sajhamcpserver

Technical Requirements Specification

Python-Based Model Context Protocol (MCP) Server  
  
Version: 1.0  
Date: October 26, 2025  
Document Type: System Requirements

# 1. EXECUTIVE SUMMARY

sajhamcpserver is a production-ready Python-based implementation of the Model Context Protocol (MCP) server that provides a standardized interface for AI tools and services. The server supports both HTTP REST APIs and WebSocket connections for real-time bidirectional communication, with comprehensive role-based access control, web-based tool discovery and testing interface, and enterprise-grade monitoring capabilities.

## 1.1 Key Features

* Full compliance with Model Context Protocol (MCP) specification
* Dual transport: HTTP REST API and WebSocket with server-side push
* Role-Based Access Control (RBAC) for secure tool access
* Plugin-based tool architecture with dynamic loading
* Web UI for tool discovery, testing, and administration
* Real-time monitoring dashboards (30-minute windows, 1-minute intervals)
* Administrative tools for tool enable/disable management
* Simple text-based authentication (user\_id/password)
* Comprehensive audit logging and metrics collection

## 1.2 Technology Stack

|  |  |
| --- | --- |
| Component | Technology |
| Language | Python 3.9+ |
| Web Framework | Flask 3.0+ |
| WebSocket | Flask-SocketIO with eventlet/gevent |
| Protocol | JSON-RPC 2.0 over HTTP/WebSocket |
| Configuration | JSON-based configuration files |
| UI Framework | Bootstrap 5.x with Jinja2 templates |
| Monitoring | Built-in metrics with 1-minute aggregation |

# 2. SYSTEM ARCHITECTURE

The sajhamcpserver implements a layered architecture with clear separation of concerns:

┌─────────────────────────────────────────────────────┐  
│ sajhamcpserver │  
├─────────────────────────────────────────────────────┤  
│ ┌──────────┐ ┌──────────┐ ┌─────────────────┐ │  
│ │ Flask │ │ Socket │ │ Web UI │ │  
│ │ Server │ │ IO │ │ (Bootstrap) │ │  
│ └────┬─────┘ └────┬─────┘ └────────┬────────┘ │  
│ └──────────────┴─────────────────┘ │  
│ ┌──────────────────────────────────────────────┐ │  
│ │ MCP Protocol Handler (JSON-RPC 2.0) │ │  
│ └────────────────────┬─────────────────────────┘ │  
│ ┌────────────────────┴─────────────────────────┐ │  
│ │ Authentication & Authorization Manager │ │  
│ └────────────────────┬─────────────────────────┘ │  
│ ┌────────────────────┴─────────────────────────┐ │  
│ │ Tools Registry (Singleton Pattern) │ │  
│ └────────────────────┬─────────────────────────┘ │  
│ ┌────────────────────┴─────────────────────────┐ │  
│ │ Tool Modules (BaseMCPTool) │ │  
│ └──────────────────────────────────────────────┘ │  
└─────────────────────────────────────────────────────┘

# 3. MCP PROTOCOL COMPLIANCE

## 3.1 Required MCP Methods

The server MUST implement the following standard MCP methods:

|  |  |
| --- | --- |
| Method | Description |
| initialize | Protocol handshake and capability negotiation |
| initialized | Confirmation of successful initialization |
| tools/list | List all tools accessible to authenticated user |
| tools/call | Execute a specific tool with provided parameters |
| ping | Health check endpoint for server availability |

## 3.2 JSON-RPC 2.0 Message Format

All MCP messages MUST conform to JSON-RPC 2.0 specification.

### Request Format:

{  
 "jsonrpc": "2.0",  
 "id": "unique-request-id",  
 "method": "tools/call",  
 "params": {  
 "name": "tool\_name",  
 "arguments": { "param1": "value1" }  
 }  
}

### Response Format:

{  
 "jsonrpc": "2.0",  
 "id": "unique-request-id",  
 "result": {  
 "content": [{"type": "text", "text": "Tool execution result"}]  
 }  
}

## 3.3 Standard Error Codes

|  |  |
| --- | --- |
| Code | Meaning |
| -32700 | Parse error - Invalid JSON |
| -32600 | Invalid Request - Malformed JSON-RPC request |
| -32601 | Method not found |
| -32602 | Invalid params - Missing or invalid parameters |
| -32603 | Internal error - Server-side processing error |
| -32001 | Unauthorized - Authentication required or failed |
| -32002 | Forbidden - User lacks permission for requested operation |

# 4. AUTHENTICATION & AUTHORIZATION

## 4.1 Users Configuration (users.json)

The system SHALL maintain user credentials and permissions in a JSON configuration file located at config/users.json.

{  
 "users": [  
 {  
 "user\_id": "admin",  
 "user\_name": "Administrator",  
 "password": "admin\_password",  
 "roles": ["admin"],  
 "tools": ["\*"],  
 "enabled": true,  
 "email": "admin@example.com",  
 "created\_at": "2025-01-01T00:00:00Z",  
 "last\_login": null,  
 "metadata": {  
 "department": "IT",  
 "description": "System Administrator"  
 }  
 },  
 {  
 "user\_id": "analyst01",  
 "user\_name": "John Analyst",  
 "password": "analyst\_pass",  
 "roles": ["analyst", "reader"],  
 "tools": ["data\_query", "report\_gen"],  
 "enabled": true,  
 "email": "john@example.com",  
 "created\_at": "2025-02-01T00:00:00Z",  
 "last\_login": "2025-10-25T14:20:00Z"  
 }  
 ]  
}

## 4.2 User Attributes

|  |  |  |
| --- | --- | --- |
| Field | Required | Description |
| user\_id | Yes | Unique identifier for authentication |
| user\_name | Yes | Display name for UI |
| password | Yes | Plain text password (simple authentication) |
| roles | Yes | List of role identifiers (e.g., admin, analyst) |
| tools | Yes | List of tool names or ["\*"] for all tools |
| enabled | Yes | Boolean flag for account status |
| email | No | User email address |
| created\_at | No | ISO 8601 timestamp of account creation |
| last\_login | No | ISO 8601 timestamp of last successful login |
| metadata | No | Additional user metadata (department, etc.) |

## 4.3 Authorization Matrix

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Admin | User (w/ Tool) | User (w/o Tool) |
| List tools | Yes (all) | Yes (filtered) | Yes (filtered) |
| View tool details | Yes | Yes | No |
| Execute tool | Yes | Yes | No |
| View monitoring | Yes | Yes (own) | Yes (own) |
| Enable/disable tools | Yes | No | No |
| Modify user config | Yes | No | No |
| Admin panel access | Yes | No | No |

# 5. TOOL MANAGEMENT

## 5.1 Tool Configuration File Structure

Each tool SHALL have a JSON configuration file located at config/tools/{tool\_name}.json:

{  
 "tool\_metadata": {  
 "name": "data\_query",  
 "version": "1.0.0",  
 "description": "Execute SQL queries against databases",  
 "author": "Data Team",  
 "category": "data\_access",  
 "tags": ["sql", "database", "query"],  
 "enabled": true  
 },  
 "mcp\_schema": {  
 "name": "data\_query",  
 "description": "Query database tables with SQL",  
 "inputSchema": {  
 "type": "object",  
 "properties": {  
 "query": {  
 "type": "string",  
 "description": "SQL query to execute"  
 },  
 "database": {  
 "type": "string",  
 "description": "Database name",  
 "enum": ["prod\_db", "test\_db"]  
 }  
 },  
 "required": ["query", "database"]  
 }  
 },  
 "implementation": {  
 "module": "tools.data.data\_query",  
 "class\_name": "DataQueryTool",  
 "endpoint\_url": "/api/tools/data\_query"  
 },  
 "sample\_request": {  
 "query": "SELECT \* FROM users LIMIT 10",  
 "database": "prod\_db"  
 },  
 "configuration": {  
 "timeout\_seconds": 30,  
 "max\_rows": 1000,  
 "allow\_write\_operations": false  
 }  
}

## 5.2 Tools Registry (Singleton Pattern)

The Tools Registry is responsible for:

* Discover and load all tool configurations from config/tools directory
* Dynamically import and instantiate tool modules
* Maintain single instance of each tool (singleton per tool)
* Provide proxy methods for tool invocation with error handling
* Cache tool schemas for fast lookup
* Track tool usage metrics (call count, execution time, errors)

## 5.3 BaseMCPTool Abstract Class

All tools MUST inherit from BaseMCPTool and implement required methods:

from abc import ABC, abstractmethod  
from typing import Dict, Any  
  
class BaseMCPTool(ABC):  
 def \_\_init\_\_(self, config: Dict[str, Any]):  
 self.config = config  
 self.name = config["tool\_metadata"]["name"]  
 self.description = config["mcp\_schema"]["description"]  
 self.input\_schema = config["mcp\_schema"]["inputSchema"]  
  
 @abstractmethod  
 def execute(self, params: Dict[str, Any]) -> Dict[str, Any]:  
 """Execute tool with provided parameters  
   
 Args:  
 params: Validated input parameters  
   
 Returns:  
 Dict with "content" key containing result  
   
 Raises:  
 ToolExecutionError: On execution failure  
 """  
 pass  
  
 def validate\_params(self, params: Dict[str, Any]) -> bool:  
 """Validate parameters against input schema"""  
 # JSON Schema validation logic  
 pass  
  
 def get\_schema(self) -> Dict[str, Any]:  
 """Return MCP tool schema"""  
 return self.config["mcp\_schema"]

# 6. WEB INTERFACE REQUIREMENTS

## 6.1 Tool Discovery Page (GET /ui/tools)

* Display all tools accessible to authenticated user
* Admin users see ALL tools, regular users see only authorized tools
* Tool cards showing: name, description, category, status (enabled/disabled)
* Search/filter by tool name, category, or tags
* Click on tool card to view detailed information
* Link to tool testing page for each tool

## 6.2 Tool Testing/Execution Page (GET /ui/tools/test)

Interactive tool testing interface with the following components:

|  |  |
| --- | --- |
| Component | Description |
| Tool Selector | Dropdown menu with all accessible tools. On selection, loads tool details. |
| Tool Info Pane | Displays: tool name, description, input schema, and sample request JSON. |
| Input Text Area | JSON editor with syntax highlighting. Pre-populated with sample request. |
| Execute Button | Validates input JSON and calls tool. Shows loading spinner during execution. |
| Response Pane | JSON-formatted response from tool execution with syntax highlighting. |
| Status Pane | Success/error messages, execution time, HTTP status. Color-coded (green/red). |

Real-Time Features:

* WebSocket connection for server-side progress updates
* Progress bar for long-running tool executions
* Ability to cancel ongoing tool execution

## 6.3 Tool Monitoring Page (GET /ui/monitoring/tools)

* Real-time dashboard showing tool usage over last 30 minutes
* Data aggregated in 1-minute intervals
* Metrics tracked per tool: call count, avg execution time, success rate, error count
* Auto-refresh every 60 seconds via WebSocket
* Charts using Chart.js or similar library (line charts for trends)
* Filter view by specific tool or show all tools

### Display Metrics:

|  |  |
| --- | --- |
| Metric | Description |
| Calls per Minute | Number of tool invocations in each 1-minute interval |
| Average Execution Time | Mean execution time in milliseconds for successful calls |
| Success Rate | Percentage of successful calls vs total calls |
| Error Count | Number of failed executions per interval |
| Peak Usage Time | Timestamp with highest call volume |

## 6.4 User Activity Monitoring Page (GET /ui/monitoring/users)

* Dashboard showing MCP server usage by user
* Admin can see all user activity
* Regular users see only their own activity
* Track: total requests, tools used, request success/failure rate
* Time-series charts showing activity over time (last 30 minutes, 1-minute intervals)
* Table view with sortable columns
* Export data to CSV

## 6.5 Admin Tool Management Page (GET /ui/admin/tools)

Administrative features (Admin only):

* List all registered tools with their status
* Enable/Disable toggle for each tool (updates tool config JSON)
* Reload tool registry without server restart
* View tool configuration details (read-only)
* View tool usage statistics (total calls, errors, last execution)
* Test tool functionality directly from admin panel
* Confirmation modal before enabling/disabling tools

Tool Status Management Process:

1. Update enabled field in tool config JSON file
2. Reload tool registry to reflect change immediately
3. Broadcast status change via WebSocket to all connected clients
4. Log action with timestamp, admin user, and tool name
5. Show success/failure notification in UI

# 7. TRANSPORT LAYER SPECIFICATIONS

## 7.1 HTTP Transport

MCP RPC Endpoint:

* Endpoint: POST /mcp/v1/rpc
* Content-Type: application/json
* Authentication: Bearer token in Authorization header
* Request Format: JSON-RPC 2.0

## 7.2 WebSocket Transport

WebSocket Endpoint:

* Endpoint: ws://host:port/mcp/v1/ws
* Authentication: Token in query parameter ?token=<token>
* Protocol: JSON-RPC 2.0 messages over WebSocket

### Server-Side Push Capabilities:

* Progress notifications for long-running tool executions
* Tool status change broadcasts (enable/disable events)
* Real-time monitoring data updates (1-minute intervals)
* System alerts and notifications
* Heartbeat/keep-alive messages

# 8. TECHNICAL SPECIFICATIONS

## 8.1 Directory Structure

sajhamcpserver/  
├── config/  
│ ├── users.json  
│ ├── server.json  
│ └── tools/  
│ ├── data\_query.json  
│ ├── report\_gen.json  
│ └── ...  
├── src/  
│ ├── server.py  
│ ├── mcp\_handler.py  
│ ├── auth\_manager.py  
│ ├── tools\_registry.py  
│ ├── base\_mcp\_tool.py  
│ ├── monitoring/  
│ │ ├── metrics\_collector.py  
│ │ └── usage\_tracker.py  
│ └── tools/  
│ ├── data/  
│ │ └── data\_query.py  
│ ├── reports/  
│ │ └── report\_gen.py  
│ └── ...  
├── templates/  
│ ├── base.html  
│ ├── login.html  
│ ├── tools\_list.html  
│ ├── tool\_test.html  
│ ├── monitoring\_tools.html  
│ ├── monitoring\_users.html  
│ └── admin\_tools.html  
├── static/  
│ ├── css/  
│ ├── js/  
│ └── images/  
├── logs/  
├── tests/  
├── requirements.txt  
└── README.md

## 8.2 Python Dependencies

# Core Framework  
flask>=3.0.0  
flask-socketio>=5.3.0  
flask-cors>=4.0.0  
eventlet>=0.33.0  
  
# JSON Schema Validation  
jsonschema>=4.19.0  
  
# Configuration & Utilities  
python-dotenv>=1.0.0  
pyyaml>=6.0  
  
# Monitoring & Metrics  
prometheus-client>=0.18.0  
  
# Logging  
python-json-logger>=2.0.0  
  
# Testing  
pytest>=7.4.0  
pytest-flask>=1.3.0  
pytest-cov>=4.1.0

# 9. PERFORMANCE & SECURITY REQUIREMENTS

## 9.1 Performance Requirements

|  |  |
| --- | --- |
| Metric | Requirement |
| Request Response Time (p95) | < 200ms (excluding tool execution time) |
| Concurrent Users | Support minimum 100 concurrent users |
| WebSocket Connections | Support minimum 200 concurrent connections |
| Requests per Second (RPS) | Handle minimum 1000 RPS |
| Memory Footprint | < 500MB base memory (excluding tools) |
| Server Startup Time | < 5 seconds |

## 9.2 Security Requirements

### Authentication Security:

* Session tokens MUST be cryptographically random (minimum 128 bits)
* Tokens MUST expire after configured duration (default 1 hour)
* Failed login attempts MUST be logged
* Implement rate limiting on login endpoint (max 5 attempts per minute)
* HTTPS MUST be used in production deployments

### Input Validation:

* All user inputs MUST be validated against JSON schema
* Maximum request payload size MUST be enforced (configurable, default 10MB)
* Sanitize all user inputs to prevent injection attacks
* Reject requests with invalid JSON-RPC 2.0 format

### Audit Logging:

* Log all authentication attempts (success and failure)
* Log all tool executions with user, timestamp, and parameters
* Log all administrative actions (enable/disable tools)
* Logs MUST be tamper-evident (write-once format)
* Implement log rotation to prevent disk space exhaustion

# 10. DEPLOYMENT REQUIREMENTS

## 10.1 System Requirements

|  |  |
| --- | --- |
| Component | Minimum Requirement |
| Python Version | Python 3.9 or higher |
| CPU | 2 cores (4 cores recommended) |
| Memory | 2GB RAM (4GB recommended) |
| Disk Space | 1GB for application + space for logs |
| Operating System | Linux (Ubuntu 20.04+), macOS, Windows Server |
| Network | Open port for HTTP/HTTPS (default 8000) |

## 10.2 Installation Steps

1. Clone repository or extract distribution package
2. Create Python virtual environment: python -m venv venv
3. Activate virtual environment
4. Install dependencies: pip install -r requirements.txt
5. Configure users.json with initial admin account
6. Configure server.json with deployment settings
7. Add tool configurations to config/tools/
8. Run server: python src/server.py
9. Verify server health at http://host:port/health

## 10.3 Production Deployment Recommendations

* Deploy behind reverse proxy (Nginx or Apache) for HTTPS termination
* Use process manager (systemd, supervisor) for automatic restart
* Configure log aggregation (ELK stack, Splunk, etc.)
* Set up monitoring alerts for server health
* Implement firewall rules to restrict access
* Regular backup of configuration files (users.json, tool configs)
* Consider containerization (Docker) for easier deployment

# 11. APPENDIX

## 11.1 Glossary

|  |  |
| --- | --- |
| Term | Definition |
| MCP | Model Context Protocol - standardized protocol for AI tool integration |
| JSON-RPC | JSON Remote Procedure Call - protocol for calling methods on remote systems |
| RBAC | Role-Based Access Control - access control paradigm using user roles |
| WebSocket | Protocol for full-duplex communication channels over TCP connection |
| Singleton | Design pattern ensuring a class has only one instance |
| Tool | Self-contained service that performs specific operations |
| Bearer Token | Security token used in Authorization header for API authentication |

## 11.2 References

1. Model Context Protocol Specification: https://modelcontextprotocol.io
2. JSON-RPC 2.0 Specification: https://www.jsonrpc.org/specification
3. Flask Documentation: https://flask.palletsprojects.com/
4. Flask-SocketIO Documentation: https://flask-socketio.readthedocs.io/
5. JSON Schema Specification: https://json-schema.org/
6. Bootstrap 5 Documentation: https://getbootstrap.com/docs/5.0/

## 11.3 Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Changes |
| 1.0 | 2025-10-26 | Ashutosh Sinha | Initial requirements document |

**--- END OF DOCUMENT ---**