COQ MONITOOL: A Web-Based Software Quality Cost Monitoring Platform

Bin Ma
Dept. of Software Engineering, College of Arts, Media, and Technology
Chiang Mai University, Chiang Mai, Thailand
652115547@cmu.ac.th

Kittitouch Suteeca
Dept. of Software Engineering, College of Arts, Media, and Technology
Chiang Mai University, Chiang Mai, Thailand
kittitouch.s@cmu.ac.th

***Abstract*—Software quality cost management is a critical challenge in modern development teams, where quality-related data is typically fragmented across multiple tools such as project management systems, defect trackers, and time-logging sheets. This paper presents COQ MONITOOL, a full-stack web application designed to centralize and automate the tracking and analysis of software quality costs using the Prevention-Appraisal-Failure (P-A-F) model. The system enforces mandatory P-A-F classification through a "one-click" stopwatch interface, automatically computes the Appraisal/Failure (A/F) ratio, and delivers real-time strategic recommendations via a WebSocket-powered dashboard. Three user roles are supported: Project Manager, SQA Engineer, and Developer. Seven user requirements were decomposed into 28 system-level requirements spanning five functional features. The system architecture follows a three-layer pattern: a Next.js/React frontend, a Python Flask COQ Engine backend, and a relational database. Development adhered to ISO/IEC 29110 guidelines using Agile methodology. A comprehensive test campaign comprising 68 unit and system test cases achieved a 100% pass rate across all five features, and a complete requirements traceability matrix confirms end-to-end coverage from URS through design artifacts to test validation.**

Keywords—Cost of Quality, P-A-F Model, Software Quality Assurance, A/F Ratio, Real-time Dashboard, WebSocket, ISO/IEC 29110, Web Application

I. Introduction

Software quality assurance (SQA) is a foundational discipline in software engineering; yet the economic dimension of quality—the Cost of Quality (COQ)—is frequently under-measured in practice. The Prevention-Appraisal-Failure (P-A-F) model [1] provides a structured framework that classifies all quality-related expenditures into three categories: Prevention Costs (activities that avert defects, e.g., code reviews, training), Appraisal Costs (activities that detect defects, e.g., testing, inspections), and Failure Costs (consequences of defects not prevented or detected, further divided into Internal Failure prior to release and External Failure after delivery). The Appraisal/Failure (A/F) ratio, defined as Total Appraisal divided by Total Failure, serves as a concise quality investment efficiency indicator; an A/F ratio below 1.0 signals that the team is spending more on fixing defects than on detecting them—a pattern associated with poor project economics [2].

Despite its theoretical utility, COQ data in real-world development teams is fragmented across disparate systems: project management tools track tasks, defect trackers record bugs, and time-logging sheets capture labor hours. Manually reconciling these sources to derive P-A-F metrics is labor-intensive and error-prone, leading most teams to forego COQ analysis entirely. This gap prevents data-driven decisions about SQA budget allocation and quality investment strategy [3].

This paper presents COQ MONITOOL, a full-stack web application that addresses this gap. The system enforces mandatory P-A-F classification at the point of time entry, centralizes all quality cost data, computes A/F ratios automatically, and delivers actionable real-time recommendations through a WebSocket-powered dashboard. The paper documents the complete software engineering lifecycle of COQ MONITOOL: requirements specification (Section II), system design (Section III), project management (Section IV), testing and verification (Section V), requirements traceability (Section VI), and conclusions with future directions (Section VII).

II. Requirements Specification

A. User Requirements (URS)

COQ MONITOOL serves three user roles defined through analysis of the P-A-F model and VSE team constraints: Project Manager (PM) — the administrator who monitors project health and makes budget decisions; SQA Engineer — who logs Appraisal activities such as testing and design reviews; and Developer — the primary data provider who logs Prevention (process improvements) and Internal Failure (bug fixes) hours. Seven top-level user requirements (URS) were elicited:

ID	Description
URS-01	System shall allow PM, SQA Engineer, and Developer to log in by username and password.

ID	Description
URS-02	System shall allow users to manage their personal profiles (e.g., email, password).
URS-03	System shall enable PM to create project teams and add team members.
URS-04	Developers/SQA Engineers must select a P-A-F category before time recording can proceed.
URS-05	System shall automatically calculate task durations and aggregate man-hours per COQ category.
URS-06	System shall calculate the A/F Ratio and compare it against the theoretical ideal value of 2.0.
URS-07	System shall provide a real-time WebSocket dashboard displaying P-A-F cost distributions.

Fig. 1. **TABLE I. User Requirements Specification (URS)**

B. Software Requirements (SRS)

Each URS was systematically decomposed into system-level requirements. Twenty-eight SRS statements (SRS-01 to SRS-28) were produced, grouped across the five functional features. Table II summarizes the key constraints per feature.

Feature	SRS IDs	Key Constraints
Authentication	SRS-01 – 04	Role-based redirect; HTTP 401 for invalid credentials; HTTP 400 for missing fields
User Management	SRS-05 – 12	Password ≥ 8 chars; PM-exclusive project/team creation; duplicate email rejected
Time Tracking	SRS-13 – 18	Start Timer disabled until Task Name + COQ Category both entered; error msg enforced
COQ Engine	SRS-19 – 24	Duration = stop − start (decimal hours); A/F = Appraisal / (Int. Failure + Ext. Failure)
Dashboard	SRS-25 – 28	Recharts pie chart; WebSocket sync; visual alert when A/F < 1.0

Fig. 2. **TABLE II. SRS Summary by Feature**

Critical behavioral constraints include: (SRS-16) the Start Timer control must remain disabled until both a Task Name is entered and a COQ Category is selected, with the error message "A COQ Category must be selected before recording time" displayed on violation; (SRS-23) the A/F Ratio is calculated as Total Appraisal / (Total Internal Failure + Total External Failure); and (SRS-24) the system must display a prominent visual warning when the A/F ratio falls below 1.0, signaling a "low-quality investment" state.

III. System Design

A. System Architecture

COQ MONITOOL adopts a three-layer client-server architecture. The Presentation Layer is built with Next.js 15 (React 19) and styled with Tailwind CSS. Recharts provides the P-A-F pie chart visualization on the PM dashboard. The Application Layer is a Python Flask server that acts as the COQ Engine: it exposes RESTful API endpoints for CRUD operations on users, projects, and time logs, performs all A/F ratio and duration calculations, and emits real-time metric updates to connected clients via Flask-SocketIO. The Data Layer uses SQLite (in development) with a MySQL-compatible schema managed through SQLAlchemy ORM. The frontend and backend communicate via REST for data submission and retrieval; WebSocket (ws://) connections maintain the persistent channel required for live dashboard updates without page refresh.

B. Database Design

The relational database schema consists of four primary entities, described in Table III.

Table	Key Fields	Purpose
users	id (PK), username, password_hash, role (ENUM)	Stores credentials and role assignments for all system users
projects	id (PK), name, description, start_date, total_dev_time	Records project metadata and aggregated development hours
project_members	project_id (FK), user_id (FK)	Manages many-to-many relationships between users and projects
time_logs	id (PK), user_id (FK), project_id (FK), task_name, coq_type (ENUM), start_time, end_time, duration	Stores every classified time entry; duration auto-computed on stop

Fig. 3. **TABLE III. Database Schema Summary**

The coq_type field in time_logs is an ENUM constrained to four values: Prevention, Appraisal, Internal Failure, and External Failure. The duration field is populated by a SQLAlchemy event hook (UTC-15) that fires on the stop event, computing end_time − start_time in decimal hours. The total_dev_time field in projects is updated automatically upon each log completion, providing an always-current denominator for percentage distribution calculations.

C. Key Design Decisions

Three principal design decisions shaped the implementation. First, mandatory category enforcement was implemented as a frontend validation rule (SRS-16) rather than a server-side check, ensuring that invalid submissions never reach the API and that user feedback is immediate. Second, the A/F ratio calculation was centralized exclusively in the Flask backend (SRS-23) to provide a single authoritative computation source, preventing discrepancies between what different UI views might independently calculate. Third, WebSocket (Socket.IO) was chosen over HTTP polling for real-time dashboard updates: polling would introduce unnecessary latency and server load, whereas WebSocket maintains a persistent connection with sub-second event propagation, satisfying the real-time requirement of SRS-27 and SRS-28.

D. Class Structure

The system's object model comprises five primary classes. The User class (attributes: id, username, password_hash, email, role) exposes two domain methods: set_password(String) hashes and stores a plaintext password using bcrypt, and check_password(String): Boolean verifies credentials at login.

The Project class (attributes: id, name, description, start_date, total_dev_time) represents a development initiative. The TimeLog class (attributes: id, user_id, project_id, task_name, coq_type, start_time, end_time, duration) includes calculate_duration(): Float which computes elapsed time, and update_duration() — a SQLAlchemy event hook that fires automatically when end_time is persisted. The MetricsService class exposes get_project_metrics(project_id): dict, which returns the aggregated P, A, IF, EF totals and the A/F ratio. The SocketIOService class handles emit('metrics_update', payload) and onConnect() to manage WebSocket lifecycle events.

IV. Project Management

The project was developed by a single engineer (Bin Ma, student ID 652115547) under the academic supervision of Dr. Kittitouch Suteeca at Chiang Mai University. Development followed the ISO/IEC 29110-4-1 Basic Profile for Very Small Entities [4], which defines two key process areas relevant to single-developer student projects: the Project Management (PM) process (covering planning, assessment, and closure) and the Software Implementation (SI) process (covering requirements analysis, architectural design, construction, and testing). Agile methodology was applied, with bi-weekly progress reviews with the advisor serving as sprint review checkpoints.

The project was structured into four sequential phases. Phase 1 (Project Proposal, 15 days: November 27 – December 10, 2025) produced the project concept document, scope definition, and feasibility assessment. Phase 2 (Progress I – Requirements and Design, 15 days: December 2025 – January 2026) delivered the Software Requirements Specification (SRS v1.0) and Software Design Document (SDD v1.0), including use case diagrams, activity diagrams, sequence diagrams, and UI mockups for all five features. Phase 3 (Progress II – Implementation and Testing, 15 days: February 20 – March 7, 2026) encompassed full-stack construction and execution of all unit and system test cases. Phase 4 (Final Submission, 30 days: March 2026) covered integration, documentation finalization, traceability verification, and submission.

Configuration management followed the naming convention COQ MONITOOL-[Document]-v[Major].[Minor]. All documents were version-controlled in Google Drive with the source code hosted on a private GitHub repository. Deliverables produced are: (1) Project Plan, (2) Software Requirements Specification, (3) Software Design Document, (4) Test Plan, (5) Test Record, (6) Traceability Record, and (7) Final Software Product (source code and executable).

V. Testing and Verification

A. Test Strategy

The test strategy for COQ MONITOOL employed two complementary techniques. White-box (unit) testing examined the internal logic of individual controller methods and model functions, verifying correct computation and error handling at the code level. Black-box (system) testing validated complete user-facing scenarios against the URS, without regard to implementation details [5]. All tests were designed following the procedure: define test objective, prepare test data, specify expected results, execute, record actual results, and classify as Pass or Fail. Pass was defined as actual result matching expected result exactly; any deviation was classified as Fail. The test environment was a Lenovo ThinkBook 16 G4+ ARA (AMD Ryzen 7 6800H, 16 GB RAM, Windows 11 Home), running the full development stack locally.

B. Unit Test Cases (UTC)

Nineteen unit test cases were designed across four controller and model classes. Table IV provides the complete summary.

UTC ID	Class / Method	Cases	Result
UTC-01	AuthController::login()	3	Pass
UTC-02	AuthController::updateProfile()	3	Pass
UTC-03	ProjectController::getProjects()	1	Pass
UTC-04	ProjectController::createProject()	3	Pass
UTC-05	ProjectController::deleteProject()	2	Pass
UTC-06	ProjectController::addMember()	2	Pass
UTC-07	ProjectController::removeMember()	2	Pass
UTC-08	ProjectController::getUsers()	2	Pass
UTC-09	ProjectController::createUser()	3	Pass
UTC-10	TimeLogController::startLog()	3	Pass
UTC-11	TimeLogController::stopLog()	3	Pass
UTC-12	User::set_password()	2	Pass
UTC-13	User::check_password()	2	Pass
UTC-14	TimeLog::calculate_duration()	2	Pass
UTC-15	TimeLog::update_duration() (Event Hook)	2	Pass
UTC-16	MetricsController::getMetrics()	2	Pass
UTC-17	Metrics::get_project_metrics()	3	Pass
UTC-18	SocketIO::emit('metrics_update')	2	Pass
UTC-19	SocketIO::onConnect()	1	Pass

Fig. 4. **TABLE IV. Unit Test Case Results (UTC-01 to UTC-19)**

Notable test cases: UTC-14 (calculate_duration) verified correct decimal hour computation for both normal durations and edge cases (e.g., zero-duration task). UTC-17 (get_project_metrics) validated the A/F ratio formula across three scenarios: normal ratio (> 2.0), low ratio (< 1.0 triggering the warning state), and zero-division guard (no failure hours logged). UTC-18 verified that the Socket.IO server correctly

broadcasts the 'metrics_update' event with the expected payload structure after each log submission.

C. System Test Cases (STC)

Nine system test cases validated complete end-to-end user workflows. Table V presents the results.

STC ID	Scenario	Linked SRS	Result
STC-01	User self-registration (login page)	SRS-02, SRS-03	Pass
STC-02	PM creates new team member account	SRS-01, SRS-04	Pass
STC-03	User updates password via profile settings	SRS-05, SRS-06, SRS-07	Pass
STC-04	PM creates a new project with name and date	SRS-08, SRS-09	Pass
STC-05	PM adds and removes team members	SRS-10, SRS-11, SRS-12	Pass
STC-06	Developer starts timer only after selecting P-A-F category	SRS-13 – 18	Pass
STC-07	A/F ratio calculated and alert shown when < 1.0	SRS-23, SRS-24, SRS-28	Pass
STC-08	Dashboard renders pie chart and KPI indicators on load	SRS-25, SRS-26	Pass
STC-09	PM dashboard auto-updates via WebSocket when dev stops timer	SRS-27, SRS-28	Pass

Fig. 5. **TABLE V. System Test Case Results (STC-01 to STC-09)**

STC-06 is particularly significant: the test confirmed that the Start Timer button remains visually disabled and non-interactive when either the task name field is empty or no COQ category radio button is selected, and that the exact prescribed error message is displayed upon any attempt to bypass the control. STC-09 validated real-time behavior: with the PM dashboard open in one browser tab and a developer account in a second, stopping the developer's timer triggered an observable, automatic update to the pie chart and A/F ratio indicator on the PM dashboard within seconds—without any page refresh.

D. Test Results Summary

All 68 test cases (counting all sub-cases across UTC-01 to UTC-19 and STC-01 to STC-09) passed successfully. Total Passed: 68. Total Failed: 0. Success Rate: 100%. No defects were identified during the test execution phase, indicating that all specified functional requirements were correctly implemented.

VI. Requirements Traceability

A Requirements Traceability Matrix (RTM) was constructed to verify that every user requirement is realised through design artifacts and validated by test cases, and conversely that every test case is grounded in a documented requirement. Table VI presents the abbreviated traceability matrix, linking each feature across all seven artifact dimensions.

Feature	URS	SRS	UC/AD	SD	UI	UTC	STC
Authentication	URS-01	01-04	UC-01–03 AD-01	SD-01	W-UI01	01,12,13	01,02
User Mgmt.	URS-02,03	05-12	UC-04–06 AD-02–04	SD-02–05	W-UI05–07	02,03	03–05
Time Track.	URS-04	13-18	UC-07–09 AD-05–07	SD-06–08	W-UI08–10	10,11	06
COQ Engine	URS-05,06	19-24	UC-10 AD-08	SD-09	W-UI11	14–17	07
Dashboard	URS-07	25-28	UC-11 AD-09	SD-10	W-UI12	18,19	08,09

Fig. 6. **TABLE VI. Requirements Traceability Matrix (RTM) — Abbreviated**

Full bidirectional traceability was achieved: every URS maps to at least one SRS; every SRS maps to at least one use case, activity diagram, and sequence diagram; every use case has a corresponding UI mockup; and every SRS block is covered by at least one UTC and one STC. No orphaned requirements (URS or SRS without test coverage) and no untethered test cases (tests without a linked requirement) were identified. This confirms that the test campaign provides complete functional coverage of the specified system.

The RTM also supports change impact analysis: should a future modification alter SRS-23 (A/F ratio formula), the matrix immediately identifies that UTC-17, UTC-16, and STC-07 must be re-executed, and that the relevant sequence diagram (SD-09) and UI mockup (W-UI11) must be reviewed. This bidirectional linkage is essential for maintaining quality assurance integrity as the system evolves.

VII. Conclusion

This paper has presented COQ MONITOOL, a web-based platform that centralizes and automates software quality cost monitoring using the P-A-F model. The system addresses the practical problem of fragmented quality cost data by enforcing P-A-F classification at the point of time entry, automating A/F ratio computation in a dedicated COQ Engine, and delivering real-time strategic recommendations to project managers via a WebSocket-powered dashboard. Three user roles, seven user requirements, and 28 system requirements were formally specified; a three-layer architecture with REST and WebSocket communication was designed; and comprehensive testing comprising 68 test cases achieved a 100% pass rate. Complete bidirectional requirements traceability was verified across seven artifact dimensions.

The results demonstrate that the P-A-F model can be effectively operationalized in a lightweight web application suitable for small software teams and Very Small Entities. The mandatory classification interface was confirmed to enforce data integrity without excessive administrative friction, and the

WebSocket-driven dashboard was confirmed to deliver sub-second metric updates.

Future work may pursue three directions: (1) integration with external tools such as Jira or GitHub Issues to automate time log ingestion and reduce manual entry burden; (2) longitudinal trend analysis to visualise A/F ratio evolution over project lifetime, enabling early detection of quality degradation patterns; and (3) multi-project portfolio analytics to support cross-project COQ comparison and organisational-level quality investment benchmarking.

VIII. REFERENCES

[1] P. B. Crosby, Quality is Free: The Art of Making Quality Certain. New York: McGraw-Hill, 1979.

[2] W. J. Morse, H. P. Roth, and K. M. Poston, Measuring, Planning, and Controlling Quality Costs. Montvale, NJ: Institute of Management Accountants, 1987.

[3] D. Galin, Software Quality: Concepts and Practice. Hoboken, NJ: Wiley-IEEE Press, 2018.

[4] ISO/IEC 29110-4-1:2011, "Software Engineering—Lifecycle profiles for Very Small Entities (VSE)—Part 4-1: Profile specifications: Basic profile," International Organization for Standardization, Geneva, Switzerland, 2011.

[5] IEEE Std 829-2008, "IEEE Standard for Software and System Test Documentation," IEEE, New York, USA, 2008.

[6] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 8th ed. New York: McGraw-Hill Education, 2014