Shortening the Last Mile of Your Release Process

A Pragmatic Approach to Managing Risk in Continuous Delivery

test IO

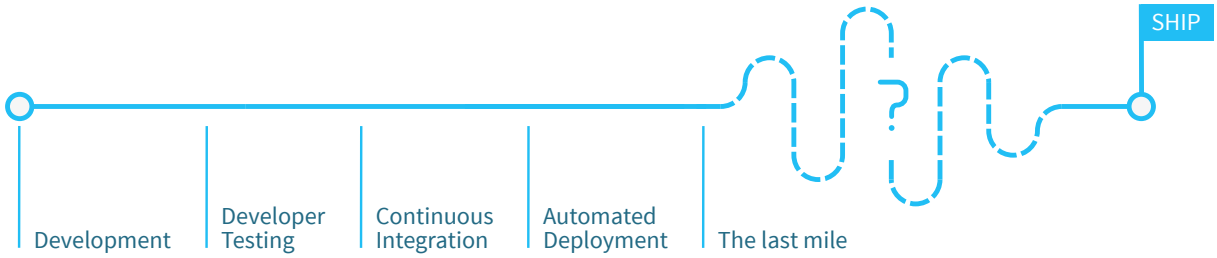
Just Because You Can, Doesn't Mean You Should

If you're like many leading edge development teams, you've worked hard to create an environment for continuous delivery of software. You probably have -- or are working on:

- Developer testing through unit, integration, and API tests
- Continuous integration that gives your developers near-immediate feedback when they check in code
- Automated mechanisms to deploy new code into production

With a solid team of developers, a QA team focused on testability and automation, regular code review, and a dev ops infrastructure that can push your code at will and then monitor its vital signs, you're feeling good about continuous delivery.

And then there's the last mile.



There's an old joke in software development that 90 percent done means you're half-way through the project. That's the last mile -- the elusive 10 percent when you need to decide whether to deploy the software to production, and for which customers. It's the time when you think the most about risk. It includes tasks such as:

- Regression testing the app to ensure that changes don't have unforeseen functional consequences;
- Identifying front-end bugs that are hard to write tests for, but cause your users fits and affect conversion rates;
- Certifying the app across all the browsers, operating systems, and form factors you need to support.
- Ensuring that the app works the way customers think it should, through the many paths they take, sometimes through manual testing, dogfooding, user acceptance tests, or beta programs.

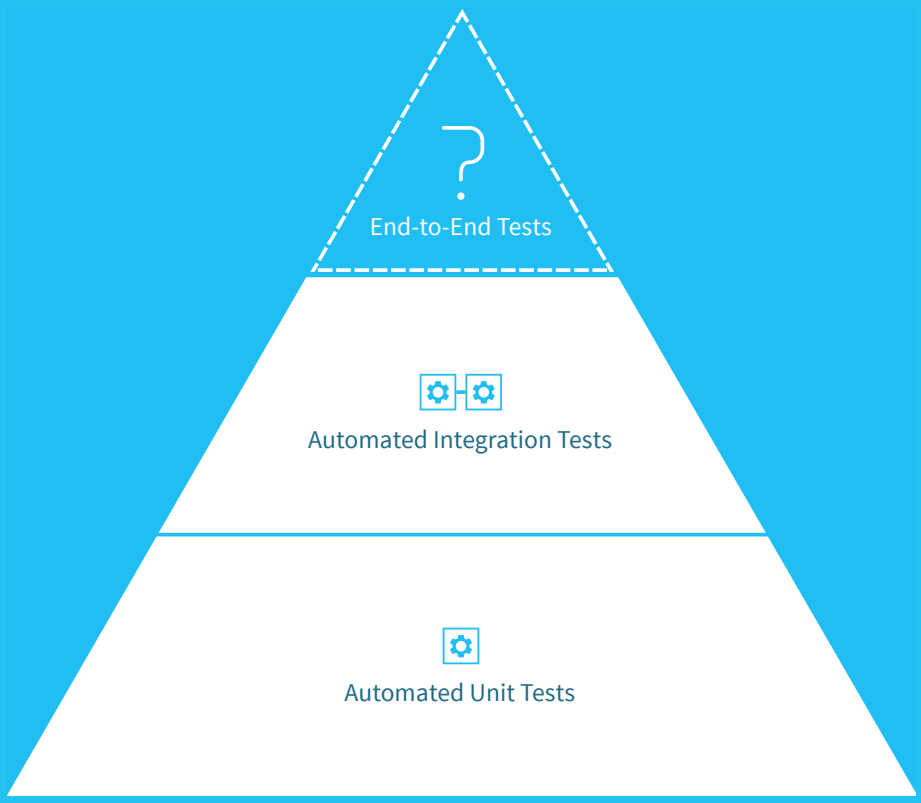
If you've done everything right, you can push code into production whenever you want, but the last mile of the process is still challenging because you need to know whether the reward for your business exceeds the possible risks -- and for all the automation you've done, you're still not sure.

If you have a consumer-facing app, you face considerable market risk: lost revenue, lost time, bad reviews, app abandonment, the embarrassment of pulling a product from the app store. As a result, even productive development teams typically slow down during the last mile, sometimes leading to multi-day regression testing cycles, inefficient bug reporting from "volunteer" testers, expensive software developers performing manual tests when they could be coding, and time spent patching instead of working on new features.

Continuous delivery focuses heavily on developer-driven test automation throughout the development process, and most teams aspiring to a continuous delivery model can boast high levels of automated test coverage -- particularly unit test coverage. Yet many teams slow down at the last mile because developer-driven test automation at the last mile is hard to do well, not only because such tests are notoriously flaky, but because human judgment is sometimes required when software is designed for humans.

In this paper we'll discuss practical strategies for optimizing the last mile: how to prioritize the development of automated tests while efficiently deploying human judgment in the testing and release process.

What Makes a Good Test



Perhaps you're familiar with the Testing Pyramid, originally developed by Mike Cohn and refined over the years by various people.

Every version of the Testing Pyramid has unit tests at the base. Integration tests, which may be of several types, typically form a second layer, and "end-to-end functional tests" top out the pyramid.

The percentage of effort will vary based on the company and the circumstances, but a rough rule of thumb suggested by Google is that 70 percent of your tests should be at the bottom of the pyramid, 20 percent in the middle, and 10 percent at the top. That 10 percent represents the tests that most closely approximate your customers' user experience.

In other words, tests that most closely approximate what your users will actually experience get the least amount of developer testing.

Why is this?

In a continuous delivery environment, the most important function of developer-driven testing is not to catch bugs before they reach your customer, it's to prevent bugs from being introduced into the codebase in the first place. To fulfill that mission, developer tests need to be fast to execute, reliable, and specific in the errors they uncover, so developers can run tests that meet these criteria every time they change code and find out pricesly what breaks.

Table 1 enumerates five characteristics that make up an ideal test. We can see from this table that compared with end-to-end automated tests, unit tests are particularly valuable as part of a developer's day-to-day work because they are fast and scale well.

Fast to execute	Developers can execute tests on their own computers and run hundreds of them in a minute. End-to-end automated tests are slower to execute.	Unit Wins
Reliable and robust	Unit tests are isolated and small, and thus execute reliably. End-to-end automated tests are often flaky and not robust to user interface changes.	Unit Wins
Scalable and timely to create	Developers create unit tests with their code, so they are timely and scale linearly with your development team. End-to-end automated tests require effort at the end of a project and are costly to maintain.	Unit Wins
Specific	Since unit tests operate against individual units of code, they give developers precise information about what has failed, down to the line number.	Unit Wins
User-sensitive	Here end-to-end automated tests beat unit tests: since they're using the full software system, they're closer to what the end-user actually sees than unit tests are.	End-to-end wins

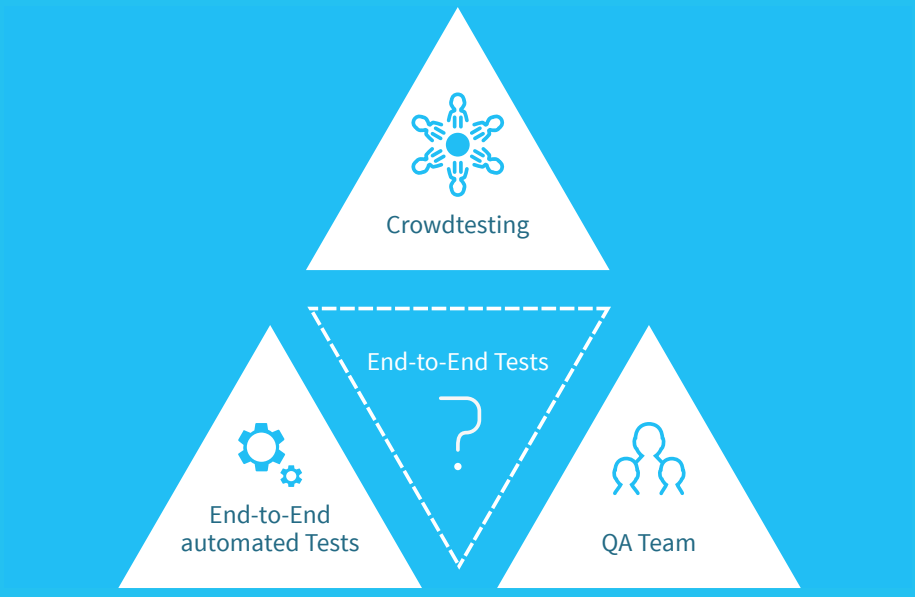
Table 1

This is why development organizations have prioritized the development of unit tests over end-to-end tests. If you haven't done this and you have a lot of end-to-end tests that are flaky or expensive to maintain, this table may explain some of your frustration.

If you're working with a legacy application, there's a strong temptation to write a lot of functional tests to automate regression

testing. As Michael Feathers has shown, this is typically a bad idea. Better to search for inflection points within the code and methodically write write tests against that code, then refactor it to make it more testable. Only your development team can do this kind of work, so from a resource management standpoint you are better off using manual testing to regress the application rather than asking your developers to write flaky tests that don't improve the code. Many organizations have found crowdtesting to be a practical way to augment their testing efforts in these cases.

Testing in the Last Mile



Now we come back to the last mile. What's the best way, once you've reach the last mile -- or the top of the testing pyramid -- to optimize your testing mix?

Recall that while unit tests prevent bad code from entering your trunk, last-mile tests help you understand the risk-reward calculus inherent in releasing software to customers. In short, does the new functionality you're offering justify the risk that you'll introduce new bugs or otherwise damage your company's brand?

Despite this difference, let's think about last mile tests using the same criteria we've already used for developer tests -- with two differences. First, an end-user's point of view becomes relatively more important the closer we are to shipping. Second, the available options are different. For the last mile, we have three different testing strategies, each with strengths and weaknesses.

- End-to-end automated tests: tests that use the browser or a mobile simulator to drive test cases that simulate human use of the application.
- QA Team (or extended team including developers): functional tests performed by your in-house team, whether a dedicated QA team or developers deputized temporarily to test the application.

- Crowdtest: functional tests, which may combine tightly scriptedtests cases and more organically defined session-based tests, performed by a group of outside software testers using real devices.

Here each type of test has its advantages, and your optimal strategy will depend on the state of your application and the risks associated with obvious customer-facing problems. Table 2 explains the trade-offs.

Criterion	Machine-Driven	Internal Manual	Crowdtest
Fast to execute	Fastest to execute.	Time consuming convene an internal team to test, and a "bug bash" is expensive if your team's time is valuable.	Quick to convene and can test in parallel, delivering results in an hour or two.
Reliable and robust	Theoretically execute the same way every time, but are not robust to small changes in code or environment and often need to tweaking.	Your internal team knows the application and knows what's changed, so they test what's changed. But small numbers of people mean higher variability.	Crowd testers are numerous, so they cover your application broadly. Human testers are not thrown off by small changes to the UI the way a machine is.
Scalable and timely to create	Challenging to write well and generally can't be written until feature development is far along.	Often difficult to find enough capacity to test when you have a big release.	Scales instantly; you can convene more testers when you have more to test.
Specific	Can be flaky and generate non-specific errors.	Particulalry helpful in debugging difficult issues.	Particulalry useful in reproduce issues across devices and environments.
User-sensitive	Only a simulation of your end-user.	Human-driven, but often too familiar with your app to test like an end-user.	Diverse and closest to your user base.

Table 2

If your application is changing very slowly and is well-covered by unit tests and integration tests, you can prioritize end-to-end automated tests. Although such tests can be flaky and expensive to maintain, the relative stability of your application means that you won't be throwing away work on your test scripts every week.

On the other hand, if your application is changing rapidly, if your unit test coverage is less than you would like, and if there is a great deal of market risk associated with failures in production, crowdtesting may be your best option for the last mile.

Crowdtests can return results as quickly as an hour or two, and unlike an internal team "bug bash" you can easily get more testers when your application has undergone more changes. Crowdtesters can test the application on real devices, and because they constitute a more diverse user community, they more closely approximate your ultimate customer base.

End-to-End Automated Tests

Do

- Write tests when you are sure the app is very stable, against very stable parts of the app.
- Create an automated "smoke test" that's fast enough to run with every build and covers the application's most crucial features.

Don't

- Spend time writing automated tests against features that are changing rapidly. Devote your developers' time to improving unit and integration test coverage instead.

QA Team

Do

- Use in-house teams to test features that are sufficiently complex that explaining them outsiders would involve significant training.
- Embed internal testers with development teams for close debugging support.

Don't

- Force developers to run tedious manual tests to "teach them the value of writing their own tests."
- Expect that your team to uncover the same kinds of issues that your customers will find. People who created an application or know it well carry expectations about how works that your customers don't share.

Crowdtesting

Do

- Use crowd tests when there are many paths through your application.
- Prioritize crowd testing when the user interface is changing rapidly.
- Use crowdtesters to cover legacy applications during significant refactors
- Use crowd tests when UX problems will impact your revenue or brand, or you're distributing the application through an app store and the price of making a small change is high.

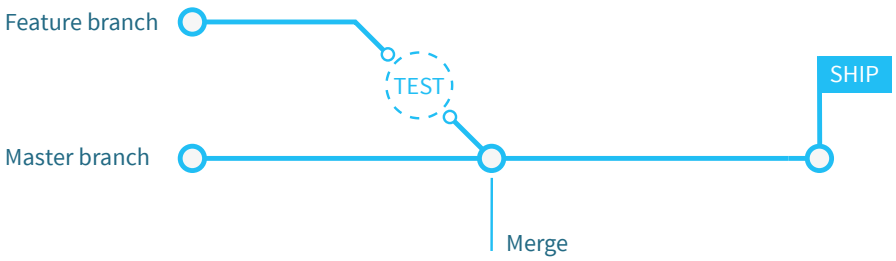
Don't

- Give crowdtesters such strict instructions that they become effectively human versions of automated tests. Take advantage of testers' skill and powers of discovery.

Mapping Your Last Mile

When your “last mile” tests happen depends on your branching and release strategy. Continuous delivery is framework of practices that ensure your code can be rapidly and safely deployed from your trunk to production at any time. But while this may be true from a technical standpoint, as we have seen, businesses must perform a risk/reward calculation in deciding when to deliver software to customers. At the highest level, we see last mile testing occurring in one of three moments: prior to the merge of new features into the main codeline, on a staging environment using the main codeline, or in production.

Feature Branch Testing



Feature branch testing has gone somewhat out of fashion in a continuous delivery environment because a feature branch is further from your production environment than tests of a true staging environment. However, depending on the types of risks your business encounters, extensive last-mile testing at the feature branch may make sense. In particular:

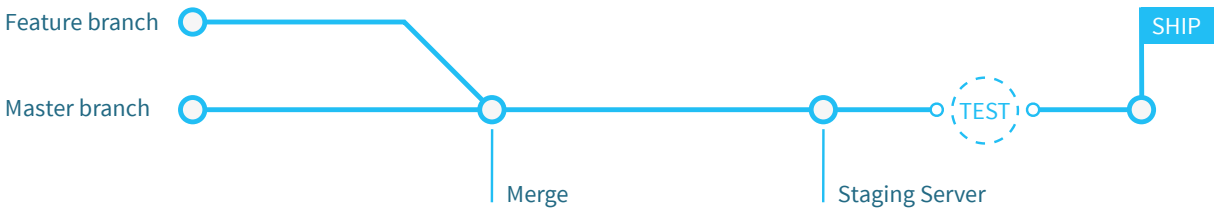
- When it is impractical to “eat your own dogfood” exposing your code to people outside your team at this point helps avoid unanticipated user experience challenges. While many organizations shy away from a “full user acceptance test” among target users because recruiting the right customers is challenging, a crowdtest is a quicker, more process-friendly alternative. Crowd testing is particularly effective for

applications with many user interface paths, and these paths should be tested earlier in the process in case any options need to be foreclosed or changed.

- When performing major refactoring, a more thorough last mile test with a full regression will reduce risks. Refactoring of front-end components is particularly risky, since automated tests often miss user interface changes that are obvious to customers.

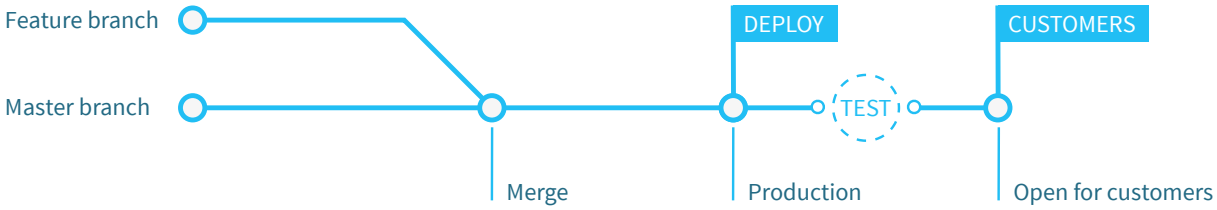
Testing on Staging

Performing your last mile tests on a staging server is the tried-and-true method, employed by most of the continuous delivery organizations we see. Here the biggest challenges mirroring production conditions as closely as possible. Some best practices include:



- Maintaining data for staging that is as close to production as possible. It is often worth automating the obfuscation of production data so tests in the staging environment reflect the latest data from production.
- Testing across a range of devices. Android devices, in particular, feature a range of screen sizes and resolutions. Last-mile tests should ensure a consistent user experience for mobile apps and responsive web apps across mobile platforms. Several options exist, from hosted emulators to cloud-based devices; crowdtesters using real devices will most closely mirror real-world conditions.

Testing in Production



It seems like heresy, but some organizations test in production -- by choice. Particularly good candidates for this are companies with good testing coverage at the unit and integration level that have reliable end-to-end tests, and whose business-critical activities are not likely to be affected by functional or UI bugs. Advantages include:

- There is no human bottleneck, so you can “push on green.”
- Your production environment is a production environment, so you’re not worried about simulations affecting the outcome of tests
- When you need to update server code to support new features in apps, you can push new code to the server and test the new apps against it in a production environment.

Some best practices for last-mile testing in production include:

- Ideally, you want to be able to push “new” code to a subset of users first using a feature flag or other means of redirecting traffic; your testing corps, including crowdtesters, can be among the vanguard users.
- You can combine testing with monitoring to get a more holistic view of errors in production, decreasing the time to resolution.
- With crowdtesting, you can schedule an in-production test for an optimal time, either when your customers are less likely to be visiting your website or when your developers can react most effectively to the results of the test.

Conclusion

Technology and DevOps methodologies are rapidly advancing, to the point where the technical barriers to distributing software to customers are almost non-existent in many cases. But with great power comes business responsibility, and the question of when -- or even whether -- it makes business sense for customers to receive new software remains challenging.

Businesses investing in continuous delivery of software need the best possible information at the appropriate time to make that judgment. A well-considered last-mile testing strategy that combines automated checks and human insight is a requirement for all such businesses.

Mitigating Risks with Crowdtesting

Apps that change a lot, depend on features of the user's device, and are sensitive to a user's location and network conditions need the most thorough last-mile testing. Multiply column A by column B then add the risk factors to see how useful crowdtesting is for you.

	Column A: Level	Column B: Importance	Risk Factor
Level of User Interface Change Release over Release	0 1 2 3 4	0 1 2 3 4	
Android Platform Support	0 1 2 3 4	0 1 2 3 4	
iOS Platform Support	0 1 2 3 4	0 1 2 3 4	
Integration with Third-Party Logins	0 1 2 3	0 1 2 3	
Integration with Third-Party Payments	0 1 2 3	0 1 2 3	
Uses GPS or has locale constraints	0 1 2 3	0 1 2 3	
Dependent on device permissions, hardware	0 1 2 3	0 1 2 3	
Responsive web UI	0 1 2	0 1 2	
User-generated content	0 1 2	0 1 2	
		Result	

1-20 Points

With strong unit and integration test coverage, automated end-to-end tests should get you where you need to go.

21-40 Points

Your application requires considerable human oversight. If your in-house team is constrained, investigate crowdtesting.

41+ Points

Crowdtesting will almost certainly help you release software faster and with greater confidence.

For Further Reading

The ice-cream cone anti-pattern:

https://watirmelon.blog/2012/01/31/introducing-the-software-testing-ice-cream-cone/

Google warns against too many end-to-end automated tests:

https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

A different view of the testing pyramid:

https://www.joecolantonio.com/2015/12/09/why-the-testing-pyramid-is-misleading-think-scales/

How Google releases software:

https://docs.google.com/presentation/d/15gNk21rjer3xo-b1ZqyQVGebOp_aPvHU3YH7YnOMxtE/edit#slide=id.g437663ce1_53_376

Dealing with Flaky Tests:

http://martinfowler.com/articles/nonDeterminism.html

Working with Legacy Code:

http://www.netobjectives.com/system/files/WorkingEffectivelyWithLegacyCode.pdf

When to automate functional tests (and when not to):

http://www.softwaretestinghelp.com/10-tips-you-should-read-before-automating-your-testing-work/

https://www.techwell.com/techwell-insights/2016/01/when-and-when-not-automate

http://techbeacon.com/dos-donts-testing-automation

Testing in production:

http://sdtimes.com/testing-in-production-risk-vs-reward/

http://www.neotys.com/blog/tips-for-testing-in-production/