GenAI API Implementation Guide

Monitor Dashboard - AI-Powered Scraper Management System

GenAI API Implementation Guide for Monitor Dashboard

Executive Summary

This document provides a comprehensive guide for implementing the GenAI solution into the Monitor Dashboard application. The system is designed to automatically detect, analyze, and fix broken web scrapers using AI-powered code generation and deployment automation.

Table of Contents

System Architecture

Technology Stack

Frontend:

- React 19.1.0
- React Router DOM 7.6.3
- Tailwind CSS (styling)
- Axios (HTTP client)

Backend:

- Node.js with Express.js 4.18.2
- SQLite/MongoDB with Sequelize ORM
- JWT authentication
- CORS enabled for cross-origin requests

Additional Dependencies:

- `bcryptjs` for password hashing
- `helmet` for security headers
- `morgan` for request logging
- `simple-git` for Git operations
- `winston` for logging

System Components

```
┌─────────────────────────────────────────────────────────────
│                        Frontend (React)
│
│  ┌─────────────────┐  ┌─────────────────┐  ┌────────────────
│  │   Dashboard     │  │  AI Fix Review  │  │   Deployment
│  │   Component     │  │    Component    │  │     Review
│  └─────────────────┘  └─────────────────┘  └────────────────
│                              │
│  ┌─────────────────┐  ┌─────────────────┐  ┌────────────────
│  │   AI Service    │  │ Deployment Svc  │  │   Constants
│  │   (Frontend)    │  │   (Frontend)    │  │   & Utilities
│  └─────────────────┘  └─────────────────┘  └────────────────
└─────────────────────────────────────────────────────────────
                             │
                             │ HTTP/REST API
                             │
┌─────────────────────────────────────────────────────────────
│                       Backend (Node.js)
│
│  ┌─────────────────┐  ┌─────────────────┐  ┌────────────────
│  │   AI Routes     │  │  Monitor Routes │  │ Deployment Rts
│  │ (Placeholder)   │  │  (Implemented)  │  │ (Placeholder)
│  └─────────────────┘  └─────────────────┘  └────────────────
│                              │
│  ┌─────────────────┐  ┌─────────────────┐  ┌────────────────
│  │   Controllers   │  │     Models      │  │   Middleware
│  │                 │  │ (Monitor, Repo) │  │
│  └─────────────────┘  └─────────────────┘  └────────────────
│                              │
│  ┌─────────────────┐  ┌─────────────────┐  ┌────────────────
│  │   Database      │  │   Git Service   │  │   AI Service
│  │   (SQLite)      │  │   (GitHub/GL)   │  │   (TO IMPL)
│  └─────────────────┘  └─────────────────┘  └────────────────
└─────────────────────────────────────────────────────────────
```

Current Implementation Status

✅ Completed Components

1. **Frontend Dashboard UI** - Fully functional

   - User authentication and role-based access - Monitor listing and management - Broken scraper visualization - AI fix review interface - Deployment confirmation workflows

   1. **Backend API Infrastructure** - Partially implemented

      - Express.js server with middleware - Database models (Monitor, Repository) - Authentication system - Basic CRUD operations for monitors

      1. **User Interface Flow** - Complete

         - Login → Dashboard → Broken Scrapers → Fix Detail → AI Review → Deployment - Responsive design with modern UI/UX - Error handling and loading states

         ## ❌ Missing Components (GenAI Integration Points)

         1. **AI Service Backend Implementation**
         2. **Deployment Service Integration**
         3. **GitHub/GitLab API Integration**
         4. **Real-time AI Processing**
         5. **Code Validation and Testing**

GenAI Integration Points

1. AI Fix Generation Workflow

```
User clicks "Generate AI Fix"
    ↓
Frontend calls: POST /api/ai/generate-fix
    ↓
Backend processes:
    - Extracts monitor configuration
    - Analyzes error patterns
    - Calls GenAI service
    - Validates generated code
    - Returns fix with explanation
    ↓
Frontend displays side-by-side comparison
    ↓
User reviews and accepts/rejects
```

2. Key Integration Points

A. Error Analysis (`ScraperFixDetail.js`)

```
const handleMagic = async () => {
  const result = await AIService.generateFix(scraperIc
    errorSummary: scraper.errorSummary,
    lastAction: scraper.lastAction,
    monitorType: scraper.name
  });
  // Process result and navigate to review
};
```

B. Code Review (`AiFixReview.js`)

- Displays old vs new code comparison

- Shows AI explanation and confidence level
- Provides deployment validation

C. Deployment Process (`DeploymentService.js`)

- Creates GitHub/GitLab pull requests
- Manages deployment status
- Handles rollback operations

API Endpoints to Implement

AI Service Endpoints

`POST /api/ai/generate-fix`

Purpose: Generate AI-powered fix for broken monitor **Request Body:**

```json
{
  "monitorId": "string",
  "errorData": {
    "errorSummary": "string",
    "lastAction": "string",
    "monitorType": "string",
    "targetUrl": "string",
    "selectors": {
      "css": ["string"],
      "xpath": ["string"]
    },
    "lastWorkingCode": "string",
    "errorLogs": ["string"]
  },
  "context": {
    "screenshotUrl": "string",
    "pageSource": "string",
    "networkLogs": ["object"]
  }
}
```

Response:

```json
{
  "success": true,
  "data": {
    "fixId": "string",
    "code": "string",
    "explanation": "string",
    "confidence": 0.95,
    "estimatedTime": "2-3 minutes",
    "changes": [
      {
        "type": "selector_update",
        "old": "#transactionsTable",
        "new": "#bankTransactions",
        "reason": "Element ID changed in new page stru
      }
    ],
    "validationResults": {
      "syntaxValid": true,
      "compatibilityCheck": true,
      "securityScan": true
    }
  },
  "message": "AI fix generated successfully"
}
```

`POST /api/ai/analyze-error`

Purpose: Deep analysis of monitor errors **Request Body:**

```json
{
  "monitorId": "string",
  "errorLogs": ["string"],
  "screenshotData": "base64_string",
  "pageSource": "string",
  "networkActivity": ["object"]
}
```

Response:

```
{
  "success": true,
  "data": {
    "rootCause": "string",
    "errorCategory": "selector_change|timeout|authenti
    "affectedElements": ["string"],
    "recommendations": ["string"],
    "severity": "low|medium|high",
    "fixComplexity": "simple|moderate|complex"
  }
}
```

Deployment Service Endpoints

`POST /api/deployments/create`

Purpose: Create deployment for AI-generated fix **Request Body:**

```
{
  "monitorId": "string",
  "fixId": "string",
  "repositoryId": "string",
  "branchName": "string",
  "commitMessage": "string",
  "deploymentType": "pull_request|direct_commit"
}
```

Response:

```
{
  "success": true,
  "data": {
    "deploymentId": "string",
    "prUrl": "string",
    "status": "pending|in_progress|completed|failed",
    "steps": [
      {
        "name": "string",
        "status": "pending|running|completed|failed",
        "timestamp": "ISO_date"
      }
    ]
  }
}
```

Data Flow and Processing

1. Monitor Health Check Process

```
graph TD
    A[Monitor Execution] --> B{Success?}
    B -->|Yes| C[Update Status: Active]
    B -->|No| D[Capture Error Details]
    D --> E[Store Error Summary]
    E --> F[Mark as Broken]
    F --> G[Trigger AI Analysis]
    G --> H[Generate Fix Recommendation]
```

2. AI Fix Generation Process

```
graph TD
    A[User Requests Fix] --> B[Collect Monitor Data]
    B --> C[Extract Error Context]
    C --> D[Call GenAI Service]
    D --> E[Generate Code Fix]
    E --> F[Validate Generated Code]
    F --> G[Return Fix with Explanation]
    G --> H[Display in UI]
    H --> I[User Reviews]
    I --> J{Accept?}
    J -->|Yes| K[Deploy Fix]
    J -->|No| L[Return to Dashboard]
```

3. Database Schema

Monitor Table

```
CREATE TABLE monitors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    target_url VARCHAR(255) NOT NULL,
    monitor_type ENUM('web_scraping', 'api_monitoring'
    status ENUM('active', 'inactive', 'broken', 'maint
    selectors TEXT, -- JSON
    repository_id INTEGER,
    last_check DATETIME,
    error_summary TEXT,
    created_at DATETIME,
    updated_at DATETIME
);
```

AI Fix History Table (New)

```
CREATE TABLE ai_fixes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    monitor_id INTEGER NOT NULL,
    fix_code TEXT NOT NULL,
    explanation TEXT,
    confidence DECIMAL(3,2),
    status ENUM('generated', 'reviewed', 'deployed', '
    deployment_id INTEGER,
    created_at DATETIME,
    FOREIGN KEY (monitor_id) REFERENCES monitors(id)
);
```

Authentication and Security

Current Authentication System

The system uses JWT-based authentication with role-based access control:

- **Admin Role**: Full access to all features
- **Operator Role**: Limited access to monitoring and fixes

Security Considerations for GenAI Integration

1. **Input Validation**

   - Sanitize all user inputs before sending to AI service - Validate generated code for security vulnerabilities - Prevent code injection attacks

   1. **Rate Limiting**

      - Implement rate limiting for AI service calls - Prevent abuse of expensive AI operations

      1. **Access Control**

- Restrict AI fix generation to authorized users - Log all AI service interactions for audit

1. **Code Safety**

   - Validate generated code syntax - Scan for potential security issues - Sandbox testing environment

   ---

Error Handling and Monitoring

Error Categories

1. **AI Service Errors**

   - Service unavailable - Invalid response format - Timeout errors - Rate limit exceeded

   1. **Code Generation Errors**

      - Syntax errors in generated code - Logic errors - Compatibility issues

      1. **Deployment Errors**

         - Git repository access issues - Merge conflicts - CI/CD pipeline failures

Monitoring and Logging

```
// Example error handling structure
const errorHandler = {
  aiService: {
    timeout: 30000,
    retryAttempts: 3,
    fallbackResponse: "AI service tempo
  },
  deployment: {
    timeout: 60000,
    retryAttempts: 2,
    rollbackOnFailure: true
  }
};
```

Testing and Validation

Testing Strategy

1. **Unit Tests**

   - Test AI service integration - Validate code generation logic - Test deployment workflows

   1. **Integration Tests**

      - End-to-end workflow testing - API endpoint testing - Database integration tests

      1. **AI Model Testing**

         - Code quality validation - Fix accuracy measurement - Performance benchmarking

Validation Framework

```
// Code validation example
const validateGeneratedCode = asy
  const results = {
    syntaxValid: await validateSy
    securityScan: await scanForVu
    compatibilityCheck: await che
    performanceScore: await analy
  };
  return results;
};
```

Deployment Strategy

Environment Configuration

Development Environment

```
AI Service Confi

AI_SERVICE_URL=http://localhost:8
AI_SERVICE_API_KEY=dev_key_12345
AI_SERVICE_TIMEOUT=30000

Deployment Confi

GITHUB_TOKEN=ghp_xxxxxxxxxxxx
GITLAB_TOKEN=glpat_xxxxxxxxxxxx
DEPLOYMENT_TIMEOUT=60000
```

Production Environment

```
AI Service Confi
```

```
AI_SERVICE_URL=https://api.ai-ser
AI_SERVICE_API_KEY=prod_key_secur
AI_SERVICE_TIMEOUT=45000
```

```
Deployment Config
```

```
GITHUB_TOKEN=ghp_prod_token
GITLAB_TOKEN=glpat_prod_token
DEPLOYMENT_TIMEOUT=120000
```

Deployment Pipeline

1. **Code Generation**

   - AI service generates fix - Validate generated code - Store in database

   1. **Review Process**

      - User reviews AI-generated fix - Side-by-side comparison - Approval workflow

      1. **Deployment Execution**

         - Create Git branch - Commit changes - Create pull request - Notify stakeholders

         ---

Performance Considerations

AI Service Performance

1. **Response Time Optimization**

   - Cache common fix patterns - Implement request queuing - Use asynchronous processing

   1. **Resource Management**

      - Monitor API usage - Implement connection pooling - Handle concurrent requests

      ## Database Performance

      1. **Query Optimization**

         - Index critical fields - Optimize complex queries - Implement caching

         1. **Data Management**

            - Archive old fix history - Implement data retention policies - Monitor database size

            ---

Future Enhancements

Phase 1: Core GenAI Integration

- Implement basic AI fix generation
- Add code validation
- Create deployment workflow

Phase 2: Advanced Features

- Multi-language support
- Complex error pattern recognition
- Automated testing integration

Phase 3: Enterprise Features

- Advanced analytics
- Custom AI model training
- Enterprise integrations

Phase 4: Scaling and Optimization

- Microservices architecture
- Performance optimization
- Advanced monitoring

Implementation Checklist

Backend Implementation

- [ ] Create AI service controller
- [ ] Implement GenAI API integration
- [ ] Add code validation logic
- [ ] Create deployment service
- [ ] Implement GitHub/GitLab API
- [ ] Add error handling and logging
- [ ] Create database migrations
- [ ] Add comprehensive tests

Frontend Updates

- [ ] Update AI service integration
- [ ] Add real-time status updates
- [ ] Implement error handling
- [ ] Add loading states
- [ ] Create admin configuration panel
- [ ] Add analytics dashboard

DevOps and Deployment

- [ ] Set up CI/CD pipeline
- [ ] Configure environment variables
- [ ] Set up monitoring and alerting
- [ ] Create deployment documentation
- [ ] Implement backup and recovery

Contact and Support

For questions regarding this implementation guide or the GenAI integration:

- **Technical Lead**: Review system architecture and API design
- **AI Team**: Consult on AI service integration and model selection
- **DevOps Team**: Coordinate deployment and infrastructure setup
- **QA Team**: Validate testing strategy and quality assurance

---

Appendix

A. API Response Examples

Successful AI Fix Generation

```
{
  "success": true,
  "data": {
    "fixId": "fix_1
    "code": "# AI-g
    "explanation":
    "confidence": (
    "estimatedTime'
    "changes": [
      {
        "type": "se
        "old": ".ol
        "new": ".ne
        "reason": '
      }
    ]
  }
}
```

Error Response

```
{
  "success": false
  "error": "AI serv
  "details": "Conne
  "retryAfter": 30
  "errorCode": "AI
}
```

B. Configuration Templates

AI Service Configuration

```
const aiConfig = {
  baseUrl: process
  apiKey: process.
  timeout: parseInt
  retryAttempts: 3
  models: {
    codeGeneration
    errorAnalysis:
    validation: "co
  }
};
```

Deployment Configuration

```
const deploymentCo
  github: {
    token: process
    baseUrl: "http
    timeout: 60000
  },
  gitlab: {
    token: process
    baseUrl: "http
    timeout: 60000
  },
  defaultBranch: "
  prTemplate: "fix
};
```

This document serves as a comprehensive guide for implementing the GenAI solution into the Monitor Dashboard application. Regular updates will be made as the implementation progresses.

Generated on 2025-07-10 21:34:29