eGUIDE

Increasing Code Quality with Automation and Machine Learning

Launchable

In the struggle to ship code out the door faster and faster, are you also creating a potential future time bomb ignoring code quality? Sometimes it can be challenging to get buy-in about the value of high-quality code. Many short-sighted managers are only worried about the here and now - and don't see the growing snowball of technical debt that is accumulating and will eventually come rolling their way. This eGuide has articles, case studies, and other resources that speak to the value of code quality and how you can use automation and machine learning to optimize it.

Increasing Code Quality with Automation and Machine Learning

6 Steps for Succeeding with Test Automation in Agile

Lots of test automation efforts in agile software development fail, or at least do not maximize their potential. This article looks at two main reasons test automation may not live up to the expectations that testers and other stakeholders in the agile development process have, then outlines six steps to avoid falling into these traps. Here's how to succeed with test automation in an agile environment.

How to Build a Data-Driven DevOps Decision Making Culture

The growth in AI and machine learning solutions can enable teams to create a modern data-driven DevOps culture that streamlines development processes significantly, maximizes resources, and improves software quality.

Dealing with a Test Automation Bottleneck

The test team uses the test automation system to execute thousands of test cases because ... why not? The tests are running automatically, for free, so there is no incentive to improve test efficiency. Just run them all! But eventually, as more and more tests are added, the system becomes overloaded. Test runs are delayed and you get a bottleneck. Don't throw more money—or new systems—at the problem; do this instead.

The Evolution of DevOps Testing Tools

The DevOps ecosystem has seen two evolutions of tools, and we are on the cusp on the third evolution. Within these three evolutions of DevOps tools there are established tools in the various channel types, from coding and code analysis, to build and QA, and overall operations.

How AI is Transforming Software Testing

The time needed to complete the slew of required test cases directly conflicts with the fast pace driven by agile-like frameworks and continuous development. The exploration of alternative and superior testing methods, such as automation and AI, is now a necessity in order to keep pace and equip QA and test teams with augmented efficiencies.

How to Deliver Code Faster With Test Automation Powered by Machine Learning

How does predictive test selection help prioritize the most important tests to run, ultimately saving you time and resources while delivering software faster?

What's Our Job When the Machines Do Testing?

While many of the tasks across our diverse test practices are similar, each of our jobs has unique challenges, so priority will be determined by the specifics of our organizational context. So your job, when the machines can do testing, is to figure out what tasks you want them to do for you.

Defensive Design Strategies to Prevent Flaky Tests

More testing isn't always better testing. If your tests are unreliable, they can cause more harm than good. "Flaky tests" aren't as rare as we'd like to think, and there are a few things you can keep in mind to avoid them and help your development lifecycle move more smoothly.

6 Steps for Succeeding with Test Automation in Agile

Bas Dijkstra

In order to keep up with the ever-shorter release cycles that come with the adoption of agile software development, many development teams are embracing test automation as a means to continuously ensure that every software release conforms to the desired level of quality.

This is a significant shift from traditional software development practices, where testing was often stuck on at the end of the development process and seen as a process burden rather than a benefit. Testers working in an organization that has adopted agile software development, moved to a DevOps culture, and embraced continuous integration and continuous delivery are therefore required to have at least a basic level of understanding of how to effectively implement test automation as part of their daily activities.

Unfortunately, lots of test automation efforts in agile software development fail, or at least do not maximize their potential.

I'd like to explore what I think are the two most important reasons for test automation not living up to the expectations that testers and other stakeholders in the agile software development process have. Then, let's look at strategies and tactics for how to avoid falling into these traps in order to succeed with test automation in an agile environment.

Unreasonable Expectations

The first of the two main reasons I see so many automation efforts fail to reach their potential is because of the unreasonable expectations that precede its implementation. Too many team leads, development and project managers, and C-level executives (although other roles are not innocent, either) see test automation as the one-stop solution to all of their testing bottlenecks.

However, reality has shown over and over again that:

- Implementing test automation takes time, effort, and specific skills
- Automation is an activity that supports testers, not replaces them
- Far from every activity related to testing can be automated

Yet there's still a widespread belief that test automation somehow, magically and with the press of a button, will perform all of the required testing for you. When, after a couple of months of diligently building and running tests, this notion turns out to be a fantasy, the people who have been working hard on getting the automation to work are often scapegoated, and sometimes even laid off.

In my opinion, the best way testers and automation engineers can deal with this problem is to think and communicate before acting. Make sure that all stakeholders are on the same level with regards to what you can reasonably expect from automation. Take a look at previous efforts, both within your organization and in the wider software testing and development community, and learn from those experiences.

What should work? What will likely not? Don't expect automation to be the magic bullet that will solve all of your testing problems.

Even though it should not be a surprise that implementing automation takes time and effort, automation is still too often one of the first things to suffer when time becomes sparse.

Lack of Attention Dedicated to Automation

The other main cause of failing automation efforts is the lack of time that development teams (and, on a larger scale, whole organizations) allow for creating useful, maintainable, and effective automation solutions. Even though it should not be a surprise that implementing automation takes time and effort, automation is still too often one of the first things to suffer when time becomes sparse.

This applies to projects, but it also applies to teams working in an agile environment. Even though automation is high on the wish list of many a software development team, when the end of a sprint comes close, delivering features almost always takes precedence over automation.

Note that I do not think that this is necessarily a bad thing. In the end, it's the features that deliver value to the end-user, not the automated tests that help safeguard them working correctly. In the long term, however, teams that allow the releasing of features to take precedence over everything might just run out of breath while trying to make deadline after deadline. It's almost as if they forget that adopting an agile way of working is about creating software at a sustainable pace while receiving feedback early and often, not about delivering features at warp speed.

Not allowing for enough time to create a solid automation solution also has an unwanted side effect: If you don't award automation the priority it deserves (and that may not be the highest priority), it is unlikely that your team members will have enough time to become skilled automation engineers. I see automation as a craft, and like any other craft, it requires continuous learning and honing of your skills.

Making Automation Part of Your Agile Development Process

Now that we've detailed two main reasons for failing automation efforts, I'd like to propose a step-by-step guide to help you avoid these pitfalls and successfully implement test automation as part of your agile software development activities. This is by no means a definitive guide, and not all steps might apply equally to your situation. But following them might just help you to be more successful in your agile automation efforts.

1. Set reasonable expectations

As I said before, the success of any automation efforts starts with reasonable expectations. I find that asking and agreeing on the "why" is a good way to set these expectations. Why are we automating in the first place? Why do we think we need test automation at all?

Note that in my opinion, there are good and, well, less good answers to this question. "Because we want to be able to get fast feedback on every commit done by a developer" is a good reason, whereas "Because we don't want to have manual testing at all" is a prime example of a source of unreasonable expectations.

2. Treat test automation as software development

Make sure that all parties involved are aware that the introduction of test automation is essentially equal to introducing a software development project within your software development project.

This applies to both the project planning aspect (you should assign resources to it and allow for time spent on developing and maintaining the automation, etc.) as well as the technical implementation of it (you're writing code, so be sure to apply good development patterns and practices and respect the fact that test automation is a craft that requires specific skills).

3. Assign dedicated resources to automation

In order to be successful with test automation in your agile software development efforts, you'll need to make sure that the people responsible for creating and maintaining the automation both have the right skill set and get enough time to do so.

The number of people who will dedicate their time toward automation depends on a number of factors, including their skills, the type of automation that needs to be created, and the complexity of and risk associated with the application to be developed. If your organization currently does not employ enough people to fulfill your automation desires, or the people lack the required experience, temporarily hiring external experts might be an option worth considering to get you started.

4. Pick a starting point

Just like with any significantly large project, it might look like a daunting task to decide where to start with test automation. I have two pieces of advice here:

- Start either with some low-hanging fruit (this helps show stakeholders the added value of test automation quickly) or with a part of your application associated with the highest risk or the highest impact of defects.
- Try to avoid starting with end-to-end test automation, like using Selenium. While this might seem like the obvious choice when you're looking to write automated regression tests, this type of test is the hardest to write, the slowest to execute, and the most prone to fail, either due to changes in the application under test or to false positives caused by synchronization or environment (e.g., test data) issues. Instead, see if you can create a solid set of meaningful unit tests, or leverage API-level testing to verify critical business logic.

5. Make automation part of your definition of done

When you're working in an agile setting, it makes sense to make test automation an integral part of your definition of done for a given feature. Try to avoid these two traps, though:

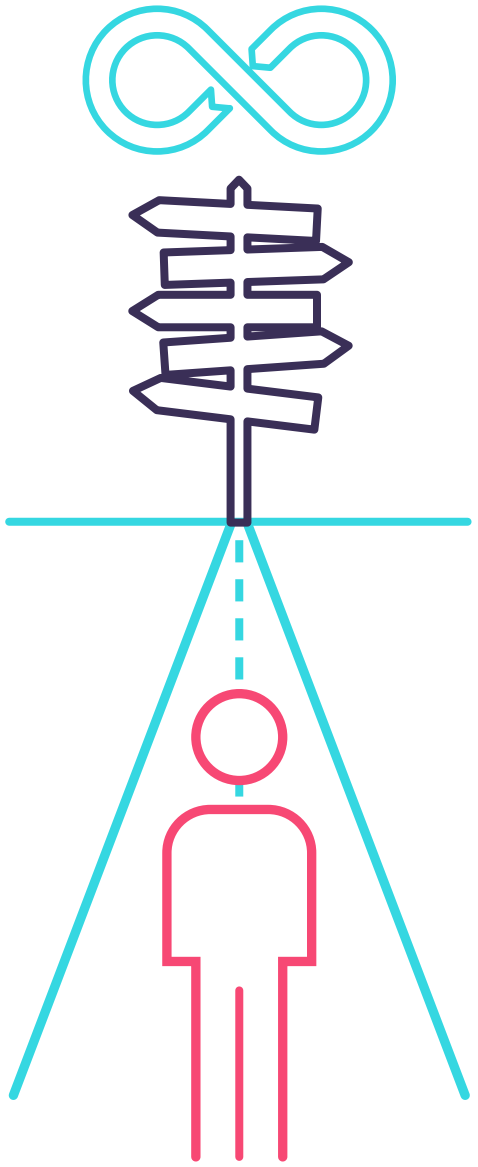
- Including a statement like "All tests should be automated" or "We should have automation in place for every feature delivered"—sometimes automation doesn't make sense, is cumbersome, or is even downright impossible to create. Instead, opt for statements like "Existing automation is updated to reflect changes brought by this feature" or "Additional automation has been created where deemed relevant and useful by the development team."
- Relying on percentages—"100 percent code coverage" is an empty statement. It says nothing about the quality or relevance of the tests. Likewise, "80 percent of all tests done are automated" does not make sense, either. For one, this relies on a one-to-one translation of tests executed to automated tests, an approach that has time and again proven to be ineffective. But more importantly, how do you define that 80 percent in the first place? Is that 80 percent of everything that can be automated or of all testing performed? I think you catch my drift here.

6. Learn and adjust

This should not come as a surprise anymore: Test automation is a software development activity, and when you're doing this in an agile way of working, it makes perfect sense to apply the agile principle of fast feedback, quick evaluation, and learning on the go.

You don't have to get it right from the start! Just like with your application under test, take the time to experiment, evaluate early and often, learn from any mistakes, and stick with what works. Over time, and with the right amount of nurturing, this should lead to an automation approach that fits hand-in-glove with your software development efforts.

Please note that every situation is different, and what works for one organization might not work as well somewhere else. Having said that, I sincerely believe that the information above will be helpful to most organizations that are struggling with effective testing and are therefore looking toward automation as a means of improving their agile testing efforts.



How to Build a Data-Driven DevOps Decision Making Culture

Adopting a Data-Driven Approach to Improving CICD Pipeline with AI

By Harpreet Singh

There is no doubt that companies today are awash in information. By itself data is of little use, but businesses who correlate it, gain insights into their operations and can change themselves for the better. This is both the opportunity as well as the challenge in creating **Data-driven DevOps**. Development teams have an abundance of data to leverage, and therefore an abundance of potential to speed up application testing and delivery, using data to drive software quality and velocity. Regardless of your team or organization size, adopting a data driven culture can start today.

Take Advantage of Existing Data

The first step in building a data drive DevOps culture requires organizations and teams to understand what data points and metrics they already have and what they are missing. Tools including Jenkins, generate growing volumes of Continuous Integration (CI) data outputs. Development teams pull that data from hundreds of libraries, modules, and services, via scripts running during various parts of the workflow. The information sped up development's front-end, but more work is needed on the back end. Taking advantage of existing data can come in different forms - one example is correlating pass/fail data of your tests to **flaky tests**. This existing data would be helpful to identify how flaky a test is likely to be

Automate Data Delivery to the Right People

Holes are also evident as developers process feedback. Individuals adding new features and fixing bugs often do not get the data they need in a timely manner because it is generated at the end rather during the testing process. Getting the developers the notifications when the first test failed would be an impactful adjustment. Another option is updating building notifications to be smarter - identifying and messaging the right developer. Running a **git-blame** on lines of code generates a build error and immediately notifies the Git author of the failure. They work on a fix immediately rather than wait hours for the full test results.

Simple Data Driven Solutions Add Up

Incorporating data into your DevOps cycle doesn't have to be a large undertaking. There are often low-hanging fruit opportunities that you can incorporate to improve productivity or efficiency. For instance, if notifications for test failures go to all engineers, take a look at your failure notifications and identify any that are caused by infrastructure failures (server disk space or database down). Those don't necessarily need to go out to all of your engineers, but likely rather a specific group or person. Setting up a simple automatic scan of the last 50 lines of the build log for specific keywords around those error types, and sending an automatic notification to the right people would reduce notifications to your developers and lessen the notification fatigue or frustration caused by errors.

Factor in the Future of DevOps

When adopting data driven DevOps, we can all learn from leading organizations incorporating it at scale and the emerging trends. Companies including Google and Facebook **have shared how they are harnessing machine learning** to prioritize tests to run for each code change. A new approach to test impact analysis is **Predictive Test Selection**, which combats long cycle times by tracking your builds and selecting a subset of tests that are critical to run. We've also done a **deep dive webinar into the state of the art of Flaky Tests**, examining how teams are tackling the prevalent pest.

Measuring Data Driven DevOps Progress

There are at least two metrics to track when incorporating data drive DevOps into your organization. The first are the key performance metrics that are helpful to your team to track and celebrate internally, like test runtime, time from code push to build complete, and code coverage. The second are metrics to show progress to leadership, which can include lead time for code changes, time to restore service, and change failure rate.

Development teams are able to collect more information today than ever before. Artificial Intelligence and machine learning products enable them to correlate that information in new ways and find solutions to long standing problems, such as slowdowns in Continuous Delivery. This expanding solutions enable teams to create a modern data driven DevOps culture that streamlines development processes significantly, maximizes resources, and improves software quality.

Artificial Intelligence and machine learning products enable them [development teams] to correlate that information in new ways and find solutions to long standing problems, such as slowdowns in Continuous Delivery.

Dealing with a Test Automation Bottleneck

By Michael Stahl

It's usually difficult to get management approval for recruiting more people. It is much easier, on the other hand, to get approval to spend money, especially when you work at a successful company with a healthy budget. Not to say that you use money as wall-to-wall carpeting, but if you can justify the expense by a positive ROI analysis, in most companies there is a good chance the request will be approved.

However, this can actually have some negative side effects, at least in the testing world.

Let's imagine that we built a wonderful test automation system. The system receives a list of tests to run and executes them in parallel on a large number of computers. The system is so good that it encourages testers to automate more tests, as doing so clears up the testers' time to do some real engineering work, like think of new and interesting test cases, develop tools that allow deeper testing, and participate actively in reviews. Besides, it's much more fun to write new code or develop a script than to execute the same test for the seventeenth time.

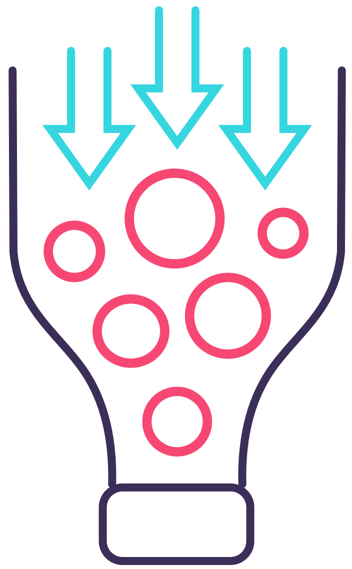
The test team uses the automated system to execute thousands of test cases—even on the daily test cycle—because … why not? The tests are running automatically, for free, so there is no reason to take the risk of missing a possible regression. Just run them all! It even improves the ROI of the automated system.

The system is indeed lightning-fast, but eventually, as more and more tests are added, it becomes overloaded. Test runs are delayed and the test automation system becomes a bottleneck. At a certain point, the program managers fail to get test results on time, and they start raising flags—and hell.

Everyone realizes we must go over all the test cases and define priorities: Which test case must be executed each day? What's enough to run once a week? Which tests are inefficient and could be made to run significantly faster?

But with thousands of test cases in the system, such prioritization or optimization of effort is a significant investment, and no one has the time to do it. Everyone is swamped with executing tests that are still manual, investigating failures, or testing possible bug fixes. If only we had some time to work on this! If only we could recruit a few junior engineers to do execution and free the senior engineers to improve the existing tests!

Recruiting new people is out of the question. But there is a solution that, in one fell swoop, will reduce the test cycle time by 50 percent and solve the problem:



Double the number of computers controlled by the automation system. This is practical and it's easy to prove the ROI, if only by showing that version release time will be shortened by half the test cycle time. And besides, it's easier to get approval to spend money than to get an open req. It also takes much less time to achieve results: A new person must learn the product, understand the technology, and find out how to operate the coffee machine. A new computer only needs a power line and network connection to start executing.

So we buy more computers and connect them to the system, and indeed, the bottleneck miraculously dis-

Any time we run a test cycle, some tests will fail—not due to a product problem, but to some problem in the test itself, the test environment, or the automation system.

appears … only to reappear a few months later, when more tests are added to the system.

One of the reasons for this phenomenon is the tragedy of the commons. The principle, according to Wikipedia, describes "a situation in a shared-resource system where individual users acting independently according to their own self-interest behave contrary to the common good of all users by depleting or spoiling that resource through their collective action." In an organization-wide test automation system where each team can submit unlimited test cases for execution, the principle explains why no team has an incentive to become more efficient.

Let me illustrate with an example.

A certain project has five test teams that share a common test automation system. The system, as expected, is overloaded and delays test cycle schedules. Each team understands that it should invest effort in reducing test time, but something is blocking their taking action.

Assume that if each team were given sole use of the system, running each team's test would take twelve hours apiece. Since all the teams are sharing the test system, the overall duration of a test cycle would be sixty hours. Team A decided to tackle the overload problem by reducing their test cycle time by 50 percent, to six hours. They improved the test time of all tests, removed redundant tests and merged others to save setup time, and did needed modifications to make their tests robust and save time lost for false failures. Eventually, they met their goal.

The overall test cycle now takes fifty-four hours. So, for the superhuman effort done by team A, the overall reduction in test cycle time was 10 percent. Since the other teams continue business as usual, the time saved by team A will quickly be consumed by new, inefficient tests added by the other teams.

This reality means that no team has an incentive to invest in improving their test efficiency.

Another problem that excessive testing brings is an increase in the number of false failures. Any time we run a test cycle, some tests will fail—not due to a product problem, but to some problem in the test itself, the test environment, or the automation system. It could be a hard drive crash, an overheated motherboard, or a flaky connection. The number of such cases depends on the skill level of the testers developing the automated tests and on the robustness of the test framework, but there always is some level of incorrect failure "noise."

In many cases, a rerun of the test passes. And while the percent of bogus failures may stay constant, the more tests we load to the system, the more false fails will occur, with the associated cost of engineering investigation and correction time. This further reduces the availability of resources to work on test efficiency improvements.

I have witnessed this process taking place not only in testing, but also in DevOps. Each check-in of code triggers a set of continuous integration (CI) tests. CI runs on a central, automated system, and the tendency of test time to get longer happens there as well.

It happens naturally because, as more functionality is added to the product, more tests must be added to the CI test suite. It can also happen when a new project is added in parallel to existing legacy projects. Whatever the reason, once the system is overloaded, delays start to occur. CI results that took only a few minutes when first implemented now take half an hour. The developers become accustomed to doing check-in before lunch. A few months later, lunchtime is not enough anymore. The developers get used to doing check-in at the end of the day and letting CI tests run at night. A few months later ... You get the idea. And once again, DevOps solves the problem by buying more computers to execute CI tests.

Once this cycle starts, it is very hard to stop it. The daily pressures and the need to make fast improvements mean that throwing money at the problem is almost always the most logical solution. Every now and then some teams manage to hold a blitz of improvements that makes a difference for a few months, but then the problems return.

Since it is so hard to fix the problem after it's started, prevention is a better approach. Here are three prevention measures you should consider:

1. Invest effort in making sure only robust tests are added to the automation system. Brittle tests will later fail intermittently during runs, wasting engineering time in investigations. This means that every automated test must pass a certain quality bar before being admitted to the system.

One approach is to create a checklist that each automated test must pass. This checklist should be realistic and short. If the list is too long, it will not be used. (Many years ago I wrote a checklist that took three days to complete; I will let you guess how much use it saw. Half an hour is a much better goal.) Once a workable checklist is in place, it will influence the design and implementation stages. Since test developers know that eventually they will be evaluated against the checklist, many of the checklist requirements will be taken care of during the test implementation phase.

2. Be aware of the impact of a long test time, and invest thought and effort to reduce the time of each test, even when it seems unnecessary.

When I tested Wi-Fi cards, almost all the tests started with a "Connect to the network" step. To ensure that a connection was achieved, we had a CheckConnection() library function, and almost every test used it. The function guaranteed a good connection by transferring a file back and forth between the Wi-Fi client and the network. To the engineer who designed it, it was a trivial check, with a runtime that looked very reasonable: 15 seconds. For one test, this is indeed not a problem, , but when used by a thousand tests cases, this routine alone was responsible for four hours of test time in each test cycle. Another example is adding a wait() to the test code to give the system time to finish some activity or to stabilize. If a wait time of one second is enough, a wait time of ten seconds, happening again and again, will eventually buy itself a few more computers.

The best time to streamline a test case is during the test development stage, when the test engineer knows all the considerations and details of the test case. Later, it is much harder to go over all the tests, one by one, and look for possible improvements.

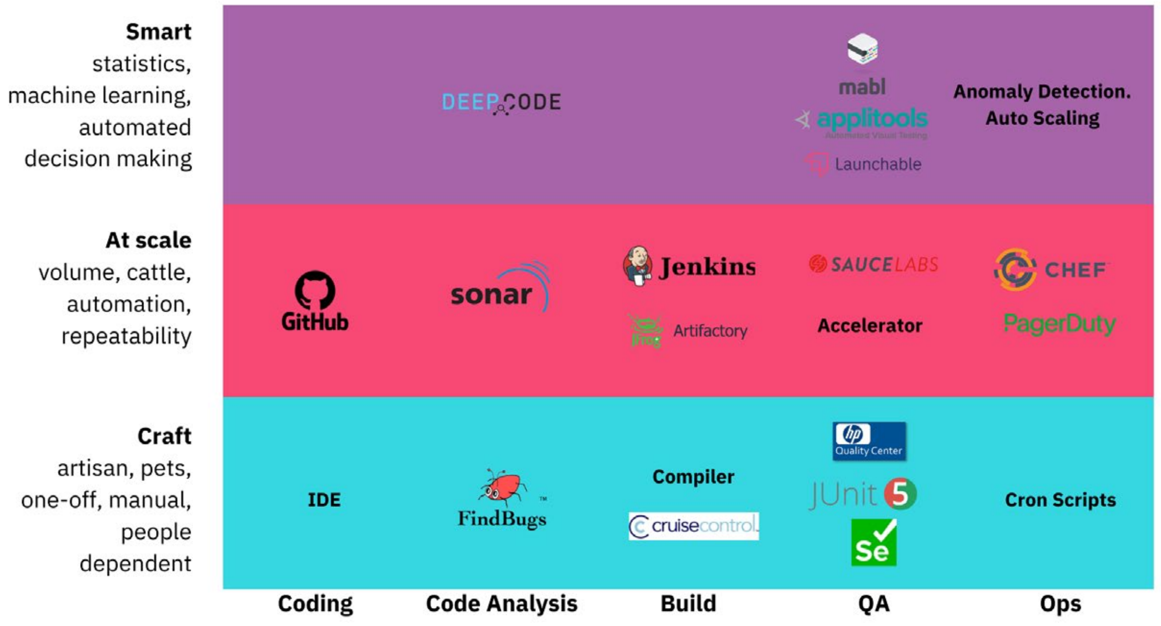
3. Define up front how much test time each test team gets on the system. This targets the tragedy of the commons and can be implemented both as a preventative measure and as an effective route to take when experiencing a bottleneck problem. Practically, this is done by allocating test machines to each team.

Going back to the previous example, if the system has thirty computers, each of the six teams would get five computers and decide how to use this resource. For team A, the efficiency effort now pays back big time because their test cycle only takes half the time of the other teams. To start with, it looks great in the reports. Additionally, each saved hour is now at team A's disposal to execute new tests. It creates a situation where each team is incentivized to improve their test efficiency.

So the next time you are tempted to solve a problem by throwing money at it, stop for a minute and ask yourself if you are a victim of the tragedy of the commons. Would spending money really get to the root cause of the problem, or would it make better sense to spend effort in making processes more efficient?

The Evolution of DevOps Testing Tools

The Next Generation of CICD Testing Automation Aims at Increasing Velocity



By Alastair Wilkes

Engineering teams face the pressure to continuously deliver new features fast. Theoretically developers could pump out new iterations multiple times a day, but parts of the DevOps cycle, including testing and Quality Assurance, aren't as fully optimized for CICD as other facets of the cycle. The next wave in machine learning testing solutions are emerging to seize this opportunity.

The DevOps ecosystem has seen two evolutions of tools, and we are on the cusp on the third evolution. Within these three evolutions of DevOps tools there are established tools in the various channel types, from coding and code analysis, to build and QA, and overall operations.

The first evolution of tools was the craft, artisan one-off manual tool approach. Within that first, craft level QA section, **Selenium** and **JUnit** laid the foundation of testing tools. The next iteration was the adoption of automated, at-scale tools that focus on repeatability. As the need for scalability grew, automated tools like Sauce Labs and Accelerator evolved. Automated testing tools have enabled us to achieve repeatable and scalable testing, but there are still significant hindrances to velocity.

The next evolution of DevOps testing tools is advancing the intelligence of the tools, making them smarter by utilizing AI and machine learning. Maturing their automation capabilities, the next phase in these tools will take the processing of tests to a new level by also prioritizing test suites further speed up test runs.

DevOps Testing Tool Limitations

Although we've come far with testing automation, several factors continue to dampen the velocity of larger testing cycles. Organizations might be working to shift left, but test suites have varying load sizes and the more complex tests require more time. User accep-

The next evolution of smarter testing automation will harnesses machine promises to bridge that gap and speed up development.

tance testing and integration tests can take time because of the variables included. Truly shifting left and running tests sooner would result in catching risks earlier.

Along with long test times taxing testing cycles, flaky tests plague development teams and waste time and resources. Typically when flaky tests are suspected, developers have to sift through large volumes of testing data to determine the problem source, and many times, there are simply not enough clear signals.

Given today's application design complexity and the dynamic nature of run time decisions, the only way to solve the problem is through deepening automation functionalities. Progressing testing automation relies on the creation of new, intelligent machine learning testing tools.

The Hurdle to the Next Evolution of DevOps Testing Automation

Developing a machine learning DevOps testing tool requires a great deal of specialized expertise from individuals trained to be data scientists. Companies need individuals like an AI QA strategist, someone who can weave AI into the DevOps process. The reality is that there is not much of that expertise available in the market. High end consulting firms, like McKinseys, try to help but the void is large.

Companies must now develop that expertise themselves, which is impractical. A self-built machine learning stack is extremely complex and expensive to deploy. The individuals who are familiar with it are rare and expensive. Unless a business has the technical depth of the Googles of the world, they lack the infrastructure needed to build their own in-house practice. The task is well beyond the purview of small and medium companies.

Corporations are trying to drive investments in pure math skills into QA teams and embed data scientists into each one. But that process may be a long term fix — maybe five to seven years — so what do they do today?

The Current Landscape of Intelligence in DevOps Testing Solutions

The latest level of DevOps tools is the next generation of automation. It builds on the automation tools and elevates the capabilities by incorporating machine learning and decision making by harnessing statistics. This next evolution consists of a number of startups emerging, and their tools can be sorted into two categories: scaling tests and prioritizing tests.

DevOps Solutions for Generating Tests

This category focuses on the issue of not enough tests and is more mature with companies like Mabl. AI is used to generate tests where none exist primarily in the UX testing space. This category addresses specific types of testing, say a bot that generates tests for mobile web pages. The more tests get automatically generated, determining what tests are effective becomes a problem.

DevOps Testing Solutions for Identifying Critical Tests to Run

This category focuses on solving the problem of having too many tests that take too long. A more nascent category, teams like **Facebook** have been using predictive test selection tools to drive efficiency in the testing process. Companies like Launchable are trying to bring this to the market. **Predictive Test Selection** that combines coding and quality into one practice. We do this by taking data from your Git changes and past test results to quickly deduce where problems lie, so developers release high quality code faster. Launchable is a layer on top of your existing CI pipeline and tooling. The Launchable CLI **interfaces with your existing build and test tools like Selenium, Maven, Gradle, and Ant**.

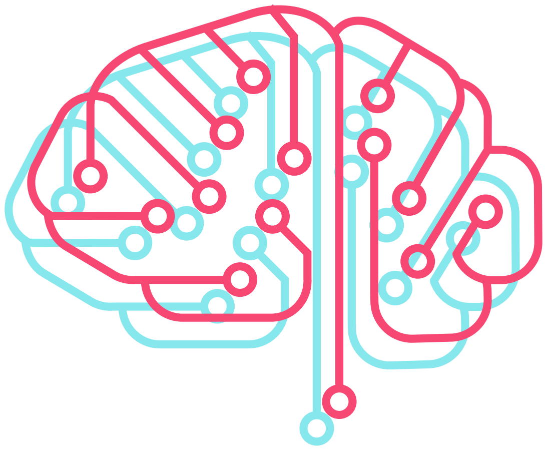
Enterprises need to push new software releases out at a faster rate. Testing velocity has been a DevOps opportunity to capitalize on. The next evolution of smarter testing automation will harnesses machine promises to bridge that gap and speed up development.

How AI Is Transforming Software Testing

By Raj Subrameyer

From mobile apps that command home appliances to virtual reality, the digital revolution is expanding to cover every aspect of the human experience. The majority of teams developing this technology are using the agile rapid delivery framework, which equates to roughly a new launch every two weeks. Of course, those apps and devices must be tested to ensure an optimal experience for the end-user before each launch, but at that pace, manual testing is simply inadequate.

The time needed to complete the slew of required test cases directly conflicts with the fast pace driven by agile-like frameworks and continuous development. The exploration of alternative and superior testing methods, such as automation and AI, is now a necessity in order to keep pace and equip QA and test teams with augmented efficiencies.



AI is showing great potential in identifying testing defects quickly and eliminating human intervention. Such advances have provided the capability to determine how a product will perform at both the machine level and the data-server level. And in the current era, where emphasis is on DevOps and continuous integration, delivery, and testing, AI can speed up these processes and make them more efficient.

Just like automation tools already have, AI is going to aid in the overall testing effort.

AI has a proven ability to function with more collective intelligence, speed, and scale than even the best-funded app teams of today. With continuous development setting an ever more aggressive pace, along with the combined pressure from AI-inspired automation, robots, and chatbots, it begs the question: Are testing and QA teams under siege? Are QA roles in jeopardy of being phased out or replaced, similar to the manufacturing industry?

Over the past decade, technologies have evolved drastically, but one aspect that remains constant is human testers' interaction with them. The same holds true for AI. To train the AI, we need good input-output combinations, which we call training data sets. We need to choose a training data set carefully, as the AI starts learning from it and creating relationships based on what we give to it. It is also important to monitor how the AI is learning as we give it different training data sets because this is going to be vital to how the software is going to be tested. We would still need human involvement in this training.

It is important to ensure that the security, privacy, and ethical aspects of the AI software are not compromised. All these factors contribute to better testability of the software. We need humans for this, too.

We will continue to do exploratory testing manually, because there are some things still best left to human minds. But we increasingly will use AI to automate processes while we do this exploration. Just like automation tools, AI would not replace manual testing, but instead complement it.

With time, the maturity level of AI automation will significantly grow. Teams will start seeing benefits and will realize the need to shift their thinking in terms of how they view software systems and how they can be tested. The future looks bright for AI-based testing solutions.

How to deliver code faster with test automation powered by machine learning

By Alastair Wilkes

Problem: Long test feedback delays

Most software teams seek an ideal state: a green build for every commit on the main branch. Every commit can be released, and developers never check out a broken build. To achieve this, a new project with a small test suite might run all of its tests on every commit or push, providing very quick feedback to developers.

This really is an ideal state, though. Over time, projects inevitably grow in scope leading to more tests. More test suites are often added, too, which changes test growth from linear to exponential. These additional tests drive up the overall test feedback delay for developers.

To keep the execution time down, you might provision more cloud testing infrastructure, but this is a non-starter for many teams such as those building embedded software.

Eventually, you need to choose which tests to run when. You need to trade off risk for feedback time. A natural consequence is to keep the fast suites earlier and move the slower suites later, as Martin Fowler explains:[1]

There is a tension around the degree of testing to provide sufficient confidence of health. Many more thorough tests require a lot of time to run, delaying feedback on whether the commit is healthy. Teams handle this by separating tests into multiple stages on a Deployment Pipeline. The first stage of these tests should run quickly, usually no more than ten minutes, but still be reasonably comprehensive. I refer to such a suite as the commit suite (although it's often referred to as "the unit tests" since the commit suite usually is mostly Unit Tests).

Ideally the full range of tests should be run on every commit. However if the tests are slow, for example performance tests that need to soak a server for a couple of hours, that isn't practical. These days teams can usually build a commit suite that can run on every commit, and run later stages of the deployment pipeline as often as they can.

As projects grow in scope, test execution times can snowball!

This means developers get fast feedback, but only for some tests. **Tests are typically distributed into the pipeline in a *coarse-grained* manner: how long an entire suite takes to run.** Distributing this way does mean some tests, like unit tests as Martin says above, do get run earlier. But if your quick unit tests usually pass but your long-running end-to-end tests fail often, developers may find themselves waiting even longer for the most important test failures.

1. *https://martinfowler.com/articles/branching-patterns.html#healthy-branch*

A predictive test selection algorithm selects the most important tests to run based on the actual changes being tested.

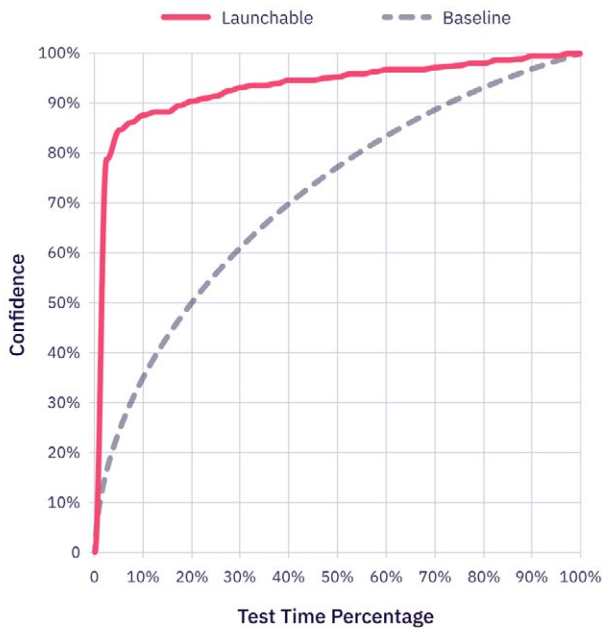
Solution: Apply machine learning to test execution

These concepts come together in the form of predictive test selection. A predictive test selection algorithm selects the most important tests to run based on the actual changes being tested. In the last few years, several published research papers have revealed the potential for predictive test selection, and now we're seeing examples of industry applications from companies like Facebook.[2] To achieve this,various historical inputs ('features') are used to build a predictive model that can then be queried for test cases in real time upon every test execution. Depending on the algorithm, these inputs could be information like file paths changed, number of lines changed, relative location of modified lines, author information, test duration, or any other metadata that might be relevant.

At Launchable, we want to unlock this potential for all software teams.

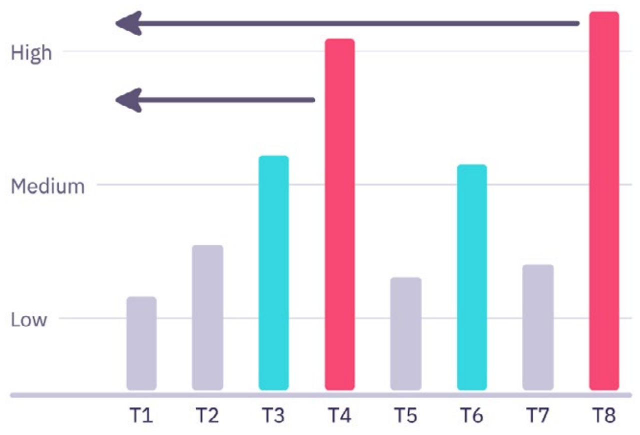
Let's see how it works in practice. First, we create a predictive model using historical data. We evaluate the quality of our predictive models using something we call a confidence curve. When we run test data through the model, this curve shows the percentage of tests that must be run in order to achieve a desired confidence level. Confidence tells us how far along the model thinks tests have come. It's modeled after the intuitive sense of the word: how much confidence do we currently have in the software being tested, given that we've already run some tests and have not run others? If it's 100%, it means we are pretty darn sure that the software has no failing tests.

For example, here's a confidence curve for a leading car company's pre-merge test suite from one of our case studies:



This curve immediately reveals the value of an intelligently selected subset: **with predictive test selection, a team can run a fraction of a test suite and still achieve high confidence.**

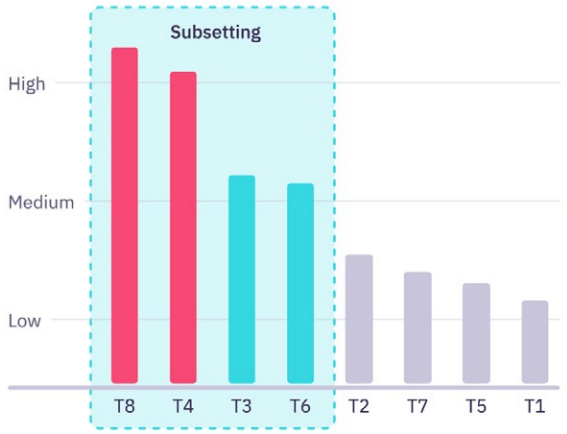
Conceptually, our approach starts by reordering tests. By default, most test suites run in an arbitrary order. This means many failures aren't caught until halfway through a run or later. Using the model created earlier, we can now reorder tests based on the changes being tested to execute the most important tests first.



Reordered execution alone is useful for reducing feedback delay for developers, because now the most important tests get run before less important ones. This means that any real-time feedback they receive (like chat notifications or streaming build logs) now comes in much sooner.

2. https://engineering.fb.com/developer-tools/predictive-test-selection/

We then take that concept one step further by removing the least important tests from execution. We call this an adaptive subset: given a set of changes, a set of tests, and an optimization target (e.g. maximum execution time or minimum confidence) the Launchable service returns a subset of those tests, specifically reordered for optimal execution.



This method reduces execution time with minimal introduction of risk. The example confidence curve above shows how we can achieve 90% confidence in only 20% of the execution time. Other example curves perform even better, such as 90% confidence in 10% time. This remarkable reduction in execution time with minimal additional risk lets teams distribute their test suite in a more finely grained manner, dramatically reducing test feedback delay.

Real world applications: delivering faster and reducing risk

However, let's go back to our original problem of ***coarse-grained*** test distribution. In contrast, **adaptive subsets enable teams to distribute their tests in a *fine-grained* manner**. They can run the tests that are more likely to fail earlier and tests that are less likely to fail later, instead of being limited to distributing by suite.

This approach doesn't just reduce feedback delay – it also reduces risk. The later a problematic test runs, the more time the risk of failure exists in the system. Adaptive subsets shift the highest risk tests earlier in the process to help teams eliminate risk while the blast radius is still small.

For example, a team can select a small subset of a long running UI test suite. The shorter execution time lets them run this suite more frequently, earlier in the lifecycle. This would provide quicker feedback on the most important tests while the full suite still runs later. Similarly, another team could shift some tests right by only running a subset of an existing suite and then running the entire suite later in the lifecycle.

Adaptive subsets are also a great solution for teams with fixed testing resources. In particular we see this with teams writing embedded software, running tests on physical hardware that is hard to acquire. Those teams don't want to waste those precious resources running tests that don't matter. Similarly, adaptive subsets can also reduce testing infrastructure costs and reduce queueing time; if you're running fewer tests (but maintaining high confidence), you're able to get more out of the hardware you already have or even reduce your spend.

This remarkable reduction in execution time with minimal additional risk lets teams distribute their test suite in a more finely grained manner, dramatically reducing test feedback delay.

What's Our Job When the Machines Do Testing?

By Geoff Meyer

The application of analytics, artificial intelligence (AI), and machine learning are transforming jobs in industries once thought to be "safe" from automation.

Law firms are using AI-based machines to do the research that used to define the role of first-year lawyers. Trading decisions for our retirement accounts are managed by algorithmic robots crunching through massive amounts of historical data and real-time market metrics. The **progress of the autonomous vehicle**, underpinned by AI, is on the verge of replacing drivers across the transportation sector. What businesses in industry after industry have uncovered is this: Processes that have large amounts of data, a representative model, and a generally understood set of rules are candidates for automation.

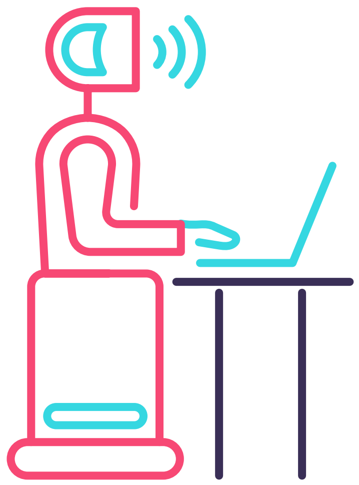
Geoff Colvin, the author of ***Humans Are Underrated***, points to the first-year lawyers described earlier as a model for those of us whose job consists of analysis, subtle interpretation, strategizing, and persuasion. These jobs are gradually—sometimes entirely—being transformed as increasing sets of tasks are delegated to a "smart assistant."

Our job in software testing is also composed of many tasks. From the nuances of requirements elaboration, the back-and-forth in establishing "expected behavior," test case analysis, and the other numerous cognitive tasks that we deal with, it's a safe bet that our jobs won't be taken over by machines anytime soon. However, for the reasons cited by Colvin, those of us in the test industry would be wise to heed cross-industry applications of analytics and machine learning and begin staking out the proper role of the machine in our testing domain.

Jason Arbon published a test autonomy maturity model, assigning a measure of an organization's automation capabilities. L0 (manual testing) is entirely in the mode of manual testing. L1 (scripted testing) has a firmly entrenched culture of scripting regression suites. L2 and above (exploratory bots, as well as human-directed, generative, and fully autonomous testing) are for organizations that have started to move into the realm of automating their cognitive and complex tasks.

For an L1 test organization well-versed in a world of automation, whether it be scripting of regression tests or the automation of build and deployment processes, analytics and machine learning represent the next generation of automation. And although it's just another tool to be considered for our testing toolbox, it's one that opens up the possibility of automating tasks in our job that we may never have imagined.

While many of the tasks across our diverse test practices are similar, each of our jobs has unique challenges,



so priority will be determined by the specifics of our organizational context. So your job, when the machines can do testing, is to **figure out what tasks you want them to do** for you.

Whether your high-priority pain points include triaging a bounty of automated test failures, avoiding wasted effort researching a defect only to find it is a duplicate, rapidly predicting the root cause of a failure, deciding whether to automate or retire a test case, or increasing regression test coverage in a time-constrained period, there's a good chance that **employing a machine as your smart assistant** might be the next automation approach for you to consider.

Defensive Design Strategies to Prevent Flaky Tests

By Steve Berczuk

Agile code is at the heart of agile software development. It's not enough to embrace change in the requirements process; you need to be able to deliver incremental changes along the way, since agile methods deal with uncertainty in the product space by enabling stakeholders to inspect and adapt. Without working code that can be delivered into production quickly, it's hard to inspect things in a meaningful way.

There are many elements to agile code. Designing with patterns to allow for flexibility with minimal overhead is essential, but not enough. You also need testing in order to get feedback.

More testing isn't always better testing, though. If your tests are unreliable, they can cause more harm than good. "Flaky tests" aren't as rare as we'd like to think, and there are a few things you can keep in mind to avoid them and help your development lifecycle move more smoothly.

Flaky tests could be the result of issues in the code, but more often they are the result of assumptions in the test code that lead to non-relatable results.

Flaky tests are tests that fail intermittently, and whose failures don't really provide useful indications of significant errors in the product code. Flaky tests could be the result of issues in the code, but more often they are the result of assumptions in the test code that lead to non-relatable results. Common examples are UI tests that rely on identifying elements based on positions, or data-driven tests that make assumptions about the initial state of the data without verifying it.

While integration tests are often the most likely to exhibit flakiness, unit tests are not immune. Unit tests based on time computations can fail on time zone transitions, or if a legacy test is written with the assumption of a future date that is fixed.

There are many reasons that tests can fail intermittently, and some can be easily avoided by applying good defensive design strategies. Sometimes the problem is that the test is too simple—for example, comparing a response to an API call as a string, when you have no control over order or even whitespace. Instead, compare fields in a structured object, checking only for the fields you expect.

Sometimes the issue is testing something irrelevant. Comparing all the fields in a response rather than just the ones that matter to the client that you are testing can lead to test failures that don't imply code failures.

Stateful integration tests can fail when you make assumptions about the data that you can't verify. Ideally, you would start with a known external system state, but if that's not possible, consider structuring tests so that the query can examine the current state, make a change, and then examine the new state.

While one can argue that tests that provide more noise than signal should just be skipped, that is not always a good practice. Even very simple tests that provide valuable insight can be flaky if not written well.

Test code **is** code, and it should be treated with the same level of good design that you treat production code. It should be adaptable and only make assumptions that it verifies. By making this a practice, you can keep all of your code more agile.

About Launchable

HOW LAUNCHABLE WORKS

BLOG

WEBINARS

YOUTUBE

CONTACT LAUNCHABLE

info@launchableinc.com www.launchableinc.com