Name	Student Number
Nicholas Dobson	u23671964
Shaylin Govender	u20498952
Lonwabo	u22516949
Mpho	u22668323
Aryan Mohanlall	u23565536

Traffic Guardian Testing Policy Doc

Team: Quantum Quenchers

quantumquenchers@gmail.com

COS301

Capstone Project

University of Pretoria

1. Introduction

This testing policy document outlines the comprehensive testing strategy, procedures, tools, and methodologies employed in the development of the Traffic Guardian system. The Traffic Guardian is an advanced incident detection and traffic management system designed to enhance road safety through real-time monitoring, AI-powered incident detection, and automated alert systems.

1.1 Purpose

The purpose of this testing policy document is to establish a systematic approach to testing that ensures the Traffic Guardian system meets all functional and non-functional requirements while maintaining high quality standards throughout the development lifecycle.

1.2 Scope

This policy covers all testing activities for the Traffic Guardian system across its complete architecture:

- Frontend React application with TypeScript
- Backend API services and databases
- AI/ML incident detection models (YOLO-based)
- Integration with external services
- Infrastructure and deployment systems

2. Test Plan Structure

Primary Objectives:

- Ensure the system satisfies all functional requirements as specified in the SRS
- Validate that non-functional requirements (performance, usability, scalability) are met
- Detect and eliminate defects before production deployment
- Establish confidence in system reliability and safety-critical operations
- Verify integration points with external systems function correctly

Quality Objectives:

- Achieve minimum 50% code coverage for all components
- Maintain zero critical security vulnerabilities
- Ensure 99.5% system availability as per quality requirements
- Validate response times under 200ms for API endpoints

2.2 Types of Testing

2.2.1 Functional Testing

- **Unit Testing**: Individual component and function validation
- **Integration Testing**: Component interaction and API integration testing
- **System Testing**: End-to-end functional validation
- **Acceptance Testing**: User story and use case validation
- **Regression Testing**: Ensuring existing functionality remains intact

2.2.2 Non-Functional Testing

- **Performance Testing**: Response time, throughput, and resource utilization
- **Scalability Testing**: System behavior under increasing load
- **Usability Testing**: User interface and user experience validation

2.4 Test Coverage Criteria

- **Requirements Coverage**: Each functional requirement must be tested
- **Code Coverage**: Minimum 50% statement and branch coverage
- **Use Case Coverage**: All implemented use cases must be tested
- **API Coverage**: All endpoint scenarios must be validated

3. Automated Testing Infrastructure

3.1 CI/CD Platform Selection

3.1.1 GitHub Actions - Primary Choice

Justification: GitHub Actions was selected over Travis CI for the following reasons:

- **Native Integration**: Seamless integration with GitHub repositories
- **Cost Effectiveness**: More generous free tier and better pricing model
- **Multi-Environment Support**: Native support for testing on Node 18.x and 20.x
- **Ecosystem**: Extensive marketplace with pre-built actions
- **Maintenance**: Lower configuration overhead compared to Travis CI
- **Performance**: Faster build times and better resource allocation

3.2 Testing Framework Selection

3.2.1 Jest - JavaScript Testing Framework

Justification:

- Industry standard for React applications
- Built-in mocking and assertion capabilities
- Snapshot testing for UI component validation
- Integrated code coverage reporting
- Excellent TypeScript support

3.2.2 Cypress - End-to-End Testing

Justification:

- Real browser testing environment
- Component testing capabilities
- Time-travel debugging features
- Visual regression testing support
- Reliable cross-browser testing

3.2.3 Python unittest - AI Model Testing

Justification:

- Native Python testing framework
- Excellent mocking capabilities for AI model dependencies
- Integration with coverage tools
- Support for complex test fixtures

4. Functional Testing Procedures

4.1 Unit Testing

4.1.1 Frontend Component Testing

Location: `frontend/src/components/*.test.tsx`

Test Categories:

- **Component Rendering**: Verify components render correctly with props
- **Event Handling**: Test user interactions and event callbacks
- **State Management**: Validate state changes and effects
- **Props Validation**: Test component behavior with various prop combinations

Coverage Requirements:

- Minimum 50% line coverage for component files
- All public methods must be tested
- Critical business logic requires 80% coverage

4.1.2 API Service Testing

Location: `frontend/src/services/*.test.ts`

Test Categories:

- **Request Formation**: Verify correct API request structure
- **Response Handling**: Test response parsing and error handling
- **Authentication**: Validate token management and refresh logic
- **Error Scenarios**: Test network failures and API error responses

4.1.3 AI Model Unit Testing

Location: `AI_Model_BB/Testing/test_*.py`

Test Categories:

- **Model Initialization**: Test YOLO model loading and configuration
- **Detection Algorithms**: Test collision detection logic
- **Data Processing**: Test input preprocessing and output formatting
- **Performance Validation**: Test processing speed requirements

4.2 Integration Testing

4.2.1 API Integration Testing

Objective: Validate interactions between frontend and backend services

Test Scenarios:

- Authentication flow integration
- Incident data CRUD operations
- Real-time alert system integration
- External API connectivity (CalTrans, PEMS)

4.2.2 Database Integration Testing

Objective: Validate data persistence and retrieval operations

Test Scenarios:

- Data insertion and validation
- Query performance and accuracy
- Transaction handling and rollback scenarios
- Concurrent access handling

4.3 System Testing

4.3.1 End-to-End User Journey Testing

Location: `frontend/cypress/e2e/`

Primary Test Flows:

1. **User Authentication Journey**
    - User registration and email verification
    - Login with valid/invalid credentials
    - Password reset functionality
    - Session management and logout
2. **Incident Management Journey**
    - Incident creation and classification
    - Real-time incident updates
    - Incident resolution workflow
    - Historical incident analysis
3. **Dashboard Functionality Journey**
    - Real-time data visualization
    - Filter and search operations
    - Export functionality
    - Mobile responsiveness

5. Non-Functional Testing Procedures

5.1 Performance Testing

5.1.1 Load Testing

Objective: Validate system performance under expected load conditions

Tools: Apache JMeter for backend API testing, Lighthouse CI for frontend performance

Test Scenarios:

- **Concurrent User Testing**: Simulate 100 concurrent users (per requirement Q5 Performance)
- **API Response Time Testing**: Ensure sub-200ms response times
- **Database Query Performance**: Validate query execution times under load
- **AI Model Processing Speed**: Ensure real-time incident detection (< 1 second)

Performance Metrics:

- Response time percentiles (50th, 90th, 95th, 99th)
- Throughput (requests per second)
- Resource utilization (CPU, memory, disk I/O)
- Error rate under load conditions

Acceptance Criteria:

- API endpoints must respond within 200ms for 95% of requests
- System must support 100 concurrent users without degradation
- AI model must process video frames in real-time (< 1 second per frame)

5.1.2 Stress Testing

Objective: Determine system breaking points and recovery behavior

Test Scenarios:

- Gradual load increase until system failure
- Sudden load spikes simulation
- Extended duration testing (24-hour runs)
- Resource exhaustion scenarios

Recovery Testing: Validate system behavior after stress conditions are removed

5.2 Scalability Testing

5.2.1 Horizontal Scaling Testing

Objective: Validate system behavior as infrastructure scales

Manual Testing Approach: Due to infrastructure complexity, scalability testing is performed manually using:

- **Load Balancer Testing**: Distribute traffic across multiple instances
- **Database Scaling**: Test read replicas and connection pooling
- **Microservice Scaling**: Independent scaling of individual services

Test Scenarios:

- Scale from 1 to 3 application instances
- Test database performance with increased connection pools
- Validate load distribution effectiveness

Metrics Measured:

- Response time consistency across scaled instances
- Database connection management
- Session persistence across instances
- Resource utilization distribution

5.3 Usability Testing

5.3.1 User Interface Testing

Objective: Validate user experience and interface design

Manual Testing Approach: Usability testing requires human judgment and cannot be fully automated

Test Categories:

- **Navigation Testing**: Verify intuitive navigation paths
- **Accessibility Testing**: compliance validation
- **Mobile Responsiveness**: Cross-device compatibility testing
- **User Workflow Testing**: Task completion efficiency

Test Procedures:

1. **Heuristic Evaluation**: Expert review using Nielsen's usability principles
2. **User Task Testing**: Observe users completing common tasks
3. **Accessibility Audit**: Screen reader and keyboard navigation testing
4. **Cross-Browser Testing**: Functionality across Chrome, Firefox, Safari, Edge

Usability Metrics:

- Task completion rate
- Time to complete common tasks
- Error recovery success rate
- User satisfaction scores

5.3.2 Documentation and Help Facility Testing

Objective: Ensure help systems and documentation are effective

Test Scenarios:

- Help system accuracy and completeness
- Documentation clarity and organization
- Error message appropriateness
- User guidance effectiveness

6. Test Reports and Documentation

6.1 Automated Test Reports

Coverage reports are automatically generated in `/coverage/` directories after every test execution, providing detailed HTML reports with line-by-line coverage analysis that enables developers to identify untested code sections quickly. These reports are generated automatically with every test run and distributed through CI/CD artifacts and GitHub Pages for easy access by all team members.

Test results reports are generated in JUnit XML format for seamless CI/CD integration and HTML format for human consumption. The reports include comprehensive test execution summaries, detailed pass/fail status for each test case, execution time analysis for performance monitoring, and historical trend analysis for quality tracking. Reports are archived in GitHub Actions artifacts to enable historical analysis and trend identification.

6.2 Manual Test Reports

Performance test reports provide comprehensive analysis of load test results, response time distribution analysis, resource utilization patterns, and capacity planning recommendations. These reports are generated weekly during active development periods and formatted as detailed PDF reports with charts and quantitative analysis. Reports are stored in the project documentation repository for easy access and historical reference.

Usability test reports include detailed analysis of task completion rates across different user scenarios, structured user feedback collection and analysis, accessibility compliance validation results, and actionable recommendations for user experience improvements. Reports are generated after each usability testing session and formatted as structured documents with clear recommendations. These reports are distributed to the development team and stakeholders to ensure user experience improvements are prioritized and implemented.

7. Quality Assurance and Continuous Improvement

7.1 Test Quality Metrics

Automated test metrics provide quantitative measures of testing effectiveness and system quality. Test coverage maintains minimum 50% line coverage across all components with a target of 70% for comprehensive validation. Critical business logic requires 80% coverage to ensure thorough validation of essential system functionality. Test execution time is limited to a maximum of 30 minutes for the complete test suite to maintain rapid feedback cycles. Test reliability maintains less than 1% flaky test rate to ensure consistent and dependable test results. Defect detection rate tracking compares bugs found through testing versus those discovered in production to measure testing effectiveness.

Manual test metrics focus on the effectiveness of human-driven testing activities. Test case execution rate measures the percentage of planned manual tests that are completed within scheduled timeframes. Defect discovery efficiency tracks critical bugs found before release compared to total critical issues to measure testing effectiveness. User satisfaction metrics collect feedback scores from usability testing sessions to ensure user experience quality. Performance compliance measures the percentage of performance requirements that are consistently met during testing cycles.

7.2 Continuous Improvement Process

Test process review occurs monthly to evaluate testing effectiveness and identify improvement opportunities. These reviews analyze test metrics trends, identify testing bottlenecks, and develop enhancement strategies. Retrospective sessions following major releases provide structured evaluation of testing processes and outcomes, enabling iterative improvement of testing procedures.

Tool evaluation includes regular assessment of testing tools and techniques to ensure the testing infrastructure remains current with industry best practices. Best practice updates incorporate evolving industry testing standards and methodologies into our testing procedures, ensuring continuous improvement of testing quality and effectiveness.

Feedback integration ensures that testing procedures evolve based on stakeholder input and real-world usage patterns. Developer feedback drives continuous improvement of test authoring experience and testing tool effectiveness. Stakeholder input helps align testing priorities with business objectives and user requirements. User feedback from production usage informs improvements to test scenarios and validation procedures. Performance data from production systems provides insights for enhancing test scenarios and acceptance criteria.

8. Conclusion

This testing policy establishes a comprehensive framework for ensuring the quality, reliability, and performance of the Traffic Guardian system through systematic application of both automated and manual testing approaches. The policy balances the efficiency of automated testing with the nuanced evaluation capabilities of manual testing to achieve thorough validation of all system aspects.

8.1 Key Success Factors

Comprehensive coverage ensures that both functional and non-functional requirements receive thorough validation through appropriate testing methodologies. The automation-first approach provides rapid feedback and continuous validation while enabling sustainable testing practices throughout the development lifecycle. Risk-based testing priorities align testing efforts with system criticality and potential user impact to maximize testing effectiveness. Continuous improvement processes ensure that testing procedures evolve with changing requirements and industry best practices.

8.2 Expected Outcomes

Following this testing policy will ensure high-quality software delivery with minimal production defects through comprehensive validation procedures. Confident deployment of safety-critical incident detection features is enabled through thorough testing of AI model accuracy and performance. Scalable and performant system delivery meets user expectations through systematic performance and scalability validation. Maintainable codebase quality is assured through comprehensive test coverage and continuous quality monitoring.

The systematic approach outlined in this policy provides a solid foundation for delivering a reliable, performant, and user-friendly Traffic Guardian system that meets all specified requirements while maintaining the flexibility to adapt to changing needs and technological advances.