Member-only story

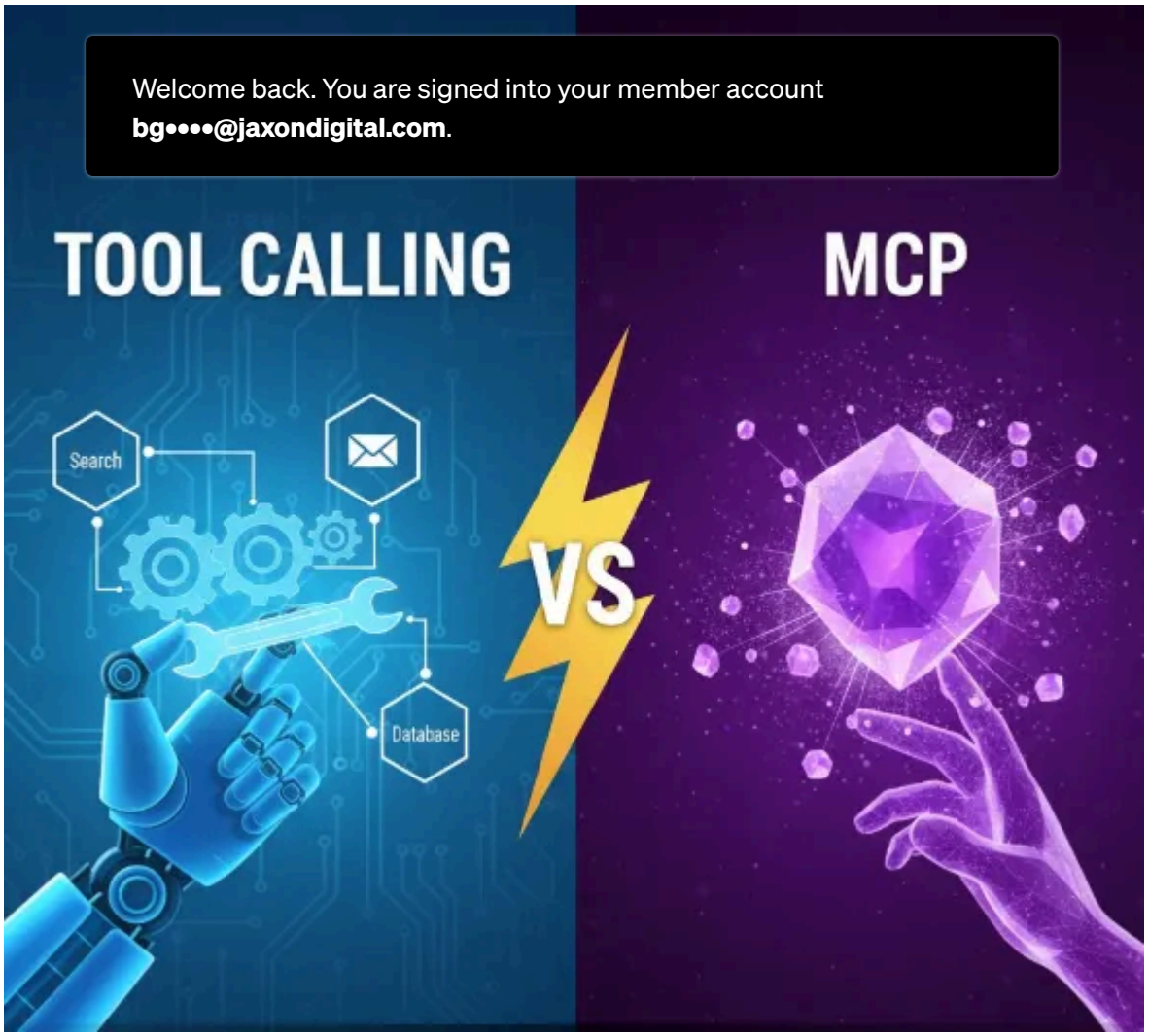
Give Your AI Superpowers: Tool Calling vs MCP Explained

13 min read · 8 hours ago

Somanath diksangi Follow

Listen Share More

If you are building AI agents, you have probably wondered about the best way to give them tools and powers. Should you use tool calling or this new thing called MCP? This guide will explain everything in simple terms with real code examples.



What Are We Talking About?

When we build AI agents, we want them to do more than just chat. We want them to search the web, send emails, calculate numbers, or connect to databases. This is where tool calling and MCP come in — they are two different ways to give your AI agent superpowers.

Think of it like this: your AI agent is like a smart assistant, but it needs tools to actually do things in the real world. Tool calling and MCP are two different toolboxes you can give to your assistant.

Understanding Tool Calling

What is Tool Calling?

Tool calling is the traditional way to give AI models the ability to use functions. You basically tell the model “here are some functions you can call” and it decides when to use them.

Here’s a simple example with OpenAI’s API:

```
import
import
```



```
# Define your tools
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_weather",
            "description": "Get the current weather for a city",
            "parameters": {
                "type": "object",
                "properties": {
                    "city": {
                        "type": "string",
                        "description": "The city name"
                    }
                },
                "required": ["city"]
            }
        }
    },
    {
        "type": "function",
        "function": {
            "name": "send_email",
            "description": "Send an email to someone",
            "parameters": {
                "type": "object",
                "properties": {
                    "to": {"type": "string"},
                    "subject": {"type": "string"},
                    "body": {"type": "string"}
                },
                "required": ["to", "subject", "body"]
            }
        }
    }
]

# Your actual functions
def get_weather(city):
    # This would call a real weather API
    return f"The weather in {city} is sunny and 25°C"

def send_email(to, subject, body):
    # This would send a real email
    return f"Email sent to {to} with subject '{subject}'"

# Function to handle tool calls
def execute_function(function_name, arguments):
    if function_name == "get_weather":
```

```
        return get_weather(**arguments)
    eli
    els
        return "Function not found"

# Chat with tool calling
client = openai.OpenAI(api_key="your-api-key")

response = client.chat.completions.create(
    model="gpt-4",
    messages=[
        {"role": "user", "content": "What's the weather like in
Mumbai?"}
    ],
    tools=tools,
    tool_choice="auto"
)

# Check if the model wants to call a function
if response.choices[0].message.tool_calls:
    tool_call = response.choices[0].message.tool_calls[0]
    function_name = tool_call.function.name
    arguments = json.loads(tool_call.function.arguments)

    # Execute the function
    result = execute_function(function_name, arguments)
    print(f"Function result: {result}")
```


Pros of Tool Calling

Simple to get started: You just define functions and pass them to the model. No complex setup needed.

Fast: Direct function calls with minimal overhead. Perfect when you need quick responses.

Full control: You decide exactly what functions the model can access and how they work.

Cons of Tool Calling

Model-specific: Each AI model (OpenAI, Claude, Gemini) has its own way of handling tools. You need different code for different models.

Hard to reuse: If you build tools for one project, it’s difficult to use them in another project without copying code.

No discovery: The model can only use tools you explicitly give it. It can’t discover new tools on its own.

Understanding MCP (Model Context Protocol)

What is M

MCP is a new standard that makes it easier for AI agents to find and use tools. Instead of hardcoding tools into your agent, MCP creates a marketplace of tools that any agent can discover and use.

Think of MCP like an app store for AI agent tools. Just like you can download apps on your phone without knowing how to code them, your AI agent can discover and use tools through MCP servers.

How MCP Works

MCP has three main parts:

1. MCP Server: Hosts tools and makes them available
2. MCP Client: Connects your agent to MCP servers
3. Tools: The actual functions your agent can use

Here's a simple MCP server example:

```
# mcp_server.py
from mcp import Server, Tool
import httpx
```

```
class WeatherServer:
    def __init__(self):
        self.server = Server("weather-server")
        self.setup_tools()

    def setup_tools(self):
        @self.server.tool("get_weather")
        async def get_weather(city: str) -> str:
            """Get current weather for a city"""
            # This would call a real weather API
            async with httpx.AsyncClient() as client:
                # Simulate API call
                return f"Weather in {city}: Sunny, 25°C"

        @self.server.tool("get_forecast")
        async def get_forecast(city: str, days: int = 5) -> str:
            """Get weather forecast for multiple days"""
            return f"{days}-day forecast for {city}: Mostly sunny"
```

```
    async def run(self, port: int = 8000):

# Run t
if __na
    import asyncio
    server = WeatherServer()
    asyncio.run(server.run())
```


Now here's how an MCP client would connect and use these tools:

```
# mcp_client.py
from mcp import Client
import asyncio
```

```
class AIAgent:
    def __init__(self):
        self.client = Client()

    async def connect_to_servers(self):
        # Connect to weather server
        await self.client.connect("http://localhost:8000")

        # You can connect to multiple servers
        # await self.client.connect("http://localhost:8001")  #
Email server
        # await self.client.connect("http://localhost:8002")  #
Database server

    async def discover_tools(self):
        """See what tools are available"""
        tools = await self.client.list_tools()
        print("Available tools:")
        for tool in tools:
            print(f"- {tool.name}: {tool.description}")
        return tools

    async def use_tool(self, tool_name: str, **kwargs):
        """Use a specific tool"""
        result = await self.client.call_tool(tool_name, **kwargs)
        return result

    async def chat_with_tools(self, user_message: str):
        """Process user message and use tools when needed"""
        # This is where you'd integrate with your LLM
        # The LLM would decide which tools to call based on
available tools

        if "weather" in user_message.lower():
```

```
            if "mumbai" in user_message.lower():
city="M
        return "I didn't understand what tool you need."

# Usage
async def main():
    agent = AIAgent()

    # Connect to MCP servers
    await agent.connect_to_servers()

    # Discover available tools
    tools = await agent.discover_tools()

    # Use tools
    response = await agent.chat_with_tools("What's the weather in
Mumbai?")
    print(response)

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


Advanced MCP Example with Multiple Tools

Here's a more complete example showing how MCP makes it easy to build complex agents:

```
# multi_tool_server.py
from mcp import Server
import json
import sqlite3
from datetime import datetime
```

```
class BusinessServer:
    def __init__(self):
        self.server = Server("business-tools")
        self.db = sqlite3.connect("business.db")
        self.setup_database()
        self.setup_tools()

    def setup_database(self):
        """Setup a simple database"""
        cursor = self.db.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS customers (
                id INTEGER PRIMARY KEY,
```

```
                name TEXT,
                created_at TEXT
            )
        """)
        self.db.commit()

    def setup_tools(self):
        @self.server.tool("add_customer")
        async def add_customer(name: str, email: str, phone: str =
None) -> str:
            """Add a new customer to the database"""
            cursor = self.db.cursor()
            cursor.execute(
                "INSERT INTO customers (name, email, phone,
created_at) VALUES (?, ?, ?, ?)",
                (name, email, phone, datetime.now().isoformat())
            )
            self.db.commit()
            return f"Customer {name} added successfully!"

        @self.server.tool("find_customer")
        async def find_customer(search_term: str) -> str:
            """Find customers by name or email"""
            cursor = self.db.cursor()
            cursor.execute(
                "SELECT * FROM customers WHERE name LIKE ? OR email
LIKE ?",
                (f"%{search_term}%", f"%{search_term}%")
            )
            results = cursor.fetchall()

            if not results:
                return "No customers found"

            customers = []
            for row in results:
                customers.append({
                    "id": row[0],
                    "name": row[1],
                    "email": row[2],
                    "phone": row[3],
                    "created_at": row[4]
                })

            return json.dumps(customers, indent=2)

        @self.server.tool("send_marketing_email")
        async def send_marketing_email(subject: str, message: str) -
> str:
            """Send marketing email to all customers"""
            cursor = self.db.cursor()
            cursor.execute("SELECT email FROM customers WHERE email
IS NOT NULL")
```

```
            emails = cursor.fetchall()
            return f"Marketing email '{subject}' sent to
{email_count} customers"

        @self.server.tool("calculate_discount")
        async def calculate_discount(original_price: float,
discount_percent: float) -> str:
            """Calculate discounted price"""
            discount_amount = original_price * (discount_percent /
100)
            final_price = original_price - discount_amount

            return json.dumps({
                "original_price": original_price,
                "discount_percent": discount_percent,
                "discount_amount": round(discount_amount, 2),
                "final_price": round(final_price, 2)
            })

    async def run(self, port: int = 8000):
        print(f"Starting business server on port {port}")
        await self.server.run(port=port)
```


And here's how an agent would use these tools:

```
# business_agent.py
from mcp import Client
import json
import asyncio
```

```
class BusinessAgent:
    def __init__(self):
        self.client = Client()
        self.tools = {}

    async def setup(self):
        """Connect to servers and discover tools"""
        await self.client.connect("http://localhost:8000")

        # Discover all available tools
        tools_list = await self.client.list_tools()
        for tool in tools_list:
            self.tools[tool.name] = tool

        print(f"Connected! Available tools:
{list(self.tools.keys())}")
```

```
    asy
```



```
        if "add customer" in request_lower:
            # Extract customer info (in a real app, use NLP for
this)
            if "john" in request_lower and "john@" in request_lower:
                result = await self.client.call_tool(
                    "add_customer",
                    name="John Doe",
                    email="john@example.com",
                    phone="9876543210"
                )
                return result

        elif "find customer" in request_lower:
            if "john" in request_lower:
                result = await
self.client.call_tool("find_customer", search_term="john")
                return f"Found customers: {result}"

        elif "send email" in request_lower:
            result = await self.client.call_tool(
                "send_marketing_email",
                subject="Special Offer!",
                message="Get 20% off this month!"
            )
            return result

        elif "discount" in request_lower:
            result = await self.client.call_tool(
                "calculate_discount",
                original_price=1000.0,
                discount_percent=20.0
            )
            return f"Discount calculation: {result}"

        return "I didn't understand your request. Available actions:
add customer, find customer, send email, calculate discount"

# Usage example
async def main():
    agent = BusinessAgent()
    await agent.setup()

    # Test different requests
    requests = [
        "Add customer John Doe with email john@example.com",
        "Find customer john",
        "Send marketing email to all customers",
        "Calculate 20% discount on 1000 rupees"
    ]
```

```
    for request in requests:
        print(request)

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


Pros of MCP

Model agnostic: Works with any AI model that supports MCP. You write tools once, use everywhere.

Tool discovery: Agents can discover new tools automatically when you connect to new servers.

Reusability: Build tools once, share them across projects and teams.

Standardization: Everyone follows the same protocol, making integration easier.

Scalability: Easy to add new tools without changing your main agent code.

Cons of MCP

More complex: Requires setting up servers and clients. Not as simple as direct function calls.

Network overhead: Tools are called over the network, which adds some latency.

Still new: MCP is relatively new, so fewer tools and examples are available compared to traditional tool calling.

Storm MCP: The Gateway Solution

Storm MCP is a special tool that acts as a gateway, making it even easier to work with MCP servers. Think of it as a traffic controller that helps your agent talk to many different MCP servers at once.

Here's how you might use Storm MCP:

```
# storm_mcp_example.py
from storm_mcp import StormGateway
import asyncio
```

```
class S
    def
```



```
    async def setup(self):
        """Setup connections through Storm MCP"""
        # Register multiple MCP servers
        await self.storm.register_server("weather",
"http://localhost:8000")
        await self.storm.register_server("business",
"http://localhost:8001")
        await self.storm.register_server("email",
"http://localhost:8002")

        print("Connected to all servers through Storm MCP!")

    async def handle_request(self, user_input: str):
        """Handle user requests using Storm MCP"""
        # Storm MCP can route requests to appropriate servers
        result = await self.storm.process_request(user_input)
        return result

# Usage
async def main():
    agent = SmartAgent()
    await agent.setup()

    # Now the agent can use tools from multiple servers seamlessly
    response = await agent.handle_request("What's the weather and
add a new customer")
    print(response)

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

Using MCP in Cursor IDE

Cursor IDE has built-in support for MCP, making it easy to give your code editor AI assistant access to custom tools.

Here's how to set it up:

1. Create an MCP configuration file (cursor_mcp.json):

```
{
  "mcpServers": {
    "weather": {
      "url": "http://localhost:8000",
      "description": "Weather tools for getting forecasts"
```

```
    },
    "bu
      '
      '
    }
  }
}
```


1. Configure Cursor settings to use your MCP servers:

```
{
  "mcp.enabled": true,
  "mcp.configFile": "./cursor_mcp.json",
  "mcp.autoConnect": true
}
```

Now when you chat with Cursor's AI, it can use your custom tools!

Real-World Comparison

Let's look at a real scenario to understand when to use each approach:

Scenario: Building a Customer Service Agent

With Tool Calling:

```
# traditional_agent.py - Tool Calling Approach
import openai
import json
import sqlite3
import smtplib
```

```
class CustomerServiceAgent:
    def __init__(self):
        self.client = openai.OpenAI()
        self.db = sqlite3.connect("customers.db")

        # Define all tools upfront
        self.tools = [
            {
                "type": "function",
                "function": {
```

```
                    "name": "search_customer",
informa
                    "type": "object",
                    "properties": {
                        "query": {"type": "string"}
                    }
                }
            }
        },
        {
            "type": "function",
            "function": {
                "name": "create_ticket",
                "description": "Create a support ticket",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "customer_id": {"type": "string"},
                        "issue": {"type": "string"},
                        "priority": {"type": "string"}
                    }
                }
            }
        }
    ]

    def search_customer(self, query):
        # Database search logic
        cursor = self.db.cursor()
        cursor.execute("SELECT * FROM customers WHERE name LIKE ?",
(f"%{query}%",))
        return cursor.fetchone()

    def create_ticket(self, customer_id, issue, priority):
        # Ticket creation logic
        ticket_id = f"TICKET-{customer_id}-{len(issue)}"
        return f"Created ticket {ticket_id}"

    def handle_request(self, message):
        response = self.client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": message}],
            tools=self.tools
        )

        # Handle tool calls...
        return "Response processed"
```


With MCP:

```
# mcp_a
from mc
import
```



```
class CustomerServiceAgent:
    def __init__(self):
        self.mcp_client = Client()

    async def setup(self):
        # Connect to different service servers
        await self.mcp_client.connect("http://customer-db-
server:8000")
        await self.mcp_client.connect("http://ticketing-
server:8001")
        await self.mcp_client.connect("http://email-server:8002")

        # Tools are discovered automatically!
        self.available_tools = await self.mcp_client.list_tools()

    async def handle_request(self, message):
        # The agent can use any discovered tools
        if "search customer" in message.lower():
            result = await
self.mcp_client.call_tool("search_customer", query="john")
            return result
        elif "create ticket" in message.lower():
            result = await
self.mcp_client.call_tool("create_ticket",

customer_id="123",
                                                    issue="Login
problem",
                                                    priority="high")
            return result

        return "Request handled with MCP tools"
```

When to Choose What

Choose Tool Calling When:

- Building a prototype: You need something working quickly and don't mind hardcoding functions.
- Simple agent: Your agent only needs 2–3 basic tools and won't grow much.
- Performance critical: You need the absolute fastest response times and can't afford network calls.

- Model-specific features: You want to use special features of one particular AI model.

Choose MCP When:

- Building for scale: You expect to add many tools over time or work with multiple projects.
- Team collaboration: Multiple developers will build and share tools.
- Model flexibility: You might switch between different AI models (OpenAI, Claude, Gemini).
- Tool reusability: You want to use the same tools across different agents and applications.
- Enterprise setup: You need standardized protocols and governance over tool usage.

Migration Strategy

If you're currently using tool calling and want to move to MCP, here's a step-by-step approach:

Step 1: Identify Reusable Tools

Look at your current tools and identify which ones could be useful in other projects:

```
# Current tool calling functions
def get_weather(city): pass
def send_email(to, subject, body): pass
def query_database(sql): pass
def calculate_taxes(income): pass
```

Step 2: Convert to MCP Servers

Create MCP servers for groups of related tools:

```
# weather_server.py
from mcp import Server
```

```
class W
    def
```



```
    @self.server.tool("get_weather")
    async def get_weather(self, city: str) -> str:
        # Your existing weather function logic
        pass
```

Step 3: Gradual Migration

Start by using MCP for new tools while keeping existing tool calling for critical functions:

```
class HybridAgent:
    def __init__(self):
        # Keep existing critical tools
        self.critical_tools = ["emergency_shutdown", "user_authentication"]

        # Use MCP for new tools
        self.mcp_client = Client()

    async def handle_request(self, request):
        # Use tool calling for critical functions
        if "emergency" in request:
            return self.handle_with_tool_calling(request)

        # Use MCP for everything else
        return await self.handle_with_mcp(request)
```

Performance Considerations

Tool Calling Performance

```
import time
```

```
# Tool calling - direct function call
start_time = time.time()
result = get_weather("Mumbai")  # ~0.1-0.5 seconds
end_time = time.time()
print(f"Tool calling took: {end_time - start_time} seconds")
```

MCP Performance


```
import time
import asyncio
```

```
# MCP - network call to server
async def test_mcp_performance():
    start_time = time.time()
    result = await mcp_client.call_tool("get_weather",
city="Mumbai")  # ~0.2-1.0 seconds
    end_time = time.time()
    print(f"MCP took: {end_time - start_time} seconds")

asyncio.run(test_mcp_performance())
```

Performance Tips for MCP:

1. Keep servers local when possible to reduce network latency
2. Batch tool calls when you need multiple operations
3. Cache results for frequently used tools
4. Use connection pooling for better network efficiency

Security Considerations

Tool Calling Security

With tool calling, security is your responsibility:

```
def secure_database_query(sql, user_role):
    # Validate user permissions
    if user_role != "admin" and any(keyword in sql.lower() for keyword in ["dro
        raise PermissionError("User not authorized for this operation")

    # Sanitize SQL to prevent injection
    # ... validation logic

    return execute_query(sql)
```

MCP Security

MCP serve


```
from mcp import Server, AuthMiddleware
```

```
class SecureBusinessServer:
    def __init__(self):
        self.server = Server("secure-business")
        self.server.use_middleware(AuthMiddleware(
            api_key_required=True,
            role_based_access=True
        ))

    @self.server.tool("delete_customer", required_role="admin")
    async def delete_customer(self, customer_id: str) -> str:
        # Only admins can delete customers
        return f"Customer {customer_id} deleted"

    @self.server.tool("view_customer", required_role="user")
    async def view_customer(self, customer_id: str) -> str:
        # Any authenticated user can view
        return f"Customer details for {customer_id}"
```

Testing Your Agents

Testing Tool Calling Agents

```
import unittest
from unittest.mock import patch
```

```
class TestToolCallingAgent(unittest.TestCase):
    def setUp(self):
        self.agent = CustomerServiceAgent()

    @patch('sqlite3.connect')
    def test_customer_search(self, mock_db):
        # Mock database response
        mock_cursor = mock_db.return_value.cursor.return_value
        mock_cursor.fetchone.return_value = ("123", "John Doe",
"john@example.com")

        result = self.agent.search_customer("John")
        self.assertEqual(result[1], "John Doe")
```

```
    def
"high")
if __name__ == "__main__":
    unittest.run()
```


Testing MCP Agents

```
import pytest
import asyncio
from mcp import MockServer
```

```
class TestMCPAgent:
    @pytest.fixture
    async def mock_mcp_server(self):
        server = MockServer("test-server")

        @server.tool("test_tool")
        async def test_tool(param: str) -> str:
            return f"Test result for {param}"

        await server.start()
        return server

    @pytest.mark.asyncio
    async def test_agent_with_mcp(self, mock_mcp_server):
        agent = CustomerServiceAgent()
        await agent.mcp_client.connect(mock_mcp_server.url)

        result = await agent.mcp_client.call_tool("test_tool",
param="testing")
        assert "Test result for testing" in result
```

Common Mistakes to Avoid

Tool Calling Mistakes

1. Not handling tool call failures properly
2. Hardcoding function names instead of using variables
3. Forgetting to validate function parameters

4. Not providing clear descriptions for tools


```
# Bad example
tools = [
    {
        "type": "function",
        "function": {
            "name": "func1",  # Unclear name
            "description": "does stuff",  # Vague description
            "parameters": {}  # Missing parameter validation
        }
    }
]
```

```
# Good example
tools = [
    {
        "type": "function",
        "function": {
            "name": "calculate_monthly_payment",  # Clear name
            "description": "Calculate monthly loan payment based on
principal, rate, and term",  # Clear description
            "parameters": {
                "type": "object",
                "properties": {
                    "principal": {
                        "type": "number",
                        "description": "Loan amount in dollars",
                        "minimum": 0
                    },
                    "annual_rate": {
                        "type": "number",
                        "description": "Annual interest rate as
decimal (e.g., 0.05 for 5%)",
                        "minimum": 0,
                        "maximum": 1
                    },
                    "years": {
                        "type": "integer",
                        "description": "Loan term in years",
                        "minimum": 1,
                        "maximum": 30
                    }
                },
                "required": ["principal", "annual_rate", "years"]
            }
        }
```

```
    }
]
```


MCP Mistakes

1. Not handling server disconnections gracefully
2. Creating too many small servers instead of grouping related tools
3. Forgetting to validate tool inputs on the server side
4. Not implementing proper error handling

```
# Bad: Too many servers for related functionality
# weather_server.py - just for current weather
# forecast_server.py - just for forecasts
# alerts_server.py - just for weather alerts
```

```
# Good: One server for related functionality
class WeatherServer:
    @server.tool("get_current_weather")
    async def get_current_weather(self, city: str): pass

    @server.tool("get_forecast")
    async def get_forecast(self, city: str, days: int): pass

    @server.tool("get_weather_alerts")
    async def get_weather_alerts(self, city: str): pass
```

Future of AI Agent Tools

The field is moving toward MCP for several reasons:

Standardization: Just like we have HTTP for web communication, we need standards for AI agent tools.

Ecosystem growth: As more tools become available through MCP, agents become more powerful without custom development.

Interoperability: Tools built for one agent can work with any other agent that supports MCP.

Here's what the future might look like:

```
# Futur
class F
    asy
        # Connect to MCP marketplace
        marketplace = MCPMarketplace()

        # Discover and install tools based on user needs
        tools = await marketplace.discover_tools(
            categories=["productivity", "business", "communication"],
            user_preferences=self.user.preferences
        )

        # Auto-connect to relevant servers
        for tool_server in tools:
            await self.client.connect(tool_server.url)

        print(f"Connected to {len(tools)} tool servers automatically!")

    async def handle_request(self, request):
        # AI automatically figures out which tools to use
        plan = await self.create_execution_plan(request)
        return await self.execute_plan(plan)
```


Conclusion

Both tool calling and MCP have their place in the AI agent world. Tool calling is great when you need something quick and simple, while MCP is better for building scalable, reusable, and maintainable agent systems.

Here's my recommendation based on your situation:

Start with tool calling if:

- You're learning and experimenting
- Building a proof of concept
- Need just 1–2 simple tools
- Working alone on a small project

Move to MCP when:

- Your agent needs more than 5 tools
- Working with a team

- Building for production
- Planni
- Want to share tools with others

Remember, you can always start with tool calling and migrate to MCP as your needs grow. The important thing is to build agents that solve real problems, regardless of which approach you choose.

The future belongs to agents that can use many tools seamlessly, and MCP is helping make that future possible. But for now, choose the approach that gets you building and learning fastest.

Happy agent building!

AI Tools Ai Agent Mcp Server Python

Written by Somanath diksangi

11 followers · 16 following

No responses yet

Bgerby

What are your thoughts?

More from

Somanath diksangi

5 Chunking Strategies For RAG

Here's the typical workflow of RAG:

Oct 1 2 1

In AWS in Plain English by Somanath diksangi

I Built a Lo

Complete RA , and Gradio interface

Oct 14 1

In Python in Plain English by Somanath diksangi

I Built a Local ChatGPT-Style RAG System with Ollama and Gradio

What we built (high level)

6d ago

Somanath diksangi

You Will NEVER Use Pandas' Describe Method After Using These Two Libraries

Generate a comprehensive data summary in seconds.

Oct 3 10 1

See all from Somanath diksangi

Recommended from Medium

CherryZhou

Claude Skills: Anthropic's New Modular System to Boost AI Agent Productivity

Claude Skills are here: modular, reusable containers that give AI specialized capabilities for any business task.

3d ago 5

Jon Capriola

From Prompts to Payloads: How Agents Safely Command the Physical World

Security. The [illegible] should turn into mo[illegible]

Sep 8

Pawel

Claude Skills: The AI Feature That Actually Solves a Real Problem

Yesterday, Anthropic quietly released what might be the most practical AI feature of 2025. It's not flashier models or better benchmarks...

4d ago 119 4

Svoboda Pavel

From Cha

I'm Pavel, an t for nearly 15 years. These days, I have an AI...

5d ago 1

In Dare To Be Better by Max Petrusenko

Claude Skills: The $3 Automation Secret That's Making Enterprise Teams Look Like Wizards

How a simple folder is replacing $50K consultants and saving companies literal days of work

4d ago 97 3

In CodeX by AI Rabbit

Anthropic Combined 3 Years of AI Lessons Into One Feature

Anthropic just released a new feature that might change the way we interact with AI entirely. After years of experience with generative AI...

2d ago 295 7

See more recommendations