',
        token_uri='https://auth.example.com/oauth/token',
        scopes=['api.read', 'api.write']
    )
)

mcp_tools = MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url='https://secure-api.example.com/mcp/stream',
        headers={'Content-Type': 'application/json'},
        timeout=30.0,
        sse_read_timeout=300.0
    ),
    auth_credential=oauth2_credential
)
```

Bearer Token with SSE/HTTP

```
from google.adk.auth.auth_credential import (
    AuthCredential, AuthCredentialTypes, HttpAuth, HttpCredentials
)

# Bearer token authentication
bearer_credential = AuthCredential(
    auth_type=AuthCredentialTypes.HTTP,
    http=HttpAuth(
        scheme='bearer',
        credentials=HttpCredentials(token='your-bearer-token')
    )
)

# With SSE
mcp_tools_sse = MCPToolset(
    connection_params=SseConnectionParams(
        url='https://api.example.com/mcp/sse'
    ),
    auth_credential=bearer_credential
)

# With HTTP
mcp_tools_http = MCPToolset(
    connection_params=StreamableHTTPConnectionParams(
        url='https://api.example.com/mcp/stream'
    ),
    auth_credential=bearer_credential
)
```


Complete Example: Production MCP Server with OAuth2

```

"""
Production MCP Server with OAuth2 Authentication
ADK 1.16.0+ SSE/HTTP Connection Example
"""

import asyncio
import os
from google.adk.agents import Agent, Runner
from google.adk.tools.mcp_tool import MCPToolset, SseConnectionParams
from google.adk.auth.auth_credential import (
    AuthCredential, AuthCredentialTypes, OAuth2Auth
)

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

async def main():
    """Demonstrate OAuth2-secured SSE MCP integration."""

    # OAuth2 configuration for SSE connection
    oauth2_credential = AuthCredential(
        auth_type=AuthCredentialTypes.OAUTH2,
        oauth2=OAuth2Auth(
            client_id=os.environ['OAUTH_CLIENT_ID'],
            client_secret=os.environ['OAUTH_CLIENT_SECRET'],
            auth_uri='https://auth.company.com/oauth/authorize',
            token_uri='https://auth.company.com/oauth/token',
            scopes=['mcp.read', 'mcp.write', 'documents.access']
        )
    )

    # Create MCP toolset with OAuth2 + SSE
    secure_mcp_tools = MCPToolset(
        connection_params=SseConnectionParams(
            url='https://mcp.company.com/sse/production',
            headers={
                'X-Client-Version': 'ADK-1.16.0',
                'X-Environment': 'production'
            },
            timeout=30.0,
            sse_read_timeout=600.0 # 10 minutes for long-running operations
        ),
        auth_credential=oauth2_credential,
        tool_name_prefix='prod_' # Avoid conflicts with other toolsets
    )

```

```

)

# Create agent with authenticated SSE MCP access
agent = Agent(
    model='gemini-2.5-flash',
    name='production_mcp_agent',
    description='Agent with OAuth2-secured SSE MCP access',
    instruction=""

You have authenticated access to production MCP servers via SSE connection.
You can:
- Access real-time data streams
- Execute long-running operations
- Handle streaming responses
- Work with authenticated enterprise resources

Connection details:
- SSE endpoint with OAuth2 authentication
- 10-minute timeout for complex operations
- Production environment access
    """.strip(),
    tools=[secure_mcp_tools]
)

# Run queries with SSE + OAuth2
runner = Runner()

print("\n" + "="*70)
print("PRODUCTION MCP SERVER WITH SSE + OAUTH2")
print("="*70 + "\n")

# Query 1: Real-time data access
result1 = await runner.run_async(
    "Get real-time sales data from the production database.",
    agent=agent
)

print("\n📊 Real-time Sales Data:\n")
print(result1.content.parts[0].text)

await asyncio.sleep(1)

# Query 2: Streaming operation
result2 = await runner.run_async(
    "Process the quarterly financial report and stream results.",
    agent=agent
)

print("\n\n📈 Streaming Financial Report:\n")
print(result2.content.parts[0].text)

```

```
print("\n" + "="*70 + "\n")

if __name__ == '__main__':
    asyncio.run(main())
```

| SSE/HTTP Connection Benefits

SSE (Server-Sent Events):

- ✓ Real-time streaming from server to client
- ✓ Automatic reconnection on connection loss
- ✓ Efficient for server-initiated updates
- ✓ Lower latency than polling
- ✓ Built-in keep-alive mechanism

HTTP Streaming:

- ✓ Bidirectional streaming communication
- ✓ Full-duplex connection (send and receive)
- ✓ Better for interactive, request-response patterns
- ✓ Supports complex authentication flows
- ✓ More flexible than SSE for advanced use cases

| Choosing Connection Type

Feature	Stdio	SSE	HTTP Streaming
Use Case	Local tools	Real-time data	Interactive APIs
Authentication	Limited	Full OAuth2	Full OAuth2
Network	Local only	Remote OK	Remote OK
Streaming	No	Server→Client	Bidirectional
Production	Development	Production	Production
Complexity	Simple	Medium	Medium-High

Recommendations:

- **Development/Local:** Use `StdioConnectionParams`
- **Real-time feeds:** Use `SseConnectionParams` + OAuth2
- **Interactive APIs:** Use `StreamableHTTPConnectionParams` + OAuth2
- **Production Enterprise:** SSE or HTTP with OAuth2 authentication

9. Troubleshooting & Common Issues

| Callback Signature Errors

Error: `TypeError: before_tool_callback() missing 1 required positional argument`

Cause: Using old callback signature with ADK 1.16.0+

```
# ❌ OLD - DON'T USE (< 1.16.0)
def before_tool_callback(callback_context, tool_name, args):
    pass

# ✅ NEW - CORRECT (1.16.0+)
def before_tool_callback(tool, args, tool_context):
    pass
```

Error: `TypeError: before_tool_callback() got an unexpected keyword argument 'tool_name'`

Cause: ADK 1.16.0 changed parameter name from `tool_name` to `tool`

Solution: Update parameter name to `tool`

Error: `AttributeError: 'str' object has no attribute 'state'`

Cause: Trying to access `callback_context.state` which doesn't exist

Solution: Use `tool_context.state` instead:

```
# ❌ WRONG
count = callback_context.state.get('count', 0)

# ✅ CORRECT
count = tool_context.state.get('count', 0)
```

Error: Tool name prints as `<google.adk.tools.mcp_tool.mcp_tool.MCPTool object at 0x...>`

Cause: `tool` parameter is a `BaseTool` object, not a string

Solution: Extract the name:

```
# ✅ CORRECT
tool_name = tool.name if hasattr(tool, 'name') else str(tool)
```

Error: `TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'`

Cause: State value is `None` instead of `0`

Solution: Use `or 0` fallback:

```
# ❌ WRONG
count = tool_context.state.get('count', 0) + 1

# ✅ CORRECT
count = tool_context.state.get('count', 0) or 0
tool_context.state['count'] = count + 1
```

| MCP Server Connection Issues

Error: `npx: command not found`

Solution: Install Node.js and npm

```
# macOS
brew install node

# Ubuntu/Debian
sudo apt install nodejs npm

# Verify
npx --version
```

Error: `ConnectionError: MCP server failed to start`

Solution: Check server path and permissions

```
# Verify server installation
connection_params=StdioConnectionParams(
    command='npx',
    args=[
        '-y', # Auto-install if missing
        '@modelcontextprotocol/server-filesystem',
        '/absolute/path/to/directory' # Use absolute paths!
    ],
    timeout=30.0 # Increase timeout if needed
)
```

Error: `EACCES: permission denied`

Solution: Check directory permissions

```
# Create directory with proper permissions
mkdir -p sample_files
chmod 755 sample_files

# Verify
ls -la sample_files
```

| HITL Approval Issues

Issue: All operations blocked, even read operations

Cause: Overly broad destructive operations list

Solution: Only block write/move/create/delete:

```
DESTRUCTIVE_OPERATIONS = {
    'write_file',
    'move_file',
    'create_directory',
    # Don't include read operations!
}
```

Issue: Auto-approve flag not working

Cause: Using wrong state scope

Solution: Use `user:` prefix for persistent approval:

```
# ❌ WRONG - session-scoped
auto_approve = tool_context.state.get('auto_approve', False)

# ✅ CORRECT - user-scoped (persists across sessions)
auto_approve = tool_context.state.get('user:auto_approve_file_ops', False)
```

Testing Issues

Error: `ImportError: cannot import name 'CallbackContext'`

Cause: Importing removed class from ADK 1.16.0

Solution: Don't import `CallbackContext`:

```
# ❌ DON'T IMPORT
from google.adk.types import CallbackContext

# ✅ USE MOCK INSTEAD
from unittest.mock import Mock

mock_context = Mock()
mock_context.state = {}
```

Issue: Tests pass but real server fails

Cause: Mock doesn't match real ADK behavior

Solution: Test with real ADK Runner:


```
# Add integration test
async def test_with_real_runner():
    from google.adk.agents import Runner

    agent = create_mcp_filesystem_agent()
    runner = Runner()

    result = await runner.run_async(
        "List files",
        agent=agent
    )

    assert result.content
```

Migration Checklist

Upgrading from ADK < 1.16.0? Use this checklist:

- [] Update callback signature to `(tool, args, tool_context)`
- [] Remove `callback_context` parameter
- [] Change `tool_name` to `tool`
- [] Extract tool name: `tool.name if hasattr(tool, 'name') else str(tool)`
- [] Replace `callback_context.state` with `tool_context.state`
- [] Add `or 0` fallbacks for state values
- [] Remove `CallbackContext` imports
- [] Run all tests (unit + integration)
- [] Test with real ADK web server
- [] Update documentation

Summary

You've mastered MCP integration and authentication for extended agent capabilities:

Key Takeaways:

- ✓ MCP provides standardized protocol for external tools

- ✓ `MCPToolset` connects agents to MCP servers
- ✓ `StdioConnectionParams` for stdio-based servers
- ✓ Filesystem server most common (file operations)
- ✓ Session pooling for efficiency
- ✓ `retry_on_closed_resource=True` for reliability
- ✓ **OAuth2 authentication** for secure production deployments
- ✓ Multiple auth methods supported (OAuth2, Bearer, Basic, API Key)
- ✓ Credential management via environment variables or Secret Manager
- ✓ 100+ community MCP servers available
- ✓ Can build custom MCP servers in Node.js

Production Checklist:

- [] Node.js/npx installed
- [] Directory paths validated
- [] `retry_on_closed_resource=True` enabled
- [] **Authentication configured** (OAuth2 for production)
- [] **Credentials stored securely** (environment variables or Secret Manager)
- [] **OAuth2 scopes** set to least-privilege
- [] Clear instructions for MCP tools
- [] Error handling for connection failures
- [] **Authentication error handling** (401, 403)
- [] Testing with actual MCP servers
- [] **Testing with expired tokens**
- [] Monitoring MCP server health
- [] Appropriate permissions for file access


Next Steps:

- **Tutorial 17:** Learn Agent-to-Agent (A2A) communication
- **Tutorial 18:** Master Events & Observability
- **Tutorial 19:** Implement Artifacts & File Management

Resources:

- [MCP Specification \(2025-06-18\)](https://spec.modelcontextprotocol.io/specification/2025-06-18/) (https://spec.modelcontextprotocol.io/specification/2025-06-18/)
- [Official MCP Servers](https://github.com/modelcontextprotocol/servers) (https://github.com/modelcontextprotocol/servers)

- [Sample: mcp_stdio_server_agent](https://github.com/google/adk-python/tree/main/contributing/samples/mcp_stdio_server_agent/) (https://github.com/google/adk-python/tree/main/contributing/samples/mcp_stdio_server_agent/)
-

 **Tutorial 16 Complete!** You now know how to extend your agents with MCP tool servers. Continue to Tutorial 17 to learn about agent-to-agent communication.

Generated on 2025-10-21 09:02:32 from 16_mcp_integration.md

Source: Google ADK Training Hub