Smart Splunk-Teams-MCP Integration Documentation

Table of Contents

Overview

The Smart Splunk-Teams-MCP Integration (`complete_smart_app.py`) is an intelligent alert processing system that:

- **Receives Splunk alerts** via webhook
- **Dynamically discovers** available Kubernetes tools from MCP servers
- **Intelligently analyzes** alerts and decides on appropriate actions
- **Automatically restarts** pods when beneficial
- **Sends rich notifications** to Microsoft Teams with analysis results

Key Features

- 🧠 **Smart Tool Discovery**: Automatically adapts to any MCP server
- 🔄 **Intelligent Pod Restart Logic**: Context-aware restart decisions
- 📊 **Rich Teams Cards**: Detailed adaptive cards with analysis
- 🛡️ **Production Safety**: Environment-aware safety checks
- 🔍 **Comprehensive Logging**: Full traceability of actions

Installation & Setup

Prerequisites

1. **Python 3.7+** with pip
2. **Node.js** (if your MCP server is Node.js-based)
3. **kubectl** configured for your Kubernetes cluster
4. **Microsoft Teams webhook URL**
5. **MCP Server** (Kubernetes-enabled)

Install Dependencies

```bash
# Clone or download the application
# Navigate to the directory containing complete_smart_app.py

# Install Python dependencies
pip install flask requests

# Or using requirements.txt (create if needed)
echo "flask>=2.0.0" > requirements.txt
echo "requests>=2.25.0" >> requirements.txt
pip install -r requirements.txt
```

MCP Server Setup

Ensure your MCP server is configured and accessible:

```bash
# Test your MCP server manually
node /path/to/your/mcp-server/index.js

# Verify kubectl access
kubectl get pods --all-namespaces
```

Configuration

Environment Variables

Set the following environment variables:

```bash
```

```
# Required: Teams webhook URL
export TEAMS_WEBHOOK_URL="https://outlook.office.com/webhook/your-webhook-url"

# Required: MCP server configuration
export MCP_SERVER_PATH="/path/to/your/mcp-server/index.js"
export MCP_SERVER_COMMAND="node"

# Optional: Timeout and port settings
export MCP_TIMEOUT="60"
export PORT="5000"
export DEBUG="false"
```

Windows Configuration

```cmd
# Windows Command Prompt
set TEAMS_WEBHOOK_URL=https://outlook.office.com/webhook/your-webhook-url
set MCP_SERVER_PATH=C:\path\to\mcp-server\index.js
set MCP_SERVER_COMMAND=node

# Windows PowerShell
$env:TEAMS_WEBHOOK_URL="https://outlook.office.com/webhook/your-webhook-url"
$env:MCP_SERVER_PATH="C:\path\to\mcp-server\index.js"
$env:MCP_SERVER_COMMAND="node"
```

Teams Webhook Setup

1. Go to your Teams channel
2. Click **"..."** → **Connectors** → **Incoming Webhook**
3. Configure the webhook and copy the URL
4. Set the URL in `TEAMS_WEBHOOK_URL` environment variable

Running the Application

Start the Server

```bash
```

```
# Run the application
python complete_smart_app.py

# Expected output:
# 2025-01-15 10:30:00 - __main__ - INFO - ✅ Teams webhook URL configured
# 2025-01-15 10:30:00 - __main__ - INFO - ✅ MCP server found: /path/to/mcp-server/index.js
# 2025-01-15 10:30:00 - __main__ - INFO - 🚀 Starting Smart Splunk-Teams-MCP Integration Server...
# 2025-01-15 10:30:00 - __main__ - INFO - 🧠 Smart tool discovery enabled - adapts to any MCP server!
#  * Running on all addresses (0.0.0.0)
#  * Running on http://127.0.0.1:5000
```

Verify Server is Running

```bash
# Test health endpoint
curl http://localhost:5000/health

# Expected response:
{
  "status": "healthy",
  "timestamp": "2025-01-15T10:30:00.000000",
  "service": "smart-splunk-teams-mcp-forwarder"
}
```

Testing Instructions

1. Basic Health Check

```bash
curl -X GET http://localhost:5000/health
```

2. Test Smart Tool Discovery

```bash
curl -X POST http://localhost:5000/test-smart-mapping \
  -H "Content-Type: application/json" \
  -d '{
    "namespace": "default",
    "pod_name": "test-pod"
  }'
```

Expected Response:

```json
{
  "status": "success",
  "message": "Smart mapping test completed",
  "results": {
    "discovery_success": true,
    "capabilities": {
      "pod_status": {
        "tool_name": "describe_pod",
        "confidence": 0.85,
        "description": "Get detailed pod information"
      },
      "pod_logs": {
        "tool_name": "get_logs",
        "confidence": 0.90,
        "description": "Retrieve pod logs"
      }
    },
    "test_calls": {
      "pod_status": {
        "success": true,
        "tool_used": "describe_pod"
      }
    }
  }
}
```

3. Test Basic Alert (No MCP)

```bash
curl -X POST http://localhost:5000/test-alert-basic \
  -H "Content-Type: application/json" \
  -d '{
    "search_name": "Basic Test Alert",
    "severity": "Medium"
  }'
```

4. Test Full Smart Alert Processing

```bash
```

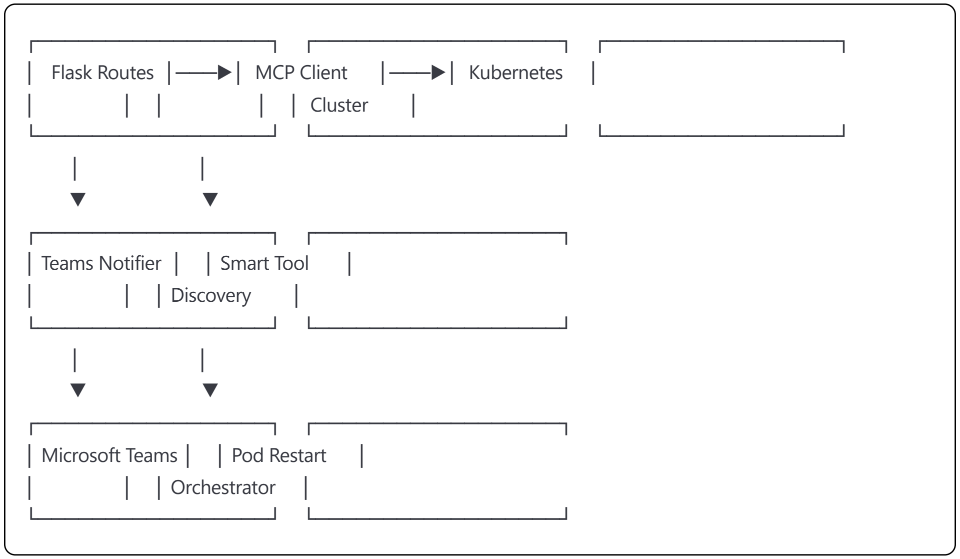
```
curl -X POST http://localhost:5000/test-alert \
  -H "Content-Type: application/json" \
  -d '{
    "search_name": "Memory Leak Detection",
    "severity": "Critical",
    "kubernetes_namespace": "staging",
    "kubernetes_container_name": "api-server",
    "_raw": "OutOfMemoryError detected in container, pod restart recommended"
  }'
```

5. Simulate Real Splunk Webhook

```bash
curl -X POST http://localhost:5000/splunk-alert \
  -H "Content-Type: application/json" \
  -d '{
    "search_name": "Pod CrashLoopBackOff Alert",
    "severity": "High",
    "host": "k8s-worker-01",
    "trigger_time": "2025-01-15 14:30:00",
    "result_count": 5,
    "view_link": "https://splunk.company.com/app/search/alert/12345",
    "kubernetes_namespace": "production",
    "kubernetes_container_name": "payment-service",
    "guid": "alert-guid-12345",
    "_raw": "Pod payment-service in production namespace is in CrashLoopBackOff state",
    "description": "Payment service pod is crashing repeatedly"
  }'
```

Code Architecture

The application follows a modular architecture with four main components:



MCP Client Implementation

SimpleMCPClient Class

The `SimpleMCPClient` handles communication with MCP (Model Context Protocol) servers via stdin/stdout.

Key Features:

- **Asynchronous communication** with MCP servers
- **JSON-RPC 2.0** protocol implementation
- **Timeout handling** and error recovery
- **Subprocess management** for MCP server processes

Core Methods:

```python
class SimpleMCPClient:
    def __init__(self):
        self.server_command = MCP_SERVER_COMMAND  # e.g., "node"
        self.server_path = MCP_SERVER_PATH        # Path to MCP server
        self.timeout = MCP_TIMEOUT                # Request timeout

    async def call_mcp_tool(self, tool_name, params):
        """Main method to call any MCP tool"""

    async def _call_mcp_server(self, mcp_request):
        """Low-level MCP server communication"""
```

Communication Flow:

1. **Request Formation**: Creates JSON-RPC 2.0 request

```python
mcp_request = {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "tools/call",
    "params": {
        "name": tool_name,
        "arguments": params
    }
}
```

2. **Process Management**: Spawns MCP server subprocess

```python
process = await asyncio.create_subprocess_exec(
    self.server_command,
    self.server_path,
    stdin=asyncio.subprocess.PIPE,
    stdout=asyncio.subprocess.PIPE,
    stderr=asyncio.subprocess.PIPE
)
```

3. **Communication**: Sends request and receives response

```python
stdout, stderr = await asyncio.wait_for(
    process.communicate(input=request_json.encode()),
    timeout=self.timeout
)
```

4. **Response Processing**: Parses JSON response and handles errors

```python
if 'result' in response:
    return response['result']
elif 'error' in response:
    raise Exception(f"MCP server error: {response['error']}")
```

Error Handling:

- **Timeout Errors**: Handles MCP server timeouts gracefully
- **JSON Parse Errors**: Validates and reports malformed responses
- **Server Errors**: Propagates MCP server-specific errors
- **Process Errors**: Manages subprocess communication failures

Smart Tool Discovery & Mapping

SmartToolMapper Class

The `SmartToolMapper` is the intelligence layer that automatically discovers and maps MCP tools to application intents.

Core Concepts:

1. **Intent Signatures**: Patterns that define what we're looking for
2. **Parameter Mapping**: Rules to adapt parameters to different tool schemas
3. **Confidence Scoring**: Algorithmic matching to find best tools
4. **Schema Adaptation**: Dynamic parameter conversion

Intent Signatures:

```python
self.intent_signatures = {
    'pod_status': {
        'name_patterns': [r'describe.*pod', r'get.*pod.*status', r'pod.*info'],
        'desc_patterns': [r'describe.*pod', r'pod.*status', r'pod.*detail'],
        'required_words': ['pod'],
        'exclude_words': ['delete', 'create', 'update', 'list'],
        'weight': {'name': 0.7, 'desc': 0.3}
    },
    'pod_logs': {
        'name_patterns': [r'get.*log', r'log', r'tail'],
        'desc_patterns': [r'log', r'tail', r'output'],
        'required_words': ['log'],
        'exclude_words': ['delete', 'create'],
        'weight': {'name': 0.8, 'desc': 0.2}
    }
    # ... more intents
}
```

Discovery Process:

1. **Tool Enumeration**: Gets all available tools from MCP server

```python
```

```
tools_response = await mcp_client.call_mcp_tool('tools/list', {})
self.available_tools = tools_response['tools']
```

2. **Pattern Matching**: Matches tools to intents using regex and keywords

python

```
def _calculate_match_score(self, tool, signature):
    # Check required words (must have at least one)
    has_required = any(word in tool_name or word in tool_desc
                      for word in signature['required_words'])

    # Check exclude words (must not have any)
    has_excluded = any(word in tool_name or word in tool_desc
                      for word in signature['exclude_words'])

    # Calculate pattern scores
    name_score = sum(1 for pattern in signature['name_patterns']
                    if re.search(pattern, tool_name))

    # Return weighted final score
    return final_score
```

3. **Confidence Ranking**: Ranks tools by match confidence and selects best

python

```
candidates.sort(key=lambda x: x['score'], reverse=True)
if candidates and candidates[0]['score'] > 0.3:
    return candidates[0]  # Only return if confidence is reasonable
```

Parameter Adaptation:

python

```
self.param_mappings = {
    'pod_name': {
        'candidates': ['name', 'pod_name', 'podName', 'pod', 'resource_name'],
        'type': 'string',
        'required': True
    },
    'namespace': {
        'candidates': ['namespace', 'ns', 'nameSpace', 'project'],
        'type': 'string',
        'required': True
    }
    # ... more mappings
}
```

The mapper automatically converts our standard parameters to whatever format the discovered tool expects.

Smart Execution:

```python
async def execute_smart_call(self, mcp_client, intent, intent_params):
    # Find the right tool for this intent
    capability = self.tool_capabilities[intent]
    tool = capability['tool']

    # Adapt our parameters to tool's expected format
    param_result = self.adapt_parameters(intent_params, tool)

    # Execute the call with adapted parameters
    result = await mcp_client.call_mcp_tool(tool['name'], param_result['params'])
```

Teams Notification System

send_alert_to_teams Function

The Teams notification system creates rich, interactive Adaptive Cards that provide comprehensive alert information and analysis results.

Key Features:

- **Adaptive Cards**: Rich, interactive cards with structured data
- **Dynamic Styling**: Color-coded based on alert severity
- **Smart Analysis Display**: Shows AI analysis with appropriate icons
- **Action Buttons**: Quick links to Splunk and other resources

- **Responsive Layout**: Works across Teams clients

Adaptive Card Structure:

python

```
adaptive_card = {
    "type": "message",
    "attachments": [{
        "contentType": "application/vnd.microsoft.card.adaptive",
        "content": {
            "type": "AdaptiveCard",
            "version": "1.4",
            "body": [
                # Header section with alert icon and title
                # Alert details (severity, host, time, etc.)
                # Kubernetes information (namespace, container)
                # AI analysis results with status indicators
                # Recommended actions
                # Raw log data (if available)
            ],
            "actions": [
                # Link to view in Splunk
            ]
        }
    }]
}
```

Alert Severity Color Mapping:

python

```
color_map = {
    'critical': 'Attention',  # Red theme
    'high': 'Warning',        # Orange theme
    'medium': 'Good',         # Green theme
    'low': 'Accent'           # Blue theme
}
```

AI Analysis Status Indicators:

python

```
if mcp_status == 'success':
    mcp_emoji = "✅"    # Success - actions completed
elif mcp_status == 'partial':
    mcp_emoji = "⚠️"    # Warning - partial success
else:
    mcp_emoji = "❌"    # Error - failed analysis
```

Data Processing:

1. **Extract Alert Data**: Pulls information from Splunk webhook

```python
alert_name = alert_data.get('search_name', 'Unknown Alert')
severity = alert_data.get('severity', 'Medium')
k8s_namespace = alert_data.get('kubernetes_namespace')
k8s_container = alert_data.get('kubernetes_container_name')
```

2. **Process MCP Analysis**: Extracts AI analysis results

```python
mcp_status = mcp_analysis.get('status', 'unknown')
analysis_text = mcp_analysis.get('analysis', 'No analysis available')
recommended_actions = mcp_analysis.get('recommended_actions', [])
restart_executed = mcp_analysis.get('restart_executed', False)
tools_discovered = mcp_analysis.get('tools_discovered', False)
```

3. **Build Dynamic Content**: Creates sections based on available data

```python
```

```
# Core facts always included
facts = [
    {"title": "Severity", "value": severity.upper()},
    {"title": "Host", "value": host},
    {"title": "Results", "value": str(result_count)},
    {"title": "Time", "value": timestamp}
]

# Add Kubernetes info if available
if k8s_namespace:
    facts.append({"title": "K8s Namespace", "value": k8s_namespace})
if k8s_container:
    facts.append({"title": "K8s Container", "value": k8s_container})

# Add action results
facts.append({"title": "Pod Restarted", "value": "Yes" if restart_executed else "No"})
facts.append({"title": "Smart Tools", "value": "Yes" if tools_discovered else "No"})
```

4. **Format Recommended Actions**: Creates bullet list for actions

```python
if recommended_actions:
    actions_text = "\n".join([f"• {action}" for action in recommended_actions])
```

5. **Handle Raw Logs**: Truncates and formats log data

```python
if raw_log:
    truncated_log = raw_log[:300] + "..." if len(raw_log) > 300 else raw_log
    # Add as monospace text block
```

HTTP Delivery:

```python
response = requests.post(TEAMS_WEBHOOK_URL, json=adaptive_card, timeout=10)

if response.status_code == 200:
    logger.info("Successfully sent alert to Teams")
    return True
else:
    logger.error(f"Teams webhook failed: {response.status_code} - {response.text}")
    return False
```

Pod Restart Orchestrator

SmartPodRestartOrchestrator Class

The orchestrator is the main intelligence engine that coordinates alert analysis, investigation, and response actions.

Workflow Overview:

```
Alert → Classify → Investigate → Decide → Act → Verify → Report
```

Key Methods:

1. **analyze_and_respond**: Main orchestration method
2. **_discover_capabilities**: MCP tool discovery
3. **_smart_investigate**: Kubernetes state investigation
4. **_classify_alert**: Alert type classification
5. **_should_restart_pod**: Restart decision logic
6. **_smart_restart_pod**: Pod restart execution
7. **_smart_verify_restart**: Post-restart verification

Alert Classification Logic:

```
python
```

```
def _classify_alert(self, alert_data):
    alert_name = alert_data.get('search_name', '').lower()
    raw_log = alert_data.get('_raw', '').lower()

    # Memory issues - very high restart likelihood
    if any(keyword in alert_name + raw_log for keyword in ['memory', 'oom', 'heap', 'leak']):
        return {
            'type': 'memory_issue',
            'restart_likelihood': 'very_high',
            'reason': 'Memory issues typically resolved by pod restart'
        }

    # Connection/timeout issues - very high restart likelihood
    if any(keyword in alert_name + raw_log for keyword in ['connection', 'timeout', 'hang', 'stuck']):
        return {
            'type': 'connection_issue',
            'restart_likelihood': 'very_high',
            'reason': 'Connection issues usually resolved by pod restart'
        }

    # More classification rules...
```

Restart Decision Matrix:

Restart Likelihood	Critical Severity	Non-Critical	Production	Non-Production
Very High	✅ Restart	✅ Restart	✅ Restart	✅ Restart
High	✅ Restart	⚠️ Manual	⚠️ Manual	✅ Restart
Medium	✅ Restart	⚠️ Manual	⚠️ Manual	⚠️ Manual
Low	⚠️ Manual	⚠️ Manual	⚠️ Manual	⚠️ Manual

Smart Investigation Process:

1. **Pod Status Check**:

python

```python
result = await self.tool_mapper.execute_smart_call(
    self.mcp_client,
    intent='pod_status',
    intent_params={
        'pod_name': container,
        'namespace': namespace
    }
)
```

2. **Log Retrieval**:

```python
result = await self.tool_mapper.execute_smart_call(
    self.mcp_client,
    intent='pod_logs',
    intent_params={
        'pod_name': container,
        'namespace': namespace,
        'log_lines': 50
    }
)
```

3. **Context Gathering**:

```python
result = await self.tool_mapper.execute_smart_call(
    self.mcp_client,
    intent='pod_list',
    intent_params={
        'namespace': namespace
    }
)
```

Restart Execution with Verification:

```python
async def _smart_restart_pod(self, alert_data):
    # Execute restart using discovered tool
    result = await self.tool_mapper.execute_smart_call(
        self.mcp_client,
        intent='pod_restart',
        intent_params={
            'pod_name': container,
            'namespace': namespace
        }
    )

    # Process results and create detailed response
    if result['success']:
        restart_result['success'] = True
        restart_result['tool_used'] = result['tool_used']
        restart_result['message'] = f"Successfully restarted {namespace}/{container} using {result['tool_used']}"
```

Post-Restart Verification:

```python
async def _smart_verify_restart(self, alert_data):
    # Wait for restart to complete
    await asyncio.sleep(30)

    # Check if pods are running again
    result = await self.tool_mapper.execute_smart_call(
        self.mcp_client,
        intent='pod_list',
        intent_params={'namespace': namespace}
    )
```

API Endpoints

Available Endpoints:

Endpoint	Method	Purpose	Description
`/health`	GET	Health Check	Basic server health verification
`/splunk-alert`	POST	Main Webhook	Primary Splunk alert processing endpoint
`/test-alert`	POST	Full Test	Test complete workflow with MCP analysis
`/test-alert-basic`	POST	Basic Test	Test without MCP (Teams notification only)
`/test-smart-mapping`	POST	Tool Discovery Test	Test smart tool discovery and mapping

Endpoint Details:

`/health` - Health Check

```bash
GET /health
```

Response:

```json
{
  "status": "healthy",
  "timestamp": "2025-01-15T10:30:00.000000",
  "service": "smart-splunk-teams-mcp-forwarder"
}
```

`/splunk-alert` - Main Webhook

```bash
POST /splunk-alert
Content-Type: application/json
```

Request Body:

```json
{
  "search_name": "Alert Name",
  "severity": "High|Medium|Low|Critical",
  "host": "server-name",
  "trigger_time": "2025-01-15 14:30:00",
  "result_count": 5,
  "view_link": "https://splunk.company.com/alert/12345",
  "kubernetes_namespace": "production",
  "kubernetes_container_name": "api-server",
  "guid": "alert-guid-12345",
  "_raw": "Raw log data from alert",
  "description": "Alert description"
}
```

Response:

```json
{
  "status": "success",
  "message": "Alert processed and forwarded to Teams with smart MCP analysis",
  "mcp_status": "success",
  "restart_executed": true,
  "tools_discovered": true
}
```

`/test-smart-mapping` - Tool Discovery Test

```bash
POST /test-smart-mapping
Content-Type: application/json
```

Request Body:

```json
```

```
{
  "namespace": "default",
  "pod_name": "test-pod"
}
```

Response:

```json
{
  "status": "success",
  "message": "Smart mapping test completed",
  "results": {
    "discovery_success": true,
    "capabilities": {
      "pod_status": {
        "tool_name": "describe_pod",
        "confidence": 0.85,
        "description": "Get detailed pod information"
      }
    },
    "test_calls": {
      "pod_status": {
        "success": true,
        "tool_used": "describe_pod",
        "has_result": true
      }
    }
  }
}
```

Troubleshooting

Common Issues and Solutions:

1. Teams Webhook Not Configured

Error: `⚠️ Teams webhook URL not configured!`

Solution:

```bash
export TEAMS_WEBHOOK_URL="https://outlook.office.com/webhook/your-actual-webhook-url"
```

2. MCP Server Not Found

Error: `⚠️ MCP server not found: /path/to/server`

Solutions:

- Verify the path exists: `ls -la /path/to/your/mcp-server/`
- Check file permissions: `chmod +x /path/to/your/mcp-server/index.js`
- Test server manually: `node /path/to/your/mcp-server/index.js`

3. Tool Discovery Failure

Error: `❌ Failed to discover MCP capabilities`

Debug Steps:

1. Test MCP server connectivity:

```bash
curl -X POST http://localhost:5000/test-smart-mapping
```

2. Check MCP server logs in application output
3. Verify kubectl access from server location:

```bash
kubectl get pods --all-namespaces
```

4. Permission Denied for Kubernetes Operations

Error: `Error: Forbidden (user "..." cannot get resource "pods")`

Solutions:

- Check kubectl configuration: `kubectl config current-context`
- Verify RBAC permissions for the service account
- Test kubectl access: `kubectl auth can-i get pods --namespace=your-namespace`

5. Teams Card Not Displaying

Issue: Teams receives notification but card doesn't render

Debug Steps:

1. Check Teams webhook response status
2. Validate Adaptive Card JSON structure
3. Test with basic alert: `curl -X POST http://localhost:5000/test-alert-basic`

6. High Memory Usage or Timeouts

Symptoms: Server becomes unresponsive or uses excessive memory

Solutions:

- Reduce MCP timeout: `export MCP_TIMEOUT="30"`
- Monitor with: `top -p $(pgrep -f complete_smart_app.py)`
- Check MCP server resource usage
- Restart the application periodically in production

Debug Mode

Enable debug logging for detailed troubleshooting:

```bash
export DEBUG="true"
python complete_smart_app.py
```

This will provide:

- Detailed HTTP request/response logs
- MCP communication traces
- Tool discovery process details
- Parameter mapping information

Log Analysis

Key log patterns to watch for:

- `✅ MCP capabilities discovered:` - Successful tool discovery
- `🔄 Restarting pod via smart mapping:` - Pod restart attempt
- `❌ Smart restart failed:` - Restart failure
- `Successfully sent alert to Teams` - Teams notification success
- `MCP server stderr:` - MCP server error output

Production Considerations

1. **Resource Limits**: Set appropriate memory limits for the container
2. **Health Checks**: Monitor the `/health` endpoint
3. **Log Rotation**: Configure log rotation for the application
4. **Backup MCP Servers**: Consider fallback MCP server configurations
5. **Rate Limiting**: Implement rate limiting for webhook endpoints

6. **Monitoring**: Set up monitoring for failed Teams notifications and MCP calls

---

This documentation covers the complete Smart Splunk-Teams-MCP Integration system. For additional support or feature requests, please refer to the application logs and error messages for specific debugging information.