MOBILE LABS

SPECIAL REPORT

Continuous Testing for Mobile

In the race to deliver new product to market, organizations are finding more and more that they need to adopt every competitive advantage they can get. One option for releasing software faster, reducing integration problems, and lowering costs is continuous testing.

In this eGuide, you will explore the reasons that organizations are embracing continuous testing, best practices for getting continuous testing right, how a DevOps approach can help improve both your quality and your speed, and other continuous testing resources.

In this Continuous Testing for Mobile Special Report

Bringing Continuous Testing to Your Organization

Consumers are more tech-savvy than they've ever been and less tolerant of defects, so applications suffering from bad behavior can have extremely negative effects on your brand. To reduce these risks, organizations are doubling down on their quality initiatives, and the software development industry is embracing continuous testing as a mainstream activity. So how do you bring continuous testing to your organization?

Why You Need Continuous Testing in DevOps

DevOps is more than adopting the right set of tools; it's a cultural shift that incorporates testing at each stage of the agile project lifecycle. Continuous testing is key to unlocking this culture change because it weaves testing activities into every part of the software design, development, and deployment processes, which helps everyone involved communicate more, collaborate better, and innovate faster.

How Continuous Testing Is Done in DevOps

DevOps does speed up your processes and make them more efficient, but companies must focus on quality as well as speed. QA should not live outside the DevOps environment; it should be a fundamental part. If your DevOps ambitions have started with only the development and operations teams, it's not too late to loop in testing. You must integrate QA into the lifecycle in order to truly achieve DevOps benefits.

A DevOps Approach to Mobile App Development

The DevOps approach has completely transformed the mobile app development scenario, and it's a good option for any business looking to create an app. DevOps facilitates collaboration between developers, operations staff, and project managers to better fulfill the objectives of an enterprise app development project. In order to adopt a DevOps approach to your app development, you will need to consider 5 important factors.

5 Mistakes to Avoid When Beginning Mobile Continuous Testing

Looking to institute continuous testing within your organization? Successful planning can yield a smooth and precise continuous testing strategy that increases performance, app quality, DevOps and overall agility. Through his work with enterprise mobility teams, see what the author has identified as the five common roadblocks that mobile developers and testers often stumble over when starting out and how to avoid them.

Why Building a Continuous Delivery Pipeline is like a game of Mousetrap

Do you remember the board game Mousetrap? If you're familiar with this game, then you know that each action in the game sets off an entire chain reaction that is necessary for you to capture your opponents' mice. Each step in the game is dependent on the step right before it and right after it. Building a continuous delivery process flow is similar. You know that you have a series of steps that must be created and successfully executed to build a mobile application. But, along the way, your team may find issues that need to be addressed quickly.

Additional Continuous Testing Resources

Bringing Continuous Testing to Your Organization

By Chris Colosimo

Businesses are focusing more and more on customer experiences as part of their go-to-market strategy, and a key piece of the customer's experience is their ability to traverse through your software in a quick and seamless way. Consumers are more tech-savvy than they've ever been, and less tolerant of defects, so applications suffering from bad behavior can have extremely negative effects on your brand.

To reduce these risks, organizations are doubling down on their quality initiatives, and the software development industry is embracing continuous testing as a mainstream activity.

What Is Continuous Testing?

Continuous testing is a principle of software testing in which all your tests are executing all the time, providing continuous feedback into the quality and health of your applications.

In order to achieve continuous testing, organizations must first adopt test automation. There are many types of test automation, spanning the breadth of an application starting at the UI layer, through the middleware systems, and even into back-end systems. Understanding how to bring in these different types of test automation practices as efficiently as possible enables you to move toward the path of continuous testing.

It All Starts with Building a Test Automation Strategy

The test pyramid, popularized by Michael Cohn and Martin Fowler, identifies that we should have different quantities of different types of testing activities. At the base, we want to establish broad coverage of the unit and API tests. These types of tests are easier to run in automation but often require technical skill to build. The top of the testing pyramid has automated UI tests and manual tests. We want this type of testing activity at the top because it's the only way to ensure the customer experience.

Most people focus on the bottom and the top of the testing pyramid with an "automate what's manual" approach for UI testing and a "developers should test" mentality for unit testing. While these practices are important, focusing on the top and bottom creates a gap in the middle, at the API layer, between the UI and the code.

This gap is only going to continue to become more problematic, with a recent report from ProgrammableWeb showing over 22,000 publicly available APIs in their directory, with the number steadily climbing. API testing is more critical than ever, and it needs to become an integrated part of the continuous testing strategy that you're building.

Choosing the Right API Testing Solution

It all starts with choosing an API testing solution that can grow with you as your API testing maturity grows. Key features that you might need depend on your industry, application, and your organization's unique requirements. But once you've chosen a tool, how do you get started?

There are three critical steps to achieve functional test automation faster.

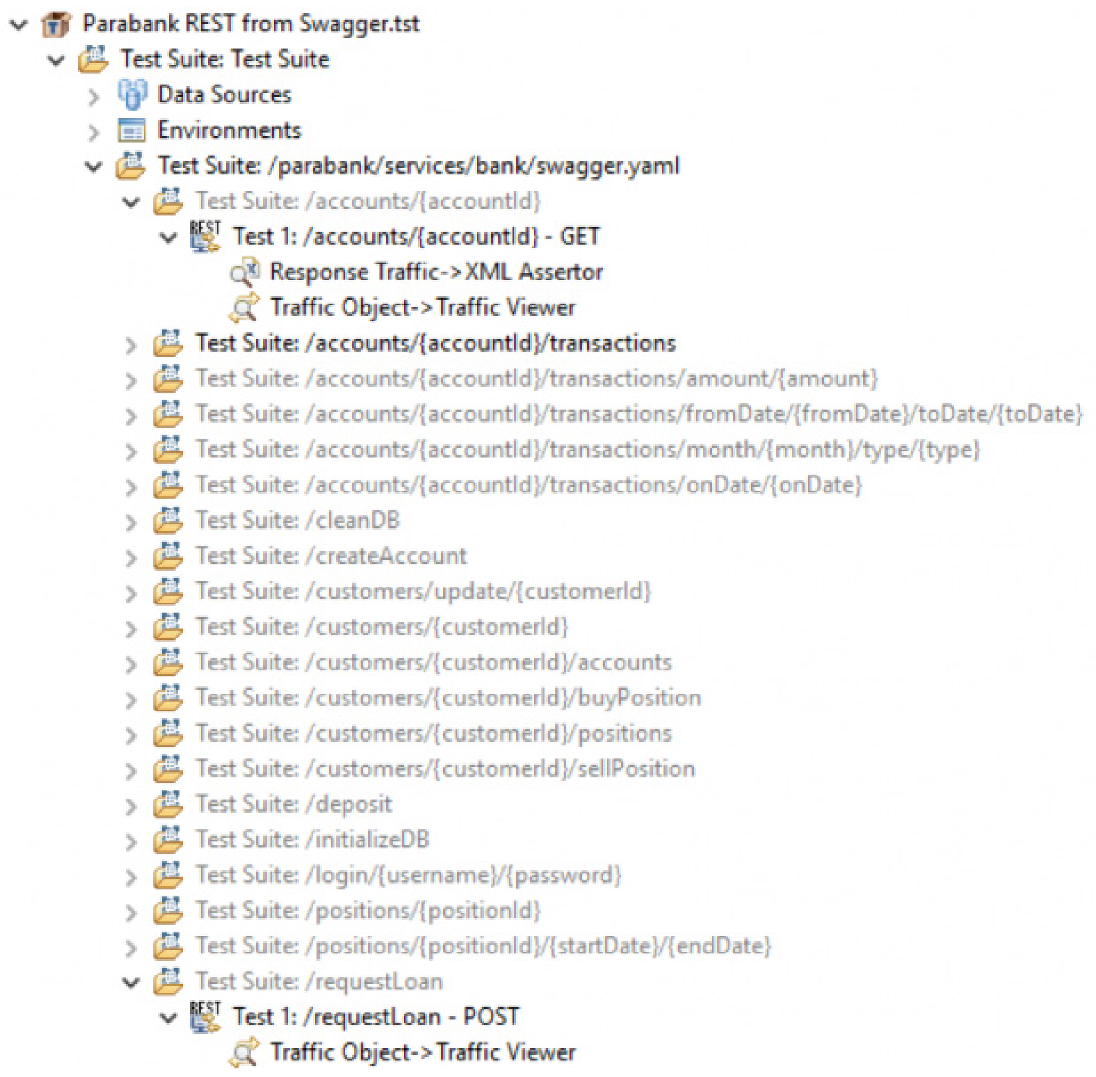
Step 1: Establish broad test coverage of your existing APIs for automation

By building scriptless tests from web recordings and API contracts, and then using those tests to continuously validate the health of your APIs, you can ensure that the APIs are working as they were designed. Think of this as unit testing for APIs, but without the grunt work.

Not only is this a valuable testing technique, but it's also one of the earliest types of functional test validation you can do, because service contracts for APIs are usually one of the first things that are written as new features or functionalities are created.

As an example, let's say that the team has added some new functionality to our application. As a first step, the development team published a new service definition. In this case, a Swagger document was generated. The new service being added was a RequestLoan service, which takes a series of inputs and responds with a loan provider for the new loan.

To test this service, I could consume the Swagger YAML and create a series of clients for each individual operation.



```
Parabank REST from Swagger.tst
  Test Suite: Test Suite
    Data Sources
    Environments
    Test Suite: /parabank/services/bank/swagger.yaml
      Test Suite: /accounts/{accountId}
        Test 1: /accounts/{accountId} - GET
          Response Traffic->XML Assertor
          Traffic Object->Traffic Viewer
      Test Suite: /accounts/{accountId}/transactions
      Test Suite: /accounts/{accountId}/transactions/amount/{amount}
      Test Suite: /accounts/{accountId}/transactions/fromDate/{fromDate}/toDate/{toDate}
      Test Suite: /accounts/{accountId}/transactions/month/{month}/type/{type}
      Test Suite: /accounts/{accountId}/transactions/onDate/{onDate}
      Test Suite: /cleanDB
      Test Suite: /createAccount
      Test Suite: /customers/update/{customerId}
      Test Suite: /customers/{customerId}
      Test Suite: /customers/{customerId}/accounts
      Test Suite: /customers/{customerId}/buyPosition
      Test Suite: /customers/{customerId}/positions
      Test Suite: /customers/{customerId}/sellPosition
      Test Suite: /deposit
      Test Suite: /initializeDB
      Test Suite: /login/{username}/{password}
      Test Suite: /positions/{positionId}
      Test Suite: /positions/{positionId}/{startDate}/{endDate}
      Test Suite: /requestLoan
        Test 1: /requestLoan - POST
          Traffic Object->Traffic Viewer
```

One of those clients would be the request loan service. I could create a series of inputs, both positive and negative, to validate that the service behaved appropriately. I could then repurpose these tests for regression purposes.

Of course, while this type of testing is extremely valuable, it's only half of the API testing puzzle, because it doesn't validate how the APIs are actually being used. Enter step 2.

Step 2: Bridge the gap between UI and API
The second part of the API testing strategy is being able to model human usage of your application into complete API test scenarios.

You can start to bridge that gap in the testing pyramid by taking advantage of artificial intelligence to augment your ability to understand what's actually happening behind the scenes when users are navigating your application, and interpret those behind-the-scenes transactions as API calls. This type of testing allows you align the user experience with your critical API tests.

AI is a key component of the strategy because we can reliably use AI to help us break down these communications into relationships and patterns, so we can understand the business rules of how our applications are being tested. We can bring this together with our unit-level API testing to get broad coverage of our API spectrum.

Continuing my example, you will notice that the request loan service requires inputs from other areas of the application:

Parameter	Description	Parameter Type	Data Type
customerId	**Customer's ID**	query	integer
amount	**Amount**	query	double
downPayment	**Down payment for the loan**	query	double
fromAccountId	**Customer funds source account**	query	integer

While I could provide customer IDs from accounts arbitrarily, I really need them to exist in my application, so I will need to create a dynamic scenario where I first query the individual user to obtain the customer ID and account ID so that I can hand that information off to the request loan service and ensure that a dynamic scenario works as described. Ensuring that I'm working with real dynamic data ensures that that behavior that exists as a result of APIs interacting with each other can be fleshed out.

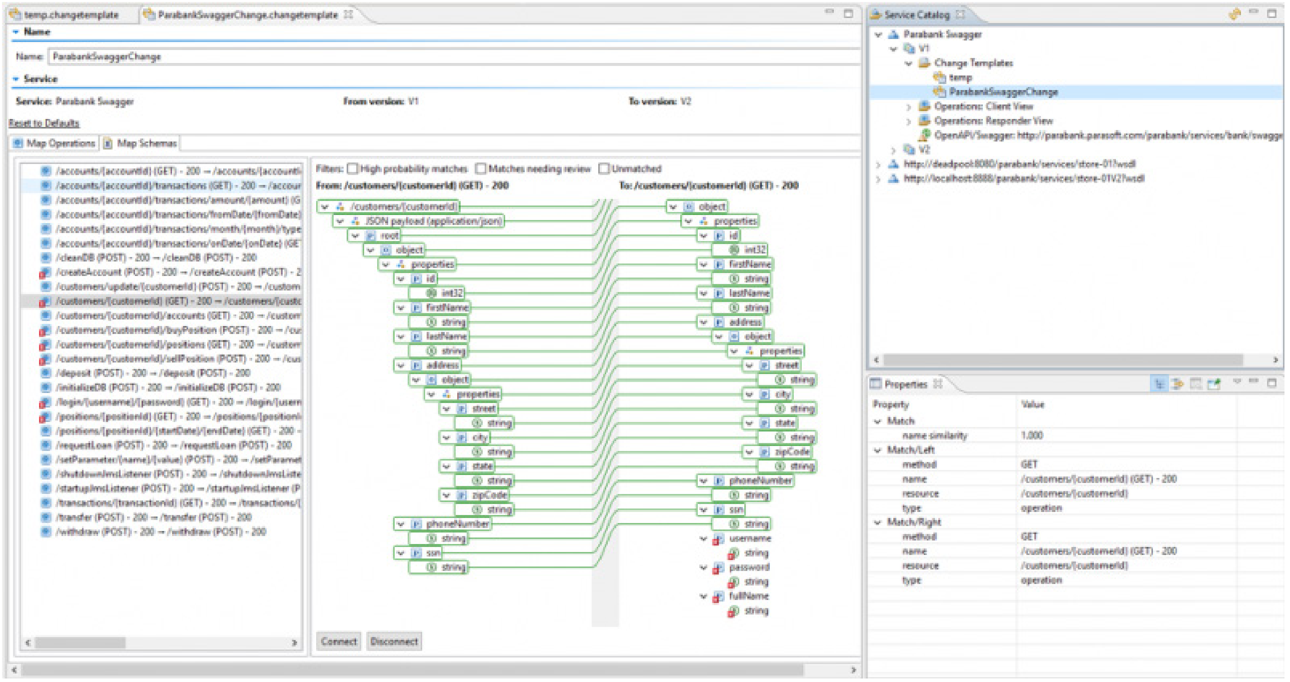
While these types of techniques will allow us to shift API testing practices left and create broad coverage of our applications in the earliest stages possible, there is a third critical component of this practice: our ability to understand and adapt to change.

Step 3: Ensure confidence through a maintainable change management process
I've spoken to so many people about their functional testing initiative that fell flat once their application changed. This is a common occurrence because testers spend the majority of their time building rich and great API tests, only to have them break when the application's APIs change. This can have the cumulative effect of reducing confidence in the API testing strategy because testers are spending a large portion of their time maintaining their API tests instead of building new value.

Change management is a key piece of any functional testing strategy, and AI can be a critical enabler here as well. By automatically scanning service definitions (yes, the same service definitions used to initially create the test cases) to identify when your APIs have changed, you can understand when you'll be impacted, and then build a template for migrating existing services to the new version.

In the first part of this example I made the statement that a new service was added to my application. This actually represents API change. Since I created the first round of baseline tests using the service definition, I can compare the different versions of the service definition to each other to identify what has changed and to build a map to update my existing test cases.

[Using a change management workflow] is arguably the most important practice you must establish when building your functional testing strategy.


temp.changetemplate | ParabankSwaggerChange.changetemplate

Name
Name: ParabankSwaggerChange

Service
Service: Parabank Swagger | From version: V1 | To version: V2

Reset to Defaults

Map Operations | Map Schemas

```
/accounts/{accountId} (GET) - 200 → /accounts/{accountI
/accounts/{accountId}/transactions (GET) - 200 → /accour
/accounts/{accountId}/transactions/amount/{amount} (G
/accounts/{accountId}/transactions/fromDate/{fromDate}
/accounts/{accountId}/transactions/month/{month}/type
/accounts/{accountId}/transactions/onDate/{onDate} (GE
/cleanDB (POST) - 200 → /cleanDB (POST) - 200
/createAccount (POST) - 200 → /createAccount (POST) - 2
/customers/update/{customerId} (POST) - 200 → /custom
/customers/{customerId} (GET) - 200 → /customers/{custo
/customers/{customerId}/accounts (GET) - 200 → /custom
/customers/{customerId}/buyPosition (POST) - 200 → /cu
/customers/{customerId}/positions (GET) - 200 → /custom
/customers/{customerId}/sellPosition (POST) - 200 → /cus
/deposit (POST) - 200 → /deposit (POST) - 200
/initializeDB (POST) - 200 → /initializeDB (POST) - 200
/login/{username}/{password} (GET) - 200 → /login/{usern
/positions/{positionId} (GET) - 200 → /positions/{positionI
/positions/{positionId}/{startDate}/{endDate} (GET) - 200 -
/requestLoan (POST) - 200 → /requestLoan (POST) - 200
/setParameter/{name}/{value} (POST) - 200 → /setParamet
/shutdownJmsListener (POST) - 200 → /shutdownJmsListe
/startupJmsListener (POST) - 200 → /startupJmsListener (P
/transactions/{transactionId} (GET) - 200 → /transactions/{
/transfer (POST) - 200 → /transfer (POST) - 200
/withdraw (POST) - 200 → /withdraw (POST) - 200
```

Filters: ☐ High probability matches ☐ Matches needing review ☐ Unmatched

From: /customers/{customerId} (GET) - 200 | To: /customers/{customerId} (GET) - 200

```
/customers/{customerId}                      object
  JSON payload (application/json)              properties
    root                                         id
      object                                       int32
        properties                               firstName
          id                                       string
            int32                                lastName
          firstName                                string
            string                               address
          lastName                                 object
            string                                   properties
          address                                      street
            object                                       string
              properties                               city
                street                                   string
                  string                               state
                city                                     string
                  string                               zipCode
                state                                    string
                  string                         phoneNumber
                zipCode                            string
                  string                         ssn
          phoneNumber                              string
            string                               username
          ssn                                      string
            string                               password
                                                   string
                                                 fullName
                                                   string
```

Connect | Disconnect

Service Catalog

```
Parabank Swagger
  V1
    Change Templates
      temp
      ParabankSwaggerChange
    Operations: Client View
    Operations: Responder View
    OpenAPI/Swagger: http://parabank.parasoft.com/parabank/services/bank/swagge
  V2
http://deadpool:8080/parabank/services/store-01?wsdl
http://localhost:8888/parabank/services/store-01V2?wsdl
```

Properties

Property	Value
Match	
name similarity	1.000
Match/Left	
method	GET
name	/customers/{customerId} (GET) - 200
resource	/customers/{customerId}
type	operation
Match/Right	
method	GET
name	/customers/{customerId} (GET) - 200
resource	/customers/{customerId}
type	operation

Upon reviewing the change template, it becomes clear that not only has a new service been added, but many of my existing services have been refactored. In the image to the left, you will notice that "GET customers" has a series of new fields that have been added. Using a change management workflow will allow you to proactively identify service changes while at the same time managing the updating of existing test cases, so you can recover from the change as quickly as possible.

This is arguably the most important practice you must establish when building your functional testing strategy. Understanding this and having a commitment to quality from the beginning will help you and your organization adopt this practice.

As part of building a sustainable API testing strategy, you'll need to build a sustainable service virtualization strategy.

What about Flaky Test Environments?

It would be easy to stop there, with your excellent test automation enabling continuous testing. But say you've spent a good amount of time building this rich and powerful functional testing strategy, you run your tests in your environment as part of your automated nightly continuous testing process, and upon reviewing the results, you see that a large portion of your tests have failed due to a system that's outside of your control.

Does this mean that your tests were bad? Are you now being held responsible for systems that are technically out of scope for your testing?

This is not an uncommon story. We know that functional tests can only be as effective as the test environments in which they execute. Unstable, unavailable, or just plain flaky test environments can reduce the return on investment we get from our functional testing tools. So I must, at least briefly, mention one of the best ways to stabilize your test environments: service virtualization.

Not to be confused with virtual machines (that's hardware virtualization), service virtualization enables you to simulate the services that are communicating between different pieces of hardware. For example, think of an application calling a database. Do you actually need that database in your test environment? What if it doesn't have the data that you need? With service virtualization, you can record the transactions with the database and then use that recording to create a simulated version of that database, with all the behaviors you want for your test environment. But of course, it doesn't stop at just databases; it can be any type of service, such as a SOAP or REST API, or even TCP and microservices.

As part of building a sustainable API testing strategy, you'll need to build a sustainable service virtualization strategy, and that starts with answering questions like:

- What services are good candidates for virtualization?
- How are virtual services created?
- How can I maintain a virtual environment?
- How can I deploy a virtual environment as part of my continuous testing strategy?

Service virtualization is a key enabler for a sustainable continuous testing strategy, but understanding where and when to bring it in and how to be as effective as possible is key to success.

Get Started with Continuous Testing

Now that you have a better understanding of how to integrate API testing as part of your continuous testing strategy, the next step is to get going! Get engaged with an API testing tool vendor, and start with step one. Understanding the end goal as you begin will help you make smart choices along the way.

Why You Need Continuous Testing in DevOps

By Tom Alexander

The agile process is all about using short, flexible development cycles to respond quickly to customer needs. Doing this effectively these days involves building a DevOps software pipeline in order to quickly get high-quality software into the hands of your customers and receive feedback.

Most DevOps initiatives start with the adoption of continuous integration (CI) practices, where code is continuously integrated to make sure everything works together. Developers start the CI process by checking code into a shared repository many times a day. Each check-in is verified by an automated build process and some fast-running tests, allowing teams to detect errors and conflicts as soon as possible. Regression tests are run at least each night to make sure any changes made during the day did not break something else.

After CI is performed, a continuous delivery process is followed, where the application is further tested and, once it passes all the required tests, it's available to release into production. The upside of continuous deployment is that it delivers new functionality to users within minutes, as well as instant feedback to the team that allows rapid response to customer demands. Effective testing during your continuous deployment process is critical because without it, there is a big risk of continuously releasing buggy software into production.

Don't Let Testing Practices Slow You Down

Continuous testing, which is often called shift-left testing, is an approach to software and system testing in which testing is performed earlier in the software lifecycle. The goals are finding defects earlier, increasing software quality, shortening long test cycles, and reducing the possibility of software defects making their way into production code during deployments. Continuous testing is critically important if your company is trying to use DevOps to deploy software frequently into production.

Done right, continuous testing provides fast and continuous insight into the health of the latest build of your application. This information can then be used to determine if the app is ready to progress through the delivery pipeline at any given time. Because testing begins early and is executed continuously, bugs are

exposed soon after they are introduced, which reduces the time and effort needed to find and fix them. Consequently, it is possible to increase the speed and frequency at which bug-free, high-quality software is delivered, as well as decrease technical debt.

By shortening the time it takes to fix buggy software, continuous testing helps pay down your technical debt by keeping these interest costs from accruing.

Technical debt refers to the price organizations pay when releasing badly designed code. It's a way of calculating the cost of additional rework caused by choosing an easy and quick solution now instead of using a better, less buggy approach that would take longer. Just like financial debt, technical debt incurs interest that must be paid, such as increased maintenance, support, or legal costs. By shortening the time it takes to fix buggy software, continuous testing helps pay down your technical debt by keeping these interest costs from accruing.

DevOps is a cultural shift that promotes collaboration among all teams, including development, quality assurance, operations, and others, such as performance management, release management, and maintenance teams. Consequently, there is no single product that can be considered the definitive DevOps tool. Often, a collection of tools from a variety of vendors is used in the stages of a DevOps process. Continuous integration is often seen as the backbone of a continuous delivery pipeline, which explains the popularity of CI tools such as Jenkins and Bamboo to build, test, and deploy applications automatically when requirements change.

Companies using DevOps often ship new software into production hundreds of times every day. These companies are delivering smaller pieces of software, collaborating, and monitoring in production to create a continuous flow of code, from check-in to production. And they're using continuous testing technology to weave testing activities into every part of their software design, development, and deployment processes.

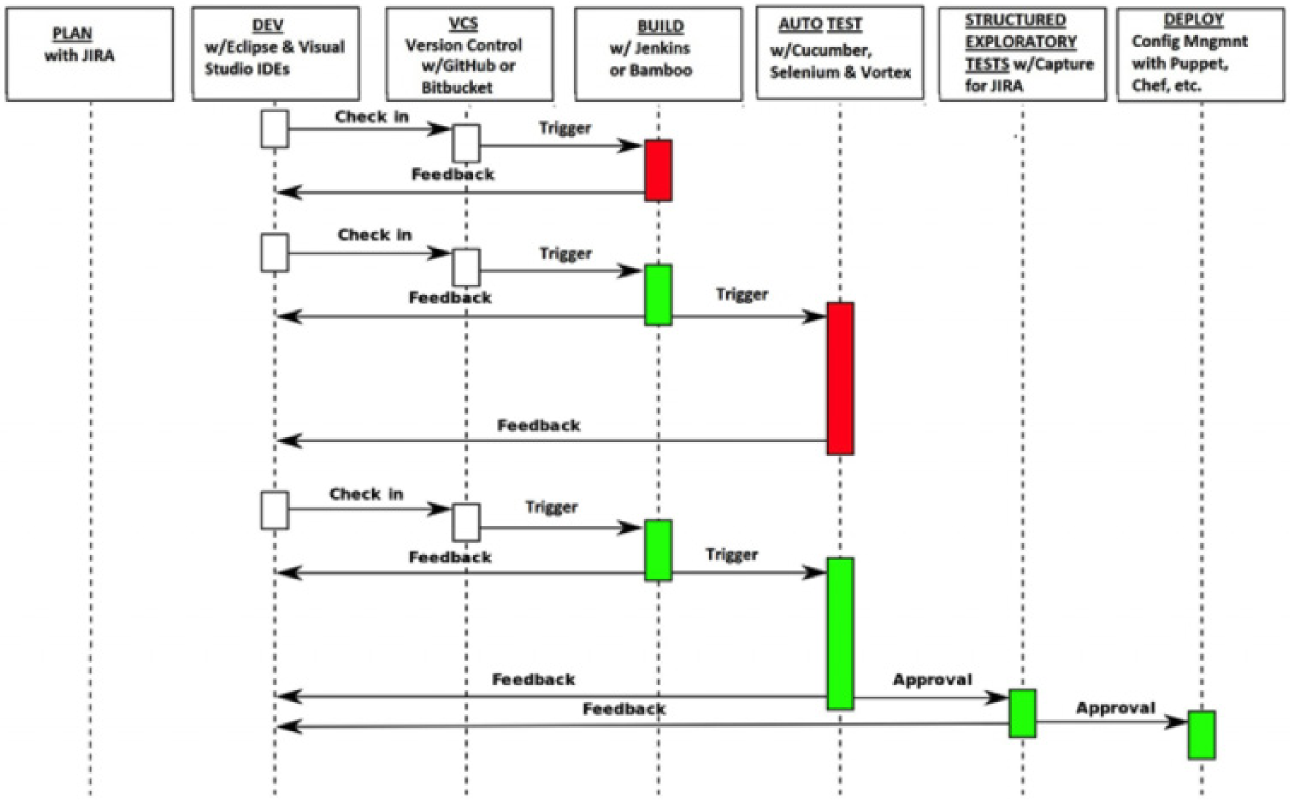
Let Tech Do the Heavy Lifting

To release high-quality code faster, your organization needs to let tech do the heavy lifting by adopting next-generation tools and practices that enable you to test early, often, automatically, and continuously.

By executing the right set of tests at the right stage of the delivery pipeline—without creating a bottleneck—these tools enforce agile principles by providing appropriate feedback at every stage of the process. This enhanced communication averts duplication of efforts and increases alignment among dev, ops, and testing teams, which will allow you to deliver software on tighter schedules.

But these streamlined schedules are only possible if test automation is seamlessly integrated into your software delivery pipeline and DevOps toolchain. Test automation works by running a large number of tests repeatedly to make sure an application doesn't break whenever new changes are introduced. Manual testers are often still involved in DevOps projects, performing testing while an automation test suite is constantly running—but their role needs to shift toward a session-based exploratory testing approach, focused on areas with the most risk or where automation is not effective.

The image below shows an example DevOps pipeline that incorporates continuous testing during check-ins, continuous integration, and continuous delivery.


PLAN with JIRA
DEV w/Eclipse & Visual Studio IDEs
VCS Version Control w/GitHub or Bitbucket
BUILD w/ Jenkins or Bamboo
AUTO TEST w/Cucumber, Selenium & Vortex
STRUCTURED EXPLORATORY TESTS w/Capture for JIRA
DEPLOY Config Mngmnt with Puppet, Chef, etc.
Check in
Trigger
Feedback
Approval

A Continuous Testing DevOps Toolchain

While not an exhaustive list of all available DevOps products, here's a checklist of tools that together make up a viable continuous testing DevOps toolchain.

Planning Tools

If you're looking for a tool that makes it easy for different teams to collaborate, **Jira** is an agile project management tool that supports any agile methodology, be it Scrum, kanban, or your own unique flavor. From agile dashboards to reports, you can plan, track, and manage all your agile software development projects. Jira's wide range of integrations also helps you connect to almost any other tool you're likely to need.

Dev Tools: Desktop or Cloud-based IDEs

While **Eclipse** and **Visual Studio** are the most popular desktop IDEs, **Cloud9,** developed by Amazon Web Services, and **JSFiddle** lead in the cloud.

Version Control Systems (VCS)

There are several web-based hosting services for DevOps version control, including Microsoft's **GitHub,** Atlassian's **Bitbucket,** and the open source **GitLab** service. All work within standard desktop or cloud IDEs to ease the processes of source code check-in and checkout.

Build Tools

Jenkins is a CI/CD server that builds applications, runs tests automatically, and pushes code through your DevOps pipeline every time a developer checks new code into the source repository. Because of the rich ecosystem of plugins, Jenkins can be used to build, deploy, and automate almost any software project.

Bamboo is a CI/CD server from Atlassian. Like Jenkins and other CI/CD servers, Bamboo allows developers to automatically build, integrate, test, and deploy source code. Bamboo is a commercial software that is integrated and supported out of the box with other Atlassian products, such as Jira for project management and Hipchat for team communication.

Automated Testing Tools

Cucumber is a tool for specifying application features and user scenarios in plain text. Cucumber runs automated acceptance tests written in a behavior-driven development (BDD) style that encourages collaboration on software projects by writing test cases in a natural language that nonprogrammers and domain experts can read.

Selenium is a suite of different open source software tools that enable automated testing of web applications across various browsers and platforms. Most often used to create robust, browser-based regression automation suites and tests, Selenium, like Jenkins, has a rich repository of open source tools that are useful for different kinds of automation problems.

Agile teams can execute one-touch control of test automation from within the Zephyr platform with **Vortex,** Zephyr's advanced add-on that allows you to integrate with a growing suite of automated testing frameworks (including eggPlant, Cucumber, Selenium, UFT, and Tricentis) with minimal configuration. Besides being able to control the execution of thousands of automated test cases, Vortex makes it easy to automatically create test cases from test scripts and to apply insights from analytics on both automated and manual testing activities.

Session-Based Exploratory Testing

PractiTest is a test management system that supports session-based exploratory testing practices. Session-based exploratory tests are created and added to a test set during testing. These tests can be combined with other types of tests, including structured manual and automated, to maintain test suites, traceability, and test coverage.

DevOps is more than adopting the right set of tools; it's a cultural shift that incorporates testing at each stage of the agile project lifecycle.

Capture for Jira helps testers on agile projects create and record exploratory and collaborative testing sessions, which are useful for planning, executing, and tracking manual or exploratory testing. **Session-based test management,** a type of structured exploratory testing, is an extremely powerful way of optimizing test coverage without incurring the costs associated with writing and maintaining test cases. Like Zephyr for Jira, Capture for Jira has a deep integration with the Jira platform, allowing users to capture screenshots within browsers, record screens in Chrome, create annotations, and validate application functionality within Jira.

Deployment Tools

Longtime "movers and shakers" in the DevOps infrastructure-as-code space, **Chef** and **Puppet** are both automated configuration management and orchestration tools used to quickly spin up compute and storage instances on demand.

Test Continuously to Deliver Faster

DevOps is more than adopting the right set of tools; it's a cultural shift that incorporates testing at each stage of the agile project lifecycle. Continuous testing is key to unlocking this culture change because it helps everyone involved communicate more, collaborate better, and innovate faster.

How Continuous Testing Is Done in DevOps

By Junaid Ahmed

The adoption of DevOps practices is dramatically increasing throughout many different industries, mostly due to companies recognizing the numerous benefits DevOps is able to deliver.

DevOps does speed up your processes and make them more efficient, but companies that solely focus on speed and ignore quality aspects are likely to suffer a huge blow. Teams should focus first on quality by reducing defects and bugs before working on speeding up their operations.

DevOps is all about better enabling the software testing process to deliver quality results within a shorter time frame. Consequently, quality assurance is an important aspect of the DevOps methodology. Integrating QA within DevOps helps companies focus on giving their clients quality software before it ships.

Integrating QA within DevOps also plays a vital role in managing risks by ensuring the application is robust and stable throughout the development process. Continuous testing as part of a DevOps method helps detect bugs quickly, when they are easier and less expensive to fix, ensuring your application is fit for usage and enhancing a good user experience.

QA should not live outside the DevOps environment; it should be a fundamental part. But if your DevOps ambitions have started with only the development and operations teams, it's not too late to loop in testing. You must integrate QA into the lifecycle in order to truly achieve DevOps benefits.

Why Should You Perform Continuous Testing In DevOps?

Software projects, websites, and applications are not static. They require regular updates and real-time changes to fulfill all the set requirements of the clients.

If continuous testing is performed properly, it delivers quick and uninterrupted insight into the quality of every new build of your software.

These changes used to be time-consuming and perilous, but now they can be attained more easily through continuous integration, continuous deployment, and continuous testing.

DevOps allows software testing teams to easily upgrade and deliver various products without interfering with their quality. That's why most DevOps enterprises begin with the adoption of continuous integration practices: to ensure that everything works together.

Continuous testing involves testing a software application beginning in its early stages and automating testing throughout the development lifecycle. This helps the team examine the quality of the product at every stage of the continuous delivery process. And this process is not limited to only testers and developers; it also involves the contribution of stakeholders, operations, and even the customer. Continuous testing is an integral factor in the DevOps equation.

If continuous testing is performed properly, it delivers quick and uninterrupted insight into the quality of every new build of your software. This information can help you analyze whether the application is prepared to go through the delivery pipeline.

For example, when the code in a source code server like Jenkins is verified by developers, a set of automated unit tests is executed in the continuous process. If the tests don't pass, the build will be rejected, and the developers will be notified about it. If the tests pass, the code will be sent to the QA servers for functional and load testing. Then those tests are executed in parallel, and if they pass the build, the application will be deployed in production.

Key Points for Continuous Testing Adoption

Before you begin implementing continuous testing in your team, there are a few points to keep in mind.

The idea of continuous testing is to implement testing early in the software development lifecycle and at each branch involved in your CI/CD (continuous integration and delivery) pipeline. So before you transition a code change to the production environment, you should already have it validated at the staging ones.

This could be challenging, as you need to make sure that all your staging environments are exact replicas of your production. This requires more resources, bandwidth, and infrastructural cost. And even if you do have all the changes pushed to the staging environments, you can't be sure about pushing them to production just because they worked in the staging environments, as your production web application will usually be facing a considerable amount of user interaction. The fact that there is more web traffic in your production environment compared to the stage environment is one of the common things testers often forget. If you are short on the resources and investment required to maintain different stage testing environments, then continuous testing is probably not a good idea for you.

You also must be ready with your tools arsenal. You need to have the right automation tools on board for effective implementation of continuous testing in DevOps.

This means you need to have the tools required for each layer of the test automation pyramid—UI testing, API testing, and unit testing—because you can't expect one tool to cover everything for you. For example, if you are performing automated browser testing, you would require unit testing frameworks to validate your code changes at an earlier stage in the continuous deployment pipeline. However, at a later stage, you would need an end-to-end test automation framework.

In continuous testing, you are testing a single change on multiple test environments before deploying into production. A release may contain a bucket of feature changes, so you will be testing numerous code changes at multiple test environments, and for each code change you also need to perform regression testing on each stage environment. All of this could be time-consuming unless you know how to put parallel testing to best use. Many automation frameworks can execute multiple test scripts simultaneously, which will speed up the continuous testing through your CI/CD pipeline.

Also be sure to recognize false negatives and false positives. They are more common than you may know! When you execute a test automation script, it may show an execution error even if the system is working fine, or the automation testing script may show the test execution as successful even when the system met with errors. While either is dangerous, false positives can be more devastating because you believe everything is good to go and then you get an outage.

Plan your rollbacks thoroughly. Even if you use all the best practices for continuous testing and CI/CD, there is no guarantee that the build won't break in production. Always be ready for the worst scenarios. Make sure your data is backed up before you push changes to production. In case things go south, you can roll back to the previous production version quickly and perform a round of smoke testing to ensure the web application functions well again.

Plan your rollbacks thoroughly. Even if you use all the best practices for continuous testing and CI/CD, there is no guarantee that the build won't break in production.

Keep in mind that when you roll back from the production environment, you can't just validate your pre-production situation and commit the changes again. You need to evaluate the entire pipeline to make sure there are no loose ends, as well as all your test environments, because you don't want to postpone the same release cycle twice.

Integrating QA and DevOps

The testing and technical teams should work hand in hand to ensure quality throughout every step in the software development lifecycle.

When integrating QA into DevOps practices—known by the term "QAOps"—the testing team should adapt and adopt certain quality processes:

- They should strive to detect bugs at the earliest point in the development lifecycle and prevent potential bugs from reappearing in the production cycle
- They are responsible for highlighting issues in the process and recommending necessary changes
- They must ensure that all the environments required for testing are standardized with automated deployment
- Apart from finding and preventing bugs, they should also focus on improving the overall quality of the product

Some people in the software industry think the requirement for QA is decreasing with the rise of DevOps due to automated processes. But that's not completely true. Though automated testing techniques are advanced and fast, they also require continuous human intervention. Companies that lack enough QA professionals and resources are not likely not to achieve their customers' requirements, mostly when it comes to updates and continuous changes.

DevOps was created to make developers think in synchronization with software testers, not to replace them.

DevOps was created to make developers think in synchronization with software testers, not to replace them. Software development processes are becoming faster through continuous deployment to meet ever-growing customer demands, and integrating QA with DevOps can help you fulfill all your objectives.

The integration of the QA process into DevOps also helps you manage various risks, making sure the end results are robust and more stable. QAOps is like a fitness regimen for software, as it makes it easy to detect bugs frequently and on time so you can tell when your applications are fit to run.

There are a couple of patterns for integration of QA into DevOps that teams can adopt:

- Try conducting an exploratory test before any new feature is merged into the master codebase. The tryout tests are designed to ensure the system is adequately covered to deliver accurate results
- Before you transmit a code change, make sure that your QA, developers, and other stakeholders involved in the continuous testing process are ready with their testing checklist. This checklist should include all the valid and invalid test scenarios they need to consider along with the expected behavior over specific test environments

Every business venture, company, and enterprise operates differently, so the ways of adopting QAOps may also differ. But the two suggestions above can be implemented across any software team.

You cannot deliver a comprehensive and quality service without a QA testing strategy, so QA is essential to the DevOps process. Integrating QA processes into DevOps operations helps both the testing and technical teams handle dynamic software environments and situations, deliver at speed throughout the CI/CD pipelines, and ensure the quality of the product—as well as customer satisfaction.

A DevOps Approach to Mobile App Development

Juned Ghanchi

Every business is now serious about having a branded mobile presence to drive growth and build a reputation. But when it comes to app development, many companies still aren't sure where to begin.

The DevOps approach has completely transformed the mobile app development scenario, and it's a good option for any business looking to create an app. DevOps facilitates collaboration between developers, operations staff, and project managers to better fulfill the objectives of an enterprise app development project.

Bringing together the development and operations teams on the platform has solved many problems in mobile app development. These are some of the key benefits of a DevOps approach:

- Continuous delivery
- Improved customer experience
- Quicker bug fixes
- Enhanced employee engagement
- Fast product delivery
- Increased stability of the development environment
- Easy and smooth deployments
- More opportunity and time for innovation
- A boost to efficiency

To adopt a DevOps approach for your app development, you need to consider these important factors.

Continuous Planning

When you are planning for a new mobile app that will be representing your business, get every functionary of your company, including developers, operations, project managers, and various other stakeholders, on board. Make every team member involved in planning and deciding the development scope.

Continuous Integration

Continuous integration is about integrating the code of one group of developers with that of others. A DevOps approach allows various teams to build different parts of the app and continuously integrate them for testing and evaluation. This approach encourages faster development while building the app.

Continuous Testing

Mobile app testing is a tricky affair because the test outcome can be different from simulators to actual devices, and you have to incorporate manual testing beside automation testing tools. To make it more challenging, the same app needs to be tested across multiple versions of all popular operating systems. A continuous monitoring of the testing results is needed across devices, users, tools, and OS versions. This approach helps deal with the multiplicity of the testing scope and problems while still ensuring continuous value addition.

Continuous Delivery

Continuous delivery is a key DevOps practice that encourages constantly updating the production environment with the changes that can influence the outcome.

Continuous Deployment

Through continuous deployment, the DevOps project helps to deploy the changes to the finished app automatically. This keeps the app updated continuously by bringing changes to the live app in real time.

While adapting to the DevOps approach of developing a mobile app has its share of challenges, the benefits are worth the effort.

5 Mistakes to Avoid When Beginning Mobile Continuous Testing

By Steve Orlando

Are you just getting started with continuous testing? Find out what you can do to avoid five common mistakes on your path to becoming a mobile continuous testing genius.

With all the excitement and buzz around continuous test automation for mobile app and mobile web testing, it's only natural to be curious. What does continuous testing mean for your developers and testers? What can your team accomplish with more speed and automation in your corner?

But, for enterprise mobility teams that are primarily leveraging manual testing the notion of moving testing from "your hand to the cloud" can be a little daunting. But there is no reason to be nervous if you plan carefully and set yourself up for success right from the beginning. A successful beginning yields a smooth and precise continuous testing strategy that increases performance, app quality, DevOps and overall agility.

Through my work with our customers and from helping their enterprise mobility teams on the path to continuous testing, I've identified five common roadblocks that mobile developers and testers often stumble over when starting out. Here's what you can do to avoid these bumps in the road and to begin your continuous testing journey on the right foot.

Mistake to Avoid #1: Using the Wrong People

When getting started with continuous testing, don't use testers that are geared for manual testing only. Because continuous testing relies on test automation, it is important to have testers on your team experienced with automation principles. While manual testers are useful for making sure that the app is easy to use, that it meets business objectives, and that it is capable of receiving 5-star ratings, test automation requires a different set of skills.

Instead you should dedicate a team in your mobile testing lab with automation experience using their

test automation tool of choice. Some automation tools like Appium® may require more experience in coding than Tricentis Tosca® or Micro Focus UFT. The team will need to work on building out a framework and a continuous testing pipeline followed by working together to establish a repeatable set of tests.

Mistake to Avoid #2: Going Rogue

I come across a lot of teams doing mobile test automation, but the most successful teams are integrated and/or work closely with the development team. Why are they more successful? I believe it is due to better communication. Let's compare the experience of being integrated with development as opposed to going at it alone below. I think you'll discover that you get more out of collaboration then being the Lone Ranger.

Working closely together means that the teams tend to solve issues faster and help each other maintain high quality apps. When working together the goals are aligned and when building out a continuous testing strategy, all the stakeholders are invested in the outcome.

The thing I love about Mobile Labs' support team is that they provide answers about any automated test scripting questions to our customers.

How does this collaboration play out in continuous testing? Well, the DevOps team will help get the automated tests to run with the application build jobs. If tests fail, then development and QA teams are both notified and issues can be solved faster.

Automated tests can be created easier with development buy in. Often the development team does not know the best way to make applications accessible for automation. Without buy-in, testing teams can struggle to write automated tests or require larger efforts than needed if developers do not make the required applications accessible (setting the content-desc and resource-id for android; name and accessibility-id for iOS.)

Don't get me wrong, there are many teams that just need to produce testing results and are suited to go at it alone. But when teams go rogue it is a harder hurdle to overcome without development buy-in to the testing efforts.

Mistake to Avoid #3: Building Your Own Testing Framework

While it may be tempting to use your team's advanced skills to build your own testing framework, it is a strategy that can sometimes backfire on you. Particularly in the area of support where issues may be hard to diagnose and fix. Often it is necessary and even comforting to have a trusted team of individuals with a knowledge base that you can tap into to help your team push toward a successful continuous testing strategy.

The thing I love about Mobile Labs' support team is that they provide answers about any automated test scripting questions to our customers. This often includes answering questions that are usually hard to diagnose, such as when a team is using a custom-built framework to manage data, test flows, reporting, and parallelization. It can be a timely process and getting to a resolution can be difficult when you're going rogue. Instead, it may be easier for teams to get up and running faster with a continuous testing framework by exploring tools and test frameworks that support many different programming languages that have broad support.

Most of these frameworks are free and open source. Leveraging these tools makes it easier for the following reasons:

Reduced startup time

An established framework can be implemented quickly, and automation tests can be written immediately. There is no time wasted building out a complex testing framework that mimics the same functionality as one already available.

Lower training and maintenance costs
When a problem or a lack of knowledge exists, it is easy to find an example online using the available framework. In most cases the custom-built framework for the company is not documented, so automation users have to rely on trial and error to work within the framework. Should the subject matter expert of the framework leave the company, the automation efforts could stall should the testing project change or new automation efforts are needed.

Can be extended for many scenarios
Most open-source frameworks are adaptable, so changing something that doesn't work in your environment can be done with little effort. For instance, if you need to have some custom functionality to connect to a device for a test, you can extend the test functions to automatically handle this and keep the framework easy to use.

Handles parallel testing execution
Implementing parallel testing is hard but worth the effort. In fact, you can see this for yourself with our handy calculator.

For example, parallel testing with Appium requires knowledge of spinning up new threads for execution which is a very complex level of development. What can happen is that some threads start up before others finish and Appium resources and devices can already be in use. Either a test will fail to run or a new test causes an existing test to fail without any evidence to why the test fails. This is difficult to diagnose and debug. Given the above advice, I will provide one word of caution, don't go overboard. I have even come across a test framework that used an Excel driven table to drive a TestNG framework that called JUnit tests.

Mistake to Avoid #4: Only Using Simulators

On the subject of testing using simulators v. real devices, I am an advocate for testing on real devices for several reasons. I recently wrote a blog post on this topic, which you can view it here for a more in-depth study.

But to briefly summarize, despite having built a powerful Simulator (Apple) and an Emulator (Android), both vendors still advocate for testing on a real device. Clearly, while simulators and emulators can help developers and testers quickly test certain elements of mobile apps and mobile websites before deployment, they should not be considered a replacement for testing on a real device.

For real results and to get the most accurate insight into how an app or mobile website will function in the real world, enterprise mobility teams should test on real devices.

Consider this excerpt from Apple's Simulator Help Overview (emphasis added):

"Simulator is a great tool for rapid prototyping and development of your app allowing you to see the results of changes quickly, debug errors, and run tests. It is also important to test your app on physical devices as there are hardware and API differences between a simulated device and a physical one. In addition to those differences, Simulator is an app running on a Mac and has access to the computer's resources, including the CPU, memory, and network connection. **These resources are likely to be very different in capacity and speed than those found on a mobile device requiring tests of performance, memory usage, and networking speed to be run on physical devices."**

Next, consider this quote from the Android Studio User Guide (emphasis added):

"When building an Android app, it's **important that you always test your app on a real device** before releasing it to users."

Mistake to Avoid #5: Building Your Own Mobile Lab

Building and running a functioning mobile testing lab on your own can be challenging. From keeping up with and managing devices, making sure that developers and testers have the tools and resources they need to deliver on time, and the volume of tests that need to be run, there are a lot of moving pieces and parts.

Also, if you're working with Appium, building and running your own Appium servers is not only time-consuming and expensive to set up on your own, but slow performance and slow scripting can cause your enterprise mobility team additional stress.

It is also important to remember that when building your own mobile testing lab that devices are typically only dedicated to functions like test automation. While for manual testers, the testing devices are still in testers' and developers' hands to keep up with and manage.

By choosing not to build your own mobile testing lab and implementing a device cloud solution your team solves all the tough challenges like Appium server setup. The team can also use the same devices for manual and automated testing from the device cloud where the devices are always online and available.

For real results and to get the most accurate insight into how an app or mobile website will function in the real world, enterprise mobility teams should test on real devices.

With so many existing solutions available, why rebuild the wheel when you don't have to?

Just don't do it. Seriously.

By avoiding these five mistakes, your mobile development and testing teams can rest assured that embarking on the continuous testing journey can be as painless as possible. With the right tools, support and methodology in place you safeguard your processes from unnecessary conflict and difficulties that could prevent your team from doing what they do best—providing superior mobile experiences.

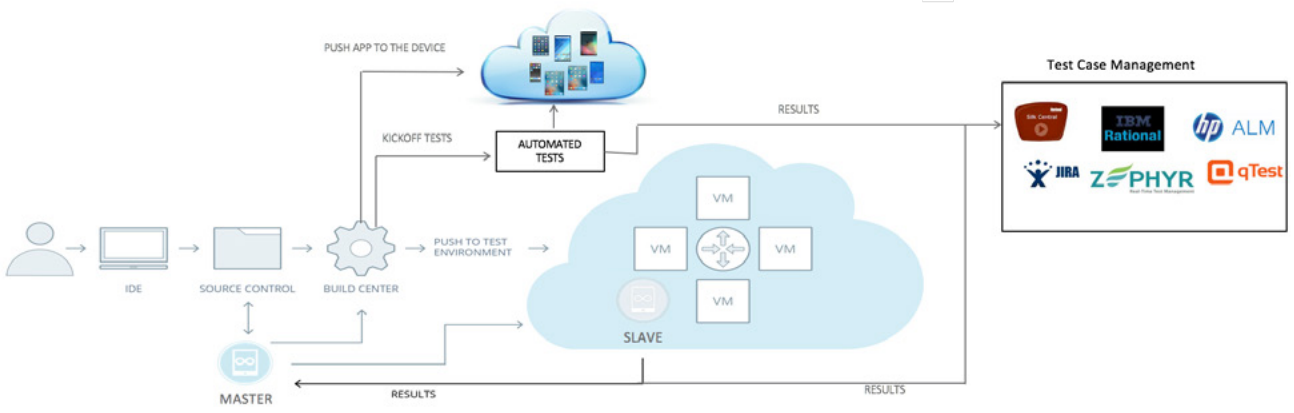
Want to learn more about building a successful continuous delivery pipeline? You can download our latest eBook on the subject here.

Why Building a Continuous Delivery Pipeline is like a game of Mousetrap

By Mobile Labs

How to Build a Continuous Delivery Pipeline

Climb into the "wayback machine" with me for a minute. Do you remember the board game Mousetrap? If you're familiar with this game, then you know that each action in the game builds toward an eventual chain reaction that is necessary for you to capture your opponents' mice. Each step in the game is dependent on the step right before it and right after it.



Let's compare this scenario to building a continuous delivery process flow. You know that you have a series of steps that must be created and successfully executed to build a mobile application. But, along the way your team may find issues that need to be addressed quickly.

By setting up a continuous delivery process flow, every member of the enterprise mobility team can have the agility and flexibility to debug and to keep the app functioning correctly.

1. Create a Delivery Process

The first step is choosing the right tool to create a delivery process. There are many CI/CD tools that you can choose, from commercial options (Team City, Bamboo, Urban Code) and open-source (Jenkins, GoCD, Concourse) that your team can explore. If you already have a CI/CD tool, then it is probably best to extend this tool to include your mobile delivery.

2. Create the Pipeline Needed to Build the App

The next step is to create the pipeline needed to build the application. This means checking out the source code and compiling the app. During the build, it is recommended to make your unit tests blocking, meaning that the entire build process will fail if a unit test fails. This gives the fastest feedback to the development team to create a fix.

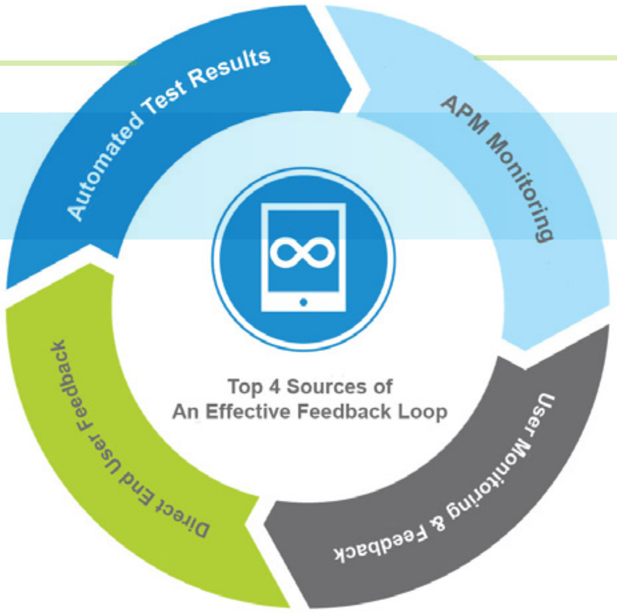
3. Install the App on a Device For Testing
Once the unit tests are passed, the next step would be to install the application to the device for testing. With a device cloud, this can be done through add-ins or API calls to upload the application to the cloud infrastructure and install it on the appropriate devices.

4. Run Automated Tests Against the Device
Now that the application is built, the unit tests are passed, and the application is uploaded and installed on the device, it is time to run automated tests against the device. To do so, simply use the built-in mechanism or add-in to execute the tests. For Appium tests, this usually means to have your test code pulled from the repository or test management system and executed through the build process of the test framework you are using. For those that are using Java, simply execute the JUnit or TestNG tests of your Maven or Gradle builds.

5. Manual Testing and QA
The test results are then used to determine if the build has passed and is ready for internal teams to manually run through the application to find any errors or admissions. This is usually the final step before declaring it ready to go out.

The good thing about this process, is that it makes it easy to repeat for every change because the whole process is automated. This speeds up delivery and provides faster feedback. The faster it is to detect an issue, usually the faster it is to fix. The goal is to speed up delivery and increase quality.

Get User Feedback



But what happens next? After successfully setting up a continuous delivery flow with all necessary tests in place, you need a way to collect results and to receive real time reports. After all, how can you catch and fix any issues if you are unaware of any trouble in the first place?

The answer lies in setting up a continuous feedback loop. By setting up a closed loop process, your team will receive feedback and notifications about any issues as soon as they occur. When you have quick feedback, your mobile developers, testers and QA can isolate issues and debug quickly to keep releases on track.

But, what should you include in your feedback loop?

For your enterprise mobility team to really be agile, you need to receive feedback in a way that is easy to digest. With so many tests running, your team could potentially receive "feedback overload" making it difficult, if not virtually impossible, for your team to sort through. To make feedback meaningful and easy for your team to act on, you want to put QA testing solutions in place to give you targeted, focused feedback. Here are four of the most important sources for your team to monitor.

4 Important sources for feedback

1. Automated test results
The automated tests that you set up, whether unit, functional or performance will feed their results back into your continuous delivery pipeline or to a test case management system. Regardless of where you are getting your results, they must be managed and analyzed by your team. Any execution issues or defects must be addressed by your team quickly to stay on track with releases. Make sure to make your automated tests rock solid so that in the event of a failure, there is no guessing if the test failed due to issues with the test or with the product.

2. APM monitoring
When making sure that your app is running at its peak performance, it is important to measure how the system is running. APM monitoring tools can help provide real time performance metrics about the health of the system. APM covers a lot of different areas such as performance and availability of the systems, transaction timing on the device and the network, as well as bottleneck detection. All of the data provided gives you insights into how your application is working from the app side all the way back to the systems that support the app. If an issue arrises you can get an immediate alert if something is going haywire. Once an alert is triggered, everyone can jump in and diagnose the issue quickly. Making sure that your critical transactions are successful is key to having a quality app.

One way to get earlier feedback from APM is to leverage the APM tools during pre-production testing. Having this data before the app hits the users can help narrow down issues that can be fixed quickly, reducing costly issues one the app has been deployed.

3. User monitoring and feedback
If you want a better understanding of how your users are interacting with your mobile apps or websites, then user monitoring reports can help you better understand their behavior. By reviewing these reports, you can see which features of an app are being used, or how users are engaging with different areas on a mobile website.

User monitoring delivers actual, real time information about key trends, user-timing and other metrics. You can use these findings to innovate and improve the mobile experience.

4. Direct End User Feedback
But, test results are not the only feedback your team receives about apps or mobile websites. Don't forget about your users, or mobile consumers. This audience has high expectations for mobile experiences, and checking out any feedback on your app or mobile website is beneficial for you team to consider when making improvements.

One way to get feedback is to monitor the app store reviews. Figuring out why your "star" rating is moving up or down gives valuable feedback about the quality of your application. Reviews can give insights into problems with the application, but it can also give you insights to what customers want more of in the app or on the mobile website.

Early feedback can also come from early release programs for your application. Apple has a program called TestFlight to get early releases of your application out to end users so you can get feedback earlier. You can also use crowd sourced testing programs, like Rainforest and Applause to get users to run through your application and provide valuable feedback.

Additional Resources

MORE INFORMATION FOR SOFTWARE PROFESSIONALS

ROI CALCULATOR	What's the ROI of Parallel Testing Your Mobile Apps?
INFOGRAPHICS	Accelerating Mobile App Transformation 10 Tips to Build a Successful Mobile Testing Lab
SURVEY REPORT	Automated Testing: How To Accelerate Mobile App Transformation
WHITE PAPER	Jump Start Mobile App Testing
CONTACT MOBILE LABS	info@mobilelabsinc.com | +1 (404) 214-5804 | www.mobilelabsinc.com