Week 5: Quality Assurance Case Study

Nate Bachmeier

TIM-7140:Software Engineering

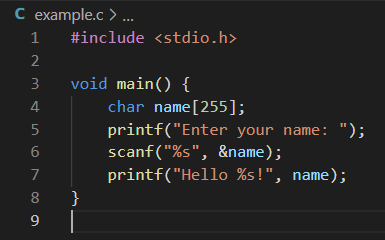
March 28, 2021

Northcentral University

# Quality Assurance Case Study

It is incredibly challenging to build scalable, reliable, and secure software. Examining the trivial example of HelloWorld (see Figure 1) produces more defects than lines of code. First, this application fails when “þêÐŔŌ” types his name. Next, any input exceeding 256 characters crashes the program with a stack overflow. Third, nothing prevents line seven from printing arbitrary memory due to the missing null-terminating marker. Since there are 255255 valid inputs, there is a good chance that quality assurance teams will miss some of these issues.

Figure 1: HelloWorld.c



Software is continuously becoming more complex (Zhang et al., 2013). While this example is relatively contrived, it highlights the existing risk proportional to additional intricacy. Distributed systems are the most complicated computing environments due to assumptions regarding the network’s reliability, security, homogeneousness, latency, bandwidth, and transport costs (van Steen & Tanenbaum, 2019, p. 986). When software systems introduce one of those fallacies into the design, it produces subtle defects under production loads.

The challenges originate from more parallel operations, asynchronously executing, under an imperfect view of the system’s state. Even an extremely reliable component with 99.99% availability fails one thousand times after processing ten million messages. Raised exceptions produce erroneous results, delay other user’s traffic, and waste system resources. Mitigating these risks requires architects to define communication patterns that assume failures will occur. However, those mitigations still require software engineers to implement them correctly.

# The Rostering Problem

School districts maintain student rostering information for tracking associations between courses, students, and teachers. Additional metadata exists within these feeds for tracking grades, assignments, and similar aspects of the academic journey. Educators compound the information volume by including historical longitudinal data. There are multiple competing formats with subtle distinctions, further complicating file processing.

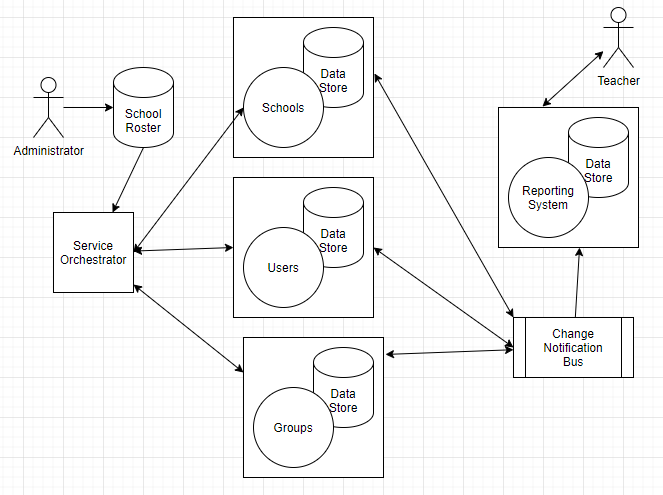
## System Design

Contoso EdTech manages this complexity through an event-driven pipeline consisting of micro-services and NoSQL data stores (see Figure 2). Administrators upload their student roster definition to begin the hydration process. Next, the roster orchestration service normalizes the input and emits events for entity-specific subsystems. For instance, the User Service fronts a key-value store containing teacher and student information with a RESTful API.

After the entity-service handles the incoming event, it outputs change notifications that must propagate across the broader system. For instance, when a student leaves the school, they also drop from their classes. Facilitating this requirement comes from the change notifications. Another standard use-case is replicating the dynamic class memberships into external reporting systems. Similar expectations exist across all life-cycle events, such as grade promotion and course completion.

There are several strengths to this system design. First, the loosely coupled micro-services promote agile methodologies. Agile development practices allow businesses to move faster and be more competitive (Corral et al., 2013; Khalid et al., 2014). Next, the eventing architectures decouple producers and consumers, enabling each component to poses unique scaling and performance characteristics (Celar et al., 2016). Lastly, the entity-specific micro-services allow the developer to align their data access patterns with specific database solutions.

Figure 2: Rostering System



## Performance and Scalability

However, the design also requires enormous event volumes to describe each aspect of the system. For instance, propagating a course enrollment means touching the student, membership, course statistics, and permissions subsystems. Each of those changes cascades another layer of updates that transactionally span dozens of private databases. Additional complexity arises from an eventual consistency model that comes with asynchronous programming. Districts like LA County and Houston ballon under this strategy and necessitate hundreds of millions of individual operations.

## Data Validation

Assuming the system is 99.99% reliable, one import will potentially encounter nearly one hundred thousand issues. Most of these failures are resolvable through a simple retry policy, but what about the remainder? For instance, Alessandra-Joesophena Ó Maoilsheachlainn-Maceachthighearna from Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch, Wales, United Kingdom, likely encounters form validation errors. The roster file might also contain typographical values, such as promoting a student from first to twelfth grade.

Distributed multi-database systems are challenging to ensure consistency because of the CAP (Consistency, Availability, and Partitioned) Theorem Principal. Essentially CAP states that systems can possess at most two of these characteristics. Many production systems opt for availability to meet their Service Level Objective. Next, they choose Partitioned to support the enormous data sets that customers demand. After degrading consistency to *eventual* consistency, systems engineers need to manage another complexity level (e.g., out-of-order events).

## Quality Gaps

Quality assurance teams create test plans to verify functionality, security, performance, and reliability. However, predicting correct outputs for a given input is difficult at scale (Jahangirova, 2017). Assessment teams partially mitigate those concerns with sampling, counting, and narrow exploration. For instance, engineers might stress test the rostering system by importing one hundred thousand students. Afterward, the test passes if there are one-hundred thousand new records. More comprehensive verification might confirm the first ten records in detail.

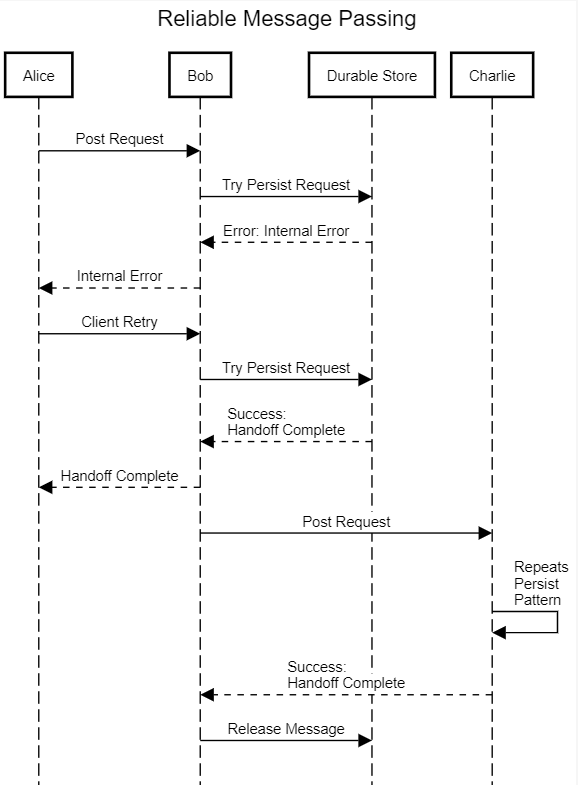
Even when extensive validation occurs, it is often under ideal configurations, such as an idle shared test environment. While this improves the chances of passing the test, it lacks real-world carnage rampant across high-volume production networks. The test cannot discover any distributed system fallacies because no constrain exists across the network’s bandwidth, latency, and available transport.

## Major Production Issues

With millions of events traversing hundreds of components across dozens of components, many opportunities exist for errors. The most common source of issues arises from messages not correctly propagating. For instance, a missing course membership entity means that Ms. Allen’s Math course is missing Timmy Smith. These scenarios are particularly complex for software engineers to troubleshoot, let alone end-users to explain.

Developers need to address these issues with proper message patterns (Li et al., 2015). Those patterns include enforcing positive hand-off between components (see Figure 3). Services can implement those techniques via durable queuing and enforcing client (not server) retries.

Figure 3: Reliable Message Passing Pattern



The second class of issues comes from exceeding service quota limits. For example, a database can only support N-connections per second. Meanwhile, a serverless function can only execute for D-seconds with G-memory. Unless an official Quality of Service (QoS) definition exists, the software engineers cannot encode these constraints into the test cases. These gaps inevitably become evident from customers using the system. Over time these discrepancies narrow with internal implementation conforming to expected behavior.

# Future Research

Implementing sophisticated software is challenging, and even the most trivial programs contain numerous defects. Given these constraints, it is not possible to prematurely uncover every failure. Instead, systems engineers must define supportable environmental conditions, component limits, explicit data contracts, and enforce positive hand-off.