GenAI Implementation Guide for Monitor Dashboard

Integrating AI Agents and Solutions with Existing API

Table of Contents

1. Overview
2. Architecture Design
3. API Integration Strategy
4. Agent Implementation
5. AI Service Setup
6. Code Implementation
7. Testing and Validation
8. Deployment Strategy
9. Monitoring and Maintenance
10. Troubleshooting

---

1. Overview

Purpose

This guide explains how to implement a GenAI solution and intelligent agent system into the Monitor Dashboard application. The implementation leverages the existing API infrastructure to provide automated problem diagnosis, code generation, and deployment management.

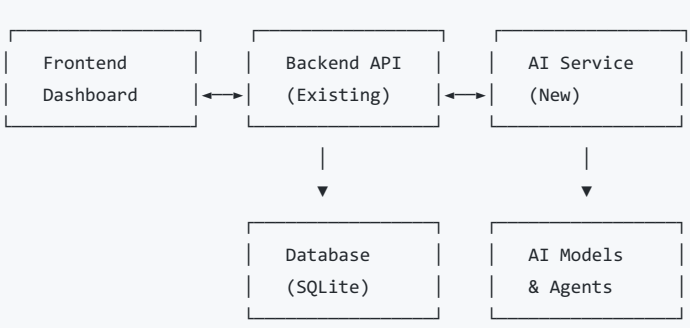
Key Components

- **AI Agent**: Intelligent system for analyzing monitor failures
- **Code Generation**: Automated fix creation for broken scrapers
- **Problem Diagnosis**: Root cause analysis of monitor issues
- **Deployment Automation**: Intelligent deployment decisions
- **Learning System**: Continuous improvement from past fixes

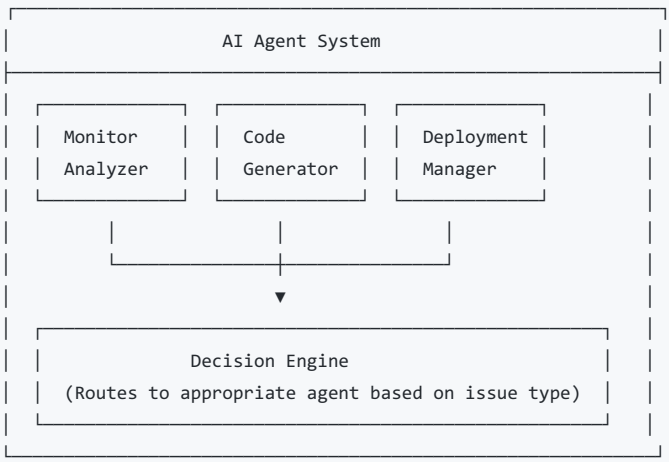
---

2. Architecture Design

High-Level Architecture



Agent Architecture



---

3. API Integration Strategy

Existing API Endpoints to Extend

```
// Current endpoints in backend/src/routes/ai.js
POST /api/ai/generate-fix      // Enhanced with agent logic
POST /api/ai/analyze-error     // Enhanced with ML analysis
POST /api/ai/review-fix        // New: AI review system
GET  /api/ai/status           // Enhanced with agent status
```

New API Endpoints to Add

```
// New endpoints to implement
POST /api/ai/agent/analyze     // Agent-based analysis
POST /api/ai/agent/generate    // Agent-based code generation
POST /api/ai/agent/deploy      // Agent-based deployment
GET  /api/ai/agent/status      // Agent system status
POST /api/ai/agent/learn       // Learning from outcomes
```

---

4. Agent Implementation

4.1 Monitor Analyzer Agent

```
// backend/src/services/agents/monitorAnalyzer.js
class MonitorAnalyzerAgent {
  constructor() {
    this.mlModel = null;
    this.patternDatabase = new Map();
  }

  async analyzeFailure(monitorData, errorLogs) {
    // 1. Extract features from monitor data
    const features = this.extractFeatures(monitorData, errorLogs);

    // 2. Classify the problem type
    const problemType = await this.classifyProblem(features);

    // 3. Identify root cause
    const rootCause = await this.identifyRootCause(features, problemType);

    // 4. Generate confidence score
    const confidence = this.calculateConfidence(features, problemType);

    return {
      problemType,
      rootCause,
      confidence,
      recommendedActions: this.getRecommendedActions(problemType),
      estimatedFixTime: this.estimateFixTime(problemType)
    };
  }

  extractFeatures(monitorData, errorLogs) {
    return {
      errorType: this.categorizeError(errorLogs),
      frequency: this.calculateFailureFrequency(errorLogs),
      pattern: this.identifyPattern(errorLogs),
      context: this.extractContext(monitorData)
    };
  }

  async classifyProblem(features) {
    // Use ML model to classify problem type
    const classification = await this.mlModel.predict(features);
    return classification.type;
  }
}
```

4.2 Code Generator Agent

```
// backend/src/services/agents/codeGenerator.js
class CodeGeneratorAgent {
  constructor() {
    this.templates = new Map();
    this.contextWindow = 4000;
  }

  async generateFix(analysis, monitorConfig) {
    // 1. Select appropriate template
    const template = this.selectTemplate(analysis.problemType);

    // 2. Generate context-aware code
    const generatedCode = await this.generateCode(template, analysis, monitorConfig);

    // 3. Validate generated code
    const validation = await this.validateCode(generatedCode, monitorConfig);

    // 4. Generate explanation
    const explanation = this.generateExplanation(analysis, generatedCode);

    return {
      code: generatedCode,
      explanation,
      confidence: validation.confidence,
      tests: validation.tests,
      rollbackPlan: this.generateRollbackPlan(monitorConfig)
    };
  }

  async generateCode(template, analysis, monitorConfig) {
    const prompt = this.buildPrompt(template, analysis, monitorConfig);

    // Call OpenAI or other LLM service
    const response = await this.callLLM(prompt);

    return this.parseResponse(response);
  }

  buildPrompt(template, analysis, monitorConfig) {
    return `
      Problem Type: ${analysis.problemType}
      Root Cause: ${analysis.rootCause}
      Monitor Configuration: ${JSON.stringify(monitorConfig)}

      Template:
      ${template}

      Generate a fix that addresses the root cause while maintaining the existing functionality.
      Include proper error handling and logging.
    `;
  }
}
```

4.3 Deployment Manager Agent

```
// backend/src/services/agents/deploymentManager.js
class DeploymentManagerAgent {
  constructor() {
    this.deploymentStrategies = new Map();
    this.riskAssessment = new RiskAssessment();
  }

  async planDeployment(fix, monitorConfig) {
    // 1. Assess deployment risk
    const riskAssessment = await this.riskAssessment.assess(fix, monitorConfig);

    // 2. Select deployment strategy
    const strategy = this.selectStrategy(riskAssessment);

    // 3. Generate deployment plan
    const plan = await this.generatePlan(strategy, fix, monitorConfig);

    // 4. Validate deployment plan
    const validation = await this.validatePlan(plan);

    return {
      strategy,
      plan,
      riskLevel: riskAssessment.level,
      estimatedTime: plan.estimatedTime,
      rollbackPlan: plan.rollbackPlan,
      requiresApproval: riskAssessment.level === 'high'
    };
  }

  async executeDeployment(plan, fix) {
    try {
      // 1. Pre-deployment checks
      await this.preDeploymentChecks(plan);

      // 2. Execute deployment
      const result = await this.executePlan(plan, fix);

      // 3. Post-deployment validation
      const validation = await this.postDeploymentValidation(result);

      // 4. Update learning system
      await this.updateLearningSystem(plan, result, validation);

      return {
        success: validation.success,
        details: result,
        validation: validation
      };
    } catch (error) {
      await this.handleDeploymentError(error, plan);
      throw error;
    }
  }
}
```

---

5. AI Service Setup

5.1 Environment Configuration

```
# backend/.env
# AI Service Configuration
AI_SERVICE_PROVIDER=openai
OPENAI_API_KEY=your_openai_api_key
OPENAI_MODEL=gpt-4
AI_SERVICE_TIMEOUT=30000
AI_SERVICE_MAX_RETRIES=3

# Agent Configuration
AGENT_LEARNING_ENABLED=true
AGENT_CONFIDENCE_THRESHOLD=0.8
AGENT_MAX_ATTEMPTS=3
AGENT_FALLBACK_ENABLED=true

# ML Model Configuration
ML_MODEL_PATH=./models/monitor_classifier
ML_MODEL_UPDATE_INTERVAL=86400
```

5.2 AI Service Integration

```
// backend/src/services/aiService.js
class AIService {
  constructor() {
    this.openai = new OpenAI(process.env.OPENAI_API_KEY);
    this.monitorAnalyzer = new MonitorAnalyzerAgent();
    this.codeGenerator = new CodeGeneratorAgent();
    this.deploymentManager = new DeploymentManagerAgent();
  }

  async generateFix(monitorId, errorData) {
    try {
      // 1. Get monitor data
      const monitor = await Monitor.findByPk(monitorId);

      // 2. Analyze the problem
      const analysis = await this.monitorAnalyzer.analyzeFailure(monitor, errorData);

      // 3. Generate fix if confidence is high enough
      if (analysis.confidence >= process.env.AGENT_CONFIDENCE_THRESHOLD) {
        const fix = await this.codeGenerator.generateFix(analysis, monitor);

        // 4. Plan deployment
        const deployment = await this.deploymentManager.planDeployment(fix, monitor);

        return {
          analysis,
          fix,
          deployment,
          success: true
        };
      } else {
        return {
          analysis,
          fix: null,
          deployment: null,
          success: false,
          reason: 'Confidence too low for automated fix'
        };
      }
    } catch (error) {
      console.error('AI Service Error:', error);
      throw new Error('Failed to generate AI fix');
    }
  }

  async analyzeError(errorData) {
    return await this.monitorAnalyzer.analyzeFailure(null, errorData);
  }

  async reviewFix(fixData) {
    // Implement AI-powered code review
    const review = await this.codeGenerator.reviewCode(fixData);
    return review;
  }
}
```

---

6. Code Implementation

6.1 Enhanced AI Controller

```
// backend/src/controllers/aiController.js
const AIService = require('../services/aiService');
```

```
const aiService = new AIService();

// Generate AI fix for broken monitor
const generateFix = async (req, res) => {
  try {
    const { monitorId, errorData } = req.body;

    const result = await aiService.generateFix(monitorId, errorData);

    res.json({
      success: result.success,
      data: result,
      message: result.success ? 'AI fix generated successfully' : 'Manual intervention required'
    });
  } catch (error) {
    console.error('Error generating AI fix:', error);
    res.status(500).json({
      success: false,
      error: 'Failed to generate AI fix',
      message: error.message
    });
  }
};

// Agent-based analysis
const agentAnalyze = async (req, res) => {
  try {
    const { monitorData, errorLogs } = req.body;

    const analysis = await aiService.monitorAnalyzer.analyzeFailure(monitorData, errorLogs);

    res.json({
      success: true,
      data: analysis,
      message: 'Agent analysis completed'
    });
  } catch (error) {
    console.error('Agent analysis error:', error);
    res.status(500).json({
      success: false,
      error: 'Agent analysis failed',
      message: error.message
    });
  }
};

// Agent-based deployment
const agentDeploy = async (req, res) => {
  try {
    const { fix, monitorConfig } = req.body;

    const deployment = await aiService.deploymentManager.executeDeployment(fix, monitorConfig);

    res.json({
      success: deployment.success,
      data: deployment,
      message: deployment.success ? 'Deployment successful' : 'Deployment failed'
    });
  } catch (error) {
    console.error('Agent deployment error:', error);
    res.status(500).json({
      success: false,
      error: 'Agent deployment failed',
      message: error.message
    });
```

```
  }
};

module.exports = {
  generateFix,
  agentAnalyze,
  agentDeploy
};
```

6.2 Enhanced AI Routes

```
// backend/src/routes/ai.js
const express = require('express');
const router = express.Router();
const aiController = require('../controllers/aiController');

// Existing endpoints (enhanced)
router.post('/generate-fix', aiController.generateFix);
router.post('/analyze-error', aiController.analyzeError);
router.get('/status', aiController.getStatus);

// New agent endpoints
router.post('/agent/analyze', aiController.agentAnalyze);
router.post('/agent/generate', aiController.agentGenerate);
router.post('/agent/deploy', aiController.agentDeploy);
router.get('/agent/status', aiController.agentStatus);
router.post('/agent/learn', aiController.agentLearn);

module.exports = router;
```

6.3 Frontend Integration

```
// src/services/aiService.js
class AIService {
  constructor() {
    this.baseURL = process.env.REACT_APP_API_URL;
  }

  async generateAIFix(monitorId, errorData) {
    try {
      const response = await fetch(`${this.baseURL}/api/ai/generate-fix`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ monitorId, errorData })
      });

      const data = await response.json();
      return data;
    } catch (error) {
      console.error('AI Service Error:', error);
      throw error;
    }
  }

  async agentAnalyze(monitorData, errorLogs) {
    try {
      const response = await fetch(`${this.baseURL}/api/ai/agent/analyze`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ monitorData, errorLogs })
      });

      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Agent Analysis Error:', error);
      throw error;
    }
  }

  async agentDeploy(fix, monitorConfig) {
    try {
      const response = await fetch(`${this.baseURL}/api/ai/agent/deploy`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ fix, monitorConfig })
      });

      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Agent Deployment Error:', error);
      throw error;
    }
  }
}

export default new AIService();
```

7. Testing and Validation

7.1 Unit Tests

```
// backend/tests/agents/monitorAnalyzer.test.js
describe('MonitorAnalyzerAgent', () => {
  let agent;

  beforeEach(() => {
    agent = new MonitorAnalyzerAgent();
  });

  test('should analyze failure correctly', async () => {
    const monitorData = { /* test data */ };
    const errorLogs = { /* test error logs */ };

    const result = await agent.analyzeFailure(monitorData, errorLogs);

    expect(result).toHaveProperty('problemType');
    expect(result).toHaveProperty('rootCause');
    expect(result).toHaveProperty('confidence');
    expect(result.confidence).toBeGreaterThan(0);
    expect(result.confidence).toBeLessThanOrEqual(1);
  });
});
```

7.2 Integration Tests

```
// backend/tests/integration/aiService.test.js
describe('AI Service Integration', () => {
  test('should generate complete fix workflow', async () => {
    const monitorId = 1;
    const errorData = { /* test error data */ };

    const result = await aiService.generateFix(monitorId, errorData);

    expect(result).toHaveProperty('analysis');
    expect(result).toHaveProperty('fix');
    expect(result).toHaveProperty('deployment');
    expect(result).toHaveProperty('success');
  });
});
```

---

8. Deployment Strategy

8.1 Staged Deployment

1. **Phase 1**: Deploy AI analysis only
2. **Phase 2**: Deploy code generation with manual approval
3. **Phase 3**: Deploy automated deployment for low-risk fixes
4. **Phase 4**: Full automation with monitoring

8.2 Monitoring and Rollback

```
// backend/src/services/monitoring/aiMonitor.js
class AIMonitor {
  constructor() {
    this.metrics = new Map();
    this.alertThresholds = {
      errorRate: 0.1,
      confidenceDrop: 0.2,
      deploymentFailure: 0.05
    };
  }

  async monitorAgentPerformance() {
    // Monitor agent performance metrics
    const metrics = await this.collectMetrics();

    // Check for anomalies
    const anomalies = this.detectAnomalies(metrics);

    // Trigger alerts if needed
    if (anomalies.length > 0) {
      await this.triggerAlerts(anomalies);
    }

    return metrics;
  }

  async rollbackIfNeeded(metrics) {
    if (metrics.errorRate > this.alertThresholds.errorRate) {
      await this.rollbackToPreviousVersion();
      return true;
    }
    return false;
  }
}
```

---

9. Monitoring and Maintenance

9.1 Performance Monitoring

- **Agent Response Time**: Monitor how long agents take to analyze and generate fixes
- **Accuracy Metrics**: Track success rate of generated fixes
- **Resource Usage**: Monitor CPU and memory usage of AI services
- **API Rate Limits**: Track usage of external AI APIs

9.2 Learning and Improvement

```
// backend/src/services/learning/agentLearning.js
class AgentLearning {
  constructor() {
    this.trainingData = [];
    this.modelUpdater = new ModelUpdater();
  }

  async learnFromOutcome(fixId, outcome) {
    // Collect training data from fix outcomes
    const trainingData = await this.collectTrainingData(fixId, outcome);

    // Update learning model
    await this.modelUpdater.updateModel(trainingData);

    // Update agent strategies
    await this.updateAgentStrategies(outcome);
  }

  async updateAgentStrategies(outcome) {
    if (outcome.success) {
      // Reinforce successful strategies
      await this.reinforceStrategy(outcome.strategy);
    } else {
      // Adjust failed strategies
      await this.adjustStrategy(outcome.strategy, outcome.failureReason);
    }
  }
}
```

---

10. Troubleshooting

10.1 Common Issues

AI Service Not Responding

```
# Check AI service status
curl http://localhost:3001/api/ai/status

# Check environment variables
echo $OPENAI_API_KEY

# Check logs
tail -f backend/logs/ai-service.log
```

Low Confidence Scores

- Review training data quality
- Check feature extraction logic
- Verify ML model is up to date
- Consider manual intervention threshold

Deployment Failures

- Check repository permissions
- Verify deployment strategy
- Review rollback procedures
- Monitor deployment logs

10.2 Debugging Tools

```
// backend/src/utils/aiDebugger.js
class AIDebugger {
  static async debugFixGeneration(monitorId, errorData) {
    console.log('=== AI Fix Generation Debug ===');
    console.log('Monitor ID:', monitorId);
    console.log('Error Data:', errorData);

    // Step through each agent
    const analysis = await aiService.monitorAnalyzer.analyzeFailure(monitorData, errorData);
    console.log('Analysis Result:', analysis);

    const fix = await aiService.codeGenerator.generateFix(analysis, monitorConfig);
    console.log('Generated Fix:', fix);

    const deployment = await aiService.deploymentManager.planDeployment(fix, monitorConfig);
    console.log('Deployment Plan:', deployment);
  }
}
```

Conclusion

This implementation guide provides a comprehensive approach to integrating GenAI solutions and intelligent agents into your Monitor Dashboard application. The modular design allows for gradual implementation and easy maintenance.

Next Steps

1. Set up the AI service infrastructure
2. Implement the agent classes
3. Integrate with existing API endpoints
4. Add monitoring and learning capabilities
5. Deploy in stages with proper testing

Resources

- OpenAI API Documentation
- Machine Learning Model Training
- Agent Architecture Patterns
- Deployment Best Practices

Version: 1.0
Last Updated: January 2024
Author: Development Team
Status: Implementation Ready