DOI:10.1145/2524713.2524721

Article development led by acmqueue
queue.acm.org

Merging the art and science of software development.

BY JAMES ROCHE

Adopting DevOps Practices in Quality Assurance

SOFTWARE LIFE CYCLE MANAGEMENT was, for a very long time, a controlled exercise. The duration of product design, development, and support was predictable enough that companies and their employees scheduled their finances, vacations, surgeries, and mergers around product releases. When developers were busy, quality assurance (QA) had it easy. As the coding portion of a release cycle came to a close, QA took over while support ramped up. Then when the product released, the development staff exhaled, rested, and started the loop again while the support staff transitioned to busily supporting the new product.

From a development team's perspective, the product release represented a closed-loop process that was repeatable and formulaic. Outside of a bug that might need expert attention from developers, most of the staff was repurposed to focus on the next version as soon as the product was made available to customers.

In this world, the QA team's responsibilities consisted largely of mimicking customers' usage patterns and minimizing the discovery of unpleasant surprises. In the waterfall model, a QA team's badge of honor was its ability to anticipate, find, and deliver reproducible scenarios that could result in problems for users.

Building out random hardware configurations, simulating puzzling network landscapes, and wiring up seemingly orthogonal factors in underlying platform architecture are all facets of a great QA engineer's knowledge. The very best ones can look at a product's design and predict where the problems might occur. With stand-alone desktop software, shipping bugs is very expensive since the fixes involve service releases, so QA was given more time and staff to verify everything was working as expected.

I remember when this era of predictable variables and finite schedules ended. I remember it very specifically, in fact. I was working for a company trying to launch a prototype of an eBay-toppling person-to-person marketplace product. We grinded for months testing the service from the user interface down. The night of the big release, the biggest conference room in the building was packed, champagne was served in cardboard coffee cups, and the whole extended team anticipated the release, eventually chanting the countdown, "5, 4, 3, 2, 1!" Hugs and toasts were everywhere as the CEO used the new site to buy silly items we joked about in the product's catalog. Then I got a tap on the shoulder. It was the engineering lead, and his words were simple: "We think you should be downstairs with us."

PHOTOGRAPH BY DAVE ROTH

Upstairs, everyone was smiling with anticipation at our product's debut. But, in a much smaller room downstairs, with door-desks rimming the walls, no one was smiling. The website "worked," but it was running too hot considering the tiny amount of traffic.

It was about 7:00 P.M. My job was primarily to sustain load on the service via a hastily prepared shell script to generate traffic. Meanwhile, half a dozen engineers hung over a single screen with logs scrolling and graphs bouncing up and down. After we tweaked the load balancers, the troublesome graph lines started pointing downward, signaling relief from the issue we were facing. The whole room breathed a cautious sigh of relief.

The party upstairs ended hours before I left the office that night. My dog and I walked home sober and tired at 11:30 P.M., satisfied but anxious to hear the fate of the product after the announcement the following day.

We made the front page of the *New York Times* the next morning, huge recognition for us, but my primary memory of that time was a newfound appreciation for the afterlife of a software product.

What products live solely on desktops anymore? How many product cycles have even 24 hours between the last bug fix and exposure to customers? More importantly, how does this change the people and the process involved in releasing, maintaining, and supporting applications?

Over the Wall

I recently came across a funny online meme called "Disaster Girl" that, admittedly, I might have been the last person on earth to encounter. In the background of the photograph, a home smolders while neighbors and firefighters stand a safe distance away. In the foreground, there is a young girl with a smarmy smile, together with loud text reading, "Worked fine in dev. Ops problem now."[5]

Part of the reason I suspect this is not a newly created joke is that the world really does not work that way anymore for most software creators. Where waterfall development once dictated the process of handing off from developers to QA to operations, the responsibilities now have hazy borders, and most engineers' skill sets span multiple disciplines.

The conflict between contributors is now best laid to rest. Cantankerous QA engineers have almost no place in a scrum team. Products grow from prototypes. The design phase has been replaced by a less cyclic, democratic evolution from development to demo.

There is no forum for contentious stand-downs within daily stand-ups. Gone too is the mystique of the operations engineer. Where there once was the Hollywood image of the nocturnal know-it-alls divining the status of a product in a room of glowing screens with indecipherable logs, the culture of DevOps has removed much of the magic and caffeine dependency from those roles.

One refreshing outcome is a steep upswing in expertise and professionalism within the industry. At the same time, engineering prowess allows teams to automate much of the tracking that once required 24/7 staffing of human system monitors, and this has improved the work/life balance of engineers across the globe. For each group—including developers, quality assurance, and operations—significant changes have happened across their roles to highlight aspects of the modern software architecture template.

So, What Exactly Has Changed?

Think back to the mid-1990s when most of what was installed onto machines came via floppy disks or CD-ROM media. The bulk of software seldom interacted with any resources outside of its own host's storage, and software that did interact with external resources did so via a limited array of enterprise products.

The development-to-support chain of command was as defined earlier. Developers worked from a spec, and the "Minimum Requirements" document was a rigid set of constraints. The testing effort consisted mainly of populating a matrix of hardware/feature pairings, driven by a daily installation, and a mix of manual and automated tests. Automation was primarily driven to gather performance metrics for benchmarking and stress tolerance. Operations were largely the responsibility of customer-support technicians who had day-to-day contact with those installing the product. Suffice it to say, there was no need for a dedicated operations effort because the landscape was limited mostly to each user's local environment.

Between this era and the present, the numerous flip-flops between thin vs. thick clients have yielded a broad spectrum of client capacity for applications. On the thick side, gaming and data-management software require very powerful clients with strong support for the myriad network options in the modern landscape. The thin-client end of the spectrum is populated with browser-based apps. Bridging the thick/thin outliers are a massive number of applications where the local processing environment is paired with data and asynchronous processing over a network.

These new offerings run far slimmer than anything discussed in the early days of the thick/thin trade-off, and now far outnumber those on any other platform. With mobile platforms growing in processing power, that thick/thin balance has a new spectrum of its own. Mobile is an interesting category, since apps are constrained by both screen size and processing capacity, and users' expectations are much lower than for a similar app run on a desktop environment.

When this migration away from isolated clients became the norm, the roles within software development teams moved in tandem. Unfortunately for many, this wasn't an immediately obvious requirement. Software cemeteries are overflowing with products whose management did not react quickly enough to the newly networked environments. Those that did adapt were forced to move roles out of the traditional dev/QA/support framework and build teams that were more collaborative in their approach to building fault-tolerant systems with a functionally infinite life span.

The waterfall is now a river; development and operations flow together. When a product launches, there are seldom plans to upgrade or deprecate functionality, and support is ongoing. The industry has generally done a good job adapting to the expectations of this new landscape.

Most notably, engineering efforts now serve multiple masters. Where the focus used to be purely on precision, now longevity and scalability hold equal footing in product requirements discussions. Your application might still move mountains, but now it needs to move mountains for everyone on earth forever.

Where QA had traditionally held the line on risk definition and maintainability, the development and operations people (shortened to DevOps) now have an equal, if not dominant, role. Bargaining for features now involves feasibility and sustainability beyond the limits of PC chipsets and media capacity. DevOps own the profile for system capacity and expandability within platform resources internal and external to the organization.

DevOps and Development, Concepts, and Misunderstandings

No standard definition exists for DevOps. Blog posts on the topic abound, but the best one can derive is that the sands are still shifting on what the term actually means. One side of the argument identifies a specific job description for DevOps, well summarized in a blog post stating that developers work mostly on code, operations work mostly with systems, and DevOps is a mix of those two skill sets.[6] The other side of the table is not specifically opposed to that notion but argues that DevOps is not really a job (in that you do not hire a "DevOp," per se), but rather that the spirit of DevOps speaks to an emerging need in the modern software development and support landscape.[3]

This issue has divided the community for some time. Engineers who proudly describe themselves as DevOps are clearly not on the same page as those who think there is no such thing, but the existence of thousands of open job listings specifically seeking "DevOps engineers" proves that many people at prominent companies believe the term describes a specific role. Others in the industry believe the term describes new criteria for development, testing, release, support, and metrics gathering. In this article I am taking no sides but use *DevOps* to describe the new criteria rather than a specific job.

In my experience, I have observed the DevOps notion emerge from many of the same industry forces that have brought the job description of QA engineer closer to that of developer. The more one knows about the code and platform for an app, the better one will be at building and fixing that app. DevOps, whether in a situation that has operations engineers picking up development tasks or one where developers work in an operations realm, is an effort to move the two disciplines closer.

This merging of software development and support came about as the need arose for a better toolset to detect and measure problems in networked systems. Hundreds, if not thousands, of different companies created homegrown solutions to common tasks, and then these became tedious to build, scale, and maintain. Not long ago, a number of products became available that gave development organizations a way to address common needs such as problem diagnosis, deployment management, task automation, and process standardization. In this way, development organizations, whether large or small, could consolidate the resources dedicated to building such mechanisms.

The biggest benefit derived from this new infrastructure is the ability to quantify aspects of the development and support effort. Without uncertainty, the development process can be described with figures that perfectly communicate the data to heads-down developers, as well as to nontechnical project and product owners. This is a huge value to teams, and one thing is clear from the aforementioned debate within the community: there is definitely a pre-DevOps and post-DevOps era of software development.

Across the entire art form, the influence of DevOps puts efficiency and process into perspective. There are specific areas, though, where the DevOps wave has served to completely and permanently improve product development and ownership, thanks specifically to a sharper focus on metrics and process standardization.

There are specific areas where the DevOps wave has served to completely and permanently improve product development and ownership, thanks specifically to a sharper focus on metrics and process standardization.

Process Standardization

The industry is once again consolidating its platform options, and system management is becoming increasingly standardized as industry outliers build shims for popular products such as Chef, Splunk, and Jenkins, among others. The handful of time-tested products that embody the hammers and screwdrivers of the DevOps toolkit are so ubiquitous that administrator-level experience with them is part of many job descriptions. While this might not seem new or odd in a small company where the infrastructure uses these products as the cornerstones for their systems, large companies with multiple teams developing and deploying separately are finding this standardization to be a valuable alternative to forcing homegrown solutions, allowing devil-may-care independence, or anything in between.

For many companies, this standardization sanitizes the release and support processes. Many organizations grew their processes around immobile objects. Desktop software could really only release around CD/DVD manufacturers and publishers, but now they have online app marketplace standards setting the guidelines. Similarly, online products with critical uptime requirements have led to costly maintenance contracts with data-center service providers. With the ability to now bring this work in-house comes a better understanding of the process and less of a dance for those responsible for publishing a new product.

True validation for any software organization's development, deployment, and support process is the concept of continuous deployment—which is, in effect, the harmonizing of all three tasks to the extent that customers are not notified, or even concerned, about software upgrades. Continuous deployment requires a development organization to achieve sufficient quality so no stone is left unturned in the short path between a source-control submission and a release. It means deployment must be undetectable to users, some of whom will have the software version changed out from under them mid-session. It also means the response to any problems encountered in the course of such a surreptitious release is both proactive and immediate.

A definition of DevOps in a 2010 blog post by Dmitriy Samovskiy[8] names one aspect of the job description "QA in Production." This undoubtedly raises the hackles of many in the industry, but thinking about how this helps define a new standard for everyone in the software industry makes it a valuable bullet point in an otherwise controversial post. Again, to the degree that the most valuable contribution from the DevOps culture surfaces in the software support role, embedding the associated culture throughout the whole product life cycle pushes valuable improvements all the way back to the design phase of an application.

Moving Software Development Metrics to a DevOps World

The aspect of the modern software landscape that most benefits creators is the availability of both thin and thick client solutions at design time. Processing power on most clients, now including most mobile devices, is on par with server-side processing once the costs of data transit are considered. DevOps makes it possible to realize the promise. When the complexity of a feature on the client brings it close to exclusion, acknowledging this trade-off is a necessity. Furthermore, accurate depiction of the give-and-take is the difference between shipping something that succeeds versus shipping something that becomes a punch line for the rest of the industry.

This brings up an interesting aspect of DevOps, in that we have now finally moved beyond the point where QA engineers' insights were consistently coupled with questionable data. DevOps at this point own the server-side numbers in the sense that load, capacity, and scalability have become easily derived figures.

When dealing primarily with client-side code, the variability of client permutations and a lack of accurate behavior measures make QA input far less objective and verifiable. For example, many QA discussions start with, "We have customers complaining about this feature running slow on their phones." Easy follow-up questions include, "What else is running on those phones?", "Are they running on Wi-Fi, 3G, or 4G?", "Is this a problem with a particular phone?", and then go on from there. A well-informed DevOps engineer can say, "Our product now uses 40% of our capacity with our 10,000 active subscribers, so if we can't double the number of instances between now and the next release, we can't take on more features or more customers." No one is going to second-guess such a data-driven DevOps ruling, whereas QA engineers were traditionally unable to support their arguments without doing a lot more research. In a way, DevOps have an easier job now in terms of defining and addressing problems with a client/service app landscape. At the same time, it is a daunting responsibility for QA to provide that same level of validity when describing issues in the application ecosystem.

There is already an emphasis on delivering much of the on-the-ground experience via client application metrics. Crash reporting is something many applications suggest users allow at install time. Browser-based applications have the ability to track just about every aspect of each interaction a user has with an app, with mobile devices delivering a similar crumb trail. The thinner the client, the more data is lost on an application crash. Without explicit permission, most browsers and all mobile operating systems prevent access to the data necessary to evaluate the cause of an application crash.

When dealing primarily with client-side code, the variability of client permutations and a lack of accurate behavior measures make QA input far less objective and verifiable.

Ironically, the source of QA's client data comes at the cost of throughput for the app. There are certainly many ways to collect data from clients running an application, but mitigating the amount of data and frequency of delivery negatively affects the performance of the client and the availability of the service. Again, browser-based apps and those running on mobile platforms experience the worst effects. While apps installed to an operating system can run sentinel processes that emit data at regular intervals whether or not the targeted application is active, browser-based applications lose all identity as a client unless they are active. So, yes, it's a frustrating circumstance for QA engineers. Innovation is constantly improving the situation for client-side data, but the trade-off between the quantity of data and the impact to the application is never going to go away.

Perhaps the best representation of the state of the art is the number of products vying for customers in the mobile platform space. Within the iOS landscape, Apple provides its native crash log reporting to all apps sold in the iTunes App Store.[4] As just described, users need to opt in at install time to provide this data to the application owner, so there is not as much data provided as with third-party providers. In the third-party space, the providers truly proliferate.[7] The products provide a variety of specialties and each comes with a DevOps-inspired graph-based user interface to track day-to-day frequency, stack-trace weak points, and provide for the global pinpointing of exception-bearing clients.

For applications running on Android, a similar proliferation of products outside of the native platform abound. The native Android SDK gives a far better venue for reporting than seen in native iOS,[2] including reporting for crash events, as well as caught and uncaught exceptions. Extensions make integration of these reports easier,[1] primarily for those keeping to the Google infrastructure. Largely, products that have the most usage in the iOS space are also available on the Android platform. These products are likely most attractive to application developers with multiplatform offerings.

Where Do QA and DevOps Converge?

After analyzing the techniques available to QA and DevOps, the comparison still begs for details on how, within the engineering phase of a development cycle, the client experience can be captured with the validity and insight that's available to an operations engineer in the DevOps era.

The currency of QA influence is manifest in the schedule, in feedback to design decisions, in prioritization of bugs, and in the decision to release despite open bugs, with the degree of convergence being largely a function of risk assessment. Using app metrics gathered from production, engineers have long been able to project severity onto discovered bugs. The task is not as easy, though, when working on new features or when a new design ends up forcing users into some completely new workflow.

The schedule for testing must move from the antiquated waterfall model to one that acknowledges the importance of testing and release simulation. This is obvious, and it constitutes a platitude of the software-development ideal. The new concept introduced by the influence of the DevOps culture promotes programmatic analysis of client scenarios. Expansion of proactive analysis of client scenarios should essentially automate the variety of use-case paths that a user could and, maybe more importantly, is likely to encounter. Issues discovered that fall more closely to the user's critical path will obviously be graded with higher severity than those that impact users falling outside of this usage pattern. The difference between the common assessment and the same analysis through DevOps lenses is the presence of data. Usage data obviously must be the basis for severity assessment.

Assessing design decisions requires expert insight into all guidance. A hallmark of the DevOps culture is a working knowledge of the overall system with the ability to profile and make predictive issue assessments based on a solvent knowledge of the entire application landscape. From the QA perspective, this insight gives credence to questions about precision, performance, and probabilities of delivering the proposed outcome of a design decision. This is where the insight of a developer is largely sufficient, but the DevOps aesthetic requires the presentation of information in the form of metrics that cannot be otherwise questioned or second-guessed.

Prioritizing bug fixes provides make-or-break influence over a software release. Mandating bug fixes prior to a release is a different responsibility. Getting bugs fixed in specific windows of the development cycle opens the opportunity for more and more interesting testing. It is obviously very important to unblock testing whenever possible, but there are frequent conflicts within the development team about when bug fixing officially begins. This is where priority must override the schedule.

When testing is blocked, testing goals are compromised. It is important to make testing goals an immobile force and to use metrics to define when two opposing forces may collide. To ensure the health of a release, testing must succeed over a certain number of days. If critical bugs are blocking that testing, the release can and should be delayed. Bug priority is required to ensure a sufficient amount of work is done to guarantee an acceptable—and successful—release.

The bottom line in the decision to release really is the bottom line, and the popularity and use of a software application is what will define its value. Whether or not this drives the financial or popular success of an effort, it is unquestionably the validator.

Release decisions must always acknowledge the presence of known customer-facing issues, and QA is frequently charged with assessing the readiness of the release. There will always be a lot of debate, but the data should speak for itself. This is where the bullet hits bone for application ownership. The data should identify what customers will see, when, and how often they will see it. An obscure bug in "Sign Up" is a lot different from an obscure bug in "Login" since the latter is a precursor to every session. Issues seen only in a dying operating system may limit exposure but also increase support costs. The goal for QA here should be to present the viable data appropriately and note issues for tracking after release.

Whether viewing DevOps as a specific role on a team or as a culture, the focus on engineering and quantified metrics gives teams the ability to make projections and trade-offs that empower product architects and management. In a way, it is a maturation of the software-development process that has been a long time coming.

No other science and few other industrial efforts make decisions without a large volume of data to help predictably validate the outcome. Where DevOps has connected the system administrative skill set to the development effort, moving that same ground shift up the chain into the quality engineering effort helps teams make more informed decisions on the development and release of their products. C

Related articles on queue.acm.org

The Antifragile Organization
Ariel Tseitlin
http://queue.acm.org/detail.cfm?id=2499552

Traipsing Through the QA Tools Desert
Terry Coatta
http://queue.acm.org/detail.cfm?id=1046955

Quality Assurance: Much More than Testing
Stuart Feldman
http://queue.acm.org/detail.cfm?id=1046943

References
1. Acra; http://acra.ch/.
2. Google Analytics. Crashes and exceptions—Android SDK, 2013; https://developers.google.com/analytics/devguides/collection/android/v2/exceptions.
3. Jones, R. How to hire DevOps. Gun.io, 2012; http://gun.io/blog/how-to-hire-devops/.
4. Mac Developer Library; https://developer.apple.com/library/mac/navigation/.
5. Meme Generator; http://memegenerator.net/instance/22605665.
6. Mueller, E. What's a DevOp? The Agile Admin, 2010; http://theagileadmin.com/2010/10/08/whats-a-devop/.
7. Rocchi, C. Overview of iOS crash reporting tools, Part 1. Ray Wenderlich, 2013; http://www.raywenderlich.com/33669/overview-of-ios-crash-reporting-tools-part-1.
8. Samovskiy, D. The rise of DevOps, 2010; http://www.somic.org/2010/03/02/the-rise-of-devops/.

James Roche is a software developer and engineering manager focusing on testing frameworks. His work spans 16 years on projects including Amazon's retail platform, Amazon Web Services; Adobe InDesign; and Adobe's Digital Publishing Suite. He is contributing to a prototype project for Adobe, looking forward to a release soon.