2018 QA Insights

LESSONS FROM TEST IO'S CROWDTESTING DATA

test IO

Table of contents

Introduction

For this report, we set out to assemble test IO's collective knowledge about quality assurance and software testing, informed by the daily operational testing we do with our customers, as well as the insights and lessons we've garnered from QA testing at scale.

The scope at which test IO is working is significant: In 2017 alone, our customers ran nearly 8,000 tests on our platform, the equivalent of around 18.5 years of testing. In those same 12 months, our testers reported over 100,000 bugs. Using this data as a starting point, we began looking for insights along many different lines of inquiry, analyzing text, frequency, and data points from app sections, submission time, bug report titles, and other areas.

For example, discovering that more of our customers test in production than in staging environments lead us to take a closer look at their testing habits in general. We examined the reality of their testing usage, rather than assuming ideal situations and what our customer success managers recommend as best practices. It also nudged us to examine more closely the types of results our customers get and how fast they know when something's wrong. You can read more of our findings about testing in production and the best practices around agile QA in the chapter "Doing It Responsibly and Safely."

test IO's customer success managers work with software teams to use in-depth tests (focused tests, in test IO parlance) and coverage tests on business-critical parts of their software, like the checkout section. Their knowledge of this common pain point lead us to examine the different kinds of critical bugs that can disrupt the checkout process on websites. Our detailed findings, including which types of bugs are most common in checkout, you'll find in the chapter "Bugs Are Damaging Your Conversion Rates."

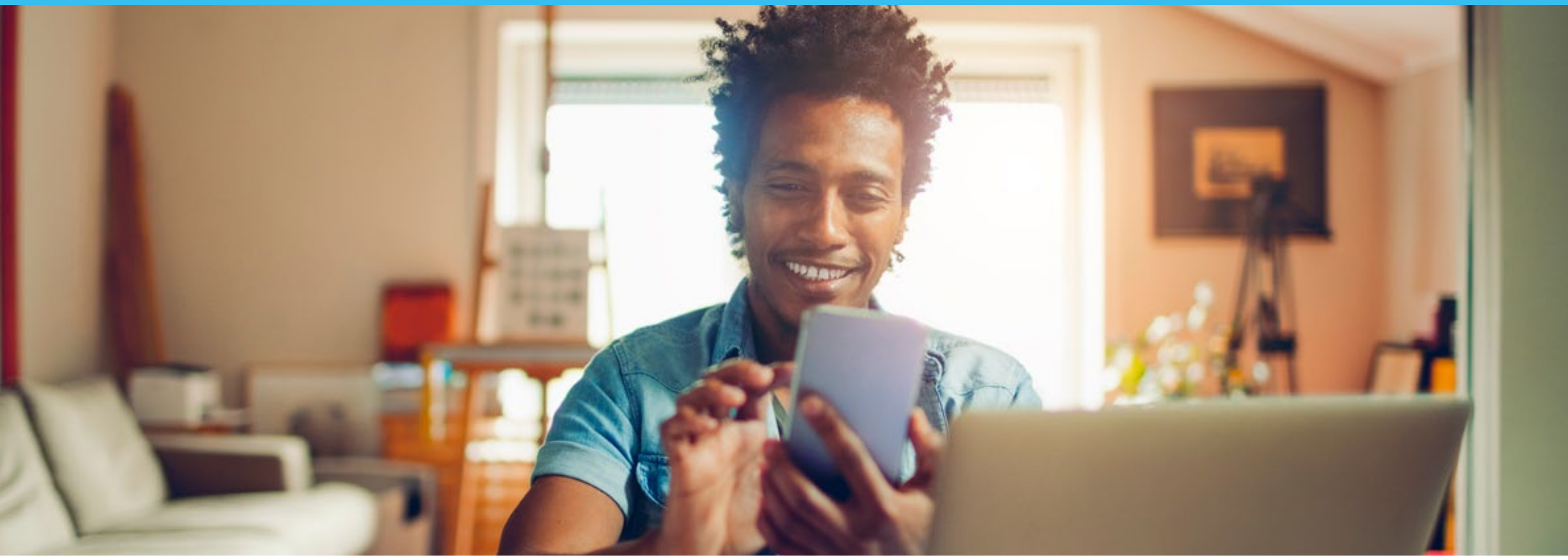
With over 6,000 detailed app crash reports from recent software tests, we delve into the challenges that are part of developing successful mobile apps. By examining the common factors that lead to mobile app crashes and the unique struggles of rigorous quality assurance testing on mobile devices, we uncover some pitfalls to avoid. These and our key lessons for avoiding the most common mobile app crashes are in the chapter "3 Reasons Why Mobile Apps Crash."

Many companies test without rigorous investigation of which platforms should be prioritized, which devices their customer use, and which ones tend to be more prone to bugs. Using our data set of over 100,000 bugs, we look for which device and operating system combinations are buggier than average and which types of environments are undertested. Our discoveries about where the most common gaps in testing coverage tend to be are in the chapter "Top Platforms You Should Be Testing But You're Not."

3 Reasons Why Mobile Apps Crash

THE CHALLENGES OF MOBILE APP TESTING

test IO



Mobile testing presents certain challenges that other kinds of software testing do not. If a company wants to take environmental factors into account, it's not possible to simulate them reliably in the lab. What are these environmental factors? Mobile phones, in addition to their wireless radios connecting them to mobile networks and Wi-Fi, have developed to include sensors like GPS, gyroscopes, accelerometers, barometers, light sensors, and compasses. Any one of these conditions is difficult to simulate in the lab; the varying combinations only compound the difficulty.

Aside from the environmental factors that make mobile testing difficult, acquiring and maintaining the library of devices to test adds significant burden to the QA process for a company that wants to make sure its mobile apps work on a broad and representative set of devices. In addition to multiple models of devices, QA teams need to have enough different mobile devices in their library to cover device size, hardware configurations, and operating system versions. These additional complexities increase the testable combinations for mobile apps exponentially, as they're usually another layer on top of the environmental, real world factors.

The consequence of these two challenges for mobile testing, combined with the limited time and resources most teams face, lead development teams to limit the scope of their testing. Often, this takes the form of "happy path" testing, in which in-house QA testers, who know what features or changes were included in a new app build, end up trying the simplest, most direct way to use a feature -- also known as the happy or golden path. For example, the latest build of a mobile ecommerce Android app includes a new feature for sharing products via email. Testing the happy path, a tester opens the app build to be checked on a freshly rebooted, newer Android device running the version of plain Android that most of the developers use. The tester then, while connected to the office wireless network, clicks through to a product in the app. After scrolling on that screen, the tester clicks the share button, and enters their work email address, and hits "send". The tester reports that the feature has been successfully tested.

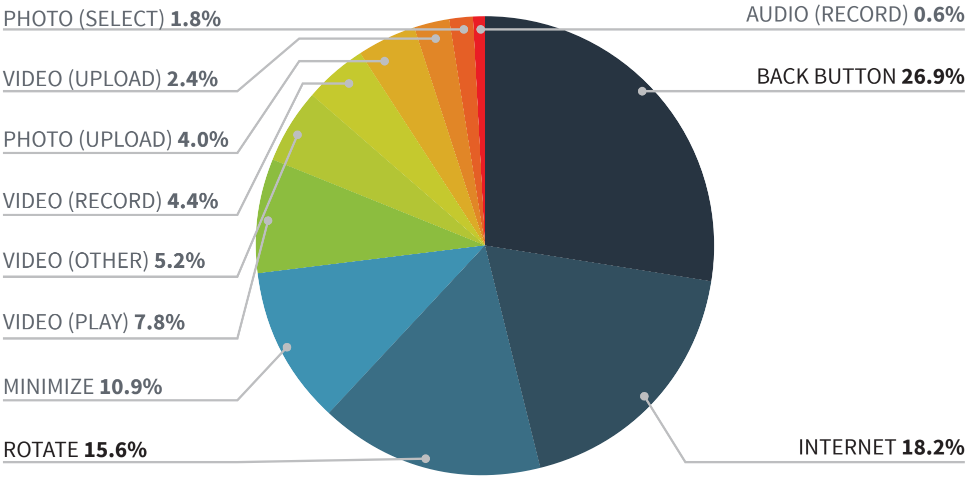
But the feature has not been rigorously tested in real-world environments on real devices by real humans. Here are some of the possible app-crashing factors that would not get tested in the above scenario:

- Real users often use multiple apps, so running your app in isolation means you miss interactions (or even RAM limitations).
- The majority of users don't upgrade their phones every year or update their operating systems regularly, so devices with older hardware running one of the older versions of Android are common.
- On Android phones in particular, there are manufacturers which have created their own versions of the operating system.
- The app may work differently (or not work at all) if it doesn't have access to strong and stable data connection. Users may have spotty Wi-Fi access. Apps also react differently to the limitations of mobile data networks. How does the app work if there is no data connection at all, as in airplane mode?
- A real user (hopefully) does more than open the app, click, and share. They'll likely explore different app sections, switch between apps, minimize the app and return, before ending up on the product page they want to share.
- Often users rotate the app into landscape to get a better or alternative view of the app.
- In a real world situation, the user might click share, then need to switch to another app to find the email address they want to send it to, tap the back button, and return to the original app to type it in.
- Users also might use a different share target -- WhatsApp, SMS, notes applications, to do lists, social media sites.

Methodology

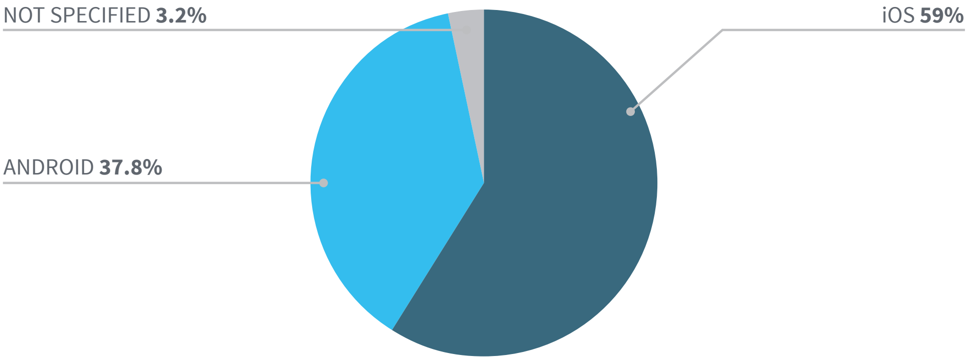
For this analysis, our goal is to determine what the major causes of app crashes are. First, we took the set of bug reports whose titles included the word "app" and "crash" (as well as the variations "crashes" and "crashed"). This yielded a set of 6,224 bug reports. Then, we performed word pair analysis on the bug titles, looking for the most common combinations of words to see if we could group the crash-related bugs into broad categories. These preliminary categories were issues related to internet connection, device rotation, and the back button. Then using keyword searches, we manually reviewed and categorized the bug reports which recurred with enough frequency to indicate recurring, significant issues. Besides the three categories derived from word pair analysis, we also found large concentrations of app crash bugs around photo, video, and audio -- in recording, selection, and uploading, as well as minimizing. Beyond these categories, which cover approximately 1,150 of the total set of bug reports, we did not find groupings that included enough similar bugs across different software products to warrant inclusion in this analysis.

Reasons for Mobile App Crashes



The mobile platform division is often an interesting facet of these crashes. Of the crashes overall, slightly more (58%) were reported in tests on iOS devices. This is not a finding on how iOS devices are more prone to crashes than Android, which we know is not the case. It does highlight the slight preponderance of iOS device testing on the test IO platform, and we also do not control for the frequency of tests run on the same product. Of the unique software products in our data set, slightly over 50% were iOS.

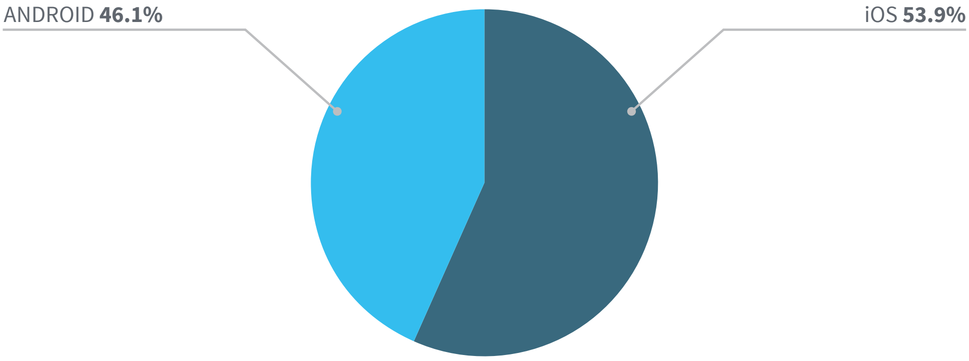
App Crashes by Operating System



Back Button

Tapping the back button in an app is the cause of more than one quarter of the app crashes in our data set. On Android devices, this primarily indicates the hardware system button, while on iOS devices, the back button is a software button written into the app interface.

Back Button Crashes by Operating System



For Android devices, the system back button navigates the app screens the user has recently worked with, in the reverse order the user interacted with them. It's separate from the app's internal screenflow or hierarchy, as the back button tracks the last several screens, independent of which app they belong to, in the "back stack."

Software back buttons on iOS and Android apps, like those in desktop browsers, only keep track of the screens and paths followed within that specific instance of that app. This enables users in ecommerce apps to return to product pages from a shopping cart, or to return to a level "higher" in the navigational hierarchy.

One common cause of back-button related crashes on Android apps occurs when app creators try to handle the back press event within the app, usually in order to prevent other crashes or problems. However, when the previous screen or activity requires some sort of input or "intent" from the previous screen, and pressing the back button doesn't include that -- which causes the app to crash.

Here's one example of an Android app that crashed during the registration process. The tester launched the app, entered their phone number, received the one-time-password, and continued on to the rest of the registration form. While on this step, the tester left the registration flow in the app and used other apps on their phone for a few minutes. Upon resuming the original app's registration process, they clicked the back button to leave the registration form. At this point, the app crashed.

For iOS, the reasons for back button-related app crashes are much more diverse and depend on the frameworks and the design of the application. In our analysis of app crashes, we found bug reports of crashes in many different areas: after entering incorrect passwords followed by the back button; clicking the back button after viewing a news article; clicking back after navigating to a screen in the app while it is still saving and updating information on a previous view; and clicking back after using the "password forgotten" feature.

Some high-level lessons from our back button bug set analysis, which apply to all mobile apps:

Is the app structured to require this kind of navigation structure at all?

Are the views hierarchical or chronological enough for users to know what a back button will do at all times?

Does the app account for the various views and situations where the back button might be used? Examples: after a user minimizes and returns to the app; after submitting something on a previous screen; after restoring an iTunes purchase.

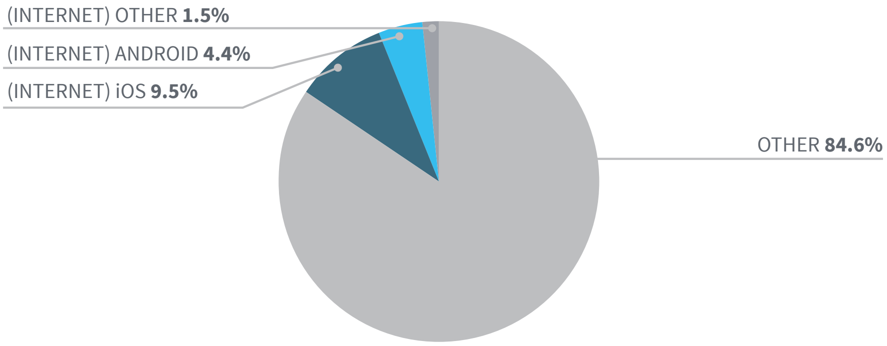
How many in-app contexts or screens does the back button apply to, and can your code handle this?

Internet Connection

Around 18% of the app crashes documented in our bug set resulted from internet connection issues. In most of these cases, the testers documented that they turned on airplane mode or turned off Wi-Fi access at a given point in the steps that led to the bug.

This kind of testing is one of the reason why mobile QA in real world environment is essential. Most developers and in-house QA teams will test while at their offices, while connected to the fastest internet connection their company can get. While conducive to testing productivity, it doesn't mirror the situation that users will often find themselves in: on mediocre Wi-Fi at home or out in public, or on their mobile device's cellular data connection.

Internet Connection-Related Mobile App Crashes



How necessary an internet connection is, is often dictated by mobile app design decisions made quite early in the process. It determines whether the app saves any information on the device, even temporarily. It also influences how much of the app that the user installs is included on the mobile device, and how much requires communication with the company's servers. Even if most of the views and activities take place on the user's device, the app might still check for updates, verification, or synchronization points that require an internet connection. Once these decisions are made, software teams still need to choose how the app reacts when the internet connection isn't present or only intermittently available.

Even the business model can come into play: if an app displays video, interactive, or other advertising to users, a poor or nonexistent connection to the internet could interrupt those ads. Some flows may need to be redesigned to take into account what happens if the app can't immediately connect to the internet with minimal impact to users.

If an application is totally dependent on a robust internet connection, that needs to be clear to users from the beginning. This is particularly important if the functions of the app don't appear to require internet connectivity to work. Should the app check the internet connection upon being started? Instead of crashing or not loading if a connection isn't present, the app can display a message. Similar behavior can also apply if internet access is lost while in operation.

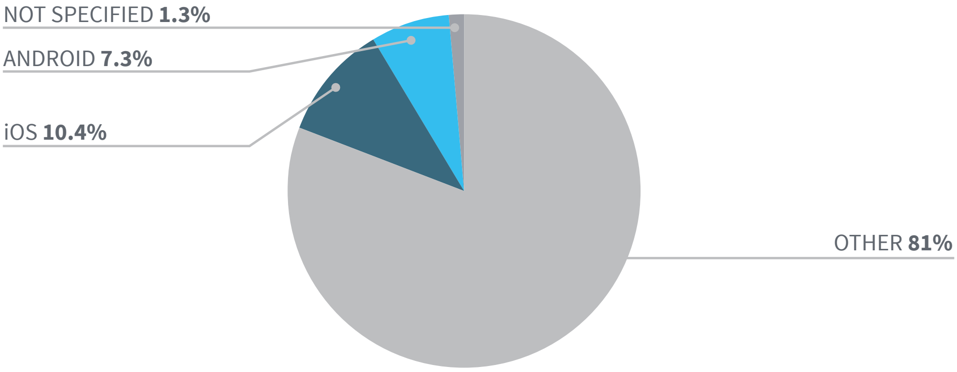
Device Rotation

On mobile phones and tablets, the built-in accelerometer sensor enables devices to determine when they're being rotated, and whether the screen should display in portrait or landscape mode. Many mobile apps use this to build two different layouts for the entire app flow. Others may only use it for displaying and capturing videos and photos. Both Android and iOS also permit users to "lock" the screen orientation so that rotating the device doesn't result in a change in the app display orientation.

From our analysis, we found that just over 15% of mobile app crashes were connected to rotation problems, either changing from landscape to portrait orientation or vice versa.

These rotation-related crashes were distributed among both Android and iOS apps, with slightly more reported on iOS. This is not out of proportion with the overall larger share of iOS products tested and overall app crashes in our data set.

Rotation-Related Mobile App Crashes



There are many possible reasons why device rotations cause apps to crash. The most likely culprit, however, is possibly missing input or view variables. When a user rotates the device and the app needs to change from portrait to landscape mode, the entire view needs to be reprocessed for the new screen orientation. If the user is on a screen reached by entering information or which required some sort of input from the previous view in the app, this same input is required to redraw the new screen orientation. When an app doesn't have the methods in place to pass this on in the event of rotation, the app may crash.

Another common cause for rotation-related app crashes is when videos are playing, or other assets like photos or input methods like keyboards are on screen. In these cases, the crash can be traced back to how the video player, photo display, or software keyboards are called from the operating system. Rotation and orientation for these views may also require specifications from your app, and if there are inconsistencies or unsaved configuration details, this can cause a crash.

Conclusion

What can a software development team do to mitigate all these app crashes? As we outlined at the beginning of this section, mobile testing is difficult and different from testing websites or desktop software. The mobile phones and tablets that these apps run on exist in the physical world in a way that software on desktop computers and within browsers does not. Not even laptops can compare with the level of physical, environmental interaction that mobile device hardware deals with. With these real world, environmental factors like device orientation or cellular signal strength that affect how your app works, testing in simulators and in labs is not enough.

To make sure your mobile apps work in all the situations and on all the devices that they should, testing by real human beings in the real world is critical. With the propensity of developers and in-house testers to take the happy path and test only what the latest version of app is supposed to do, putting your app in the hands of professional testers who will tap twice on a button or rotate a device while it's loading a part of your app is the only way to make sure that it won't crash on your real users at a crucial moment. After all, the real world is messy, and who hasn't opened the camera in an app, hit the back button, and then rotated their phone to take a photo while in a cell service dead spot? That's all three of the most common reasons for an app crash in a single use.

Doing It Safely and Responsibly

NO, NOT SEX — SOFTWARE TESTING IN PRODUCTION

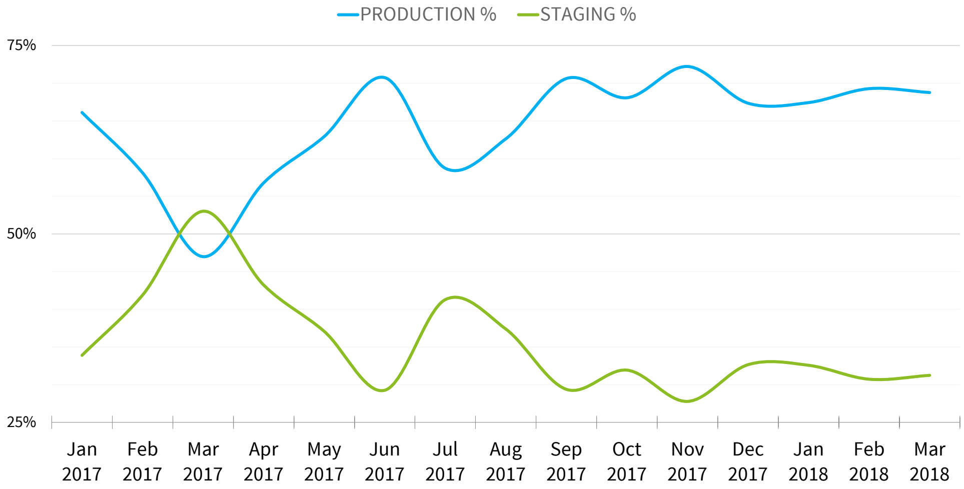
test IO

Who tests in production?

Development cycles are shortening and companies face more pressure to ship the latest software updates and features to customers as quickly as possible. Where testing takes place in the software development cycle has shifted over time, both as a matter of methodology and due to these market realities. Historically, most product managers and technical leads have considered pre-production user testing the gold standard. Any deviation from this practice is viewed dangerous, irresponsible, and the hallmark of an unprofessional technical organization. However, that's not the reality that many teams are living in. If someone tells you they're testing in production, it's not the punchline anymore. It's their process.

As teams shift to agile development methodologies and change code more quickly, testing and quality assurance cycles have become more frequent. The responsibility for testing has shifted to include developers in addition to QA testers. Sometimes testing in the live deployment environment is the best way to see how a complex application will behave, especially when it's not possible to reproduce all factors in a staging environment. At test IO, we see this reflected in how our customers test: a higher share of test IO customers test in production than don't, and more tests are run in production. Since 2017, almost two-thirds of tests cycles are in production, while just 35% are in staging environments.

Share of Test Cycles by Test Environment



In this article, we focus on the reasons why more companies should test in production. This isn't carte blanche to upend all QA process. Instead, we lay out when and why software teams should feel comfortable doing so. In particular, as testing in production becomes not only a matter of necessity but a new reality, there are best practices teams can put in place to minimize disruptions for users resulting from software issues in production.

The largest technology companies are pushing the envelope on testing in production. With huge user bases, Facebook and Google roll out new code constantly, but they use feature toggles to activate updates selectively. They'll almost always test internally with their employees first, and they have also invested heavily in test automation at all levels. This enables these big companies to take small incremental steps with tiny subsets of their users (but even 0.01% of Facebook's 2 billion users is 200,000 guinea pigs) to detect the impact these code changes have on their infrastructure and user experience. If there's an issue, they can toggle it back off or troubleshoot the problem quickly.

This is a relatively safe and responsible way of testing in production, but most software teams aren't working on products with the same scale and reach that would enable segmented feature rollouts. Indeed, the financial stakes for smaller companies of errors in production may be considerable: a bug in an order flow for an ecommerce site, for example, can mean lost revenue. Despite this, all companies face the same pressure to ship as quickly, if not more quickly, than larger companies, since "agility" and responsiveness to market needs are structural advantages that small organizations have over their larger competitors. Code ends up being deployed to production without having been tested, and without a QA strategy to catch the inevitable issues directly after the release or with a particular segment of users.

Different Testing Environments:

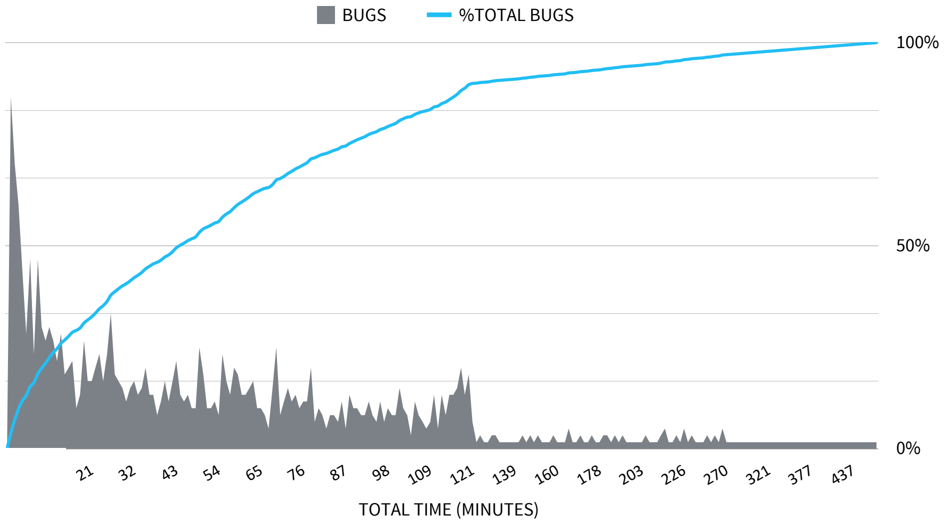
Staging and Production

According to our analysis, the overall number of bugs found per test cycle did not vary significantly between software tested in production versus staging. However, testers did find more critical bugs in staging than in production environments. test IO categorizes critical bugs as those preventing a core function of the app or website, that cause a potential loss of income for the company running the app or website, like an app crash. As received wisdom tells us, it's more effective, less disruptive, and less expensive to pinpoint bugs before they're exposed to users. If this isn't feasible due to other factors, the speed at which bugs are discovered in production becomes more of a concern.

Testing at test IO is exploratory, unscripted testing of software by a real person, a QA professional, reacting as a real user of the system might. Across all types of software tests run by test IO's customers in production, 70% of critical bugs are discovered within the first 2 hours. This means you'll have a high chance of discovering and being able to prioritize which serious bugs to tackle in a new version of your software that has been pushed to production.

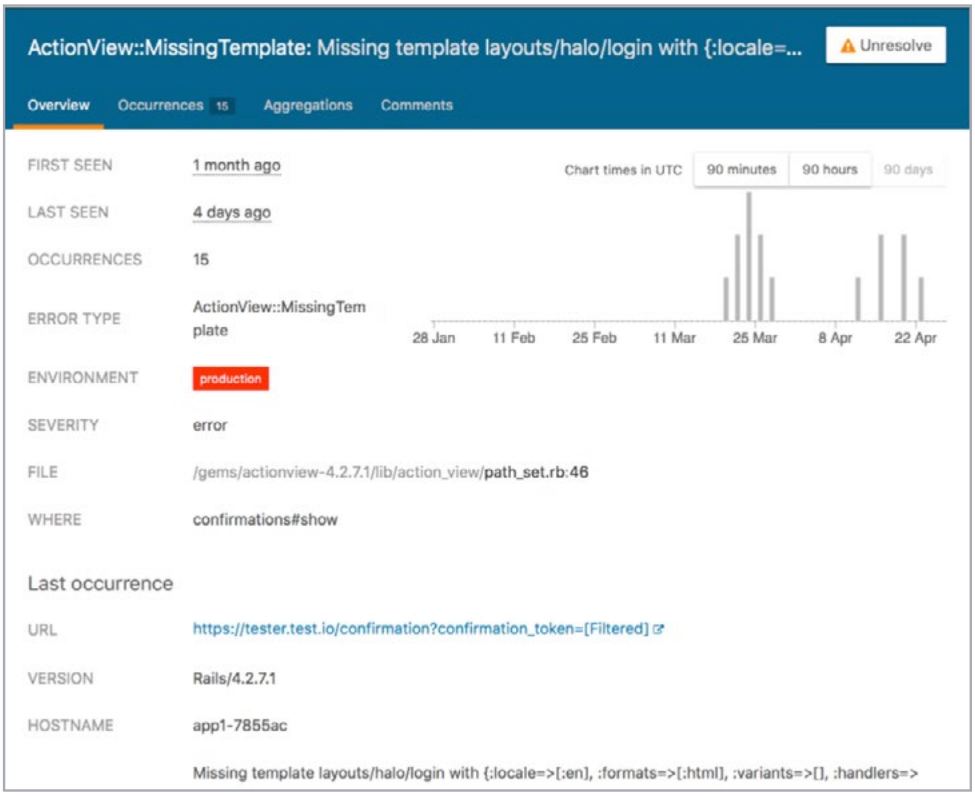
Speed is essential for tests in production where real users and customers are just as likely to be impacted by serious software issues. Rapid tests at test IO are exploratory, software quality tests designed to deliver results quickly, by checking core product sections for major or critical issues. When we look at the critical bugs in these rapid tests, testers report almost 90% of critical bugs within the first 2 hours.

Critical Bugs in Rapid Tests in Production



These rapid results enable developers to react quickly to issues that testers find in production, to either toggle the code back or to roll out a fix before a critical bug affects many users. The combination of production telemetry from monitoring systems (see example below) and detailed bug reports that crowdtesters provide cuts down on the time required to debug an issue. Compare this having the same issue identified by a paying customer, who might leave traces of her frustrating experience in the error logs or possibly in a complaint form, but will not demonstrate concrete steps to reproduce or provide a screencast of the issue. Moreover, a bug found in production is by definition not a bug caused by a subtle environmental difference between staging and production, so developers spend less time chasing potentially spurious bugs.

This is a screenshot of one of an error from Airbrake, an example of data provided by a monitoring system.



ActionView::MissingTemplate: Missing template layouts/halo/login with {:locale=...

FIRST SEEN	1 month ago
LAST SEEN	4 days ago
OCCURRENCES	15
ERROR TYPE	ActionView::MissingTemplate
ENVIRONMENT	production
SEVERITY	error
FILE	/gems/actionview-4.2.7.1/lib/action_view/path_set.rb:46
WHERE	confirmations#show

Last occurrence

URL	https://tester.test.io/confirmation?confirmation_token=[Filtered]
VERSION	Rails/4.2.7.1
HOSTNAME	app1-7855ac
	Missing template layouts/halo/login with {:locale=>[:en], :formats=>[:html], :variants=>[], :handlers=>

How to test responsibly in production

Testing your software through a human-driven process before releasing a customer-facing build is the theoretical ideal, but we've seen that it's not always feasible. While our statistical research does not prove this, anecdotally we see two basic paradigms at work for customers who test in production. You might call them two sides of a DevOps divide:

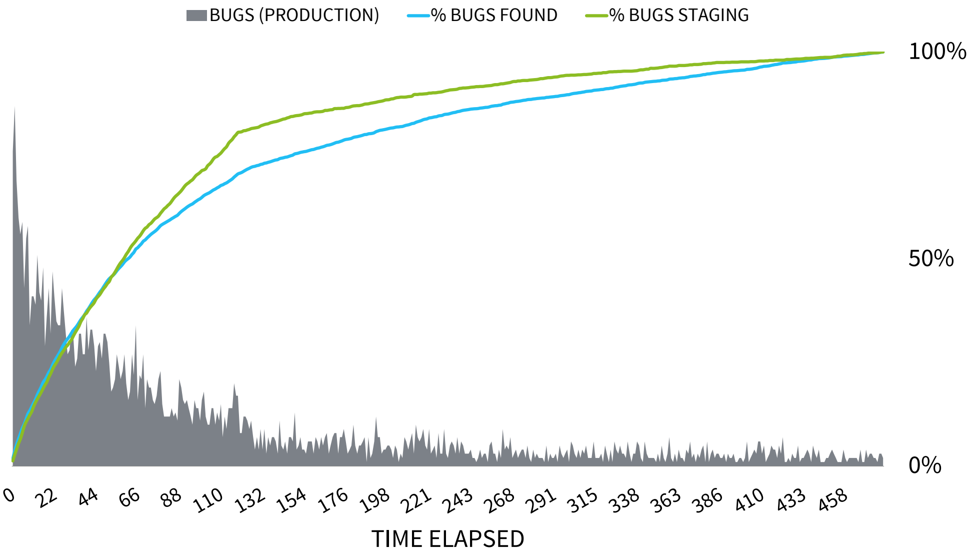
- **The Split:** in some organizations, there's sufficient distance between the people who develop the software and the people who are responsible for keeping it running that the latter need their own QA team to characterize issues they find and get them fixed.
- **The DevOps Nirvana:** in other organizations, test automation is sufficiently advanced and relationships between the development team and the production team are close enough that dev-to-production pipelines run extremely fast.

Without casting judgment on either approach, any software deployment contains the potential for critical functional issues that can severely impact your business and forfeit your users' trust. If testing in production is part of the reality of your release plan, as it is for many companies, there are strategies you can put in place to limit the risk of disruption to your product and to your customers.

These strategies fall into three main categories: limiting the time the software is "out in the wild" untested; limiting the users exposed by using feature flags; and limiting the users exposed by targeted geographic rollouts.

Running crowdtests for new software builds that have just gone live is the strategy with the least disruption for a software team's process. Some test IO customers have automated this process by creating tests that are always initiated when certain release milestones are reached or statuses are changed in their code and project management tools. Testers can begin right away on your website or on the new version of the application, and the issues they discover through exploratory testing are precisely the kinds that real human users will run into. In crowdtesting test cycles on production environments, 50% of critical bugs are reported within the first hour of testing.

Critical Bugs by Minute of Testing in Production



The second strategy for minimizing the exposure of users to critical bugs in production, which can be used in combination with crowdtesting, is the use of feature flags. Feature flags are toggles in your code to activate new code for certain segments of users. Large consumer technology organizations like Facebook and Google use these flags to turn on sections of their code for a portion of their user base, allowing them to test the rollout of new features or optimizations without changing the experience for all of their billions of users. If a limited rollout surfaces critical issues, the toggles can be deactivated or internal testers can be directed to document the precise environment and locations of the problematic code.

Smaller organizations can use feature flagging to roll out new code to specific target groups, like employees or crowdtesters. Using flags, software development teams can deploy updated code at will, but choose to enable new changes only for testers, for example. Once the new version has passed through the QA process, and any significant issues have been fixed (and tested), the feature can be toggled on for all users.

The third method for minimizing risk while testing in production applies primarily to companies with users in different countries. Whether they're rolling out new versions of applications to different locales via feature flagging or through 3rd-party distribution mechanisms like the Google Play or Apple App store, testing a new version of a software product in a non-critical market can align with certain business contexts.

These are real advantages to distributing new software releases to a non-core market. Real users come into contact with the product, and their behavior reflects that of users familiar with the software. Issues reported real users, even in peripheral market, get taken more seriously by internal stakeholders. Finally, if bugs are very serious, pushing the release in another country can avoid any severe loss of reputation or loss of customer trust.

A report in The Economist looks at how New Zealand has become a testing ground for social networks, app and game developers, and other kinds of software developers. Chosen for its relative isolation, English-speaking population, and relative affluence, companies like Facebook, Yahoo, and Microsoft feel comfortable trying out new software and new features in New Zealand or other regions because any issues can be ironed out without news of the problems reaching customers or media outlets in bigger markets.

Crowdtesting can also be part of a non-core market strategy. An organization can release to a smaller market and direct crowdtesters to the production version via a VPN or special access code. In this way, organizations can obtain detailed bug reports and fix the issues before releasing the application to a larger market.

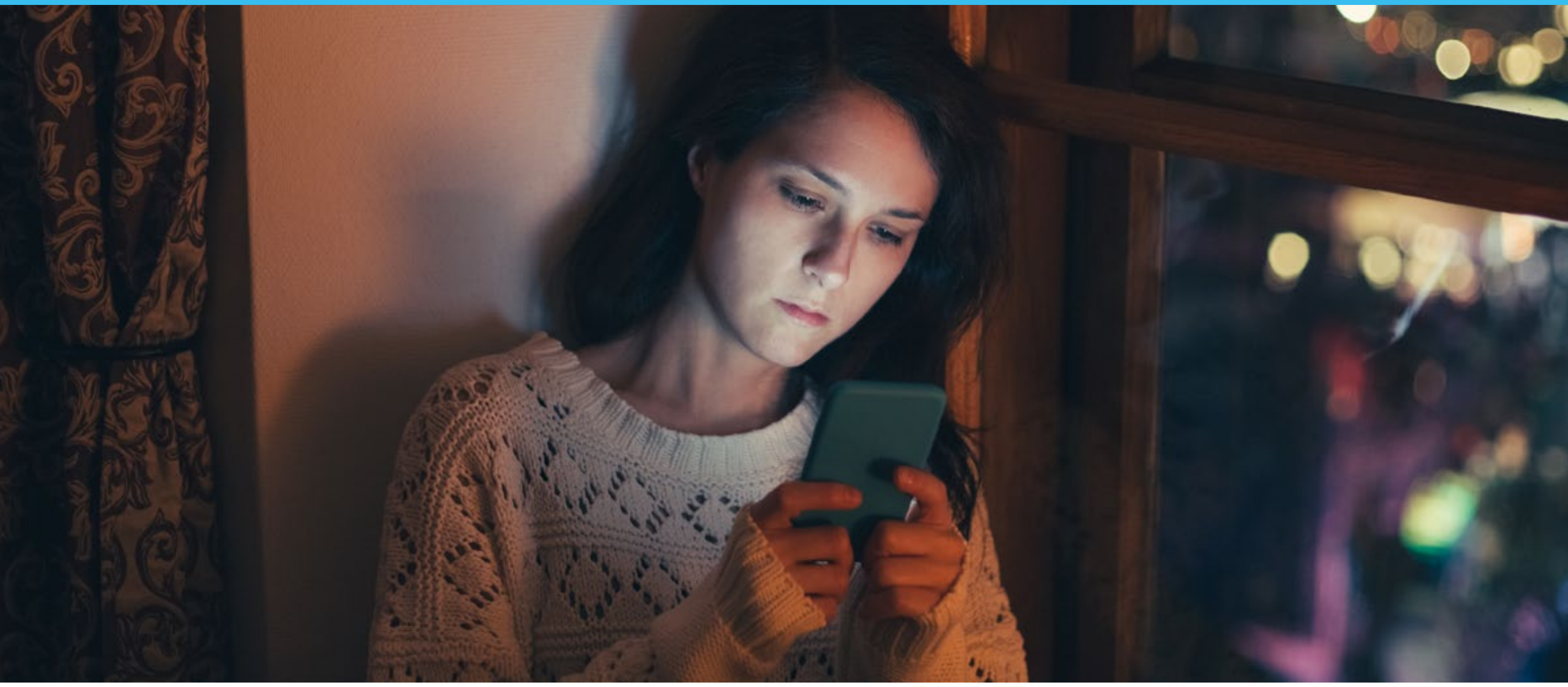
Conclusion

Just as public health advocates call for harm reduction strategies that take into account the lived realities of at-risk populations, software quality advocates also need to recommend practices that acknowledge the business, logistical, and resource pressures on product teams. That means looking at how teams are actually creating, releasing, and testing their software, not only the ideal conditions. That means instead of only calling for more QA before releasing to customers, it's practical and reasonable to implement development and testing strategies that recognize code changes may be pushed to production before being fully tested. By admitting that they do test in production, software organizations can find ways to reduce the disruption that severe bugs cause: by testing right after code releases, by putting feature flags in place, and by releasing first to nonessential geographic markets.

Bugs Are Damaging Your Conversion Rates

THE TOP CONVERSION KILLERS ACCORDING TO SCIENCE

test IO

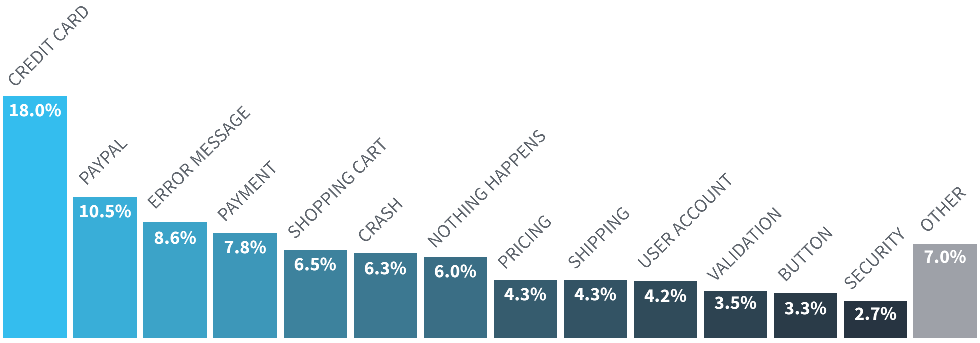


For this investigation, we examined bugs that testers reported for product areas (known at test IO as "app sections") that had the word "checkout" in them. This nomenclature is standard among our ecommerce customers. We limited the bugs to critical-level issues to focus on serious software defects. At test IO, we define critical-level bugs as those that prevent a main function of the app or website, or cause a potential loss of income for the company running the app or website. Ecommerce managers call these bugs "conversion killers," because they are issues in a software product that can prevent potential customers from making a purchase or frustrate them enough to stop trying, thus "killing" the sale, and the conversion of potential customer to an actual one.

Based on these criteria, the set of bugs included a total of 768 bugs, across 92 products from 44 companies.

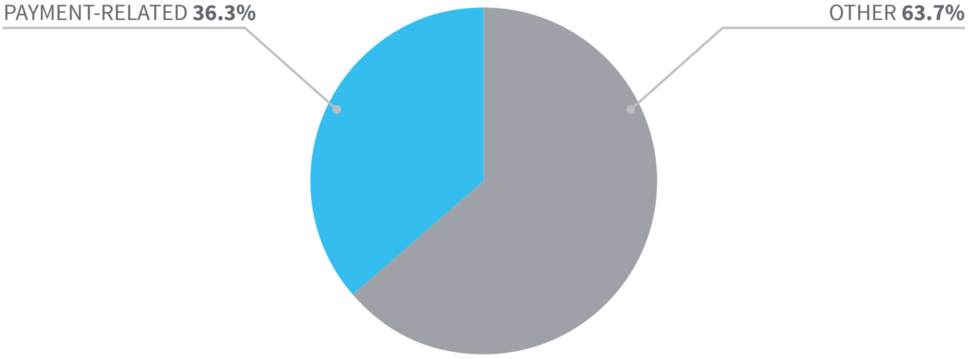
First, using word pairs generated from the titles of bug reports, we developed general categories in which to group the bugs. Then, by examining bug titles and descriptions, we refined these categories into more discrete groupings, separating in the process, for example, general payment-related issues into more specific categories like "credit card" or "PayPal" problems.

Conversion killers



Payments

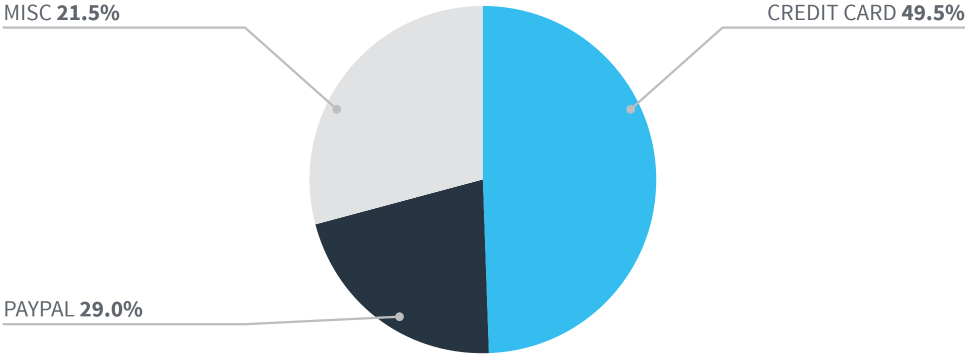
Payment-related Critical Checkout Bugs



Over one-third of critical bugs in checkout sections (36.3%) are payment-related. This is by far the largest grouping we discovered.

Within the broader payment category, we have separated the critical bugs into three more specific subcategories: credit-card related issues, PayPal problems, and all other payment issues. Nearly half (49.5%) of all payment issues involve credit cards. These include verification of numbers, dates, security codes; connections with credit card networks like Visa and Mastercard; as well as also assorted technical website issues on the credit card payment page.

Critical Payment Issues During Checkout



PayPal issues are similarly broad, encompassing problems as diverse as Paypal authentication; the integration of various PayPal services into a company's checkout process; and problems transferring checkout data between the product and PayPal. In the remaining section, the remaining kinds of payment issues include difficulties with other payment methods like bank transfers and direct debits; to voucher and coupon code redemption difficulties; to payment information validation.

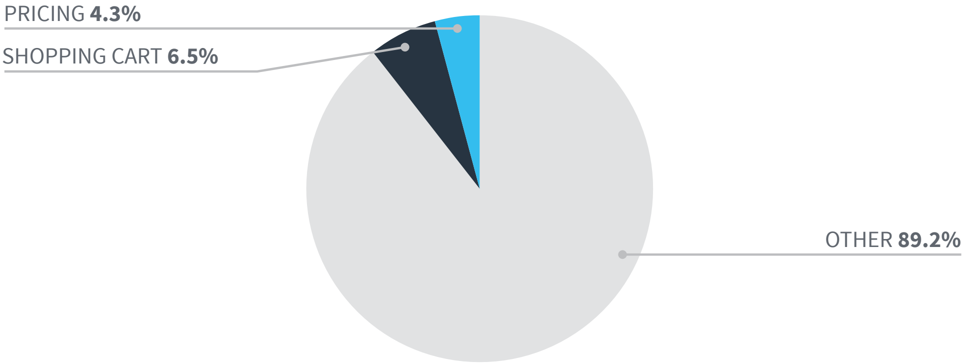
Accepting and integrating different forms of payments is a complex and challenging endeavor for software teams building websites and apps that include ecommerce functionality. There are many interlocking factors to take into consideration, like the varying requirements of different payment methods. In this survey of the payment-related checkout bugs, the following payment methods were mentioned: bank transfer, direct debit, Visa, Mastercard, and other credit card processing networks; PayPal and PayPal Express Checkout; Amazon Payments; Apple Pay; iDEAL; Sofortüberweisung; and SEPA. Each of these payment methods requires different fields for customer data and has differing security and authentication methods.

Creating and testing so many different types of payment systems can be difficult and unwieldy. Above all, it calls for familiarity with the implementation details of the payment systems, particularly for testers. For example, one recurring bug documented in multiple checkout sections prevented crowdtesters from using Visa cards with 19 digits. While credit cards with 16-digit payment card numbers are most common, the lengths of card numbers range from 12 digits up to 19 digits. Through an expansion on Visa's VPAY sub-brand, 19-digit cards are becoming more common.

Shopping Cart & Pricing

Pricing and shopping-cart related issues make up just over 10% of all critical-level issues that occur in checkout that we examined. Other than payment-related issues which we cover in the previous section, most other categories contain around 3-8% of total critical bugs. In this section, we look at shopping cart and pricing issues together. Shopping cart and pricing problems often display similar effects to users and testers.

Pricing Cart-related Checkout Bugs



Critical software bugs grouped into the "shopping cart" category include adding items to cart, removing items, changing quantities, discrepancies in item quantities, missing items, and faulty transitions between cart and checkout. Software bugs we categorized as "pricing" range from calculation errors, incorrect tax rates, delivery fee misapplication, prices not updating when shopping cart changes, incorrect pricing logic for promotions, and other assorted price-related errors.

Pricing and shopping cart issues can be difficult for automated tests to detect. They can be arise from a combination of specific products, manipulations, and other intersecting calculations, like tax and shipping rates. For example, on one particular software product, test IO's crowdtesters discovered a variety of intersecting pricing problems: shipping costs did not get included in checkout pricing; wrong tax rates were applied when the testers selected certain countries; and in other cases, tax information did not get displayed at all.

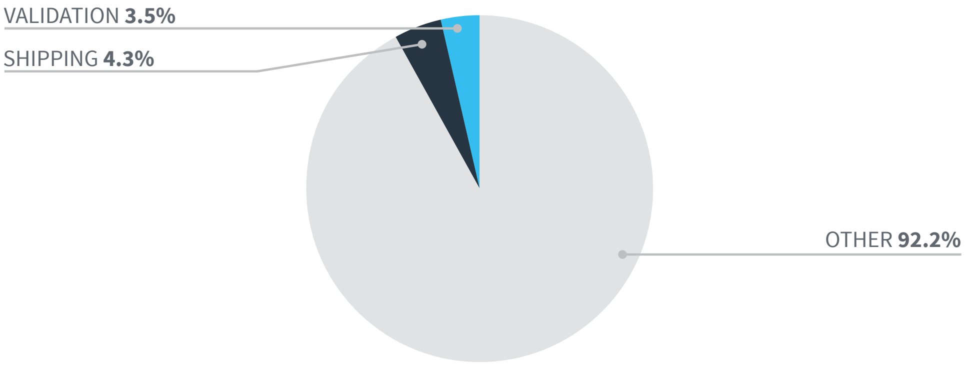
The key lesson to take away from our analysis of pricing and shopping cart difficulties is to not underestimate the complexity involved, and to avoid relying on "happy path" testing. Calculating prices for ecommerce orders can include promotional discounts, vouchers, coupons, shipping charges, and overlapping tax rates based on jurisdiction and product type. Furthermore, your shopping cart will need to display this information clearly, correctly, and without errors, in addition to showing correct quantities of items (and allowing their manipulation). If a potential customer changes the item quantity in the shopping cart, but the tax rate and shipping costs don't update correctly to reflect the new total price, that loss of confidence about what the final cost will be leads directly to a lost sale or distrust in the vendor.

Shipping and Validation

Errors around shipping and addresses, though less than 5% of the total pool of critical bugs in the data set, have an outsize impact. If there are difficulties entering a shipping address or calculating what the shipping cost is, then the potential customer won't feel assured that the ordered items will be sent to the correct address -- if they're able to enter it at all. Some of the shipping-related errors we uncovered in our investigation include

- Not being able to select a state
- Not being able to check out with an international address
- Not being able to add or save billing or shipping addresses
- Shipping to multiple addresses causing phantom items

Shipping & Validation Checkout Bugs



Validation problems in form fields are just as disruptive to converting customers. While they overlap to a degree with shipping issues, in mailing address fields, for example, errors that occur when validating other user-entered fields like email addresses, coupon codes, payment information, and phone numbers, impact many critical conversion points. In our analysis of critical bugs in checkout sections, 3.5% fall into the grouping of these types of validation errors.

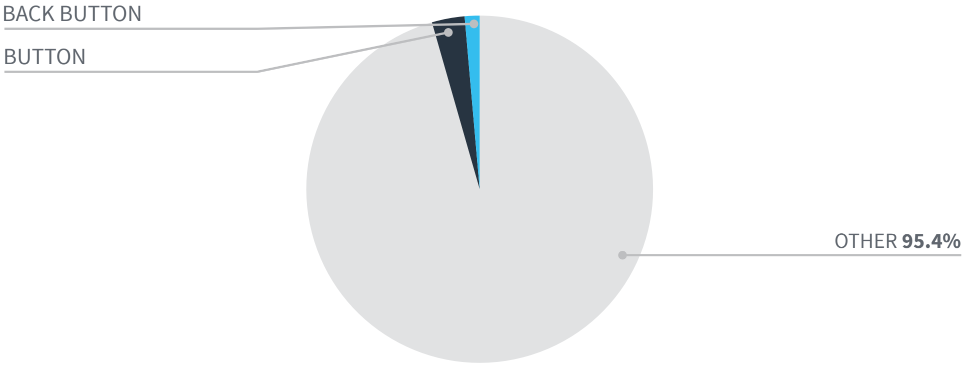
While the level of complexity in implementing shipping and form validation doesn't quite approach that of pricing and taxation, these are still areas where the average programmer rarely can plan for all edge cases and how they interact with each other. Choosing the right validation methods (libraries), modeling complicated interactions like shipping and checkout with care, and working with a broad range of testers who know what real-world problems can crop up in are a few strategies to avoid conversion-killing software bugs that can seriously impact revenue.

Buttons (Back & others)

Button-related errors are an even smaller proportion of the revenue-impacting bugs in our data set. However, as we cover in the "3 Reasons Why Mobile Apps Crash" chapter, they are the cause of a significant number of application crashes overall, so we're spending a little time examining how they affect checkout sections. The category of "back button" here refers to the browser back button, not to the hardware back button found on many Android devices.

The other button bugs typically concern interface elements which don't appear, disappear upon a certain action, aren't functional, or are disabled.

Button-related Checkout Bugs



Button software issues can be hard for automated tests to detect, since many of them are caused by graphical user interface display problems. In our experience, these are the kinds of problems that human testers uncover exceedingly well: when automated clicks can't tell that the button is out of sight of the screen, invisible, or covered by another UI element.

For back button bugs in the browser, many problems we discover relate to maintaining continuity across different checkout phases. For example, one bug report documents how all products a tester adds to the shopping cart are removed when they use the browser back button. This is not the intended behavior, nor is it the conducive to helping potential customers complete transactions successfully. Another issue report on a different product describes a similar bug: if the user clicks back from the PayPal payment page, they receive a "cart empty" message.

To avoid some of these back button issues, developers can store the state of the user shopping cart. By doing so, if the user clicks the browser back button, the contents of their cart doesn't get lost. This has the added benefit of allowing users to visit other sites and return to finish their purchase. It's also recommended not to break the back button functionality with automatic redirects. These redirects prevent users from navigating through a site using the normal browser buttons and history.

How to stop bugs from eating your lunch

After looking at all these different areas where you can mess up your software and lose customers, it's easy to feel like there's an insurmountable barrier to creating a website or an app that inspires confidence in your customers and doesn't prevent them from making a purchase. In our analysis of each grouping of bugs, we offered a few suggestions on how to directly address those types of problems. The good news, however, is that all of these bugs were found by software testers: they weren't reported by customers. The software teams had the opportunity to fix these bugs before they had any serious impact of revenue.

Some of these bugs were reported in staging environments, others were discovered during tests of live apps or websites in production. By using a crowd of human testers to check any complicated interactions like shopping carts and to handle graphical user interface issues like buttons, your team can focus on creating the automated tests for the types of bugs that those tests can catch more easily. Furthermore, though your team may not be able to fix all the bugs that are uncovered, you'll be working from a position of knowledge: you can prioritize the issues which have the biggest potential impact on the bottom line and your business. As long as your team commits to testing as part of the release process, it's possible to prevent significant revenue loss, offer a seamless experience to customers, and avoid losing customer trust.

Top Platforms You Should Be Testing But You’re Not

test IO

Introduction

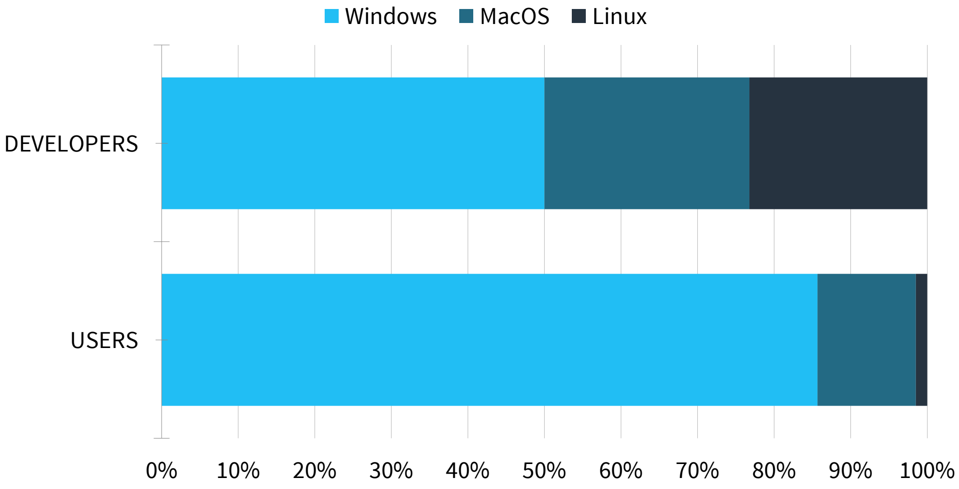
Choosing which devices, operating systems, and browsers to test is a perennial struggle for developers of websites and mobile applications. On one side, there's the aspiration to check your software and make sure it functions for as many potential users as possible. On the other side, the constraints are time and money: it takes time to execute functional tests on hundreds of devices; financial and staffing constraints limit the number of people testing on computers, tablets, mobile phones, and other devices. Once the issues are documented, it's non-trivial for a team to organize and prioritize the feedback.

Even with tools like emulators, simulators, and remote testing resources expanding what's possible for QA teams to handle on their own, testing strategies still need to prioritize which platforms should be tested, and subsequently, which issues should be fixed first.

Companies developing software can use data to determine which devices and environments to focus on in QA, but this type of prioritization overlooks one major type of manual, exploratory testing that's already happening: the kind done by developers while they're writing software. When developers work on feature branches or troubleshoot fixes to reported issues, they do this on their own machines. These smoke tests and other incidental manual testing all take place on the developers' local environment, leading to a significant bias around which issues are uncovered earlier in the development process and which ones get fixed.

In the Stack Overflow 2018 survey of developers, just over one quarter used macOS on their desktop computers, over 20% on Linux, with Windows representing slightly more than half of developers' operating system of choice. According to StatCounter's data on desktop operating systems for all users, over 85% run Windows, while about 12% are on macOS, and less than 2% use Linux.

Desktop Operating Systems



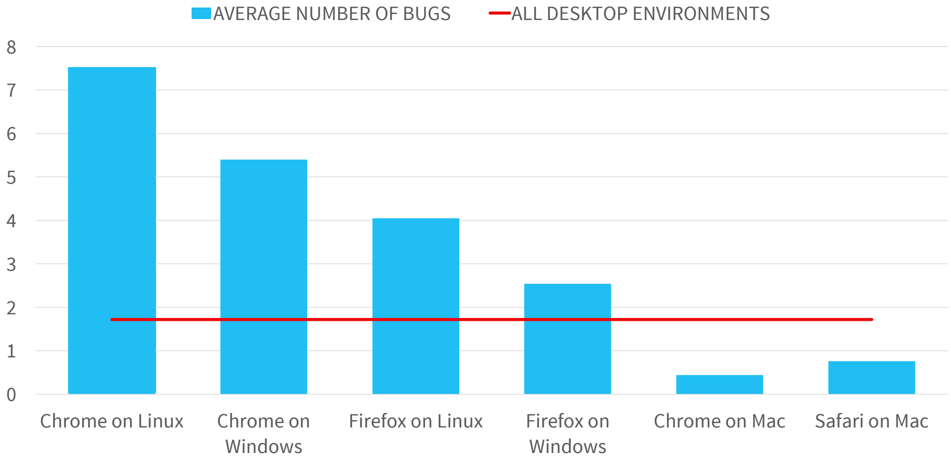
At test IO, we conduct thousands of exploratory, functional tests as well as usability tests on hundreds of software products. In 2017, our testers reported over 100,000 bugs to our customers. Each bug report contains screenshots, steps to reproduce, as well as the all-important details about the device, operating system, and browser (if applicable) of the tester. Using this treasure trove of metadata, we've analyzed which environments software are tested on to help determine which platforms software teams should prioritize in both running tests and fixing critical issues.

To that end, we're examining which environments are tested on most frequently, which device and software combinations have higher rates of bug discovery, and therefore which ones are often "undertested." We're also combining our testing data with public data on the prevalence of platforms, as well as surveys of developer data.

Testing Environments

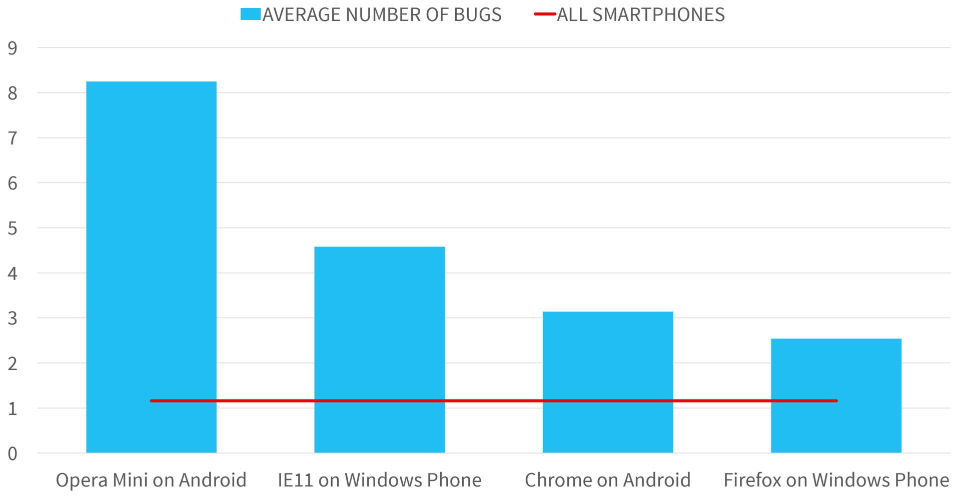
When a customer runs a test on with test IO's crowdtesting platform, they indicate what kind of software it is (mobile app, website, desktop software, etc.), what environments they want tested (devices, operating systems, browsers), and which sections of the app or website should be tested. There are different types of functional testing defaults that test IO offers, emphasizing either speed, device coverage, or feature functionality. Each test run, or test cycle, may run anywhere from 2 to 48 hours, depending on the type of test.

Desktop Test Cycles



The testing environments averaging the most bugs per cycle on desktop computers are Chrome on Linux (7.5), Chrome on Windows (5.4), Firefox on Linux (4.05), and Firefox on Windows (2.54).

Smartphone Test Cycles

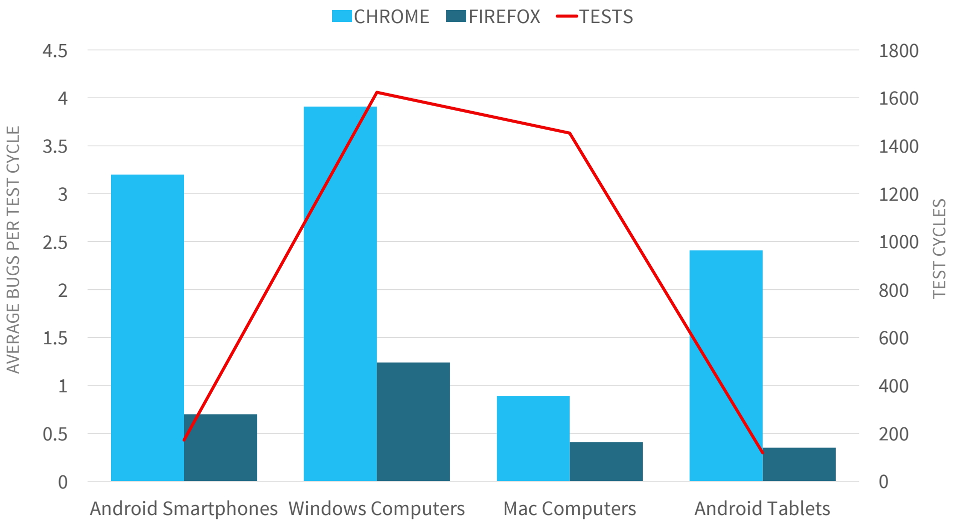


The testing environments that average the most bugs per cycle on smartphones are Opera Mini on Android (8.25), IE11 on Windows Phone (4.58), Chrome on Android (3.14), and Firefox on Windows (2.54). Beyond these environments, average number of bugs per test cycle was also above the overall smartphone rate (the red line in the chart above) for Android Browser, Firefox on Android, UC Browser on Windows Phone, Safari on iOS, as well as Opera Coast on iOS.

Which platforms are buggier? Direct Comparisons

For this part of our analysis, we compare test environments to determine which platforms are more prone to bugs. For these comparisons, we only looked at test cycles which invited testers to both environments. We then looked at the number of bugs reported by all testers for those testing environments and calculated an average across test cycles.

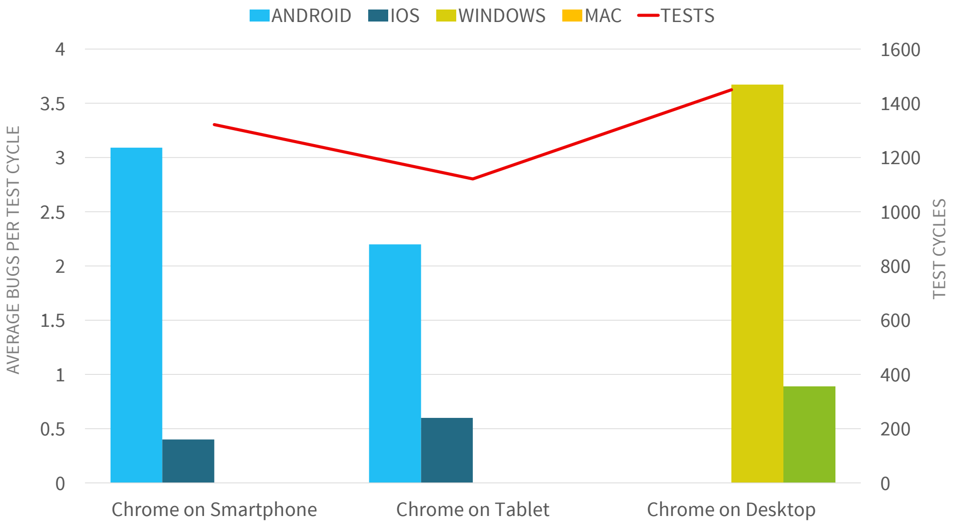
Chrome vs. Firefox Across Platforms



Our first set of comparisons all include the Firefox and Chrome browsers on specific operating systems and devices. Our analysis found significantly higher numbers of bug reported on Chrome per test cycle compared to Firefox on four platforms: Android smartphones, Windows computers, Mac computers, and Android tablets. The gap was largest on Windows, where the average number of bugs uncovered on Chrome per test cycle is 3.9, but only 1.24 for Firefox. The smallest difference is on iOS tablets, where testers uncovered an average 1.3 bugs on Chrome per test cycle versus 1.2 bugs on Firefox.

Though our analysis does not uncover any reproducible reasons for these discrepancies, we do have specific reports from testers that provide some potential insights. For example, on Windows and Mac computers, the tendency to install extensions and add-ons sometimes leads to particular issues in Chrome, and to lesser degree, on Firefox. On iOS tablets, the minimal difference between Chrome and Firefox can be attributed to the fact that on iOS, 3rd-party browsers all must use the Safari rendering engine, through WKWebView API. The comparison doesn't account for the higher incidence of certain browser and device combinations among testers. In essence, the popularity of the Chrome across all devices may also lead to more bugs being reported first on Chrome instead of on Firefox.

Chrome on Android, iOS, Windows, and Mac



Our next set of bug-frequency comparisons looks at Chrome on several different operating systems. Again, we looked at test cycles where both operating systems were tested. In the above chart, there were 1,321 tests which included Chrome on iOS and Android smartphones. The average number of bugs per test cycle for Chrome on Android was 3.09, while testers only found 0.4 bugs per test cycle on Chrome for iOS. The same pattern holds for Chrome on iOS and Android tablets, though the difference is not as large.

We also compared the incidence of bugs reported for the Chrome browsers on computers running the two most common operating systems: Windows and MacOS. For this comparison, we found 1,450 test cycles which required both operating system environments. The average number of bugs found on Chrome on Windows computers was 3.67, while Chrome on Mac averaged 0.89 bugs per test.

Minding the Gaps in Your Test Coverage

There are not strong correlations between the most common developer environments and the testing environments which average higher incidences of bugs. From our analysis of comparing bug incidences on different platforms, we do see that there is great variation in the experienced software quality for users. Whether a given operating system or browser is inherently buggier than others, while satisfying to discover, is less important than acknowledging that some platforms are tested more while other seem buggier to users.

Without a plan and process in place to test software rigorously across multiple platforms and devices, especially those which your team does not normally work on, you skew the bugs you find. Since users' common devices are different from those that your team works on, there will be gaps between what your users' experience of your software and what your team experiences as they're building it.

One way to deal with this discrepancy would be to force your team to adopt a set of devices to work and test on that more closely mirrors the breadth of operating systems, browsers, and hardware of your users. That's the kind of dogfooding that, while well-intentioned, may not make your team very productive or happy. At Facebook, management realized too many of their employees were using iPhones, which the company issued to them by default, leading to skewed internal usage and a notoriously bad Facebook Android app. To combat this, the company started a campaign to make employees aware that they could also request Android devices, known as "Droidfooding". While this was effective for Facebook, it's not possible for every company to do the same. No amount of posters in bathrooms will convince your employees to trade in their laptops for a slow, older machine running an outdated operating system.

For a more systematic and rigorous strategy to close the gap between your developers' and your users' regularly used platforms, you can bring in crowdtesters, like the ones test IO offers. test IO's crowdtesting harnesses the efficiency and advantages of crowdsourcing for software testing. We help distribute your software to a large group of people, our testers. These crowdtesters run your software or your website on their own computers or mobile phones to inspect it for defects, not on sanitized test devices. That means your software is tested systematically on a huge variety of devices in many different real-world environments. This includes the same privacy settings, ad blockers, and locales the testers normally use to create realistic software testing experiences as unlike your in-house testing as possible.

Conclusion

In this report, we investigated software testing and quality assurance practices, using the aggregate data from test IO's professional crowdtesting operations. In four areas, we examined the trends and behavior of our customers and our crowdtesters: ecommerce checkouts, testing in production, mobile app crashes, and software platforms. Here are our findings and recommendations from analyzing the different data sets in the relevant areas.

It's possible to reduce the negative impact of bugs on ecommerce conversion rates by having exploratory testers check the complex interdependencies of shopping carts, payments, and form validations. Prioritize human beings for these kinds of tests, and allocate in-house resources to test automation. Knowing what kinds of bugs you're dealing with enables you to work from a position of knowledge, fixing the bugs which are business-impacting.

Most teams don't plan to test in production; they end up doing so. Design your quality assurance strategy around the reality of your software development process, rather than aspirational and ideal testing procedure. By working within your actual restraints, you can achieve the best results for customers and your business. Acknowledging this truth enables teams to plan testing around limiting the time undiscovered bugs are in production environments.

We also see the need to grapple with real-world factors in mobile testing. The top three types of mobile app crashes we identified have environmental factors and real-user behaviors in common. Many of the difficulties of testing mobile applications relate to physical device features on mobile phones and the real-world environment they're in. Real user experience of an app is more varied than can be modeled in a simulator or in the confines of a lab.

In our investigation of which testing environments and software platforms have higher incidences of bugs, we discovered that there is wide variation in users' experience of the software quality. However, there is no simple answer that one particular combination (for example, Chrome on iOS smartphones) is under-tested and prone to bugs. Instead, our findings recommend looking at which devices and platforms your software team routinely develops on, and setting up a testing plan that tests rigorously on a broad range of devices, operating systems, and form factors. There's no substitute for a systematic approach.

The common thread in all of these findings is the need for an approach to quality assurance and software quality that is both structured and pragmatically rooted in reality, in the real-world experience of users and in the company's practical business context. Quality assurance needs to be structured and have sufficient testing resources to minimize "happy path" testing and to cover the platform, device, and real-world factor combinations that real software users have. At the same time, pragmatic prioritization of software testing recognizes limited resources and business demand for speed. This means recognizing that companies will test in production and focus on issues that directly impact revenue, like ecommerce features.

About test IO

test IO helps software teams ship high-quality software faster.

As a global leader in software crowdtesting, we speed up fast-moving software development teams with a platform for on-demand QA testing throughout the entire development cycle. Test setup takes just minutes, and we dynamically allocate human testers in real-world conditions to fit your specific testing needs. No more QA bottlenecks at the end of your sprints -- test IO makes software teams both faster, and more flexible.

Our community of tens of thousands of professional QA testers ensures on-demand availability when you need testing, and guarantees coverage across all the devices, operating systems, regions and languages that matter to you. Test results are delivered in as little as an hour within the development tools you already have in place, or via web app.

Founded in Berlin in 2011, test IO is headquartered in San Francisco, and is the trusted testing partner of leading companies such as Lonely Planet, Thumbtack, Headspace, and Carnival Cruise Line.

test IO

✉ hello@test.io
☎ +1 (415) 937-6859
🌐 www.test.io
📍 535 Mission St.
14th Floor
San Francisco, CA
94104