# Technical Debt Management Strategy

## Overview

Implement a comprehensive technical debt management strategy to ensure Audityzer's long-term maintainability, performance, and scalability. This strategy focuses on identifying, prioritizing, and systematically addressing technical debt while preventing its accumulation.

# Technical Debt Assessment

## Current Technical Debt Analysis

```
// Technical Debt Assessment Framework
class TechnicalDebtAssessment {
  constructor() {
    this.debtCategories = {
      'code_quality': {
        weight: 0.3,
        metrics: ['complexity', 'duplication', 'test_coverage', 'documentation']
      },
      'architecture': {
        weight: 0.25,
        metrics: ['coupling', 'cohesion', 'modularity', 'scalability']
      },
      'security': {
        weight: 0.2,
        metrics: ['vulnerabilities', 'outdated_dependencies', 'security_practices']
      },
      'performance': {
        weight: 0.15,
        metrics: ['response_time', 'memory_usage', 'cpu_utilization', 'scalability']
      },
      'maintainability': {
        weight: 0.1,
        metrics: ['documentation_quality', 'code_readability', 'test_quality']
      }
    };
  }

  async assessTechnicalDebt(codebase) {
    const assessment = {
      timestamp: new Date(),
      overallScore: 0,
      categoryScores: {},
      criticalIssues: [],
      recommendations: [],
      estimatedEffort: 0
    };

    for (const [category, config] of Object.entries(this.debtCategories)) {
      const categoryScore = await this.assessCategory(codebase, category, config);
      assessment.categoryScores[category] = categoryScore;
      assessment.overallScore += categoryScore.score * config.weight;
    }

    assessment.criticalIssues = await this.identifyCriticalIssues(codebase);
    assessment.recommendations = await this.generateRecommendations(assessment);
    assessment.estimatedEffort = await this.estimateRemediationEffort(assessment);

    return assessment;
  }

  async assessCategory(codebase, category, config) {
    const categoryAssessment = {
      category,
      score: 0,
      metrics: {},
      issues: [],
      recommendations: []
    };
```

```
    for (const metric of config.metrics) {
      const metricScore = await this.assessMetric(codebase, metric);
      categoryAssessment.metrics[metric] = metricScore;
      categoryAssessment.score += metricScore.score / config.metrics.length;
    }

    return categoryAssessment;
  }

  async assessMetric(codebase, metric) {
    switch (metric) {
      case 'complexity':
        return await this.assessComplexity(codebase);
      case 'duplication':
        return await this.assessDuplication(codebase);
      case 'test_coverage':
        return await this.assessTestCoverage(codebase);
      case 'documentation':
        return await this.assessDocumentation(codebase);
      case 'vulnerabilities':
        return await this.assessVulnerabilities(codebase);
      case 'outdated_dependencies':
        return await this.assessDependencies(codebase);
      default:
        return { score: 0.5, details: 'Metric not implemented' };
    }
  }

  async assessComplexity(codebase) {
    const complexityMetrics = await this.runComplexityAnalysis(codebase);

    const score = this.calculateComplexityScore(complexityMetrics);
    const issues = this.identifyComplexityIssues(complexityMetrics);

    return {
      score,
      details: complexityMetrics,
      issues,
      recommendations: this.generateComplexityRecommendations(issues)
    };
  }

  calculateComplexityScore(metrics) {
    // Cyclomatic complexity scoring
    const avgComplexity = metrics.averageCyclomaticComplexity;
    const maxComplexity = metrics.maxCyclomaticComplexity;
    const highComplexityFunctions = metrics.highComplexityFunctions;

    let score = 1.0;

    // Penalize high average complexity
    if (avgComplexity > 10) score -= 0.3;
    else if (avgComplexity > 7) score -= 0.2;
    else if (avgComplexity > 5) score -= 0.1;

    // Penalize very high maximum complexity
    if (maxComplexity > 20) score -= 0.4;
    else if (maxComplexity > 15) score -= 0.2;
```

```
    // Penalize high number of complex functions
    const complexityRatio = highComplexityFunctions / metrics.totalFunctions;
    if (complexityRatio > 0.2) score -= 0.3;
    else if (complexityRatio > 0.1) score -= 0.1;

    return Math.max(score, 0);
  }
}
```

### Debt Categorization System

```yaml
technical_debt_categories:
  critical:
    priority: "immediate"
    impact: "high"
    examples:
      - "Security vulnerabilities"
      - "Performance bottlenecks"
      - "Critical bugs"
      - "Broken functionality"

  high:
    priority: "next_sprint"
    impact: "medium-high"
    examples:
      - "Code duplication > 20%"
      - "Functions with complexity > 15"
      - "Missing test coverage < 70%"
      - "Outdated major dependencies"

  medium:
    priority: "next_quarter"
    impact: "medium"
    examples:
      - "Code smells"
      - "Minor performance issues"
      - "Documentation gaps"
      - "Refactoring opportunities"

  low:
    priority: "backlog"
    impact: "low"
    examples:
      - "Code style inconsistencies"
      - "Minor optimizations"
      - "Nice-to-have improvements"
      - "Legacy code cleanup"
```

---

## Debt Prevention Strategies

### Code Quality Gates

```
// Automated Quality Gates
class QualityGates {
  constructor() {
    this.gates = [
      new ComplexityGate(),
      new CoverageGate(),
      new DuplicationGate(),
      new SecurityGate(),
      new PerformanceGate()
    ];
  }

  async evaluateChanges(pullRequest) {
    const results = {
      passed: true,
      gates: [],
      blockers: [],
      warnings: []
    };

    for (const gate of this.gates) {
      const gateResult = await gate.evaluate(pullRequest);
      results.gates.push(gateResult);

      if (gateResult.status === 'failed' && gateResult.blocking) {
        results.passed = false;
        results.blockers.push(gateResult);
      } else if (gateResult.status === 'warning') {
        results.warnings.push(gateResult);
      }
    }

    return results;
  }
}

class ComplexityGate {
  constructor() {
    this.thresholds = {
      maxFunctionComplexity: 15,
      maxFileComplexity: 50,
      maxComplexityIncrease: 5
    };
  }

  async evaluate(pullRequest) {
    const complexityAnalysis = await this.analyzeComplexity(pullRequest);

    const violations = [];

    // Check function complexity
    for (const func of complexityAnalysis.functions) {
      if (func.complexity > this.thresholds.maxFunctionComplexity) {
        violations.push({
          type: 'function_complexity',
          file: func.file,
          function: func.name,
          complexity: func.complexity,
```

```
          threshold: this.thresholds.maxFunctionComplexity
        });
      }
    }

    // Check file complexity
    for (const file of complexityAnalysis.files) {
      if (file.complexity > this.thresholds.maxFileComplexity) {
        violations.push({
          type: 'file_complexity',
          file: file.path,
          complexity: file.complexity,
          threshold: this.thresholds.maxFileComplexity
        });
      }
    }

    return {
      gate: 'complexity',
      status: violations.length > 0 ? 'failed' : 'passed',
      blocking: true,
      violations,
      message: this.generateMessage(violations)
    };
  }
}

class CoverageGate {
  constructor() {
    this.thresholds = {
      minOverallCoverage: 80,
      minNewCodeCoverage: 90,
      maxCoverageDecrease: 2
    };
  }

  async evaluate(pullRequest) {
    const coverageReport = await this.analyzeCoverage(pullRequest);

    const violations = [];

    // Check overall coverage
    if (coverageReport.overall < this.thresholds.minOverallCoverage) {
      violations.push({
        type: 'overall_coverage',
        current: coverageReport.overall,
        threshold: this.thresholds.minOverallCoverage
      });
    }

    // Check new code coverage
    if (coverageReport.newCode < this.thresholds.minNewCodeCoverage) {
      violations.push({
        type: 'new_code_coverage',
        current: coverageReport.newCode,
        threshold: this.thresholds.minNewCodeCoverage
      });
    }
```

```
    // Check coverage decrease
    const coverageDecrease = coverageReport.baseline - coverageReport.overall;
    if (coverageDecrease > this.thresholds.maxCoverageDecrease) {
      violations.push({
        type: 'coverage_decrease',
        decrease: coverageDecrease,
        threshold: this.thresholds.maxCoverageDecrease
      });
    }

    return {
      gate: 'coverage',
      status: violations.length > 0 ? 'failed' : 'passed',
      blocking: true,
      violations,
      coverageReport
    };
  }
}
```

## Development Workflow Integration

```yaml
# .github/workflows/quality-gates.yml
name: Quality Gates

on:
  pull_request:
    branches: [ main, develop ]

jobs:
  quality-gates:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3
      with:
        fetch-depth: 0

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run complexity analysis
      run: |
        npm run analyze:complexity
        npm run quality:complexity-gate

    - name: Run test coverage
      run: |
        npm run test:coverage
        npm run quality:coverage-gate

    - name: Check code duplication
      run: |
        npm run analyze:duplication
        npm run quality:duplication-gate

    - name: Security scan
      run: |
        npm audit --audit-level=high
        npm run security:scan

    - name: Performance benchmarks
      run: |
        npm run test:performance
        npm run quality:performance-gate

    - name: Generate quality report
      run: npm run quality:report

    - name: Comment PR with results
      uses: actions/github-script@v6
      with:
        script: |
```

```
            const fs = require('fs');
            const report = JSON.parse(fs.readFileSync('quality-report.json', 'utf8'));

            let comment = '##   Quality Gates Report\n\n';

            for (const gate of report.gates) {
              const status = gate.status === 'passed' ? ' ' : ' ';
              comment += `${status} **${gate.gate}**: ${gate.status}\n`;

              if (gate.violations && gate.violations.length > 0) {
                comment += `    - ${gate.violations.length} violation(s) found\n`;
              }
            }

            if (report.blockers.length > 0) {
              comment += '\n###   Blocking Issues\n';
              for (const blocker of report.blockers) {
                comment += `- **${blocker.gate}**: ${blocker.message}\n`;
              }
            }

            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: comment
            });
```

---

## Refactoring Strategy

### Systematic Refactoring Approach

```
class RefactoringStrategy {
  constructor() {
    this.refactoringTypes = {
      'extract_method': {
        priority: 'high',
        effort: 'low',
        impact: 'medium',
        automation: 'partial'
      },
      'extract_class': {
        priority: 'medium',
        effort: 'medium',
        impact: 'high',
        automation: 'manual'
      },
      'eliminate_duplication': {
        priority: 'high',
        effort: 'medium',
        impact: 'high',
        automation: 'partial'
      },
      'simplify_conditionals': {
        priority: 'medium',
        effort: 'low',
        impact: 'medium',
        automation: 'partial'
      },
      'improve_naming': {
        priority: 'low',
        effort: 'low',
        impact: 'low',
        automation: 'manual'
      }
    };
  }

  async planRefactoring(codebase, constraints) {
    const refactoringPlan = {
      phases: [],
      totalEffort: 0,
      expectedBenefits: {},
      risks: []
    };

    // Identify refactoring opportunities
    const opportunities = await this.identifyOpportunities(codebase);

    // Prioritize based on impact and effort
    const prioritized = this.prioritizeOpportunities(opportunities, constraints);

    // Group into phases
    refactoringPlan.phases = this.groupIntoPhases(prioritized, constraints);

    // Calculate effort and benefits
    refactoringPlan.totalEffort = this.calculateTotalEffort(refactoringPlan.phases);
    refactoringPlan.expectedBenefits = this.calculateBenefits(refactoringPlan.phases);

    // Identify risks
```

```
    refactoringPlan.risks = this.identifyRisks(refactoringPlan.phases);

    return refactoringPlan;
  }

  async identifyOpportunities(codebase) {
    const opportunities = [];

    // Find code duplication
    const duplication = await this.findDuplication(codebase);
    for (const dup of duplication) {
      opportunities.push({
        type: 'eliminate_duplication',
        location: dup.locations,
        effort: this.estimateEffort(dup),
        impact: this.estimateImpact(dup),
        description: `Eliminate ${dup.lines} lines of duplicated code`
      });
    }

    // Find complex methods
    const complexMethods = await this.findComplexMethods(codebase);
    for (const method of complexMethods) {
      opportunities.push({
        type: 'extract_method',
        location: method.location,
        effort: this.estimateEffort(method),
        impact: this.estimateImpact(method),
        description: `Extract methods from ${method.name} (complexity: ${meth-
od.complexity})`
      });
    }

    // Find large classes
    const largeClasses = await this.findLargeClasses(codebase);
    for (const cls of largeClasses) {
      opportunities.push({
        type: 'extract_class',
        location: cls.location,
        effort: this.estimateEffort(cls),
        impact: this.estimateImpact(cls),
        description: `Extract classes from ${cls.name} (${cls.lines} lines)`
      });
    }

    return opportunities;
  }

  prioritizeOpportunities(opportunities, constraints) {
    return opportunities
      .map(opp => ({
        ...opp,
        score: this.calculatePriorityScore(opp, constraints)
      }))
      .sort((a, b) => b.score - a.score);
  }

  calculatePriorityScore(opportunity, constraints) {
    const typeConfig = this.refactoringTypes[opportunity.type];
```

```
    let score = 0;

    // Impact weight (40%)
    score += opportunity.impact * 0.4;

    // Effort weight (30% - inverse, lower effort = higher score)
    score += (1 - opportunity.effort) * 0.3;

    // Type priority weight (20%)
    const priorityMap = { high: 1, medium: 0.6, low: 0.3 };
    score += priorityMap[typeConfig.priority] * 0.2;

    // Constraint adjustments (10%)
    if (constraints.timeConstraint === 'tight' && typeConfig.effort === 'low') {
      score += 0.1;
    }
    if (constraints.riskTolerance === 'low' && typeConfig.automation === 'partial') {
      score += 0.05;
    }

    return score;
  }
}
```

## Automated Refactoring Tools

```
class AutomatedRefactoring {
  constructor() {
    this.tools = {
      'jscodeshift': new JSCodeshiftTool(),
      'eslint': new ESLintTool(),
      'prettier': new PrettierTool(),
      'custom': new CustomRefactoringTool()
    };
  }

  async executeRefactoring(refactoringPlan) {
    const results = {
      completed: [],
      failed: [],
      skipped: []
    };

    for (const phase of refactoringPlan.phases) {
      console.log(`Starting refactoring phase: ${phase.name}`);

      for (const task of phase.tasks) {
        try {
          const result = await this.executeTask(task);
          results.completed.push({ task, result });
        } catch (error) {
          console.error(`Refactoring task failed: ${task.description}`, error);
          results.failed.push({ task, error });
        }
      }
    }

    return results;
  }

  async executeTask(task) {
    const tool = this.selectTool(task);

    if (!tool) {
      throw new Error(`No suitable tool found for task: ${task.type}`);
    }

    // Create backup
    await this.createBackup(task.files);

    try {
      // Execute refactoring
      const result = await tool.execute(task);

      // Validate result
      await this.validateRefactoring(task, result);

      // Run tests
      await this.runTests(task.affectedTests);

      return result;
    } catch (error) {
      // Restore backup on failure
      await this.restoreBackup(task.files);
```

```
      throw error;
    }
  }

  selectTool(task) {
    const toolMap = {
      'eliminate_duplication': 'jscodeshift',
      'extract_method': 'jscodeshift',
      'simplify_conditionals': 'jscodeshift',
      'improve_formatting': 'prettier',
      'fix_lint_issues': 'eslint'
    };

    const toolName = toolMap[task.type];
    return toolName ? this.tools[toolName] : null;
  }
}

class JSCodeshiftTool {
  async execute(task) {
    const transform = this.getTransform(task.type);
    const options = this.buildOptions(task);

    const result = await this.runJSCodeshift(transform, task.files, options);

    return {
      tool: 'jscodeshift',
      filesModified: result.filesModified,
      transformations: result.transformations,
      stats: result.stats
    };
  }

  getTransform(taskType) {
    const transforms = {
      'eliminate_duplication': './transforms/eliminate-duplication.js',
      'extract_method': './transforms/extract-method.js',
      'simplify_conditionals': './transforms/simplify-conditionals.js'
    };

    return transforms[taskType];
  }

  async runJSCodeshift(transform, files, options) {
    const { execSync } = require('child_process');

    const command = [
      'npx jscodeshift',
      `-t ${transform}`,
      files.join(' '),
      Object.entries(options).map(([key, value]) => `--${key}=${value}`).join(' ')
    ].join(' ');

    const output = execSync(command, { encoding: 'utf8' });

    return this.parseJSCodeshiftOutput(output);
  }
}
```

## Debt Monitoring & Metrics

### Continuous Monitoring System

```
class TechnicalDebtMonitoring {
  constructor() {
    this.metrics = [
      'code_complexity',
      'test_coverage',
      'code_duplication',
      'dependency_freshness',
      'security_vulnerabilities',
      'performance_metrics'
    ];

    this.alertThresholds = {
      complexity_increase: 0.1, // 10% increase
      coverage_decrease: 0.05, // 5% decrease
      duplication_increase: 0.02, // 2% increase
      new_vulnerabilities: 1, // Any new vulnerability
      performance_degradation: 0.2 // 20% degradation
    };
  }

  async collectMetrics() {
    const metrics = {
      timestamp: new Date(),
      codebase: {},
      trends: {},
      alerts: []
    };

    // Collect current metrics
    for (const metric of this.metrics) {
      metrics.codebase[metric] = await this.collectMetric(metric);
    }

    // Calculate trends
    metrics.trends = await this.calculateTrends(metrics.codebase);

    // Check for alerts
    metrics.alerts = await this.checkAlerts(metrics.codebase, metrics.trends);

    // Store metrics
    await this.storeMetrics(metrics);

    return metrics;
  }

  async collectMetric(metricName) {
    switch (metricName) {
      case 'code_complexity':
        return await this.measureComplexity();
      case 'test_coverage':
        return await this.measureCoverage();
      case 'code_duplication':
        return await this.measureDuplication();
      case 'dependency_freshness':
        return await this.measureDependencyFreshness();
      case 'security_vulnerabilities':
        return await this.measureVulnerabilities();
      case 'performance_metrics':
```

```
        return await this.measurePerformance();
      default:
        throw new Error(`Unknown metric: ${metricName}`);
    }
  }

  async calculateTrends(currentMetrics) {
    const historicalData = await this.getHistoricalMetrics(30); // Last 30 days
    const trends = {};

    for (const [metric, value] of Object.entries(currentMetrics)) {
      const historical = historicalData.map(d => d.codebase[metric]).filter(v => v !==
undefined);

      if (historical.length > 0) {
        const average = historical.reduce((sum, val) => sum + val, 0) / historic-
al.length;
        const change = ((value - average) / average) * 100;

        trends[metric] = {
          current: value,
          average,
          change,
          direction: change > 0 ? 'increasing' : change < 0 ? 'decreasing' : 'stable'
        };
      }
    }

    return trends;
  }

  async checkAlerts(metrics, trends) {
    const alerts = [];

    // Check complexity increase
    if (trends.code_complexity?.change > this.alertThresholds.complexity_increase *
100) {
      alerts.push({
        type: 'complexity_increase',
        severity: 'warning',
        message: `Code complexity increased by $
{trends.code_complexity.change.toFixed(1)}%`,
        metric: 'code_complexity',
        threshold: this.alertThresholds.complexity_increase * 100,
        actual: trends.code_complexity.change
      });
    }

    // Check coverage decrease
    if (trends.test_coverage?.change < -this.alertThresholds.coverage_decrease * 100) {
      alerts.push({
        type: 'coverage_decrease',
        severity: 'error',
        message: `Test coverage decreased by ${Math.abs(trends.test_coverage.change).to
Fixed(1)}%`,
        metric: 'test_coverage',
        threshold: -this.alertThresholds.coverage_decrease * 100,
        actual: trends.test_coverage.change
      });
```

```
    }

    // Check new vulnerabilities
    if (metrics.security_vulnerabilities > this.alertThresholds.new_vulnerabilities) {
      alerts.push({
        type: 'new_vulnerabilities',
        severity: 'critical',
        message: `${metrics.security_vulnerabilities} new security vulnerabilities de-
tected`,
        metric: 'security_vulnerabilities',
        threshold: this.alertThresholds.new_vulnerabilities,
        actual: metrics.security_vulnerabilities
      });
    }

    return alerts;
  }
}
```

**Debt Dashboard**

```
class TechnicalDebtDashboard {
  constructor() {
    this.widgets = [
      'debt_overview',
      'trend_charts',
      'alert_panel',
      'refactoring_progress',
      'team_metrics'
    ];
  }

  async generateDashboard() {
    const dashboard = {
      title: 'Technical Debt Dashboard',
      generatedAt: new Date(),
      widgets: {}
    };

    // Generate each widget
    for (const widget of this.widgets) {
      dashboard.widgets[widget] = await this.generateWidget(widget);
    }

    return dashboard;
  }

  async generateWidget(widgetType) {
    switch (widgetType) {
      case 'debt_overview':
        return await this.generateDebtOverview();
      case 'trend_charts':
        return await this.generateTrendCharts();
      case 'alert_panel':
        return await this.generateAlertPanel();
      case 'refactoring_progress':
        return await this.generateRefactoringProgress();
      case 'team_metrics':
        return await this.generateTeamMetrics();
      default:
        return { error: `Unknown widget: ${widgetType}` };
    }
  }

  async generateDebtOverview() {
    const currentMetrics = await this.getCurrentMetrics();
    const debtScore = await this.calculateDebtScore(currentMetrics);

    return {
      type: 'debt_overview',
      title: 'Technical Debt Overview',
      data: {
        overallScore: debtScore.overall,
        categoryScores: debtScore.categories,
        totalIssues: debtScore.totalIssues,
        criticalIssues: debtScore.criticalIssues,
        estimatedEffort: debtScore.estimatedEffort,
        trend: debtScore.trend
      },
```

```
      visualization: {
        type: 'gauge',
        config: {
          min: 0,
          max: 100,
          value: debtScore.overall,
          thresholds: [
            { value: 30, color: 'red', label: 'Critical' },
            { value: 60, color: 'yellow', label: 'Warning' },
            { value: 100, color: 'green', label: 'Good' }
          ]
        }
      }
    };
  }

  async generateTrendCharts() {
    const historicalData = await this.getHistoricalMetrics(90); // Last 90 days

    return {
      type: 'trend_charts',
      title: 'Technical Debt Trends',
      data: {
        timeRange: '90 days',
        metrics: this.prepareTrendData(historicalData)
      },
      visualization: {
        type: 'line_chart',
        config: {
          xAxis: 'date',
          yAxes: [
            { metric: 'code_complexity', color: 'blue', label: 'Complexity' },
            { metric: 'test_coverage', color: 'green', label: 'Coverage %' },
            { metric: 'code_duplication', color: 'red', label: 'Duplication %' }
          ]
        }
      }
    };
  }
}
```

---

## Success Metrics

### Technical Debt KPIs

- **Overall Debt Score**: Maintain above 70/100
- **Code Complexity**: Keep average cyclomatic complexity below 7
- **Test Coverage**: Maintain above 85%
- **Code Duplication**: Keep below 5%
- **Security Vulnerabilities**: Zero critical, < 5 high severity
- **Dependency Freshness**: 90% of dependencies within 6 months

### Process Metrics

- **Debt Introduction Rate**: < 2% per sprint

- **Debt Resolution Rate**: > 5% per sprint
- **Refactoring Velocity**: 20+ story points per sprint
- **Quality Gate Pass Rate**: > 95%
- **Time to Fix Critical Issues**: < 24 hours

### Team Metrics

- **Developer Satisfaction**: > 4/5 with codebase quality
- **Onboarding Time**: < 2 weeks for new developers
- **Bug Fix Time**: < 4 hours average
- **Feature Development Velocity**: Maintain or improve

---

## Implementation Roadmap

### Phase 1: Assessment & Tooling (Months 1-2)

- [ ] Implement technical debt assessment framework
- [ ] Set up automated quality gates
- [ ] Deploy monitoring and alerting system
- [ ] Create technical debt dashboard

### Phase 2: Process Integration (Months 3-4)

- [ ] Integrate quality gates into CI/CD pipeline
- [ ] Establish refactoring workflow
- [ ] Train team on debt management practices
- [ ] Implement automated refactoring tools

### Phase 3: Systematic Reduction (Months 5-8)

- [ ] Execute high-priority refactoring initiatives
- [ ] Implement preventive measures
- [ ] Optimize development workflow
- [ ] Establish debt reduction targets

### Phase 4: Continuous Improvement (Months 9-12)

- [ ] Refine monitoring and alerting
- [ ] Optimize refactoring processes
- [ ] Expand automation capabilities
- [ ] Achieve target debt levels

This comprehensive technical debt management strategy ensures Audityzer maintains high code quality, performance, and maintainability while supporting rapid development and innovation.