MCP Manager - Phase 2 PRD: Server Management

Timeline: Weeks 5-8

Goal: Add support for traditional Node.js/Python MCP servers, health monitoring, and basic logging

Success Criteria: Can manage both containerized and traditional MCP servers with health monitoring

Phase 2 Architecture Overview

```
┌─────────────────────────────────────────┐
│           MCP Manager Container          │
├─────────────────┬───────────────────────┤
│   Frontend      │   Backend API         │
│   (React)       │   (Node.js)           │
│   - Monitoring  │   - Process Manager   │
│   - Logs View   │   - Health Monitor    │
│   - Server Mgmt │   - Log Aggregator    │
└─────────────────┴───────────────────────┘
          │                 │
    Docker Socket     Host Process
          │             Management
          │                 │
┌─────────▼─────┐   ┌───────▼──────┐
│Docker         │   │Host          │
│Containers     │   │Processes     │
│               │   │              │
│┌────────────┐ │   │┌──────────┐  │
││MCP-1       │ │   ││npm run   │  │
││Container   │ │   ││server.js │  │
│└────────────┘ │   │└──────────┘  │
│┌────────────┐ │   │┌──────────┐  │
││MCP-2       │ │   ││python    │  │
││Container   │ │   ││server.py │  │
│└────────────┘ │   │└──────────┘  │
└───────────────┘   └──────────────┘
```

Enhanced Technical Stack

Backend Additions

- **Process Management**: child_process module
- **Health Monitoring**: Custom health check service

- **Log Management**: Log aggregation and streaming
- **WebSocket**: Enhanced real-time updates

Database Schema Updates

```sql
-- Add new columns to servers table
ALTER TABLE servers ADD COLUMN command TEXT;
ALTER TABLE servers ADD COLUMN args TEXT; -- JSON array of arguments
ALTER TABLE servers ADD COLUMN working_directory TEXT;
ALTER TABLE servers ADD COLUMN process_id INTEGER;
ALTER TABLE servers ADD COLUMN health_endpoint TEXT;
ALTER TABLE servers ADD COLUMN last_health_check DATETIME;
ALTER TABLE servers ADD COLUMN health_status TEXT DEFAULT 'unknown'; -- 'healthy', 'unhealthy',

-- Create health_metrics table
CREATE TABLE health_metrics (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  server_id TEXT NOT NULL,
  status TEXT NOT NULL, -- 'healthy', 'unhealthy', 'unknown'
  response_time INTEGER, -- in milliseconds
  error_message TEXT,
  timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (server_id) REFERENCES servers(id) ON DELETE CASCADE
);

CREATE INDEX idx_health_metrics_server_timestamp ON health_metrics(server_id, timestamp);
CREATE INDEX idx_health_metrics_timestamp ON health_metrics(timestamp);

-- Create server_logs table
CREATE TABLE server_logs (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  server_id TEXT NOT NULL,
  level TEXT NOT NULL, -- 'info', 'warn', 'error', 'debug'
  message TEXT NOT NULL,
  source TEXT, -- 'stdout', 'stderr', 'system'
  timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (server_id) REFERENCES servers(id) ON DELETE CASCADE
);

CREATE INDEX idx_server_logs_server_timestamp ON server_logs(server_id, timestamp);
CREATE INDEX idx_server_logs_level ON server_logs(level);
CREATE INDEX idx_server_logs_timestamp ON server_logs(timestamp);
```

Core Requirements

REQ-2.1: Process Management System

Priority: P0
Estimated Effort: 20 hours

Functional Requirements:

- Spawn Node.js and Python MCP server processes
- Monitor process health and lifecycle
- Handle process termination and cleanup
- Environment variable management
- Working directory configuration

Technical Implementation:

```
javascript
```

```
const { spawn } = require('child_process');
const path = require('path');
const fs = require('fs').promises;

class ProcessManager {
  constructor() {
    this.processes = new Map(); // serverId -> process info
    this.setupCleanupHandlers();
  }

  async spawnServer(serverConfig) {
    const { id, type, command, args, workingDirectory, environment } = serverConfig;

    try {
      // Validate executable exists
      await this.validateExecutable(command, type);

      // Prepare spawn options
      const spawnOptions = {
        cwd: workingDirectory || process.cwd(),
        env: { ...process.env, ...environment },
        stdio: ['pipe', 'pipe', 'pipe'], // stdin, stdout, stderr
        detached: false
      };

      // Spawn process
      const childProcess = spawn(command, args || [], spawnOptions);

      // Store process info
      this.processes.set(id, {
        process: childProcess,
        pid: childProcess.pid,
        startTime: new Date(),
        config: serverConfig
      });

      // Setup process event handlers
      this.setupProcessHandlers(id, childProcess);

      return {
        pid: childProcess.pid,
        started: true
      };
    } catch (error) {
      throw new Error(`Failed to spawn process: ${error.message}`);
    }
  }

  setupProcessHandlers(serverId, process) {
    // Handle process exit
    process.on('exit', (code, signal) => {
      console.log(`Process ${serverId} exited with code ${code}, signal ${signal}`);
      this.handleProcessExit(serverId, code, signal);
    });

    // Handle process errors
    process.on('error', (error) => {
      console.error(`Process ${serverId} error:`, error);
      this.handleProcessError(serverId, error);
    });

    // Capture stdout
    process.stdout.on('data', (data) => {
      this.handleProcessOutput(serverId, 'stdout', data.toString());
    });

    // Capture stderr
    process.stderr.on('data', (data) => {
      this.handleProcessOutput(serverId, 'stderr', data.toString());
    });
  }

  async stopProcess(serverId) {
    const processInfo = this.processes.get(serverId);
    if (!processInfo) {
      throw new Error(`Process ${serverId} not found`);
    }

    try {
      const { process } = processInfo;

      // Graceful shutdown first
      process.kill('SIGTERM');

      // Wait for graceful shutdown
      await new Promise((resolve) => {
        const timeout = setTimeout(() => {
          // Force kill if not stopped gracefully
          process.kill('SIGKILL');
          resolve();
        }, 10000); // 10 second timeout

        process.on('exit', () => {
          clearTimeout(timeout);
          resolve();
        });
      });

      this.processes.delete(serverId);
      return true;
    } catch (error) {
      throw new Error(`Failed to stop process: ${error.message}`);
    }
  }

  async restartProcess(serverId) {
    const processInfo = this.processes.get(serverId);
    if (!processInfo) {
      throw new Error(`Process ${serverId} not found`);
    }

    const config = processInfo.config;
    await this.stopProcess(serverId);
    return await this.spawnServer(config);
  }

  getProcessInfo(serverId) {
    const processInfo = this.processes.get(serverId);
    if (!processInfo) {
      return null;
    }

    const { process, pid, startTime } = processInfo;
    return {
      pid,
      startTime,
      uptime: Date.now() - startTime.getTime(),
      memory: process.memoryUsage ? process.memoryUsage() : null,
      status: process.killed ? 'killed' : 'running'
    };
  }
```

```
async validateExecutable(command, type) {
  try {
    switch (type) {
      case 'nodejs':
        // Check if npm/npx/node is available
        if (['npm', 'npx', 'node'].includes(command)) {
          const { spawn } = require('child_process');
          return new Promise((resolve, reject) => {
            const test = spawn(command, ['--version'], { stdio: 'ignore' });
            test.on('exit', (code) => {
              code === 0 ? resolve() : reject(new Error(`${command} not available`));
            });
            test.on('error', reject);
          });
        }
        break;
      case 'python':
        // Check if python/python3 is available
        if (['python', 'python3', 'pip', 'pip3'].includes(command)) {
          return new Promise((resolve, reject) => {
            const test = spawn(command, ['--version'], { stdio: 'ignore' });
            test.on('exit', (code) => {
              code === 0 ? resolve() : reject(new Error(`${command} not available`));
            });
            test.on('error', reject);
          });
        }
        break;
    }
  } catch (error) {
    throw new Error(`Executable validation failed: ${error.message}`);
  }
}

handleProcessExit(serverId, code, signal) {
  // Update database status
  this.updateServerStatus(serverId, code === 0 ? 'stopped' : 'error');

  // Log the exit
  this.logServerEvent(serverId, 'info', `Process exited with code ${code}, signal ${signal}`)

  // Clean up process info
  this.processes.delete(serverId);
  }

  handleProcessError(serverId, error) {
    // Update database status
    this.updateServerStatus(serverId, 'error');

    // Log the error
    this.logServerEvent(serverId, 'error', `Process error: ${error.message}`);
  }

  handleProcessOutput(serverId, source, data) {
    // Parse log level from output if possible
    const level = this.parseLogLevel(data);

    // Store in database
    this.logServerEvent(serverId, level, data.trim(), source);

    // Emit to WebSocket clients
    this.emitLogUpdate(serverId, { level, message: data.trim(), source, timestamp: new Date() }
  }

  parseLogLevel(message) {
    const lowerMessage = message.toLowerCase();
    if (lowerMessage.includes('error') || lowerMessage.includes('err')) return 'error';
    if (lowerMessage.includes('warn') || lowerMessage.includes('warning')) return 'warn';
    if (lowerMessage.includes('debug')) return 'debug';
    return 'info';
  }

  setupCleanupHandlers() {
    // Cleanup on process exit
    process.on('SIGINT', () => this.cleanup());
    process.on('SIGTERM', () => this.cleanup());
    process.on('exit', () => this.cleanup());
  }

  async cleanup() {
    console.log('Cleaning up spawned processes...');
    const promises = Array.from(this.processes.keys()).map(id =>
      this.stopProcess(id).catch(err => console.error(`Failed to stop ${id}:`, err))
    );
    await Promise.allSettled(promises);
  }
}
```

```
module.exports = ProcessManager;
```

Enhanced Server Manager:

javascript

```javascript
const ProcessManager = require('./ProcessManager');
const DockerService = require('./DockerService');
const Database = require('./Database');

class EnhancedServerManager {
  constructor() {
    this.docker = new DockerService();
    this.processManager = new ProcessManager();
    this.db = new Database();
  }

  async createServer(serverData) {
    const serverId = generateId();

    try {
      if (serverData.type === 'container') {
        // Create container (existing logic from Phase 1)
        const containerId = await this.docker.createContainer({
          ...serverData,
          serverId
        });

        await this.db.createServer({
          id: serverId,
          name: serverData.name,
          type: 'container',
          image: serverData.image,
          status: 'stopped',
          port: serverData.port,
          container_id: containerId,
          config: JSON.stringify(serverData.config || {})
        });
      } else {
        // Create process-based server
        await this.db.createServer({
          id: serverId,
          name: serverData.name,
          type: serverData.type, // 'nodejs' or 'python'
          command: serverData.command,
          args: JSON.stringify(serverData.args || []),
          working_directory: serverData.workingDirectory,
          status: 'stopped',
          port: serverData.port,
          health_endpoint: serverData.healthEndpoint,
          config: JSON.stringify(serverData.config || {})
        });
      }

      return { id: serverId };
    } catch (error) {
      throw new Error(`Failed to create server: ${error.message}`);
    }
  }

  async startServer(serverId) {
    try {
      const server = await this.db.getServer(serverId);
      if (!server) throw new Error('Server not found');

      if (server.type === 'container') {
        // Start container (existing logic)
        await this.docker.startContainer(server.container_id);
      } else {
        // Start process
        const config = {
          id: serverId,
          type: server.type,
          command: server.command,
          args: JSON.parse(server.args || '[]'),
          workingDirectory: server.working_directory,
          environment: JSON.parse(server.config || '{}').environment || {}
        };

        const result = await this.processManager.spawnServer(config);

        // Update database with process ID
        await this.db.updateServer(serverId, {
          process_id: result.pid,
          status: 'running'
        });
      }

      await this.db.updateServerStatus(serverId, 'running');
      return true;
    } catch (error) {
      await this.db.updateServerStatus(serverId, 'error');
      throw error;
    }
  }

  async stopServer(serverId) {
    try {
      const server = await this.db.getServer(serverId);
      if (!server) throw new Error('Server not found');

      if (server.type === 'container') {
        await this.docker.stopContainer(server.container_id);
      } else {
        await this.processManager.stopProcess(serverId);
        await this.db.updateServer(serverId, { process_id: null });
      }

      await this.db.updateServerStatus(serverId, 'stopped');
      return true;
    } catch (error) {
      await this.db.updateServerStatus(serverId, 'error');
      throw error;
    }
  }

  async getServerStatus(serverId) {
    const server = await this.db.getServer(serverId);
    if (!server) throw new Error('Server not found');

    if (server.type === 'container' && server.container_id) {
      try {
        const containerInfo = await this.docker.getContainerInfo(server.container_id);
        return {
          status: containerInfo.status === 'running' ? 'running' : 'stopped',
          details: containerInfo
        };
      } catch (error) {
        return { status: 'error', error: error.message };
      }
    } else if (server.process_id) {
      const processInfo = this.processManager.getProcessInfo(serverId);
      return {
        status: processInfo ? 'running' : 'stopped',
        details: processInfo
      };
    }
```

```
      return { status: server.status };
    }
  }

  module.exports = EnhancedServerManager;
```

Acceptance Criteria:

- [ ] Can spawn Node.js MCP servers using npm/npx commands
- [ ] Can spawn Python MCP servers using python commands
- [ ] Processes are properly monitored and cleaned up
- [ ] Process output is captured and logged
- [ ] Process termination is handled gracefully
- [ ] Environment variables are passed correctly
- [ ] Working directory is set appropriately

REQ-2.2: Health Monitoring System

Priority: P0
Estimated Effort: 16 hours

Functional Requirements:

- Regular health checks for all servers
- HTTP endpoint health probing
- Process health validation
- Health history tracking
- Real-time health status updates

Technical Implementation:

```
javascript
```

```javascript
const axios = require('axios');

class HealthMonitorService {
  constructor(database, websocketService) {
    this.db = database;
    this.ws = websocketService;
    this.healthChecks = new Map(); // serverId -> interval
    this.defaultInterval = 30000; // 30 seconds
  }

  async startMonitoring(serverId) {
    try {
      const server = await this.db.getServer(serverId);
      if (!server) throw new Error('Server not found');

      // Clear existing monitoring
      this.stopMonitoring(serverId);

      // Start periodic health checks
      const interval = setInterval(async () => {
        await this.performHealthCheck(serverId);
      }, this.defaultInterval);

      this.healthChecks.set(serverId, interval);

      // Perform initial health check
      await this.performHealthCheck(serverId);

    } catch (error) {
      console.error(`Failed to start monitoring for ${serverId}:`, error);
    }
  }

  stopMonitoring(serverId) {
    const interval = this.healthChecks.get(serverId);
    if (interval) {
      clearInterval(interval);
      this.healthChecks.delete(serverId);
    }
  }

  async performHealthCheck(serverId) {
    const startTime = Date.now();
    let healthResult = {
      serverId,
      status: 'unknown',
      responseTime: null,
      error: null,
      timestamp: new Date()
    };

    try {
      const server = await this.db.getServer(serverId);
      if (!server) {
        healthResult.status = 'unhealthy';
        healthResult.error = 'Server not found in database';
        return await this.recordHealthCheck(healthResult);
      }

      if (server.status !== 'running') {
        healthResult.status = 'unhealthy';
        healthResult.error = `Server status is ${server.status}`;
        return await this.recordHealthCheck(healthResult);
      }

      // Perform health check based on server type
      if (server.health_endpoint) {
        // HTTP health check
        healthResult = await this.performHttpHealthCheck(server, startTime);
      } else if (server.type === 'container') {
        // Container health check
        healthResult = await this.performContainerHealthCheck(server, startTime);
      } else {
        // Process health check
        healthResult = await this.performProcessHealthCheck(server, startTime);
      }

    } catch (error) {
      healthResult.status = 'unhealthy';
      healthResult.error = error.message;
      healthResult.responseTime = Date.now() - startTime;
    }

    return await this.recordHealthCheck(healthResult);
  }

  async performHttpHealthCheck(server, startTime) {
    try {
      const url = server.health_endpoint.startsWith('http')
        ? server.health_endpoint
        : `http://localhost:${server.port}${server.health_endpoint}`;

      const response = await axios.get(url, {
        timeout: 5000, // 5 second timeout
        validateStatus: (status) => status < 500 // Accept 4xx as healthy
      });

      return {
        serverId: server.id,
        status: 'healthy',
        responseTime: Date.now() - startTime,
        error: null,
        timestamp: new Date(),
        details: {
          httpStatus: response.status,
          contentLength: response.headers['content-length']
        }
      };
    } catch (error) {
      return {
        serverId: server.id,
        status: 'unhealthy',
        responseTime: Date.now() - startTime,
        error: `HTTP health check failed: ${error.message}`,
        timestamp: new Date()
      };
    }
  }

  async performContainerHealthCheck(server, startTime) {
    try {
      // Use Docker API to check container health
      const docker = require('./DockerService');
      const dockerService = new docker();
      const containerInfo = await dockerService.getContainerInfo(server.container_id);

      const isHealthy = containerInfo.status === 'running';

      return {
        serverId: server.id,
        status: isHealthy ? 'healthy' : 'unhealthy',
        responseTime: Date.now() - startTime,
        error: isHealthy ? null : `Container status: ${containerInfo.status}`,
        timestamp: new Date(),
        details: {
          containerStatus: containerInfo.status,
          uptime: containerInfo.uptime
        }
      };
    } catch (error) {
      return {
        serverId: server.id,
        status: 'unhealthy',
        responseTime: Date.now() - startTime,
        error: `Container health check failed: ${error.message}`,
        timestamp: new Date()
      };
    }
  }

  async performProcessHealthCheck(server, startTime) {
    try {
      const processManager = require('./ProcessManager');
      const pm = new processManager();
      const processInfo = pm.getProcessInfo(server.id);

      const isHealthy = processInfo && processInfo.status === 'running';

      return {
        serverId: server.id,
        status: isHealthy ? 'healthy' : 'unhealthy',
        responseTime: Date.now() - startTime,
        error: isHealthy ? null : 'Process not running',
        timestamp: new Date(),
        details: processInfo
      };
    } catch (error) {
      return {
        serverId: server.id,
        status: 'unhealthy',
        responseTime: Date.now() - startTime,
        error: `Process health check failed: ${error.message}`,
        timestamp: new Date()
      };
    }
```

```
}

async recordHealthCheck(healthResult) {
  try {
    // Store in database
    await this.db.insertHealthMetric({
      server_id: healthResult.serverId,
      status: healthResult.status,
      response_time: healthResult.responseTime,
      error_message: healthResult.error,
      timestamp: healthResult.timestamp
    });

    // Update server health status
    await this.db.updateServer(healthResult.serverId, {
      health_status: healthResult.status,
      last_health_check: healthResult.timestamp
    });

    // Emit real-time update
    this.ws.broadcast('health:update', {
      serverId: healthResult.serverId,
      status: healthResult.status,
      responseTime: healthResult.responseTime,
      error: healthResult.error,
      timestamp: healthResult.timestamp
    });

    return healthResult;
  } catch (error) {
    console.error('Failed to record health check:', error);
    throw error;
  }
}

async getHealthHistory(serverId, timeRange = '1h') {
  try {
    const ranges = {
      '1h': 60 * 60 * 1000,
      '24h': 24 * 60 * 60 * 1000,
      '7d': 7 * 24 * 60 * 60 * 1000
    };

    const since = new Date(Date.now() - (ranges[timeRange] || ranges['1h']));
```

```
      return await this.db.getHealthMetrics(serverId, since);
    } catch (error) {
      throw new Error(`Failed to get health history: ${error.message}`);
    }
  }

  async getHealthSummary(serverId) {
    try {
      const recent = await this.getHealthHistory(serverId, '24h');

      const total = recent.length;
      const healthy = recent.filter(m => m.status === 'healthy').length;
      const unhealthy = recent.filter(m => m.status === 'unhealthy').length;

      const avgResponseTime = recent
        .filter(m => m.response_time)
        .reduce((sum, m) => sum + m.response_time, 0) / total || 0;

      return {
        uptime: total > 0 ? (healthy / total) * 100 : 0,
        totalChecks: total,
        healthyChecks: healthy,
        unhealthyChecks: unhealthy,
        averageResponseTime: Math.round(avgResponseTime),
        lastCheck: recent[0] || null
      };
    } catch (error) {
      throw new Error(`Failed to get health summary: ${error.message}`);
    }
  }

  // Start monitoring all running servers on service startup
  async initializeMonitoring() {
    try {
      const runningServers = await this.db.getServersByStatus('running');

      for (const server of runningServers) {
        await this.startMonitoring(server.id);
      }

      console.log(`Initialized health monitoring for ${runningServers.length} servers`);
    } catch (error) {
      console.error('Failed to initialize health monitoring:', error);
    }
  }

  // Cleanup on shutdown
  cleanup() {
    for (const [serverId, interval] of this.healthChecks) {
      clearInterval(interval);
    }
    this.healthChecks.clear();
  }
}

module.exports = HealthMonitorService;
```

Health Monitoring UI Components:

tsx

```
// Health status indicator with more detail
const HealthIndicator: React.FC<{ server: MCPServer }> = ({ server }) => {
  const getHealthColor = (status: string) => {
    switch (status) {
      case 'healthy': return 'text-green-500';
      case 'unhealthy': return 'text-red-500';
      default: return 'text-gray-500';
    }
  };

  const getHealthIcon = (status: string) => {
    switch (status) {
      case 'healthy': return '●';
      case 'unhealthy': return '⚠';
      default: return '?';
    }
  };

  return (
    <div className="flex items-center space-x-2">
      <span className={`${getHealthColor(server.healthStatus)} text-lg`}>
        {getHealthIcon(server.healthStatus)}
      </span>
      <div className="text-sm">
        <div className={`font-medium ${getHealthColor(server.healthStatus)}`}>
          {server.healthStatus || 'Unknown'}
        </div>
        {server.lastHealthCheck && (
          <div className="text-gray-500 text-xs">
            Last checked: {new Date(server.lastHealthCheck).toLocaleTimeString()}
          </div>
        )}
      </div>
    </div>
  );
};

// Health metrics chart component
const HealthChart: React.FC<{ serverId: string; timeRange: string }> = ({ serverId, timeRange }
  const [healthData, setHealthData] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchHealthData = async () => {
      try {
        const response = await api.get(`/servers/${serverId}/health/history?range=${timeRange}`
        setHealthData(response.data.data);
      } catch (error) {
        console.error('Failed to fetch health data:', error);
      } finally {
        setLoading(false);
      }
    };

    fetchHealthData();
    const interval = setInterval(fetchHealthData, 30000); // Update every 30 seconds

    return () => clearInterval(interval);
  }, [serverId, timeRange]);

  if (loading) return <LoadingSpinner />;

  return (
    <div className="bg-white p-4 rounded-lg border">
      <h3 className="text-lg font-medium mb-4">Health History ({timeRange})</h3>
      <div className="h-64">
        {/* Simple health status timeline */}
        <div className="flex h-full items-end space-x-1">
          {healthData.map((check, index) => (
            <div
              key={index}
              className={`flex-1 min-w-0 ${
                check.status === 'healthy' ? 'bg-green-500' : 'bg-red-500'
              }`}
              style={{ height: check.status === 'healthy' ? '100%' : '20%' }}
              title={`${check.status} - ${new Date(check.timestamp).toLocaleString()}`}
            />
          ))}
        </div>
      </div>
    </div>
  );
};
```

Acceptance Criteria:

- [ ] Health checks run automatically every 30 seconds for running servers
- [ ] HTTP endpoint health checks work correctly
- [ ] Process and container health validation works
- [ ] Health history is stored and retrievable
- [ ] Real-time health updates are broadcasted via WebSocket
- [ ] Health monitoring stops when servers are stopped
- [ ] Health summary calculations are accurate

REQ-2.3: Logging System

Priority: P1
Estimated Effort: 14 hours

Functional Requirements:

- Capture process stdout/stderr
- Log level detection and parsing
- Real-time log streaming
- Log history storage and search
- Log export functionality

Technical Implementation:

javascript

```
class LoggingService {
  constructor(database, websocketService) {
    this.db = database;
    this.ws = websocketService;
    this.logBuffers = new Map(); // serverId -> circular buffer
    this.maxBufferSize = 1000; // Keep last 1000 log entries in memory
  }

  async logMessage(serverId, level, message, source = 'system') {
    const logEntry = {
      server_id: serverId,
      level,
      message: message.trim(),
      source,
      timestamp: new Date()
    };

    try {
      // Store in database
      await this.db.insertLogEntry(logEntry);

      // Add to in-memory buffer
      this.addToBuffer(serverId, logEntry);

      // Broadcast to WebSocket clients
      this.ws.broadcast('logs:new', {
        serverId,
        ...logEntry
      });

      return logEntry;
    } catch (error) {
      console.error('Failed to log message:', error);
    }
  }

  addToBuffer(serverId, logEntry) {
    if (!this.logBuffers.has(serverId)) {
      this.logBuffers.set(serverId, []);
    }

    const buffer = this.logBuffers.get(serverId);
    buffer.push(logEntry);
    // Keep buffer size limited
    if (buffer.length > this.maxBufferSize) {
      buffer.shift(); // Remove oldest entry
    }
  }

  async getRecentLogs(serverId, limit = 100) {
    try {
      // Try memory buffer first
      const buffer = this.logBuffers.get(serverId);
      if (buffer && buffer.length >= limit) {
        return buffer.slice(-limit);
      }

      // Fall back to database
      return await this.db.getServerLogs(serverId, limit);
    } catch (error) {
      throw new Error(`Failed to get recent logs: ${error.message}`);
    }
  }

  async searchLogs(serverId, query, options = {}) {
    try {
      const {
        level,
        source,
        startDate,
        endDate,
        limit = 100
      } = options;

      return await this.db.searchServerLogs(serverId, query, {
        level,
        source,
        startDate,
        endDate,
        limit
      });
    } catch (error) {
      throw new Error(`Failed to search logs: ${error.message}`);
    }
  }
```

```javascript
async exportLogs(serverId, format = 'json', options = {}) {
  try {
    const logs = await this.db.getServerLogs(serverId, options.limit || 10000);

    switch (format) {
      case 'json':
        return JSON.stringify(logs, null, 2);

      case 'csv':
        const headers = 'timestamp,level,source,message\n';
        const csvRows = logs.map(log =>
          `"${log.timestamp}","${log.level}","${log.source}","${log.message.replace(/"/g, '""
        );
        return headers + csvRows.join('\n');

      case 'txt':
        return logs.map(log =>
          `[${log.timestamp}] ${log.level.toUpperCase()} (${log.source}): ${log.message}`
        ).join('\n');

      default:
        throw new Error(`Unsupported export format: ${format}`);
    }
  } catch (error) {
    throw new Error(`Failed to export logs: ${error.message}`);
  }
}

// Parse log level from message content
parseLogLevel(message) {
  const patterns = {
    error: /\b(error|err|fatal|exception|fail)\b/i,
    warn: /\b(warn|warning|caution)\b/i,
    debug: /\b(debug|trace|verbose)\b/i,
    info: /\b(info|information)\b/i
  };

  for (const [level, pattern] of Object.entries(patterns)) {
    if (pattern.test(message)) {
      return level;
    }
  }

  return 'info'; // default
}

// Clean up old logs periodically
async cleanupOldLogs(retentionDays = 30) {
  try {
    const cutoffDate = new Date();
    cutoffDate.setDate(cutoffDate.getDate() - retentionDays);

    const deleted = await this.db.deleteOldLogs(cutoffDate);
    console.log(`Cleaned up ${deleted} old log entries`);

    return deleted;
  } catch (error) {
    console.error('Failed to cleanup old logs:', error);
  }
}

// Start periodic cleanup
startPeriodic
```