MCP Server Implementation Guide

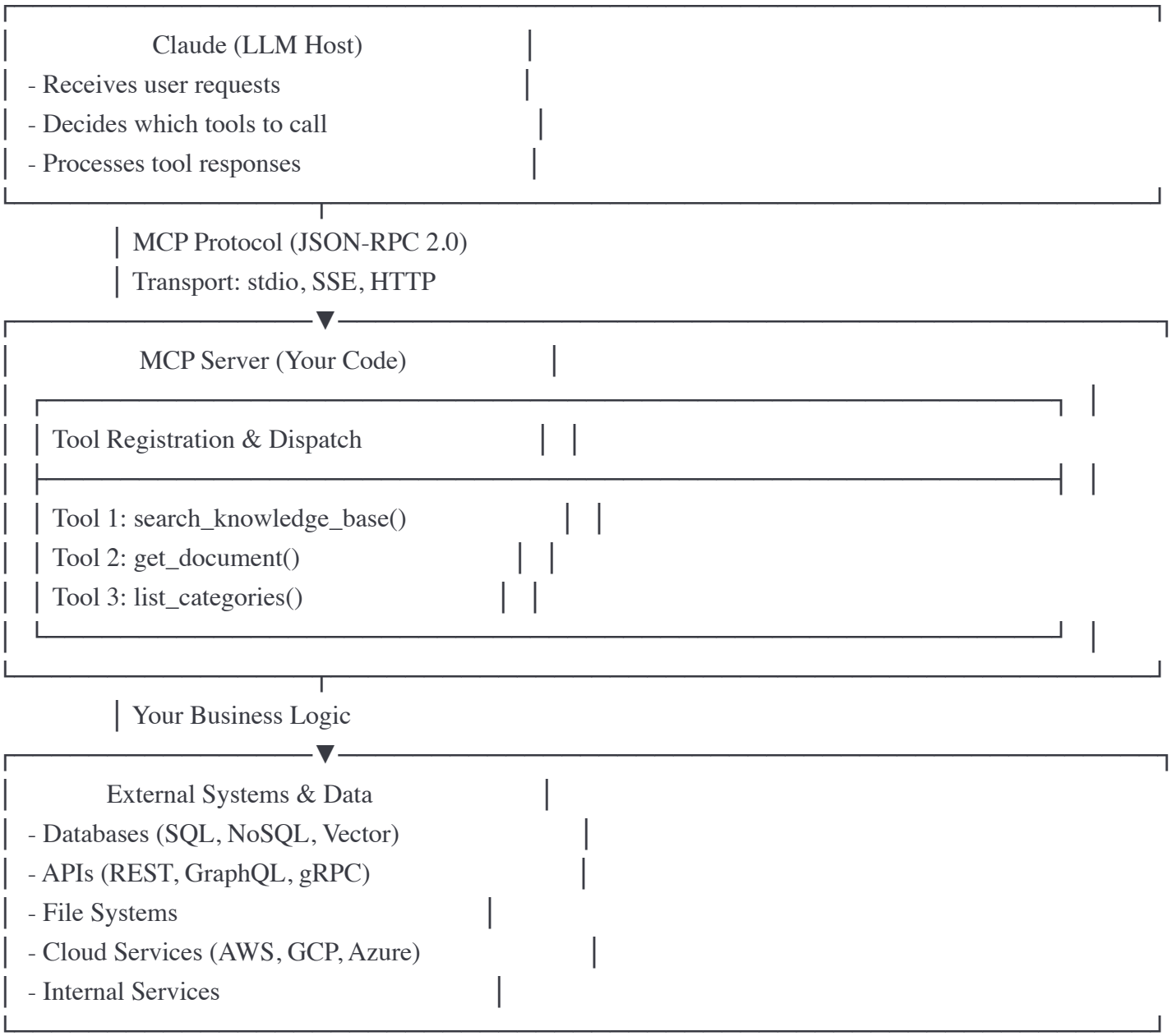
For Agentic Automation Solution Architects

This guide provides a comprehensive understanding of MCP (Model Context Protocol) server implementation with production-ready examples and architectural patterns.

Table of Contents

Architecture Overview

The MCP Stack



Communication Flow

1. User Message → Claude
2. Claude analyzes message → Determines tool needed
3. Claude → MCP Server: call_tool(name="search_knowledge_base", args={...})
4. MCP Server → External System: Fetch data
5. External System → MCP Server: Return data
6. MCP Server → Claude: Format and return result
7. Claude → User: Synthesize response with tool data

Core Concepts

1. Tools (Primary Mechanism)

Tools are functions that Claude can invoke. They are the primary way to extend Claude's capabilities.

Key Characteristics:

- **Discoverable**: Claude can list all available tools
- **Self-describing**: Each tool has a detailed description and schema
- **Typed**: Input/output schemas enforce type safety
- **Stateless**: Each call is independent (though you can implement state)

Tool Metadata:

python

```
@server.list_tools()
async def list_tools() -> list[Tool]:
    return [
        Tool(
            name="tool_name",
            description="What it does, when to use it, examples",
            inputSchema=InputModel.model_json_schema()
        )
    ]
```

2. Resources (Optional)

Resources are data that Claude can read. Unlike tools (which Claude calls), resources are passive data sources.

Use cases:

- Configuration files
- Templates
- Static documentation
- System state

3. Prompts (Optional)

Pre-defined prompt templates that users can invoke.

Use cases:

- Common workflows
- Standardized analysis patterns
- Templated interactions

4. Transports

How Claude communicates with your server:

- **stdio**: Standard input/output (most common, simplest)
- **SSE**: Server-Sent Events (for web apps)
- **HTTP**: Standard HTTP requests

Implementation Walkthrough

Step 1: Project Structure

```
my-mcp-server/
├── server.py              # Main server code
├── requirements.txt       # Dependencies
├── config.json            # Configuration
├── README.md              # Documentation
└── tests/                 # Test suite
    └── test_server.py
```

Step 2: Dependencies

requirements.txt:

```
mcp>=0.9.0
pydantic>=2.0.0
aiohttp>=3.9.0          # If calling HTTP APIs
asyncpg>=0.29.0         # If using PostgreSQL
redis>=5.0.0            # If using Redis
```

Install:

bash

```
pip install -r requirements.txt
```

Step 3: Core Server Structure

The example `knowledge_base_server.py` demonstrates the essential components:

A. Imports and Configuration

python

```python
from mcp.server import Server
from mcp.types import Tool, TextContent
from pydantic import BaseModel, Field, ConfigDict
```

B. Input Models (Type Safety)

python

```python
class SearchInput(BaseModel):
    model_config = ConfigDict(extra='forbid')  # Reject unknown fields

    query: str = Field(
        description="Search query",
        min_length=1,
        max_length=500,
        examples=["authentication best practices"]
    )
```

Why Pydantic?

- Automatic validation
- JSON schema generation
- Type checking
- Clear error messages

C. Server Initialization

python

```python
server = Server("my-server-name")
```

D. Tool Registration

python

```python
@server.list_tools()
async def list_tools() -> list[Tool]:
    return [Tool(...)]
```

E. Tool Implementation

python

```python
@server.call_tool()
async def call_tool(name: str, arguments: Any) -> list[TextContent]:
    if name == "my_tool":
        input_data = MyInput(**arguments)
        result = await do_something(input_data)
        return [TextContent(type="text", text=result)]
```

F. Main Entry Point

python

```python
async def main():
    from mcp.server.stdio import stdio_server

    async with stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            server.create_initialization_options()
        )

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

Design Patterns for Architects

Pattern 1: Search → Fetch Pattern

Problem: Large data sets can't fit in one response

Solution: Two-step pattern

1. Search tool returns IDs and summaries
2. Fetch tool retrieves full details by ID


python

```
# Step 1: Search (returns 10 results with IDs)
search_knowledge_base(query="kubernetes")
→ Returns: [doc-005: "Container Orchestration...", ...]

# Step 2: Fetch specific document
get_document(document_id="doc-005")
→ Returns: Full content of document
```

Benefits:

- Respects context limits
- Allows refinement
- More efficient token usage

Pattern 2: Pagination Pattern

Problem: Large result sets

Solution: Return results in pages with tokens


python

```python
class SearchInput(BaseModel):
    query: str
    page_size: int = Field(default=10, le=100)
    page_token: Optional[str] = None

# Response includes next_page_token
{
    "results": [...],
    "next_page_token": "abc123"
}
```

Pattern 3: Format Flexibility Pattern

Problem: Different use cases need different formats

Solution: Support both JSON and Markdown


python

```python
class SearchInput(BaseModel):
    format: Literal["json", "markdown"] = "markdown"
    detail_level: Literal["concise", "detailed"] = "concise"
```

When to use each:

- **Markdown**: Human-readable, final answers
- **JSON**: Machine-readable, intermediate steps, data processing

Pattern 4: Composite Operations Pattern

Problem: Common workflows require multiple API calls

Solution: Create workflow tools that combine operations


python

```python
# Instead of:
#   1. check_availability()
#   2. create_event()
#   3. send_invites()

# Create:
async def schedule_meeting(params):
    """One tool that does all three"""
    available_slots = await check_availability(...)
    if available_slots:
        event = await create_event(...)
        await send_invites(...)
        return event
```

Pattern 5: Error Recovery Pattern

Problem: External systems fail

Solution: Actionable error messages


python

```
except RateLimitError as e:
    return [TextContent(
        type="text",
        text=f"Rate limit exceeded. Try again in {e.retry_after} seconds.\n\n"
             f"Alternatively, reduce max_results or use more specific filters."
    )]
```

Production Considerations

1. Context Window Management

Character Limits:


python

```python
CHARACTER_LIMIT = 25000  # ~6,250 tokens

def truncate_text(text: str, max_chars: int = CHARACTER_LIMIT) -> str:
    if len(text) <= max_chars:
        return text
    return text[:max_chars - 50] + "\n\n[Content truncated]"
```

Best Practices:

- Default to concise responses
- Offer detail_level parameter
- Implement pagination
- Truncate gracefully

2. Authentication & Security

python

```python
import os
from typing import Optional

class ServerConfig:
    def __init__(self):
        # Never hardcode secrets
        self.api_key = os.environ.get("API_KEY")
        self.database_url = os.environ.get("DATABASE_URL")

        if not self.api_key:
            raise ValueError("API_KEY environment variable required")

config = ServerConfig()
```

Security Checklist:

- ✅ Use environment variables for secrets
- ✅ Validate all inputs
- ✅ Sanitize outputs (no sensitive data leakage)
- ✅ Implement rate limiting
- ✅ Use HTTPS for external APIs
- ✅ Log security events (not sensitive data)

3. Error Handling

python

```python
async def call_tool(name: str, arguments: Any) -> list[TextContent]:
    try:
        # Validate input
        input_data = InputModel(**arguments)

        # Execute operation
        result = await perform_operation(input_data)

        # Return result
        return [TextContent(type="text", text=result)]

    except ValidationError as e:
        # Pydantic validation errors
        return [TextContent(
            type="text",
            text=f"Invalid input: {e}\n\nPlease check parameter types and values."
        )]

    except ExternalAPIError as e:
        # External service errors
        return [TextContent(
            type="text",
            text=f"External service error: {e}\n\nThe service may be temporarily unavailable."
        )]

    except Exception as e:
        # Unexpected errors
        logger.exception(f"Unexpected error in {name}")
        return [TextContent(
            type="text",
            text=f"An unexpected error occurred. Please try again or contact support."
        )]
```

4. Performance Optimization

python

```python
# Connection pooling for databases
from asyncpg import create_pool

pool = await create_pool(
    database_url,
    min_size=5,
    max_size=20,
    command_timeout=10
)

# Caching for frequent requests
from functools import lru_cache

@lru_cache(maxsize=1000)
def get_cached_data(key: str):
    return expensive_operation(key)

# Parallel requests
import asyncio

async def fetch_multiple(ids: list[str]):
    tasks = [fetch_one(id) for id in ids]
    return await asyncio.gather(*tasks)
```

5. Logging and Observability

python

```python
import logging
import structlog

# Structured logging
logger = structlog.get_logger()

async def call_tool(name: str, arguments: Any):
    logger.info(
        "tool_called",
        tool_name=name,
        arguments=arguments
    )

    start_time = time.time()
    try:
        result = await execute_tool(name, arguments)

        logger.info(
            "tool_succeeded",
            tool_name=name,
            duration_ms=(time.time() - start_time) * 1000
        )

        return result

    except Exception as e:
        logger.error(
            "tool_failed",
            tool_name=name,
            error=str(e),
            duration_ms=(time.time() - start_time) * 1000
        )
        raise
```

Testing and Debugging

Unit Testing Tools

python

```python
import pytest
from knowledge_base_server import search_documents

@pytest.mark.asyncio
async def test_search_documents():
    results = await search_documents(
        query="authentication",
        max_results=5
    )

    assert len(results) > 0
    assert all('id' in doc for doc in results)
    assert results[0]['title']  # Has content

@pytest.mark.asyncio
async def test_search_no_results():
    results = await search_documents(
        query="nonexistent_term_xyz",
        max_results=5
    )

    assert len(results) == 0
```

Integration Testing

python

```python
import subprocess
import json

def test_server_startup():
    """Test that server starts without errors"""
    process = subprocess.Popen(
        ['python', 'knowledge_base_server.py'],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE
    )

    # Send initialize request
    initialize_request = {
        "jsonrpc": "2.0",
        "id": 1,
        "method": "initialize",
        "params": {
            "protocolVersion": "2024-11-05",
            "capabilities": {},
            "clientInfo": {
                "name": "test-client",
                "version": "1.0.0"
            }
        }
    }

    process.stdin.write(json.dumps(initialize_request).encode() + b'\n')
    process.stdin.flush()

    # Check response (simplified)
    # In real testing, parse JSON-RPC response
    process.terminate()
    process.wait(timeout=5)
```

Debugging Tips

1. Enable Verbose Logging:

python

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

2. Test in tmux:

bash

```
# Terminal 1: Run server
tmux new -s mcp
python knowledge_base_server.py

# Terminal 2: Send test requests
# (Use Claude Desktop or MCP inspector)
```

3. Use MCP Inspector:

bash

```
npx @modelcontextprotocol/inspector python knowledge_base_server.py
```

This opens a web UI where you can:

- List available tools
- Call tools with custom inputs
- See JSON-RPC messages
- Debug responses

Advanced Patterns

Pattern 1: Database Integration

python

```python
import asyncpg

class DatabaseMCPServer:
    def __init__(self):
        self.pool = None

    async def initialize(self):
        self.pool = await asyncpg.create_pool(
            'postgresql://user:pass@localhost/db'
        )

    async def query_database(self, sql: str, params: list):
        async with self.pool.acquire() as conn:
            return await conn.fetch(sql, *params)

# Tool implementation
async def search_users(query: str):
    sql = """
        SELECT id, name, email
        FROM users
        WHERE name ILIKE $1 OR email ILIKE $1
        LIMIT 10
    """
    results = await server.query_database(sql, [f'%{query}%'])
    return format_results(results)
```

Pattern 2: API Integration with Authentication

python

```python
import aiohttp
from typing import Optional

class APIClient:
    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = "https://api.example.com"
        self.session: Optional[aiohttp.ClientSession] = None

    async def __aenter__(self):
        self.session = aiohttp.ClientSession(
            headers={"Authorization": f"Bearer {self.api_key}"}
        )
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        if self.session:
            await self.session.close()

    async def get(self, endpoint: str, params: dict = None):
        async with self.session.get(
            f"{self.base_url}/{endpoint}",
            params=params,
            timeout=aiohttp.ClientTimeout(total=30)
        ) as response:
            response.raise_for_status()
            return await response.json()

# Usage in tool
async def fetch_user_data(user_id: str):
    async with APIClient(config.api_key) as client:
        user = await client.get(f"users/{user_id}")
        return format_user(user)
```

Pattern 3: File System Access


python

```python
import os
import aiofiles
from pathlib import Path

async def search_files(directory: str, pattern: str):
    """Search files by content"""
    results = []
    base_path = Path(directory).resolve()

    # Security: Prevent directory traversal
    if not str(base_path).startswith(str(Path.home())):
        raise ValueError("Access denied: outside home directory")

    for file_path in base_path.rglob(pattern):
        if file_path.is_file():
            async with aiofiles.open(file_path, 'r') as f:
                content = await f.read(1000)  # First 1000 chars
                results.append({
                    'path': str(file_path),
                    'preview': content
                })

    return results
```

Pattern 4: Vector Database Integration

python

```
from typing import List
import numpy as np

class VectorSearchServer:
    def __init__(self):
        # In production, use Pinecone, Weaviate, Qdrant, etc.
        self.embeddings = {}  # document_id -> vector
        self.documents = {}   # document_id -> content

    async def embed_text(self, text: str) -> np.ndarray:
        """Generate embedding (use OpenAI, Cohere, etc.)"""
        # Simplified - use actual embedding model
        return np.random.rand(768)

    async def semantic_search(self, query: str, top_k: int = 5):
        query_vector = await self.embed_text(query)

        # Calculate cosine similarity
        similarities = {}
        for doc_id, doc_vector in self.embeddings.items():
            similarity = np.dot(query_vector, doc_vector) / (
                np.linalg.norm(query_vector) * np.linalg.norm(doc_vector)
            )
            similarities[doc_id] = similarity

        # Return top-k results
        top_docs = sorted(similarities.items(), key=lambda x: x[1], reverse=True)[:top_k]

        return [
            {
                'id': doc_id,
                'content': self.documents[doc_id],
                'score': score
            }
            for doc_id, score in top_docs
        ]
```

Pattern 5: Workflow Orchestration

python

```python
async def create_and_deploy_service(params: dict):
    """
    Multi-step workflow:
    1. Create git repository
    2. Initialize project structure
    3. Deploy to cloud
    4. Configure DNS
    5. Set up monitoring
    """

    results = {
        'steps': [],
        'status': 'in_progress'
    }

    try:
        # Step 1: Create repo
        repo_url = await create_github_repo(params['repo_name'])
        results['steps'].append({'step': 'create_repo', 'status': 'success', 'url': repo_url})

        # Step 2: Initialize project
        await initialize_project_structure(repo_url, params['template'])
        results['steps'].append({'step': 'initialize', 'status': 'success'})

        # Step 3: Deploy
        deployment = await deploy_to_cloud(repo_url, params['cloud_provider'])
        results['steps'].append({'step': 'deploy', 'status': 'success', 'url': deployment['url']})

        # Step 4: DNS
        await configure_dns(params['domain'], deployment['ip'])
        results['steps'].append({'step': 'dns', 'status': 'success'})

        # Step 5: Monitoring
        await setup_monitoring(deployment['url'])
        results['steps'].append({'step': 'monitoring', 'status': 'success'})

        results['status'] = 'completed'

    except Exception as e:
        results['status'] = 'failed'
        results['error'] = str(e)
```

```
    return results
```

Configuration and Deployment

Claude Desktop Configuration

Edit `~/Library/Application Support/Claude/claude_desktop_config.json` (Mac):


json

```
{
  "mcpServers": {
    "knowledge-base": {
      "command": "python",
      "args": ["/absolute/path/to/knowledge_base_server.py"],
      "env": {
        "PYTHONUNBUFFERED": "1",
        "API_KEY": "your-api-key-here"
      }
    }
  }
}
```

Linux: `~/.config/Claude/claude_desktop_config.json`

Windows: `%APPDATA%\Claude\claude_desktop_config.json`

Docker Deployment


dockerfile

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY knowledge_base_server.py .

CMD ["python", "knowledge_base_server.py"]
```

Kubernetes Deployment


yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mcp-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mcp-server
  template:
    metadata:
      labels:
        app: mcp-server
    spec:
      containers:
      - name: mcp-server
        image: your-registry/mcp-server:latest
        env:
        - name: API_KEY
          valueFrom:
            secretKeyRef:
              name: mcp-secrets
              key: api-key
        resources:
          requests:
            memory: "256Mi"
            cpu: "200m"
          limits:
            memory: "512Mi"
            cpu: "500m"
```

Real-World Use Cases for Architects

1. Internal Knowledge Base

- Connect to Confluence, Notion, or internal wikis
- Enable natural language search across company docs
- Reduce time spent searching for information

2. Database Query Interface

- Allow natural language database queries

- Implement read-only views for security
- Generate reports and analytics on-demand

3. DevOps Automation

- Check deployment status
- View logs and metrics
- Trigger deployments (with approval workflows)
- Manage infrastructure

4. Customer Support Integration

- Search support tickets
- Fetch customer information
- Create and update tickets
- Analyze support trends

5. Code Repository Management

- Search code across repos
- Review PRs and issues
- Generate documentation
- Analyze code quality metrics

Key Takeaways for Architects

1. **Design for Agents, Not Humans**
   - Tools should enable workflows, not just wrap APIs
   - Optimize for limited context
   - Make errors actionable
2. **Start Simple, Iterate**
   - Begin with 2-3 core tools
   - Test with real users (Claude)
   - Add tools based on actual needs
3. **Security First**
   - Validate all inputs
   - Use environment variables for secrets
   - Implement proper authentication
   - Log security events
4. **Performance Matters**
   - Use connection pooling
   - Implement caching
   - Set reasonable timeouts
   - Handle rate limits gracefully
5. **Observability is Critical**
   - Log all tool invocations
   - Track success/failure rates
   - Monitor response times
   - Alert on anomalies

Next Steps

1. ✅ **Run the Example**: Test the knowledge base server
2. 🔨 **Build Your First Server**: Start with a simple use case
3. 📚 **Read the Docs**: Study MCP protocol at https://modelcontextprotocol.io
4. 🧪 **Create Evaluations**: Test your server with realistic scenarios
5. 🚀 **Deploy**: Move to production with monitoring

Additional Resources

- **MCP Protocol Docs**: https://modelcontextprotocol.io
- **Python SDK**: https://github.com/modelcontextprotocol/python-sdk
- **TypeScript SDK**: https://github.com/modelcontextprotocol/typescript-sdk
- **Example Servers**: https://github.com/modelcontextprotocol/servers
- **Claude Documentation**: https://docs.claude.com

Questions or feedback? Open an issue or contribute to the MCP community!