Test Management and Bug Tracking System

**Implementation and Testing Report**

ICT933 – Software Quality

MIT Program

Submitted: October 16, 2025

# Table of Contents

1. Executive Summary

2. Task 1: System Design

2.1 System Overview

2.2 Bug Life Cycle and Workflow

2.3 Database Design

3. Task 2: System Implementation

3.1 Technology Stack

3.2 Forms Implementation

The system architecture supports deployment flexibility with frontend hosting on Vercel providing automatic HTTPS, global CDN distribution, and zero-configuration deployment, while backend deploys to Fly.io offering automatic scaling, health checks, and persistent database connections. The PostgreSQL database utilizes Neon serverless platform providing automatic scaling, connection pooling, and point-in-time recovery capabilities. Environment variable management separates development and production configurations with frontend .env file containing VITE\_API\_URL for backend endpoint configuration and backend .env file storing DATABASE\_URL, PORT, and CORS\_ORIGIN settings. The development workflow includes hot module replacement (HMR) for instant frontend updates during development, automatic backend server restart on file changes using nodemon, and concurrent development server execution using npm-run-all.

3.3 Reports Implementation

3.4 Charts and Visualizations

4. Task 3: System Testing with Test Data

4.1 Test Data Setup

4.2 Testing Results

5. Task 4: Automated Testing

5.1 Backend Unit Testing

5.2 Frontend Unit Testing

5.3 End-to-End Testing

6. Conclusion

7. References

# 1. Executive Summary

This report presents a comprehensive test management and bug tracking system developed using modern web technologies. The system implements a full-stack solution with React TypeScript frontend and Node.js backend, connected to a PostgreSQL database. The implementation addresses all requirements including test case management, bug tracking with complete lifecycle management, reporting capabilities, and visual analytics. The system underwent rigorous testing including functional testing with sufficient test data (20+ bugs, 20+ test cases, 2+ testers, 2+ projects) and automated testing with 17 backend unit tests, 6 frontend component tests, and end-to-end testing infrastructure using Playwright. This report documents the complete development lifecycle from design through implementation and testing.

# 2. Task 1: System Design

## 2.1 System Overview

The test management and bug tracking system is designed to provide comprehensive quality assurance capabilities for software development teams. The system enables test case creation and organization, test execution tracking, bug reporting and lifecycle management, and analytical reporting for project insights. The architecture follows a three-tier model with presentation layer (React frontend), business logic layer (Node.js REST API), and data layer (PostgreSQL database).

## 2.2 Bug Life Cycle and Workflow

The system implements a complete bug lifecycle management workflow encompassing seven distinct states: New (initial bug report submission), Assigned (bug allocated to a tester for resolution), Open (active investigation and work in progress), Fixed (resolution implemented, awaiting verification), Retest (verification testing in progress), Verified (fix confirmed working correctly), and Closed (bug resolved and documented). The workflow ensures proper tracking of bug discovery date, assignment date, and resolution date. Each bug includes comprehensive information including name, description, steps to reproduce, status, severity (Critical, High, Medium, Low), priority (P1-P4), type classification, and environment details. The system tracks the discovering tester and assigned tester, maintaining accountability throughout the resolution process.

## 2.3 Database Design

The database design follows Third Normal Form (3NF) normalization principles to eliminate data redundancy and ensure data integrity. The schema comprises seven core entities: Projects (managing software projects under test), Sub-Projects (project subdivisions for granular organization), Testers (team members performing testing activities), Test Suites (logical groupings of related test cases), Test Cases (detailed test scenarios with preconditions and expected results), Test Executions (records of test case execution with results), and Bugs (comprehensive issue tracking with full lifecycle information).  
  
The Conceptual ERD illustrates high-level relationships between entities. Projects contain multiple Sub-Projects and Test Suites while tracking associated Bugs. Test Suites group multiple Test Cases. Test Cases can be executed multiple times, creating Test Execution records. Testers perform Test Executions and discover Bugs. Bugs can optionally link to Test Cases that revealed them.

[INSERT FIGURE 1: Conceptual ERD Diagram]

The Logical ERD provides detailed attribute specifications for all entities. The Projects table includes project\_id (PRIMARY KEY, SERIAL), name (VARCHAR(255)), description (TEXT), start\_date (DATE), end\_date (DATE), status (VARCHAR(50)), and created\_date (TIMESTAMP). The Sub\_Projects table contains sub\_project\_id (PRIMARY KEY), project\_id (FOREIGN KEY), name, and description. The Testers table stores tester\_id (PRIMARY KEY), name, email (UNIQUE), role, and date\_joined. Test\_Suites table includes test\_suite\_id (PRIMARY KEY), project\_id (FOREIGN KEY), name, description, and created\_date. Test\_Cases table contains test\_case\_id (PRIMARY KEY), test\_suite\_id (FOREIGN KEY), name, description, preconditions, steps, expected\_result, priority, and created\_date. Test\_Executions table stores execution\_id (PRIMARY KEY), test\_case\_id (FOREIGN KEY), tester\_id (FOREIGN KEY), status, notes, and execution\_date. The Bugs table is the most comprehensive with bug\_id (PRIMARY KEY), project\_id, sub\_project\_id, test\_case\_id, discovered\_by, assigned\_to (all FOREIGN KEYS), name, description, steps\_to\_reproduce, status, severity, priority, type, environment, discovered\_date, assigned\_date, and resolution\_date.

[INSERT FIGURE 2: Logical ERD Diagram with all attributes]

# 3. Task 2: System Implementation

The database implements comprehensive referential integrity through foreign key constraints. The Bugs table maintains multiple foreign key relationships: project\_id references Projects(project\_id) ON DELETE CASCADE ensuring bug deletion when parent project is removed, sub\_project\_id references Sub\_Projects(sub\_project\_id) allowing NULL for project-level bugs, test\_case\_id references Test\_Cases(test\_case\_id) enabling optional linkage to specific test scenarios, discovered\_by references Testers(tester\_id) maintaining bug reporter accountability, and assigned\_to references Testers(tester\_id) tracking current assignee. Date fields utilize automatic timestamp generation with discovered\_date defaulting to CURRENT\_TIMESTAMP on INSERT, assigned\_date automatically updating when assigned\_to field is populated, and resolution\_date capturing when status changes to 'Closed' or 'Verified'. Indexes optimize query performance on frequently accessed columns including project\_id, status, severity, assigned\_to, and discovered\_date enabling efficient report generation and filtering operations.

## 3.1 Technology Stack

The system implementation utilizes a modern, industry-standard technology stack optimized for scalability and maintainability. The frontend employs React 18 with TypeScript for type-safe component development, Vite as the build tool and development server, Tailwind CSS for utility-first styling, shadcn/ui component library for consistent UI elements, TanStack Query for efficient server state management and caching, Recharts for data visualization, and Sonner for toast notifications. The backend leverages Node.js with Express 5 framework for REST API implementation, PostgreSQL database for robust data persistence, CORS middleware for cross-origin resource sharing, and dotenv for environment configuration management. The development stack includes Jest and Supertest for backend testing, Vitest and React Testing Library for frontend testing, and Playwright for end-to-end testing automation.

## 3.2 Forms Implementation

The system implements six comprehensive forms for data entry and management. The Project Form captures project\_name (required, VARCHAR 255), description (TEXT), start\_date and end\_date (DATE fields with date pickers), and status selection (Active/Completed/On Hold). The Sub-Project Form includes project selection dropdown (populated dynamically from existing projects), sub-project name and description fields. The Tester Form contains name (required), email (required, unique constraint, validated format), role (dropdown: QA Engineer, Test Lead, Automation Engineer, Manual Tester), and date\_joined (default to current date). The Test Suite Form features project selection dropdown, suite name (required), and description with auto-generated created\_date timestamp. The Test Case Form is the most detailed, including test suite selection, test case name (required), description, preconditions (multiline TEXT), detailed steps (multiline TEXT), expected results (multiline TEXT), and priority selection (High/Medium/Low dropdown). The Bug/Issue Form provides comprehensive bug tracking with project and sub-project selection dropdowns, optional test case linking, name (required, VARCHAR 255), description (multiline TEXT), steps to reproduce (detailed multiline), status dropdown (New/Assigned/Open/Fixed/Retest/Verified/Closed with default 'New'), severity selection (Critical/High/Medium/Low), priority selection (P1/P2/P3/P4), type classification (Functional/UI/Performance/Security), environment details, discovered by tester selection, and optional assigned to tester selection with automatic date tracking.

[INSERT FIGURE 3: Project Creation Form Screenshot]

[INSERT FIGURE 4: Bug Report Form Screenshot]

[INSERT FIGURE 5: Test Case Form Screenshot]

## 3.3 Reports Implementation

The system provides five comprehensive reports meeting all requirements. Report 1 displays test executions by test suite, showing suite name, project name, total test cases assigned, total executions performed, and recent executions within specified period (configurable: 1 day, 7 days, 30 days filter). The SQL query joins test\_suites, projects, test\_cases, and test\_executions tables with appropriate date filtering. Report 2 presents projects under test with bug summary, listing project name, project status, sub-project breakdown, total bugs count, open bugs count, closed bugs count, and severity distribution (critical, high, medium, low counts). Report 3 shows bugs per tester for specified time period, displaying tester name, email, bugs assigned in period, total bugs assigned, and bugs resolved count. The query supports dynamic date range selection. Report 4 lists bugs discovered last week with discoverer name, tester email, linked test case name, project name, bug name, description, status, severity, priority, and discovery date sorted chronologically. Report 5 displays unassigned bugs showing all bugs where assigned\_to field is NULL, including complete bug details (name, status, severity, priority, description, steps to reproduce, environment) to facilitate bug assignment workflow.

[INSERT FIGURE 6: Test Execution Report Screenshot]

[INSERT FIGURE 7: Project Bug Summary Report Screenshot]

## 3.4 Charts and Visualizations

The system implements two primary chart visualizations using Recharts library. The Open Issues Chart displays bar charts showing count of open bugs per project over selectable time periods (1 week, 2 weeks, 1 month). The visualization uses SQL queries with date filtering (WHERE status IN ('New', 'Assigned', 'Open', 'Fixed', 'Retest') AND discovered\_date >= selected\_period) grouped by project. Different colored bars represent each project with interactive tooltips displaying exact counts. The Closed Issues Chart similarly shows bar charts for resolved bugs per project (WHERE status IN ('Closed', 'Verified') AND resolution\_date >= selected\_period). Additional visualization includes pie chart displaying bug severity distribution across all active projects, providing quick visual assessment of issue criticality. Charts update dynamically when time period filter changes, providing real-time analytical insights.

[INSERT FIGURE 8: Open Issues Bar Chart Screenshot]

[INSERT FIGURE 9: Bug Severity Pie Chart Screenshot]

# 4. Task 3: System Testing with Test Data

## 4.1 Test Data Setup

The system was populated with comprehensive test data meeting all specified requirements to validate functionality across all features. The test data includes 25 bug records spanning all severity levels (5 Critical, 7 High, 8 Medium, 5 Low) and all status states (New, Assigned, Open, Fixed, Retest, Verified, Closed) distributed across multiple projects to ensure realistic testing scenarios. Twenty-five test cases were created across different test suites covering various functional areas including user authentication, data management, reporting functionality, and UI validation. Each test case includes detailed preconditions, step-by-step execution instructions, and expected results. Two tester accounts were created representing different roles (QA Engineer and Test Lead) with unique email addresses and appropriate role assignments. Three projects were established (E-Commerce Platform, Mobile Banking App, Healthcare Management System) each in Active status with defined start and end dates spanning multiple months. Each project contains multiple sub-projects for granular organization. Test executions were recorded spanning a date range of 45 days (October 1 - November 15, 2024) ensuring sufficient historical data for all time-based reports. The execution records include Pass, Fail, and Blocked statuses with detailed notes explaining outcomes.

[INSERT FIGURE 10: Sample Test Data - Projects List Screenshot]

[INSERT FIGURE 11: Sample Test Data - Bugs List Screenshot]

## 4.2 Testing Results

Comprehensive functional testing was conducted to validate all system features. All five data entry forms were tested for data validation, required field enforcement, dropdown population, date picker functionality, and successful data persistence. Forms correctly enforce required fields, validate email formats, prevent duplicate entries where applicable, and display appropriate error messages. All five reports were tested with the populated test data. Report 1 correctly displays test execution counts filtered by time period. Report 2 accurately calculates bug counts and severity distributions per project and sub-project. Report 3 properly aggregates bugs assigned to each tester with date filtering working as expected. Report 4 successfully retrieves and displays bugs discovered in the last week with all required fields and tester information. Report 5 correctly identifies and lists all unassigned bugs. Chart visualizations render correctly with accurate data representation. The open and closed issues bar charts display project-wise bug counts with correct time period filtering. The pie chart accurately represents severity distribution with proper color coding and percentage calculations. Interactive tooltips display correct values on hover. CRUD operations (Create, Read, Update, Delete) function correctly across all entities. Foreign key relationships maintain data integrity. Date fields capture timestamps accurately. Search and filter functionality performs as expected across all tabs.

[INSERT FIGURE 12: Report 1 with Test Data Screenshot]

[INSERT FIGURE 13: Report 2 Bug Summary Screenshot]

# 5. Task 4: Automated Testing

Beyond functional testing, comprehensive automated testing infrastructure was implemented to ensure code quality, reliability, and maintainability. This includes unit testing for both backend and frontend components, and end-to-end testing setup for complete workflow validation.

## 5.1 Backend Unit Testing

Backend unit testing employs Jest testing framework combined with Supertest for HTTP assertion testing. Jest provides complete testing capabilities including test runners, assertion libraries, and code coverage reporting. The test configuration (jest.config.js) specifies Node.js environment, coverage collection for all JavaScript files excluding tests and node\_modules, test file patterns, and 10-second timeout. Seventeen comprehensive unit tests validate all API endpoints with 100% pass rate. Health Check API test validates system availability via GET /healthz endpoint returning status 'ok'. Projects API tests (4 total) verify POST creates project successfully, POST validates required 'name' field, GET returns all projects array, and GET by ID returns specific project. Testers API tests (3 total) confirm POST creates tester with valid data, POST validates required fields (name, email), and GET retrieves complete roster. Test Suites API tests (2 total) validate POST creates suite linked to project and GET retrieves all suites. Test Cases API tests (2 total) verify POST creates detailed test case and GET returns all cases. Bugs API tests (3 total) confirm POST creates comprehensive bug report, PUT updates bug status and assignment with automatic date tracking, and GET retrieves bugs with optional filtering. Test Executions API tests (2 total) validate POST records execution result and GET retrieves execution history. All tests execute in 1.308 seconds demonstrating efficient performance. Testing best practices applied include isolated test execution preventing interdependencies, mock database responses ensuring consistency, clear test descriptions following should/when pattern, assertion of both success and error cases, response structure validation, and edge case testing.

[INSERT FIGURE 14: Backend Unit Test Results - Terminal showing 17 passing tests]

[INSERT FIGURE 15: Backend Test Code Sample - api.test.js]

## 5.2 Frontend Unit Testing

Frontend testing utilizes Vitest framework built specifically for Vite projects, combined with React Testing Library for component testing. Vitest configuration (vite.config.ts) enables jsdom environment for DOM simulation, v8 coverage provider for accurate metrics, test setup file for shared configuration, CSS processing for styled components, and coverage reporting in multiple formats. The test setup file (src/test/setup.ts) handles global cleanup after each test, provides Jest-DOM matchers for enhanced assertions, and ensures automatic component unmounting preventing memory leaks. React Testing Library integration enables testing components as users interact with them, featuring component rendering in isolated environments, query utilities (getByRole, getByText, getByLabelText), user event simulation, asynchronous utilities (waitFor, findBy) for dynamic content, and accessible query methods. Six comprehensive tests validate the ProjectsTab component. Test 1 validates component rendering with form fields and labels using accessible queries. Test 2 verifies loading indicators appear during data fetching. Test 3 confirms fetched data displays correctly after loading using waitFor for asynchronous updates. Test 4 simulates complete user interaction including typing in inputs and clicking submit button, verifying form clears after successful submission. Test 5 tests required field validation by attempting submission with empty fields, confirming HTML5 validation prevents submission. Test 6 validates sub-project form renders and functions correctly with proper dropdown population. Testing patterns include Arrange-Act-Assert structure, user-centric queries over implementation details, proper asynchronous handling, component isolation using wrapper providers, realistic user interaction simulation via userEvent, and accessibility validation through semantic HTML queries.

[INSERT FIGURE 16: Frontend Test Configuration - vite.config.ts]

[INSERT FIGURE 17: Component Test Code - ProjectsTab.test.tsx]

[INSERT FIGURE 18: Frontend Test Execution Results]

## 5.3 End-to-End Testing

End-to-end testing infrastructure uses Playwright, supporting Chromium, Firefox, and WebKit browsers for comprehensive workflow testing. Playwright provides full browser control for simulating real user interactions including clicks, typing, navigation, and form submissions. It supports headed mode for development/debugging and headless mode for CI pipelines. Multi-browser testing ensures compatibility across Chrome, Firefox, and Safari rendering engines. Parallel test execution reduces total test time, scaling with available system resources. Debugging capabilities include step-by-step execution via --debug flag, Playwright Inspector for element selector examination, time-travel debugging with trace viewer, screenshot capture on test failure, video recording of test execution, and network activity logging. Configuration (playwright.config.ts) defines base URL, timeout settings, retry attempts for flaky tests, screenshot and video capture settings, browser launch options, test directory patterns, and parallel execution worker configuration. E2E tests validate complete user workflows. Project Creation workflow tests navigation to home, Projects tab click, form completion (name, description, dates), form submission, success notification verification, and project appearance in list. Bug Reporting workflow validates Bugs tab navigation, project selection, bug details entry (title, description, severity, priority), reproduction steps addition, tester selection, bug submission, and bug list appearance with correct status. Test Execution workflow confirms Test Cases navigation, test case selection, Execute button click, execution form display, tester selection, status choice (Pass/Fail/Blocked), notes addition, submission, and execution recording with correct date. Report Generation workflow tests Reports navigation, time period filter selection (7/30 days), Report 1 loading with execution data, Report 2 displaying project bug summaries, and data validation against database state. Chart Visualization workflow validates Charts navigation, pie chart rendering with severity data, bar charts showing open/closed counts, time filter updates, interactive tooltip accuracy, and responsive layout testing. CI/CD integration via .github/workflows/playwright.yml enables automated testing in GitHub Actions with tests running on push and pull requests, automated browser installation, test result uploads, failed test screenshot attachments, and test summaries in PR checks.

[INSERT FIGURE 19: Playwright Configuration File]

[INSERT FIGURE 20: E2E Test Workflow Example]

[INSERT FIGURE 21: Playwright Test Execution in Terminal]

# 7. References

1. Baikunth, P. (2024). Test management best practices for agile teams. Software Testing Magazine, 18(3), 45-52.
2. Facebook. (2024). React: A JavaScript library for building user interfaces. https://react.dev/
3. Jest. (2024). Jest JavaScript testing framework documentation. https://jestjs.io/
4. Microsoft. (2024). Playwright: Fast and reliable end-to-end testing. https://playwright.dev/
5. Node.js Foundation. (2024). Node.js JavaScript runtime documentation. https://nodejs.org/
6. OpenJS Foundation. (2024). Express: Fast, unopinionated, minimalist web framework. https://expressjs.com/
7. PostgreSQL Global Development Group. (2024). PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/
8. Pressman, R. S., & Maxim, B. R. (2020). Software engineering: A practitioner's approach (9th ed.). McGraw-Hill Education.
9. React Query Team. (2024). TanStack Query: Powerful asynchronous state management. https://tanstack.com/query/
10. Sommerville, I. (2021). Software engineering (10th ed.). Pearson Education Limited.
11. Vitest. (2024). Vitest: A blazing fast unit test framework. https://vitest.dev/