MCP Manager - Product Requirements Document (MVP)

Project Overview

Product Name: MCP Manager
Version: 1.0.0 (MVP)
Project Type: Open Source Container Management Platform
Target Deployment: Docker Container (similar to Portainer)
Business Model: Open Source Core + Paid Enterprise Features

Problem Statement

Model Context Protocol (MCP) servers are becoming critical infrastructure for AI applications, but managing them is fragmented and complex. Developers struggle with:

- Manual server configuration and deployment
- Mixed deployment patterns (containers vs. traditional processes)
- No centralized management interface
- Lack of monitoring and health checking
- Difficult team collaboration and access control

Solution Overview

MCP Manager is a web-based management platform that provides Portainer-style simplicity for MCP server orchestration. It supports both containerized and traditional MCP server deployment patterns through a unified interface.

Target Users

Primary (MVP):

- Individual developers working with multiple MCP servers
- Small teams (2-10 developers) sharing MCP infrastructure
- AI application developers who need reliable MCP server management

Secondary (Future):

- Enterprise teams requiring advanced access control
- DevOps teams managing MCP servers across environments
- Organizations needing compliance and audit features

Core Value Propositions

1. **Unified Management**: Single interface for all MCP servers regardless of deployment method
2. **Zero-Config Setup**: One-click deployment of popular MCP servers
3. **Hybrid Support**: Works with both containerized and traditional npm/Python servers
4. **Real-time Monitoring**: Health checks, logs, and performance metrics
5. **Team Collaboration**: Shared configurations and access management

Technical Architecture

Deployment Model

- Ships as single Docker container
- Mounts Docker socket for container orchestration
- Volume mounts for configuration persistence
- Web UI + REST API architecture

Supported MCP Server Types

1. **Containerized Servers**: Docker containers from registries
2. **Node.js Servers**: npm packages executed via child processes
3. **Python Servers**: Python modules executed via child processes

MVP Feature Requirements

1. Core Server Management

1.1 Server Discovery and Registration

Requirement ID: CORE-001
Priority: P0 (Must Have)

Functional Requirements:

- Auto-discover running MCP servers on local machine
- Manual server registration via web UI
- Support for server metadata (name, description, version, type)
- Server categorization (filesystem, API, database, etc.)

Technical Implementation:

```javascript
// Server discovery interface
interface MCPServer {
  id: string;
  name: string;
  type: 'container' | 'nodejs' | 'python';
  status: 'running' | 'stopped' | 'error';
  port?: number;
  containerName?: string;
  command?: string;
  args?: string[];
  environment?: Record<string, string>;
  healthEndpoint?: string;
}

// Discovery service
class ServerDiscoveryService {
  async discoverRunningServers(): Promise<MCPServer[]>;
  async registerServer(config: ServerConfig): Promise<MCPServer>;
  async validateServerConfig(config: ServerConfig): Promise<boolean>;
}
```

Acceptance Criteria:

- [ ] Can detect existing MCP servers running on ports 3000-4000
- [ ] Can register new servers through web form
- [ ] Server list updates in real-time
- [ ] Server metadata is persisted across restarts

1.2 Server Lifecycle Management

Requirement ID: CORE-002
Priority: P0 (Must Have)

Functional Requirements:

- Start/stop/restart individual servers
- Bulk operations (start all, stop all)
- Server configuration editing
- Process health monitoring

Technical Implementation:

```javascript
class ServerLifecycleManager {
  async startServer(serverId: string): Promise<void>;
  async stopServer(serverId: string): Promise<void>;
  async restartServer(serverId: string): Promise<void>;
  async getServerStatus(serverId: string): Promise<ServerStatus>;
  async bulkOperation(serverIds: string[], operation: 'start'|'stop'): Promise<void>;
}

// Container orchestration
class ContainerManager {
  async createContainer(config: ContainerConfig): Promise<string>;
  async startContainer(containerId: string): Promise<void>;
  async stopContainer(containerId: string): Promise<void>;
  async getContainerLogs(containerId: string, lines?: number): Promise<string>;
}

// Process management
class ProcessManager {
  async spawnProcess(command: string, args: string[], env: Record<string, string>): Promise<Chi
  async killProcess(pid: number): Promise<void>;
  async getProcessHealth(pid: number): Promise<boolean>;
}
```

Acceptance Criteria:

- [ ] Can start/stop containerized MCP servers via Docker API
- [ ] Can start/stop Node.js MCP servers via child processes
- [ ] Server status updates in real-time in UI
- [ ] Failed operations show clear error messages
- [ ] Bulk operations work reliably

1.3 Configuration Management

Requirement ID: CORE-003

Priority: P0 (Must Have)

Functional Requirements:

- Generate MCP client configuration files
- Environment variable management

- Secret/token management (basic)
- Configuration templates for popular servers

Technical Implementation:

```javascript
interface ClientConfig {
  mcpServers: Record<string, {
    command: string;
    args: string[];
    env?: Record<string, string>;
  }>;
}

class ConfigurationManager {
  async generateClientConfig(enabledServers: string[]): Promise<ClientConfig>;
  async saveConfiguration(config: ClientConfig): Promise<void>;
  async exportConfiguration(format: 'json' | 'yaml'): Promise<string>;
  async validateConfiguration(config: ClientConfig): Promise<ValidationResult>;
}

// Built-in templates
const SERVER_TEMPLATES = {
  'filesystem': {
    image: 'mcp/filesystem:latest',
    defaultConfig: {
      allowedPaths: ['/tmp']
    }
  },
  'github': {
    image: 'mcp/github:latest',
    requiredEnv: ['GITHUB_TOKEN']
  }
};
```

Acceptance Criteria:

- [ ] Generates valid claude_desktop_config.json files
- [ ] Can export configurations in JSON/YAML formats
- [ ] Environment variables are managed securely
- [ ] Templates work for 5+ popular MCP servers

2. Monitoring and Observability

2.1 Health Monitoring

Requirement ID: MONITOR-001
Priority: P0 (Must Have)

Functional Requirements:

- Real-time server health checks
- Uptime tracking
- Connection count monitoring
- Basic resource usage (CPU, memory for containers)

Technical Implementation:

```javascript
interface HealthMetrics {
  status: 'healthy' | 'unhealthy' | 'unknown';
  uptime: number;
  responseTime: number;
  connections: number;
  lastCheck: Date;
  resources?: {
    cpu: number;
    memory: number;
  };
}

class HealthMonitor {
  async checkServerHealth(serverId: string): Promise<HealthMetrics>;
  async startContinuousMonitoring(serverId: string, interval: number): Promise<void>;
  async getHealthHistory(serverId: string, timeRange: string): Promise<HealthMetrics[]>;
}
```

Acceptance Criteria:

- [ ] Health checks run every 30 seconds
- [ ] Health status visible in dashboard
- [ ] Failed health checks trigger status updates
- [ ] Resource usage shown for containerized servers

2.2 Logging

Requirement ID: MONITOR-002
Priority: P1 (Should Have)

Functional Requirements:

- Real-time log streaming
- Log history and search
- Log level filtering
- Export logs

Technical Implementation:

```javascript
class LogManager {
  async streamLogs(serverId: string): Promise<ReadableStream>;
  async getLogHistory(serverId: string, options: LogQueryOptions): Promise<LogEntry[]>;
  async searchLogs(serverId: string, query: string): Promise<LogEntry[]>;
  async exportLogs(serverId: string, format: 'txt' | 'json'): Promise<string>;
}

interface LogEntry {
  timestamp: Date;
  level: 'info' | 'warn' | 'error' | 'debug';
  message: string;
  source: string;
}
```

Acceptance Criteria:

- [ ] Logs stream in real-time in web UI
- [ ] Can search logs by text and date range
- [ ] Log levels are color-coded
- [ ] Logs persist across server restarts

3. User Interface

3.1 Dashboard

Requirement ID: UI-001
Priority: P0 (Must Have)

Functional Requirements:

- Overview of all servers with status indicators
- Quick action buttons (start/stop/restart)
- Server metrics summary
- System resource overview

UI Components:

```javascript
// Dashboard page components
<Dashboard>
  <ServerStatCards />        // Total, Running, Stopped, Error counts
  <ServerList />             // Table with actions
  <QuickActions />           // Bulk operations
  <SystemResources />        // Overall resource usage
</Dashboard>

// Server list table columns
const COLUMNS = [
  'name',           // Server name with type icon
  'status',         // Status indicator with icon
  'uptime',         // Time since start
  'connections',    // Active connection count
  'actions'         // Start/stop/restart/configure buttons
];
```

Acceptance Criteria:

- [ ] Dashboard loads in <2 seconds
- [ ] Server status updates in real-time
- [ ] All core actions accessible from dashboard
- [ ] Responsive design works on tablet/mobile

3.2 Server Detail Views

Requirement ID: UI-002
Priority: P0 (Must Have)

Functional Requirements:

- Individual server configuration pages

- Log viewer with real-time updates
- Health metrics and charts
- Configuration editor

Technical Implementation:

```javascript
<ServerDetail serverId={id}>
  <ServerInfo />           // Basic info, status, uptime
  <ConfigurationTab />     // Environment variables, settings
  <LogsTab />              // Real-time log stream
  <MetricsTab />           // Health and performance metrics
  <ActionsPanel />         // Start/stop/restart controls
</ServerDetail>
```

Acceptance Criteria:

- [ ] Server details load quickly
- [ ] Configuration changes persist correctly
- [ ] Logs stream without performance issues
- [ ] All server actions work from detail view

4. Server Marketplace/Catalog

4.1 Built-in Server Templates

Requirement ID: CATALOG-001
Priority: P1 (Should Have)

Functional Requirements:

- Pre-configured templates for popular MCP servers
- One-click deployment from templates
- Template customization before deployment
- Community template sharing (future)

Built-in Templates (MVP):

```javascript
const BUILTIN_TEMPLATES = {
  'filesystem': {
    name: 'Filesystem Access',
    description: 'Access local filesystem with configurable paths',
    type: 'container',
    image: 'mcp/filesystem:latest',
    configSchema: {
      allowedPaths: { type: 'array', required: true }
    }
  },
  'github': {
    name: 'GitHub Integration',
    description: 'Access GitHub repositories and APIs',
    type: 'container',
    image: 'mcp/github:latest',
    configSchema: {
      token: { type: 'string', required: true, secret: true }
    }
  },
  'postgres': {
    name: 'PostgreSQL Database',
    description: 'Query and manage PostgreSQL databases',
    type: 'container',
    image: 'mcp/postgres:latest',
    configSchema: {
      connectionString: { type: 'string', required: true, secret: true }
    }
  }
};
```

Acceptance Criteria:

- [ ] Can deploy from template in <30 seconds
- [ ] Template configuration validates before deployment
- [ ] Templates work out-of-box with minimal config
- [ ] At least 5 popular server types available

Technical Implementation Details

Backend Architecture

Core Technologies

- **Runtime**: Node.js 18+
- **Framework**: Express.js or Fastify
- **Database**: SQLite (MVP) / PostgreSQL (future)
- **Container Management**: Docker Engine API
- **Process Management**: Node.js child_process
- **WebSocket**: Socket.io for real-time updates

API Design

```javascript
// RESTful API endpoints
GET    /api/servers                    // List all servers
POST   /api/servers                    // Create new server
GET    /api/servers/:id                // Get server details
PUT    /api/servers/:id                // Update server config
DELETE /api/servers/:id                // Remove server
POST   /api/servers/:id/start          // Start server
POST   /api/servers/:id/stop           // Stop server
POST   /api/servers/:id/restart        // Restart server
GET    /api/servers/:id/logs           // Get logs
GET    /api/servers/:id/health         // Get health status

GET    /api/templates                  // List server templates
POST   /api/templates/:id/deploy       // Deploy from template

GET    /api/system/info                // System information
GET    /api/config/export              // Export configuration
POST   /api/config/import              // Import configuration

// WebSocket events
'server:status'                        // Server status changes
'server:logs'                          // Real-time log stream
'system:resources'                     // System resource updates
```

Database Schema

```sql
-- SQLite schema for MVP
CREATE TABLE servers (
  id TEXT PRIMARY KEY,
  name TEXT NOT NULL,
  type TEXT NOT NULL, -- 'container', 'nodejs', 'python'
  status TEXT NOT NULL DEFAULT 'stopped',
  config TEXT NOT NULL, -- JSON configuration
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE server_logs (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  server_id TEXT NOT NULL,
  level TEXT NOT NULL,
  message TEXT NOT NULL,
  timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (server_id) REFERENCES servers(id)
);

CREATE TABLE health_metrics (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  server_id TEXT NOT NULL,
  status TEXT NOT NULL,
  response_time INTEGER,
  connections INTEGER,
  cpu_usage REAL,
  memory_usage REAL,
  timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (server_id) REFERENCES servers(id)
);
```

Frontend Architecture

Core Technologies

- **Framework**: React 18 with TypeScript
- **Styling**: Tailwind CSS
- **State Management**: React Context + useReducer (MVP) / Redux Toolkit (future)
- **Real-time**: Socket.io-client

- **HTTP Client**: Axios
- **Charts**: Recharts
- **Icons**: Lucide React

Component Structure

```
src/
├── components/
│   ├── Dashboard/
│   │   ├── ServerStatCards.tsx
│   │   ├── ServerList.tsx
│   │   └── QuickActions.tsx
│   ├── ServerDetail/
│   │   ├── ServerInfo.tsx
│   │   ├── ConfigurationTab.tsx
│   │   ├── LogsTab.tsx
│   │   └── MetricsTab.tsx
│   ├── Marketplace/
│   │   ├── TemplateGrid.tsx
│   │   └── DeployModal.tsx
│   └── Common/
│       ├── StatusIndicator.tsx
│       ├── ActionButton.tsx
│       └── LoadingSpinner.tsx
├── hooks/
│   ├── useServers.ts
│   ├── useRealtime.ts
│   └── useServerLogs.ts
├── services/
│   ├── api.ts
│   ├── websocket.ts
│   └── docker.ts
└── types/
    ├── server.ts
    ├── config.ts
    └── api.ts
```

DevOps and Deployment

Docker Configuration

dockerfile

```dockerfile
# Multi-stage build
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production

COPY . .
RUN npm run build

FROM node:18-alpine AS runtime
WORKDIR /app

# Install Docker CLI for container management
RUN apk add --no-cache docker-cli

# Copy built application
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./

# Create non-root user
RUN addgroup -g 1001 -S mcpmanager && \
    adduser -S mcpmanager -u 1001

# Create directories
RUN mkdir -p /app/data /app/configs /app/logs && \
    chown -R mcpmanager:mcpmanager /app

USER mcpmanager

EXPOSE 3000
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1

CMD ["node", "dist/server.js"]
```

Docker Compose for Development

```yaml
version: '3.8'
services:
  mcp-manager:
    build: .
    ports:
      - "3000:3000"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./data:/app/data
      - ./configs:/app/configs
      - ./logs:/app/logs
    environment:
      - NODE_ENV=development
      - LOG_LEVEL=debug
      - DATABASE_URL=sqlite:///app/data/mcp-manager.db
    restart: unless-stopped
```

Business Model

Open Source Core (MIT License)

Free Features:

- All core server management functionality
- Basic monitoring and logging
- Web UI for up to 10 servers
- Community support via GitHub
- Basic templates and marketplace

Enterprise Features (Paid)

Pricing: $29/month per team (5+ developers) **Features:**

- Unlimited servers
- Advanced access control and user management
- Multi-environment support (dev/staging/prod)
- Advanced monitoring and alerting
- Compliance and audit logging
- Priority support

- Custom integrations
- Team collaboration features

Support Tiers

1. **Community** (Free): GitHub issues, community forum
2. **Professional** ($99/month): Email support, documentation
3. **Enterprise** ($299/month): Dedicated support, SLA, custom features

Success Metrics (MVP)

Product Metrics

- **Active Installations**: 1,000+ Docker pulls within 3 months
- **User Engagement**: 70%+ weekly active users
- **Server Management**: Average 5+ MCP servers per installation
- **Performance**: <2s dashboard load time, 99% uptime

Technical Metrics

- **Reliability**: 99.5% successful server operations
- **Performance**: <500ms API response times
- **Compatibility**: Support 20+ popular MCP server types
- **Resource Usage**: <512MB RAM, <1 CPU core for manager

Community Metrics

- **GitHub Stars**: 500+ within 6 months
- **Contributors**: 10+ community contributors
- **Issues**: <48hr response time on issues
- **Documentation**: Complete API docs and user guides

Development Timeline (MVP)

Phase 1: Core Foundation (Weeks 1-4)

- [ ] Project setup and architecture
- [ ] Basic Docker container with web UI
- [ ] Server discovery and registration
- [ ] Basic start/stop functionality for containers

Phase 2: Server Management (Weeks 5-8)

- [ ] Support for Node.js and Python servers
- [ ] Configuration management
- [ ] Health monitoring
- [ ] Basic logging

Phase 3: User Interface (Weeks 9-12)

- [ ] Complete dashboard implementation
- [ ] Server detail views
- [ ] Real-time updates
- [ ] Basic templates/marketplace

Phase 4: Polish and Release (Weeks 13-16)

- [ ] Testing and bug fixes
- [ ] Documentation
- [ ] Docker Hub publishing
- [ ] Community launch

Risk Assessment

Technical Risks

- **Docker Socket Security**: Mitigated by running in containers with limited permissions
- **Process Management**: Child process zombies - implement proper cleanup
- **Resource Constraints**: Monitor and limit resource usage per server
- **MCP Protocol Changes**: Stay updated with spec changes, version compatibility

Market Risks

- **Docker Competition**: Differentiate through specialized MCP features
- **Adoption Rate**: Focus on developer experience and ease of use
- **Enterprise Sales**: Start with open source, build enterprise features iteratively

Mitigation Strategies

- Maintain backward compatibility with MCP protocol versions
- Build strong community through open source
- Focus on developer experience and documentation

- Implement robust testing and monitoring

Future Roadmap (Post-MVP)

Version 2.0 Features

- Kubernetes operator for enterprise deployments
- Advanced monitoring with custom dashboards
- Plugin system for extensibility
- Multi-region deployment support
- Advanced security features

Integration Opportunities

- CI/CD pipeline integration
- Cloud provider marketplaces (AWS, GCP, Azure)
- IDE plugins (VS Code, Cursor)
- Monitoring tool integrations (Grafana, Datadog)

Conclusion

MCP Manager addresses a clear market need for centralized management of MCP servers. By starting with an open source MVP focused on core functionality and developer experience, we can build a strong community foundation while validating enterprise feature demand.

The hybrid approach supporting both containerized and traditional deployment patterns positions us uniquely in the market, providing a migration path as the ecosystem evolves toward containerization.

Success depends on excellent developer experience, comprehensive documentation, and building a strong open source community while developing sustainable enterprise features.