

MCPO MCP Server/Proxy Implementation Plan

Comprehensive upgrade to add MCP server capabilities alongside existing OpenAPI functionality

Executive Summary

Transform MCPO from an **MCP-to-OpenAPI proxy** into a **bidirectional MCP aggregation proxy** that can:

- Continue serving REST API clients on port 8000 (existing)
- Serve MCP protocol clients (Claude Desktop, etc.) on port 8001 (new)
- Aggregate multiple backend MCP servers into a unified interface
- Maintain all existing functionality and security features

Phase 1: Research & Analysis (1-2 days)

1.1 MCP Protocol Research

Research the latest MCP specification:

- Current MCP protocol version and changes
- SSE transport implementation details
- Stdio transport requirements
- Tool schema requirements and validation
- Error handling protocols
- Authentication mechanisms
- Performance considerations

Key Questions to Research:

1. What's the latest MCP protocol version?
2. Have there been changes to tool schema formats?
3. What are the current transport requirements?
4. Are there new security considerations?
5. How do other MCP servers handle aggregation?

1.2 Architecture Analysis

Current MCPO Analysis:

- Review existing MCP client implementations in `main.py`
- Analyze current tool aggregation logic
- Understand error envelope system
- Document timeout and retry mechanisms
- Map configuration management system

Phase 2: Core Infrastructure (2-3 days)

2.1 MCP Server Implementation

Add MCP Server Dependencies:

```
# Add to requirements.txt
mcp>=1.0.0 # Latest version from research
```

Core MCP Server Setup:

```
# src/mcpo/mcp_server.py (new file)
from mcp.server import Server
from mcp.server.sse import sse_server
from mcp.types import Tool, TextContent, InitializationOptions
import asyncio
import json
from typing import List, Dict, Any

class MCPProxyServer:
    def __init__(self, config: Dict[str, Any], mcp_clients: Dict):
        self.config = config
        self.mcp_clients = mcp_clients
        self.server = Server("mcpo-proxy")
        self.setup_handlers()

    def setup_handlers(self):
        @self.server.list_tools()
        async def list_tools():
            """Aggregate tools from all enabled MCP servers"""
            return await self._aggregate_tools()

        @self.server.call_tool()
        async def call_tool(name: str, arguments: dict):
            """Route tool calls to appropriate MCP server"""
            return await self._route_tool_call(name, arguments)

        async def _aggregate_tools(self) -> List[Tool]:
            """Aggregate tools with proper namespacing and filtering"""
            # Implementation details below
            pass

        async def _route_tool_call(self, name: str, arguments: dict):
            """Route namespaced tool calls to backend servers"""
            # Implementation details below
            pass
```

2.2 Tool Aggregation Logic

Enhanced Tool Discovery:

```

async def _aggregate_tools(self) -> List[Tool]:
    tools = []
    proxy_config = self.config.get('claude_proxy', {})

    # Filter servers based on configuration
    enabled_servers = proxy_config.get('server_filter',
list(self.mcp_clients.keys()))
    namespace_tools = proxy_config.get('namespace_tools', True)

    for server_name, client in self.mcp_clients.items():
        # Skip disabled servers
        if server_name not in enabled_servers:
            continue

        # Check server state from existing infrastructure
        server_state = get_server_state(server_name)
        if not server_state.get('enabled', True):
            continue

    try:
        # Use existing timeout and error handling
        server_tools = await asyncio.wait_for(
            client.list_tools(),
            timeout=self.config.get('tool_timeout', 30)
        )

        for tool in server_tools.tools:
            # Namespace tools to prevent conflicts
            if namespace_tools:
                original_name = tool.name
                tool.name = f"{server_name}_{original_name}"
                tool.description = f"[{server_name}] {tool.description}"

            # Preserve all schema information
            tools.append(tool)

    except Exception as e:
        # Use existing error logging
        logger.error(f"Failed to list tools from {server_name}: {e}")
        continue

    # Optionally include MCPO management tools
    if proxy_config.get('expose_management_tools', False):
        tools.extend(self._get_management_tools())

return tools

```

Tool Call Routing:

```

async def _route_tool_call(self, name: str, arguments: dict):
    """Route tool calls with full error handling"""

    # Parse namespaced tool name
    if '_' not in name:
        raise ValueError(f"Invalid tool name format: {name}")

    server_name, tool_name = name.split('_', 1)

    # Validate server exists and is enabled
    if server_name not in self.mcp_clients:
        raise ValueError(f"Unknown server: {server_name}")

    client = self.mcp_clients[server_name]
    server_state = get_server_state(server_name)

    if not server_state.get('enabled', True):
        raise ValueError(f"Server {server_name} is disabled")

    try:
        # Use existing timeout system
        timeout = arguments.pop('--tool-timeout',
                               self.config.get('tool_timeout', 30))

        # Call original tool with preserved arguments
        result = await asyncio.wait_for(
            client.call_tool(tool_name, arguments),
            timeout=timeout
        )

        # Return exactly what the backend server returned
        return result.content

    except Exception as e:
        # Use existing error envelope system
        error_response = create_error_envelope(
            error_type="tool_execution_error",
            message=f"Tool {name} failed: {str(e)}",
            server=server_name,
            tool=tool_name
        )
        raise RuntimeError(json.dumps(error_response))

```

2.3 Transport Layer Implementation

SSE Transport (Recommended):

```

# src/mcpo/transports.py (new file)
import asyncio
from mcp.server.sse import sse_server

```

```

from fastapi import FastAPI, Request
from fastapi.responses import StreamingResponse

class MCPSESTransport:
    def __init__(self, mcp_server: MCPProxyServer, port: int = 8001):
        self.mcp_server = mcp_server
        self.port = port
        self.app = FastAPI(title="MCPO MCP Server")
        self.setup_routes()

    def setup_routes(self):
        @self.app.get("/sse")
        async def sse_endpoint(request: Request):
            """SSE endpoint for MCP clients"""
            return await self._handle_sse_connection(request)

    async def _handle_sse_connection(self, request: Request):
        """Handle SSE connection with proper error handling"""
        async def event_stream():
            try:
                async with sse_server() as (read, write):
                    await self.mcp_server.server.run(
                        read, write,
                        InitializationOptions(
                            server_name="mcpo-proxy",
                            server_version="2.0.0"
                        )
                    )
            except Exception as e:
                logger.error(f"SSE connection error: {e}")
                yield f"data: {json.dumps({'error': str(e)})}\n\n"

        return StreamingResponse(
            event_stream(),
            media_type="text/event-stream",
            headers={
                "Cache-Control": "no-cache",
                "Connection": "keep-alive",
                "Access-Control-Allow-Origin": "*"
            }
        )

```

Phase 3: Configuration Integration (1 day)

3.1 Configuration Schema Extension

Add Claude Proxy Configuration:

```
{
  "claude_proxy": {
```

```

    "enabled": true,
    "port": 8001,
    "transport": "sse",
    "server_name": "mcpo-proxy",
    "server_version": "2.0.0",
    "namespace_tools": true,
    "expose_management_tools": false,
    "server_filter": ["perplexity", "time", "playwright"],
    "timeout_override": 45,
    "max_concurrent_calls": 10
  },
  "mcpServers": {
    // existing configuration
  }
}

```

Configuration Validation:

```

# src/mcpo/config_models.py (extend existing)
from pydantic import BaseModel, Field
from typing import List, Optional, Literal

class ClaudeProxyConfig(BaseModel):
    enabled: bool = False
    port: int = Field(default=8001, ge=1024, le=65535)
    transport: Literal["sse", "stdio"] = "sse"
    server_name: str = "mcpo-proxy"
    server_version: str = "2.0.0"
    namespace_tools: bool = True
    expose_management_tools: bool = False
    server_filter: Optional[List[str]] = None
    timeout_override: Optional[int] = None
    max_concurrent_calls: int = 10

# Add to main config model
class McpoConfig(BaseModel):
    # existing fields...
    claude_proxy: Optional[ClaudeProxyConfig] = None

```

3.2 Dynamic Configuration Support

UI Configuration Panel:

```

// Add to existing UI configuration section
const ClaudeProxyConfig = {
  render: function() {
    return `
      <div class="config-section">

```

```

        <h3>Claude Desktop Proxy</h3>
        <div class="form-group">
            <label>
                <input type="checkbox" id="claude-proxy-enabled">
                Enable MCP Server Proxy
            </label>
        </div>
        <div class="form-group">
            <label>Port:</label>
            <input type="number" id="claude-proxy-port" value="8001"
min="1024" max="65535">
        </div>
        <div class="form-group">
            <label>Server Filter:</label>
            <select multiple id="claude-server-filter">
                ${this.renderServerOptions()}
            </select>
        </div>
        <button onclick="ClaudeProxyConfig.save()">Save
Configuration</button>
        <button onclick="ClaudeProxyConfig.restart()">Restart MCP
Server</button>
    </div>
    `;
},
save: async function() {
    // Implementation for saving configuration
},
restart: async function() {
    // Implementation for restarting MCP server
}
};

```

Phase 4: Service Integration (1-2 days)

4.1 Dual Server Architecture

Main Application Integration:

```

# src/mcpo/main.py (modifications)
import asyncio
from .mcp_server import MCPProxyServer
from .transports import MCPSSETTransport

class MCPOApplication:
    def __init__(self):
        self.fastapi_app = None # existing FastAPI app
        self.mcp_server = None

```

```

        self.mcp_transport = None
        self.background_tasks = set()

    async def start_mcp_server(self):
        """Start MCP server if enabled in configuration"""
        config = get_current_config()
        proxy_config = config.get('claude_proxy', {})

        if not proxy_config.get('enabled', False):
            logger.info("Claude proxy disabled in configuration")
            return

        try:
            # Initialize MCP server with current MCP clients
            self.mcp_server = MCPProxyServer(config, get_mcp_clients())

            # Start appropriate transport
            transport_type = proxy_config.get('transport', 'sse')
            if transport_type == 'sse':
                self.mcp_transport = MCPSSETransport(
                    self.mcp_server,
                    proxy_config.get('port', 8001)
                )
                await self._start_sse_transport()
            else:
                raise ValueError(f"Unsupported transport: {transport_type}")

        except Exception as e:
            logger.error(f"Failed to start MCP server: {e}")
            raise

    async def _start_sse_transport(self):
        """Start SSE transport server"""
        import uvicorn

        config = uvicorn.Config(
            self.mcp_transport.app,
            host="127.0.0.1", # MCP server only on localhost for security
            port=self.mcp_transport.port,
            log_level="info"
        )

        server = uvicorn.Server(config)
        task = asyncio.create_task(server.serve())
        self.background_tasks.add(task)
        task.add_done_callback(self.background_tasks.discard)

        logger.info(f"MCP server started on port {self.mcp_transport.port}")

```

4.2 Lifecycle Management

Startup Integration:

```
async def startup():
    """Enhanced startup with MCP server support"""
    # Existing startup logic...

    # Start MCP server if configured
    app = get_mcpo_application()
    await app.start_mcp_server()

    logger.info("MCPO fully started with OpenAPI and MCP servers")

async def shutdown():
    """Enhanced shutdown with proper cleanup"""
    # Shutdown MCP server
    app = get_mcpo_application()
    if app.mcp_transport:
        await app.mcp_transport.shutdown()

    # Cancel background tasks
    for task in app.background_tasks:
        task.cancel()

    # Existing shutdown logic...
```

Hot Reload Support:

```
async def reload_mcp_server():
    """Reload MCP server when configuration changes"""
    app = get_mcpo_application()

    # Stop existing MCP server
    if app.mcp_transport:
        await app.mcp_transport.shutdown()

    # Restart with new configuration
    await app.start_mcp_server()

    return {"status": "reloaded", "timestamp": datetime.utcnow().isoformat()}

# Add to management API
@router.post("/_meta/mcp-server/reload")
async def reload_mcp_server_endpoint():
    """Reload MCP server configuration"""
    try:
        result = await reload_mcp_server()
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Phase 5: Security & Validation (1-2 days)

5.1 Security Considerations

Access Control:

```
# MCP server security configuration
SECURITY_CONFIG = {
    "bind_localhost_only": True, # Only bind to 127.0.0.1
    "require_authentication": False, # MCP doesn't define auth yet
    "rate_limiting": {
        "enabled": True,
        "calls_per_minute": 100,
        "burst_size": 10
    },
    "tool_filtering": {
        "enabled": True,
        "dangerous_tools": ["mcpo_restart", "mcpo_shutdown"]
    }
}
```

Tool Filtering:

```
def filter_dangerous_tools(tools: List[Tool]) -> List[Tool]:
    """Filter out potentially dangerous management tools"""
    config = get_security_config()
    if not config.get('tool_filtering', {}).get('enabled', False):
        return tools

    dangerous_patterns = config.get('tool_filtering',
    {}).get('dangerous_tools', [])

    filtered_tools = []
    for tool in tools:
        if not any(pattern in tool.name for pattern in dangerous_patterns):
            filtered_tools.append(tool)
        else:
            logger.info(f"Filtered dangerous tool: {tool.name}")

    return filtered_tools
```

5.2 Input Validation

Enhanced Schema Validation:

```

def validate_tool_arguments(tool_name: str, arguments: dict) -> dict:
    """Validate arguments against tool schema with enhanced checks"""
    # Get tool schema from aggregated tools
    tool_schema = get_tool_schema(tool_name)
    if not tool_schema:
        raise ValueError(f"Unknown tool: {tool_name}")

    # Use existing JSON schema validation
    try:
        jsonschema.validate(arguments, tool_schema.inputSchema)
    except jsonschema.ValidationError as e:
        raise ValueError(f"Invalid arguments for {tool_name}: {e.message}")

    # Additional security checks
    sanitized_args = sanitize_arguments(arguments)
    return sanitized_args

def sanitize_arguments(arguments: dict) -> dict:
    """Sanitize arguments to prevent injection attacks"""
    # Remove potentially dangerous keys
    dangerous_keys = ['__proto__', 'constructor', 'prototype']
    sanitized = {k: v for k, v in arguments.items() if k not in dangerous_keys}

    # Sanitize string values
    for key, value in sanitized.items():
        if isinstance(value, str):
            sanitized[key] = html.escape(value)

    return sanitized

```

Phase 6: Testing & Validation (2-3 days)

6.1 Unit Tests

MCP Server Tests:

```

# tests/test_mcp_server.py (new file)
import pytest
import asyncio
from unittest.mock import Mock, AsyncMock
from src.mcpo.mcp_server import MCPProxyServer

class TestMCPProxyServer:
    @pytest.fixture
    async def mcp_server(self):
        config = {
            "claude_proxy": {
                "enabled": True,
                "namespace_tools": True,

```

```

        "server_filter": ["test_server"]
    }
}

mock_clients = {
    "test_server": AsyncMock()
}

server = MCPProxyServer(config, mock_clients)
return server

@pytest.mark.asyncio
async def test_tool_aggregation(self, mcp_server):
    """Test that tools are properly aggregated and namespaced"""
    # Mock tool response
    mock_tool = Mock()
    mock_tool.name = "test_tool"
    mock_tool.description = "Test tool"
    mock_tool.inputSchema = {"type": "object"}

    mcp_server.mcp_clients["test_server"].list_tools.return_value = Mock(
        tools=[mock_tool]
    )

    # Test aggregation
    tools = await mcp_server._aggregate_tools()

    assert len(tools) == 1
    assert tools[0].name == "test_server_test_tool"
    assert "[test_server]" in tools[0].description

@pytest.mark.asyncio
async def test_tool_call_routing(self, mcp_server):
    """Test that tool calls are properly routed"""
    # Mock successful tool call
    expected_result = Mock(content=["Test result"])
    mcp_server.mcp_clients["test_server"].call_tool.return_value =
expected_result

    # Test routing
    result = await mcp_server._route_tool_call(
        "test_server_test_tool",
        {"arg1": "value1"}
    )

    assert result == ["Test result"]
    mcp_server.mcp_clients["test_server"].call_tool.assert_called_with(
        "test_tool", {"arg1": "value1"}
    )

```

Integration Tests:

```

# tests/test_mcp_integration.py (new file)
import pytest
import httpx
from fastapi.testclient import TestClient

class TestMCPIntegration:
    @pytest.fixture
    def client(self):
        from src.mcpo.main import app
        return TestClient(app)

    def test_mcp_server_management_api(self, client):
        """Test MCP server management endpoints"""
        # Test status endpoint
        response = client.get("/_meta/mcp-server/status")
        assert response.status_code == 200

        data = response.json()
        assert "enabled" in data
        assert "port" in data
        assert "active_connections" in data

    def test_mcp_configuration_validation(self, client):
        """Test MCP server configuration validation"""
        # Test invalid configuration
        invalid_config = {
            "claude_proxy": {
                "enabled": True,
                "port": 9999, # Invalid port
                "transport": "invalid"
            }
        }

        response = client.post("/_meta/config", json=invalid_config)
        assert response.status_code == 422

```

6.2 End-to-End Testing

Claude Desktop Integration Test:

```

# tests/test_claude_integration.py (new file)
import pytest
import asyncio
import websockets
import json

@pytest.mark.integration
class TestClaudeIntegration:
    @pytest.mark.asyncio

```

```

async def test_sse_connection(self):
    """Test SSE connection establishment"""
    import httpx

    async with httpx.AsyncClient() as client:
        async with client.stream("GET", "http://localhost:8001/sse") as
response:
            assert response.status_code == 200
            assert response.headers["content-type"] == "text/event-stream"

@pytest.mark.asyncio
async def test_tool_discovery(self):
    """Test tool discovery through MCP protocol"""
    # This would require a more complex test setup
    # with actual MCP client library
    pass

@pytest.mark.asyncio
async def test_tool_execution(self):
    """Test tool execution through MCP protocol"""
    # This would test end-to-end tool execution
    pass

```

Phase 7: Documentation & Deployment (1 day)

7.1 Documentation Updates

README Updates:

```

# MCPO - Multi-Protocol Orchestration Proxy

MCPO now supports both OpenAPI REST and MCP protocol interfaces:

## Dual Interface Support

- **REST API** (Port 8000): Traditional HTTP endpoints for web applications
- **MCP Server** (Port 8001): Native MCP protocol for Claude Desktop and other
MCP clients

## Claude Desktop Integration

Add to your `claude_desktop_config.json`:

```json
{
 "mcpServers": {
 "mcpo-proxy": {
 "command": "curl",
 "args": ["-N", "http://localhost:8001/sse"],
 "env": {}
 }
 }
}
```

```

```
        }
    }
}
```

This gives Claude Desktop access to all your configured MCP servers through a single connection.

```
**Configuration Documentation:**  
```markdown  
Claude Proxy Configuration

Add the following section to your `mcpo.json`:

```json  
{  
  "claude_proxy": {  
    "enabled": true,  
    "port": 8001,  
    "transport": "sse",  
    "namespace_tools": true,  
    "server_filter": ["perplexity", "time", "playwright"],  
    "expose_management_tools": false  
  }  
}
```

Options:

- **enabled**: Enable/disable MCP server functionality
- **port**: Port for MCP server (default: 8001)
- **transport**: Transport protocol ("sse" currently supported)
- **namespace_tools**: Prefix tool names with server names (recommended)
- **server_filter**: Only expose specific backend servers to Claude
- **expose_management_tools**: Include MCPO management tools (use carefully)

```
### 7.2 Deployment Considerations  
  
**Production Checklist:**  
- [ ] MCP server binds only to localhost (127.0.0.1)  
- [ ] Firewall rules prevent external access to MCP port  
- [ ] Tool filtering configured for security  
- [ ] Rate limiting enabled  
- [ ] Logging configured for debugging  
- [ ] Health checks include MCP server status  
- [ ] Backup configuration includes claude_proxy section  
  
## Risk Assessment & Mitigation
```

High Risk Items:

1. **Tool Name Conflicts**: Mitigated by mandatory namespacing
2. **Security Exposure**: Mitigated by localhost-only binding and tool filtering
3. **Performance Impact**: Mitigated by connection reuse and timeout management
4. **Configuration Complexity**: Mitigated by UI configuration panel

Medium Risk Items:

1. **Memory Usage**: Monitor aggregated tool counts
2. **Error Propagation**: Use existing error envelope system
3. **Claude Compatibility**: Extensive testing required

Low Risk Items:

1. **Port Conflicts**: Configurable port selection
2. **Transport Issues**: SSE is well-supported
3. **Schema Changes**: Existing schema inference handles this

Success Metrics

- [] Claude Desktop can connect and discover tools
- [] All backend MCP servers accessible through Claude
- [] Tool calls execute successfully with proper error handling
- [] Configuration changes apply without Claude reconnection
- [] Performance impact < 10% on existing OpenAPI functionality
- [] All existing tests continue to pass
- [] Security audit passes with no critical issues

Rollback Plan

If issues arise:

1. Set `claude_proxy.enabled = false` in configuration
2. Restart MCPO (only FastAPI will start)
3. Remove MCP server code if necessary
4. Restore from git backup

The existing OpenAPI functionality remains completely unchanged and unaffected.

Implementation Timeline

Total Estimated Time: 7-10 days

- Phase 1 (Research): 1-2 days
- Phase 2 (Core Infrastructure): 2-3 days
- Phase 3 (Configuration): 1 day
- Phase 4 (Service Integration): 1-2 days
- Phase 5 (Security & Validation): 1-2 days
- Phase 6 (Testing): 2-3 days
- Phase 7 (Documentation): 1 day

Next Steps

1. **Conduct MCP Protocol Research** (Phase 1.1)

2. **Review Current Architecture** (Phase 1.2)
3. **Begin Core Implementation** (Phase 2.1)

This plan provides a comprehensive roadmap for implementing MCP server capabilities while maintaining the existing robust OpenAPI functionality that MCPO already provides.