

Software Quality Assurance Fall 2025

Bahria University Islamabad Campus

Saira Hameed

Assignment No 02

BS(CS) 8B

Shams ul Islam (01-134212-166)

Due Date: 16th Oct 2025

Assignment should be in pdf format.

Question No 1

[Marks 05]

Software Quality Assurance (SQA) and Software Quality Control (SQC) both aim to ensure the delivery of high-quality software, yet their focus, timing, and methods differ.

Critically analyze a real-world or hypothetical software project where ineffective Quality Control (QC) led to software failure or performance issues, despite well-defined SQA activities being implemented.

- 1) In your answer, clearly discuss:**
- 2) The scenario or context of the project,**

Project: Online trading platform (order-execution service) for a brokerage, a high-throughput, low-latency system that accepts market orders from retail clients, validates them, and forwards them to market/exchange gateways.

Stakeholders: product owners, exchange integrators, ops, QA engineers, SQA team. The organisation has documented development processes, coding standards, mandatory peer code review policy, and an SQA function that maintains process checklists and performs periodic process audits (documented in the company's SQA handbook). These SQA activities were "in place and mature" training, process audits, code-style checks and requirement reviews were happening regularly.

Despite that, during a major release the platform executed a faulty order-routing configuration and a legacy test stub was enabled in production, causing a flood of malformed orders into the

market gateway for ~45 minutes. The company lost large sums (client trading disruption, regulatory fines, market-position exposure) and required emergency rollback and extended remediation.

3) How and why QC failed,

In this scenario QC failed in several concrete ways:

- Insufficient pre-deployment testing of production configuration: configuration changes were not validated against a production-like environment; a configuration flag that switches routing to a legacy stub was not exercised under realistic load, so the failure mode was invisible in lower environments.
- Poor release-control and deployment verification: the CI/CD pipeline allowed a build to be promoted without an automated pre-prod smoke step that verifies critical flows; manual sign-off was cursory and missed the misconfiguration.
- Lack of environment parity (test \neq production): tests ran in environments with different middleware versions and feature flags, so integration tests passed but the production combination exposed a mismatch.
- Inadequate regression and end-to-end tests: unit and component tests existed (and passed), but there were not enough high-coverage end-to-end tests exercising the production routing and failure modes.
- Weak monitoring/alerting for the specific failure mode: the monitoring focused on server CPU and latency but not on the semantic correctness of outbound orders, so the problem was discovered late.

4) The limitations of SQA practices in that situation, and

In this case the organisation had SQA activities (process documents, audits, peer-review rules) but the failure still occurred because:

- SQA focused on process presence, not effectiveness. Having a policy (peer review, audits) does not guarantee that every release follows release-control checklists strictly or that checklist items are effective at catching deployment/config issues.
- SQA did not ensure adequate product-oriented verification (SQC). The lecture differentiates SQA (process) from SQC (product inspection/testing). The team relied on SQA process compliance but delegated product correctness to weak QC activities.

- Gap between process and operational reality. Processes were designed assuming environments and dependencies are static; they did not account for configuration drift, third-party adapter behavior, or legacy stubs left in codebase.
- Insufficient traceability from requirements to tests. SQA audits checked that requirement documents existed, but there was poor mapping from critical requirements (e.g., never route live orders to test stubs) to high-level test cases and release gates.
- Audit cadence too coarse. Periodic SQA audits and training cannot replace gating tests and automated verifications executed on every release.

5) Specific improvements that could have prevented the issue.

A. Strengthen QC (test & release controls)

- Require production-like pre-production environment (environment parity). Ensure configuration, middleware versions, feature flags, and external adapters in staging mirror production.
- Automated pre-release smoke and integration tests in pipeline. Add mandatory automated checks that exercise critical end-to-end flows (order entry → validation → routing → exchange). Pipeline must block promotion on failure.
- Add regression tests for configuration-driven behaviors. Tests should explicitly verify that routing/config flags cannot enable legacy test stubs in production. Include negative tests for forbidden states.
- Canary / phased rollouts with automatic rollback. Deploy to a small portion of traffic first and validate live correctness metrics before full rollout; automate rollback if anomalies exceed thresholds.
- Release checklist + enforced gating. Convert manual sign-offs into enforced gates in CI/CD that require specific test artifacts and evidence (test results, config diffs, verification scripts) before deployment.

B. Improve SQA to close the process–product gap

- Shift SQA from paper compliance to outcome checks. SQA audits should verify that process outputs (test coverage reports, environment parity reports, CI artifacts) exist and are effective. e.g., check that the CI pipeline actually runs the tests and that tests cover critical business rules.
- Traceability matrix (requirements → tests → release). For critical requirements (safety, money flow, regulatory), maintain traceability showing which test cases and release gates cover them. SQA should enforce completeness.

- Independent verification & validation for critical releases. A small independent QC team or external reviewer should perform targeted verification of release-critical paths (business rules, configuration controls).
- Improve configuration management policies. Enforce immutability of production configuration where possible (e.g., use feature flags stored in audited, access-controlled stores, with change approvals).
- Post-deployment verification (synthetic testing) and alerting. Run synthetic transactions after deploy to verify semantic correctness (not just health). Add alerts for business-metric deviations (e.g., sudden change in order-rejection patterns).

C. Observability and production safety nets

- Semantic monitoring & business assertions. Monitor not only resource metrics but semantic correctness (e.g., ratio of rejected orders, orders routed to test gateways). Alerts must trigger immediate rollback/mitigation actions.
- Runbooks & automated mitigation. Maintain runbooks for release failures and automate mitigations (e.g., flip feature flag, route to safe mode).
- Chaos / failure-mode testing in pre-prod. Regularly practice fault-injection to expose gaps in how the system handles misconfiguration or external adapter failures.

D. People & culture

- Release rehearsals and blameless postmortems. Include SQA and ops in rehearsals. After incidents do root-cause analysis and ensure corrective actions are added to SQA process and tracked.
- Train developers on deployment risks. Ensure developers understand configuration risks and the importance of integration tests.