Continuous Testing 101 – The Key to In-Sprint Testing

BY **TIP SPRINGSTEAD**

CONTENTS

Faster. Better. Cheaper. These are the basic goals of manufacturing. We get there by maintaining smaller batch sizes, maintaining a low amount of work in progress (WIP), and automating everything we can. This accelerates throughput, shortens the time to market, and increases revenue. DevOps practices and Agile methodologies have taught us that, despite their linear versus iterative workflows, software development isn't all that different from manufacturing. The goals are the same. We measure our progress by the frequency of our releases and the volume of user stories we can promote to production.

Less quantifiable, however, is quality. Quality is more like an aspirational goal and it tends to be an implied outcome. It doesn't really exist until it's absent. It's an expectation that can only really be measured by lack, and it manifests through defect tracking, reworks, delays, and user feedback. The impact of quality gaps, in some ways, are a foregone conclusion to the outcomes of business as usual. It's easy to sweep up the pieces and push them under the rug since "business" is always broken, and humans are imperfect. However, defects must be fixed and we're forced into introspection:

- Was it a coding error?
- Were the requirements ambiguous?
- How thorough was our test case design?
- Did we understand our testing coverage?
- Were we using the wrong test data?
- Did we lack the tools to simulate and test dependencies?
- Was something misconfigured in the QA environment?
- Was there a mistake in our deployment sequence?

The list can go on, but notice how, in the snippet of questions just mentioned, less than half pertain to what a traditional QA department would own. That's because quality must span the entire software development life cycle (SDLC). Since manufacturing is such an apt analogy in several ideological levels, consider TQM (total quality management) and Lean Manufacturing. In an assembly line, each stage of production requires quality control checks. For instance, what would happen if the wrong material was unknowingly delivered to the 12th station in an assembly line? Every batch that passed station 12 would represent instant rework. That rework is then compounded across each subsequent station until the defect is caught. Imagine the impact if there are 25 stations total and it's not caught until right before the product is packaged. Depending on the product, the plant could easily now be a day behind schedule and thousands of dollars in the hole. This is why each major stage change typically has a Quality Control person standing around in a white lab coat and maybe a hardhat taking and testing samples.

SO, WHAT IS CONTINUOUS TESTING?

It's best to define Continuous Testing by first taking a look at what it isn't. The first concept to come to mind would probably be test automation – having tools in place constantly running tests and providing an ongoing feedback loop. Maybe you think of Continuous Integration – ensuring daily builds aren't failing. Also, developers are constantly checking to see their code is working through unit testing. Is it that? Whatever it is, it's probably done in the QA Environment and is a QA-led activity, right?

Wrong on all accounts. Continuous Testing isn't solved by any one tool with a specific feature set, and it's not the responsibility of just one team. Rather, Continuous Testing is a much broader concept, and it's an interdependent discipline with Continuous Delivery. Meaning, you can't have one without the other. Why? Because of quality. Continuous Delivery can be defined as the ability to release any application to any environment at any time with zero touches. That's a whole lot of automation, but what would be the benefit of shorter release cycles if each application code change produced an adverse effect somewhere else in the system for users to experience? The reality is, you would be testing somewhere along the way and if quality wasn't being vigorously and continuously tested, the pipeline would be continuously halted. At the end of the day, the expectation is, "it (the application, the feature, the button,

Continuous Testing

Enable continuous testing across your software delivery lifecycle.

Adopt next-generation testing practices to test early, often, automatically, and continuously.

Only CA offers a continuous testing strategy that's automated and build upon end-to-end integrations and open source. Enable your DevOps and continuous delivery practices today.

Explore ca.com/continuous-testing

ca technologies

the accounting functions, etc.) should work." If our goals are to lower costs and shorten the time to market with higher quality, then quality assurance must be present throughout the SDLC.

As obvious as this is, let me ask another question: why then do we rely so much on functional UI testing? It would be absurd to manufacture an automobile AND THEN run an airbag circuit test to see if the gold conductor 1) completes the circuit, 2) releases the charge, and 3) deploys the airbag and rips the interior apart. Within the manufacturing industry, there are systems in place to catch defects when they occur – maintaining the systems integrity and balance of WIP. Therefore, the definition of Continuous Testing for DevOps is: **The practice of testing across every activity in the SDLC to uncover and fix unexpected behaviors as soon as they are injected.**

10 KEY ELEMENTS OF CONTINUOUS TESTING TECHNOLOGY

While "Test Automation" does not define Continuous Testing, it certainly is a part of it. In fact, of the five "It's not" statements mentioned earlier, "It's not a QA-led activity" is the only one that doesn't represent some part of Continuous Testing. Instead, a true Continuous Testing initiative would come from the DevOps management level. And just like any good discipline, concept, or initiative, in order to achieve a "true" Continuous Testing practice, there are 10 tenants for success. They are more like 10 key elements, spread out across the SDLC, representing certain processes and technology which have not yet reached their potential inside of most organizations or, at the very least, been combined to such a degree.

1. AUTOMATICALLY GENERATE TEST AUTOMATION SCRIPTS FROM REQUIREMENTS

Quality is owned by everyone in the SDLC. Each stage of the life cycle needs a quality control. However, building quality into an application should start at the source of it all; the requirements. That's where a requirements model comes into play. Models enable requirements to be defined in an unambiguous and testable manner so that test cases and automated test scripts can be automatically generated from requirements. That happens as soon as requirements are defined, before a single line of code is written. Additionally, models are self-healing. If you change the design of your requirements, you've already changed your test cases and automation scripts. You have a record of versioning, previous test cases, and metrics indicating the functional test case coverage of the application.

2. SIMULATE YOUR TEST ENVIRONMENT

With tests in hand after defining requirements, the next step is to wait for the application code to be deployed. But once that occurs, the team quickly realizes the first roadblock to overcome is the lack of available environments and interfaces needed to run the previously created tests. So, you need to be able to virtualize those environments and interfaces to remove those testing constraints whether your testing at the unit, integration, or functional levels. Virtual end points and request response pairs can also be a part of the requirements model, pushing the elimination of environmental road blocks even further to the left.

3. ACCESS TEST DATA ON-DEMAND & ENSURE PII COMPLIANCE

With the tests ready to run, and interfaces available (real or virtual), the next roadblock the team has to tackle is test data. It takes too much time to find or make the right test data and ensure it's available to be consumed during testing. While masking and subsetting or creating virtualized, individual, archived copies of production are helpful, it's sometimes like asking for a haystack to get a needle. To tackle the challenges of test data, we need to think about test data differently. We assume the best place to get test data is production. If the purpose of test data is testing, then we need to set up generation rules to create fit-for-purpose, synthetic test data and give the ownership to execute those rules to the testers. Taking it further, test data must be fully correlated across systems and databases and tied to the test cases. This means capturing data generation rules at the requirements and planning phases. The data is then captured by design, desensitized, and ready once the code hits the QA environment.

4. START MULTI-LAYER / BACKEND "REQUEST & RESPONSE" TESTING PRIOR TO UI

Now that we have the tests, interfaces, and test data ready to go, the next step is to run the tests (finally!). That's when teams will need a test execution engine. Executing tests and consuming test data across multiple applications and system layers (backend, API, UI, etc.) is essential to achieving the speed and quality all teams are looking for.

5. DEMOCRATIZE PERFORMANCE TESTING

Up to this point we've framed the conversation from an Agile development and testing perspective. Performance testing historically was for preproduction and maybe QA, but now performance testing can be done even earlier. By democratizing performance and load testing we can shift further to both the left and right. We can know at the unit level if our code will cause degradation of the overall system, and continue testing it through to post-production.

6. INTEGRATE OPEN SOURCE TESTING TOOLS

Great open source testing tools are out there. Think Cucumber, Selenium, JMeter, Gatling, and Locust to name a few. Developers and testers love them. Like smartphones, they come with the employees, and we should work with them wherever possible. For one, it saves money, but it also means we can keep pace with the industry and embrace new technologies. It's important to not only include them in our tool boxes but expand our integration with them so that we can automate the interactions and test at scale.

7. ENSURE COMPREHENSIVE CLOUD-BASED API TESTING

With virtually limitless scalability, teams can throttle up or down their testing against web applications and APIs according to their needs. The rapid adoption of microservices architecture is only causing the proliferation of both private and public APIs to accelerate and become revenue-generating opportunities. Lack of test coverage results in broken APIs coupled with the challenge of pinpointing issues once deployed, thus reducing development

velocity. SaaS–based functional API, testing solutions can auto-generate and execute comprehensive tests for each API allowing teams to test earlier, more often, and prevent defects from occurring.

8. BUILT-IN AUTOMATED APPLICATION SECURITY TESTING

With that perfectly working code that is perfectly performing, before putting it in the hands of your users, teams must ensure that code is secure. That's where static and dynamic security tests come into play. It's critical to treat the remediation of security risks the same as we do defects. Therefore, we must begin testing application security at the development, pre-production, and production levels to lower the risk of significant financial losses. Security must be at the core of all we do.

9. PROMOTE ARTIFACTS WHEN TESTS PASS BY ORCHESTRATING AND AUTOMATING THE PIPELINE

For teams to coordinate all those different activities we need an orchestration engine, which is responsible for defining what is supposed to happen and when, as well as capturing the data from all the involved tools to ensure the team can quickly understand what is going on across the different parts of the lifecycle. It's more than just a release engine. It's a manager of managers that automates all the tools within the entire pipeline. For example, the orchestrating engine would:

- Launch virtual services mentioned in Element 2 based off the models in Element 1, as well as provisioning test data from Element 3 which was also tied to the models.
- Activate Element 4's automation that picks up the test scripts created from the same models in Element 1 to feed them into the test automation engine and execute them.
- After receiving the all-clear, start the performance, API, and any other testing in Elements 5, 6, and 7; and so on.

10. HARNESS APPLICATION INSIGHT ACROSS THE SDLC TO IMPROVE USER EXPERIENCE

To top it off, before and after the application is released to users, teams must continuously monitor the technology, user, and business metrics to correlate them and learn whether users are perceiving the value the team originally intended to deliver. Based on those lessons learned, new hypotheses are created, and ideas for new features or changes are defined and added to the next sprint's backlog so the team can go at it again.

CONTINUOUS TESTING AS A PRACTICE – LEVERAGING DEPENDENCIES

Continuous Delivery is dependent on Continuous Testing, Continuous Testing is dependent upon the technology you employ, and the technology is dependent upon the processes you put in place to utilize them. "Dependency," in itself, is a neutral word. It's not until the dependency becomes a perceived or measurable hindrance that the term becomes slandered. If only we could create technology to code all the ideas in our head, but we can't. Quality software is dependent upon us.

Despite the tools that have been mentioned which span across the entire SDLC from ideation to postproduction, we're really only talking about four objectives in achieving quality. Let's call them, "The 4 Pillars of Continuous Testing," and pull the covers back a little.

- **Quality Code:** Language, company and style guides, security, and industry standards.
- **Quality Pipeline Automation:** Tool automation and integration with orchestration.
- **Quality Application:** Satisfying the business at the story, feature, and epic levels.
- **Quality Customer Experience:** The perceived value held by the end user.

EXAMPLE 1: THE DEVELOPMENT ENVIRONMENT

Let's say a developer is using a popular Continuous Integration automation tool to build, test, and deploy code. The tool requires configuration to kick off code scans once the code is checked in. The configuration is comprised of success criteria, triggers, tolerance levels, and thresholds. If code is scanned and there are more warnings than the criteria should allow, the build should be stopped. So, how thorough are we testing our assumptions which drive our automation? This is a dependency of the QA team. They're expecting quality to be an inherent trait of the code coming their way and it's the developer's responsibility.

EXAMPLE 2: THE TEST ENVIRONMENT

The team is moving into a second sprint. How many previous assets are reusable? Virtual Services need to be provisioned. We can use stored endpoints as long as the needs haven't changed. Maybe the endpoints are good as they are but we need a different scenario of test data. Test data needs to be designed and provisioned or queued up but not until any relevant schema changes have been registered and synthetic data rules created. Again, it's not enough to test code. Continuous Testing as a practice brings quality full-circle, back to testing ourselves. The beauty of testing our assumptions is that technology is allowing us to refine our processes, capitalize on previous work, and to work in parallel.

EXAMPLE 3: THE STAGING ENVIRONMENT

Scale, scale, scale. Like the test environment, the staging environment might undergo similar testing and maybe greater regression scrutiny, but regardless, it will be more robust. There will be more virtual services and data will need to be richer and more comprehensive. High performance loads will be pushed through, and the data will need to be there to support it.

EXAMPLE 4: THE RELEASE

Tests and performance are stellar. This should be easy, but a configuration error fails the deployment. Outside of having the right tools to roll this back and identify the error, it's discovered that because the topology of production is different than

production, despite all our automation and testing, the build configurations weren't regression tested.

CONTINUOUS TESTING IN-SPRINT AND BEYOND

Some people get offended when software development is compared to manufacturing. It's not always palatable to imply that iterative development is somehow likened to linear, sequential, Waterfall-like execution. If we're honest with ourselves, when it comes to most of our present testing practices, it's more like we've been trying to fit a square, Waterfall peg into a round, Agile hole. The manufacturing guys figured it out, why shouldn't we learn from them? While Dev teams are now working in sprints and collaborating with Ops more effectively, we're still testing the old way. Here is what ends up happening.

If development teams have a scrum master and work in sprints, but are completely detached from the time tables and activities of central QA, and if releases are dependent upon the time it takes for QA and Dev to produce something ready to release, the organization is still in Waterfall. You could call that, "Water-Scrum-fall." Continuous Delivery isn't possible and most teams are using the same technology and processes as before. The only thing supporting DevOps is people.

If your teams are more closely aligned, but still suffering from +1 sprint lag, completing testing in the second sprint and releasing in the third, then the organization is "Agile-Fall." Typically, this would indicate dev teams are working on smaller batch sizes, maintaining WIP fairly well, and perhaps they've adopted some new technologies like Agile planning tools. However, the QA team is kicking and screaming because they want to do a better job but can't due to time constraints. Developers are still stuck with costly reworks because defects are making it to production. Whether they learn from the second sprint or a month later, they have to refocus and remember what they did.

But by adopting the right technology and supporting it with best practices, ensuring that everyone has a Continuous Testing mindset, there's no reason your organization can't be completely Agile – completing both testing and releases in-sprint. Developers get instant feedback on the code they've just written, not just from a unit or integration perspective, but a functional and performance perspective as well. Everyone moves together, dare I say, like a well-oiled assembly line, packaging code to be picked up by automation and deployed into production.

In business, everything is about quality. In light of Continuous Testing, the adage, "time is money," is still true. If too much time is spent in rework fixing bugs resulting from our assumptions that were derived from ambiguous requirements or previous configuration thresholds, money is being wasted. However, to reach our goals of faster, better, and cheaper software, "Quality is money," has a nicer ring to it. After all, the better the quality, the better the money.

ABOUT THE AUTHOR

CHRISTOPHER SPRINGSTEAD (nicknamed "Tip") is a Product Marketing Manager for CA Technologies whose focus is on Continuous Testing. After spending 10 years as a functional business analyst, Tip found his calling and passion in educating people on best practices. "It's not enough for me to constantly find and uncover the obstacles and limitation of our business practices. I want to help drive the catalytic forces that change things for the better."

DZone

BROUGHT TO YOU IN PARTNERSHIP WITH

ca technologies

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com