A RATIONAL GUIDE TO

Testing in Production

by Rob Zuber

In the increasingly complex world of risk management in software delivery, testing in production has been promoted from its status as a joke to a key tool in the validation toolbox. Therefore, we have an obligation to our businesses, as well as the well-being of our users, to look at what it would mean, and what it would cost, to do it well.

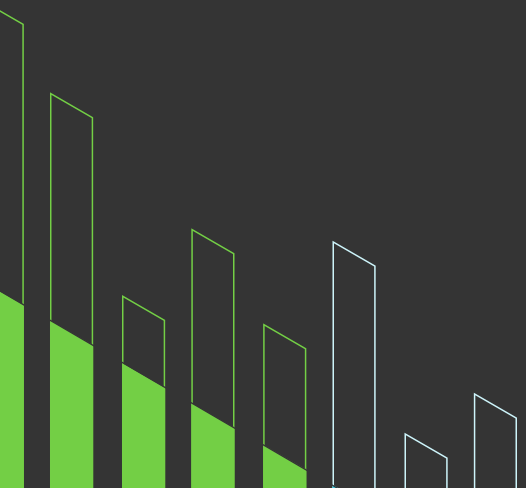
CircleCI CTO Rob Zuber, will walk through what thorough risk assessment looks like in today's software landscape, and provide frameworks and tools to help companies of all sizes and industries make prudent and well-reasoned testing choices.

Change has become a massive supply chain challenge that every company is now dealing with.

Over the last 10 years, the tech industry has adopted new testing methodologies to help address the ever-growing sources of change. As our choices for how to evaluate the correctness of our software in both pre-production environments and beyond have increased, we need a better way to assess how each testing methodology fits into the landscape of cost and risk. This type of evaluation is especially critical when we consider testing in production. The combination of ever-increasing software complexity and vastly improved operational tooling has led to a world where production validation is the right business decision in certain cases.

Testing in production can be a confusing term, because it's often associated with practices that don't look like traditional testing. For that reason, we think of testing in production as an extension of the spectrum of validation. Production validation, when used in conjunction with other testing methodologies, helps teams mitigate risk as trends like increased use of third-party services and larger and larger datasets make it more difficult to fully validate code before it hits production. That's where the idea of continuous validation comes in. It's important to validate that all changes and dependencies that developers rely on are working, and that the ground isn't changing under their feet. Thinking about testing in production as another component of your validation strategy (rather than a replacement for it) helps teams bolster their confidence and increase speed to market. There are, of course, costs and risks associated with testing in production, and I'll explore when this strategy is most useful and how to go about it safely.

COST & RISK



Many people taking roles as software engineers today are trained in computer science. The growth from the study of computer science to the practice of engineering is about taking that theoretical knowledge and learning how to apply it to a constrained world. One of the major considerations of real-world projects that goes beyond the theoretical, is thinking about the costs and risks that your projects pose to your business.

> *"Over time we realized that these larger and larger planning efforts were actually increasing risk, not reducing it."*

A concrete comparison

While a common practice in engineering is performing a cost comparison across projects to find the greatest return on time and resources, a second common comparison is to look at possible adjustments to the cost for the same return. In an example taken from civil engineering, the concrete used in the pillars of highrise buildings is about five times stronger than concrete used in a sidewalk, and for good reason. Not only is the weight to bear higher on the highrise pillar, but the impact of a failure in that pillar is orders of magnitude higher than the result of a cracking sidewalk.

More certainty in the output requires higher cost.

On the surface, this example appears to be just about the cost of materials, but going one level deeper, the cost of producing higher-strength concrete reflects the additional work that goes into it: stricter gates on the quality of the source materials, more waste, and higher levels of testing to ensure that standards are met.

It turns out that the increasing cost associated with reducing defects in construction materials has more than a few first-order parallels to software.

What causes risk in software delivery?

For the purposes of this discussion, software risk is limited to the risk of delivering software to end users that does not behave as they would expect. The primary driver of this type of risk comes from changes made to the software itself.

In the early days of software engineering, we had a fundamentally broken view of risk, which was that with enough effort, we could eliminate it. Or, if not eliminate it altogether, then minimize it through ever-increasing investments in upfront analysis, which we expected would pay dividends in risk mitigation.

Over time we realized that these larger and larger planning efforts were actually increasing risk, not reducing it. That is because as preparation and planning increased, delivering to users – whether as

a SaaS deploy, App Store submission, or otherwise – became a climactic event. Each one had more code and more lead time than the last. This in turn created a much bigger surface area for possible failures associated with releases, more risk, and higher costs.

Since every change is a source of risk, conventional thinking focused on introducing fewer changes and having each of these changes well-validated with vetted designs, many sign-offs, and large QA cycles. However, by increasing batch size, we now know that we were also ultimately increasing risk. This method had the added effects of slowing delivery and increasing the time to resolve issues due to forgotten context.

Predicting, constraining, and embracing risk

Eventually, our industry realized that by slicing changes into the smallest possible increments, we could reduce the risk associated with any one change.

Since adopting the model of making more frequent, smaller changes to a codebase, the industry has become much better at quickly responding to surprises and failures. We've created TDD, Agile methodologies, and continuous deployment, which all help us design for and respond to uncertainties. Using these methodologies, we've been able to greatly reduce the risk of unforeseen bugs leaking into production environments where they could affect end users.

In practice, reducing change into smaller increments often looks like deploying multiple changes to a production codebase per day. The tooling that allows us to do that, such as CI/CD with comprehensive test coverage, gives us the confidence to move quickly because we know we will not deploy anything to a production environment until it's been tested and validated. By thoroughly testing code before it ever reaches production, we've been able to maintain the benefits of more lightweight planning cycles, shorter feedback loops with realtime user feedback, all with higher confidence in our code and reduced risk. This has been a good thing.

But technology continues to evolve, so while we were focused on minimizing risk, the problem space shifted under our feet, and introduced new kinds of risks that didn't exist before (we'll cover these new developments in depth in the next chapter). Over the past few years, increasing complexities in the landscape of software development have made comprehensive pre-production testing more complicated and in some cases, prohibitively difficult or expensive. As such, we've seen a rising enthusiasm for "testing in production" — a term once used to invoke imagery of disastrous negligence, but which has shifted to represent intentional practices of modern software delivery. In some cases, it is the most practical choice.

Despite the many factors enabling broader adoption of production validation, it's equally important, if not more important, to rationally assess the risk of any testing decision, applying inputs from your particular business and risk profile, in order to ascertain the appropriate testing protocol. Yes, the world has gotten more complex, but just as in a decision about concrete quality, software deployment decisions have real costs and risks to consider. We'll always be testing something in production, but our success is more dependent than ever on having good frameworks for assessing the risk, the cost of change, and the potential upside to those choices.

> *"But technology continues to evolve, so while we were focused on minimizing risk, the problem space shifted under our feet, and introduced new kinds of risks that didn't exist before."*

THE RISING COST (AND RISK) OF PRE-PRODUCTION TESTING

The changing world of software development

As software becomes a competitive differentiator for more and more companies (not just in the tech industry but in more traditional verticals like retail, health, and finance), software teams are optimizing for faster and faster delivery. A few trends have appeared alongside these goals:

- Greater use of third-party services and tools
- Microservice architectures
- Larger and larger data sets

While these factors have helped us refine and optimize our software development, they have also reduced our ability to be confident in complete validation within a pre-production environment, so they're important to discuss here. Put another way, the presence of any (or all) of these factors have increased the cost of achieving the same confidence in our code before we ship it. Let's go through them one by one to see how they can make it more difficult to thoroughly test your code.

Greater use of third-party services and tools

The introduction of more and more third-party services makes it increasingly difficult to be certain (or even aware) of all the changes to your codebase. In a world where we build on larger and larger frameworks, it's not uncommon for a small fraction of your codebase to be originated by your developers — the rest might be shared services and libraries. These resources, while efficient for development, are less predictable than code written by your team, and in the case of third-party services (such as payment or analytics platforms), it's possible for their behavior to change without any change in your own code. Therefore, it's much more difficult to have real insight into all the changes which may affect your users.

Microservice architectures

Microservices have allowed us to move a great deal of complexity out of single monolithic pieces of software. Minimizing a piece of software is great for a developer tasked with working on it, but that complexity doesn't just go away — instead it's moved into the interactions *between* the pieces of software.

Done well, microservices are intended to be effectively independent. As a developer, I should be able to manage my service without knowing about services that depend on it. However, if I wanted to test my service thoroughly, I would have to consider these dependencies. For full confidence in the output of my tests, and the performance of my service in the real world, I would have to set up a complete environment (including all the services that are dependent on mine) to make sure my changes didn't break any of them. Doing so would pose a huge cost, and also undermine my independence and agility to push changes.

Larger and larger data sets

As an industry, we've become obsessed with data-driven applications. Along with our goal of collecting as much data as possible to improve the capabilities of our applications, we're also

subjecting ourselves and our systems to processing these ever larger scales of data. Storage is cheaper, computers are fast. But with increasingly large data sets comes the responsibility of managing larger test data sets as well: with appreciable costs related to moving them around, storing, and archiving them. This also affects the cost and feasibility of thorough pre-production testing. At CircleCI, for example, we have terabytes of data in our production environment — it would be an immense undertaking to inject that into every test run.

There are additional considerations for large data sets as well. Things that look like they'll operate effectively in a CI run can see different results in a production environment.

For example, cardinality, a measurement of the amount of uniqueness of data, is something that people don't usually plan for. Let's say I have a column in my database called "Last Name." In some countries, I'll get huge diversity within that column, while in others, I might have very little diversity. In those cases, indexing on last name won't be a super fast lookup. Given real data instead of test data, I might make a different performance decision; for example, indexing on first name instead of last name. Some teams try to solve this problem by pulling production data into their test environment, but privacy and security concerns lead to anonymization, resulting in data that *still* doesn't represent the real world. Imagining all the edge cases, creating the appropriate test data sets, and operating on them would be an immense task. By putting it into production and using real production data, you can attempt to minimize the blast radius of mistakes while seeing if your service will really work.

Each trend in its own right has made it harder to feel fully confident when performing software validation in a pre-production environment. Combined, attempting to itemize the possible issues is overwhelming. Combating this risk with increasingly complex test platforms results in an upward trajectory of cost that eventually becomes unsustainable. While testing in production brings its own set of risks, it is another tool in the toolkit that can be thoughtfully applied to cost-effectively identify failures and edge cases with real-world data.

THE INFLECTION POINT

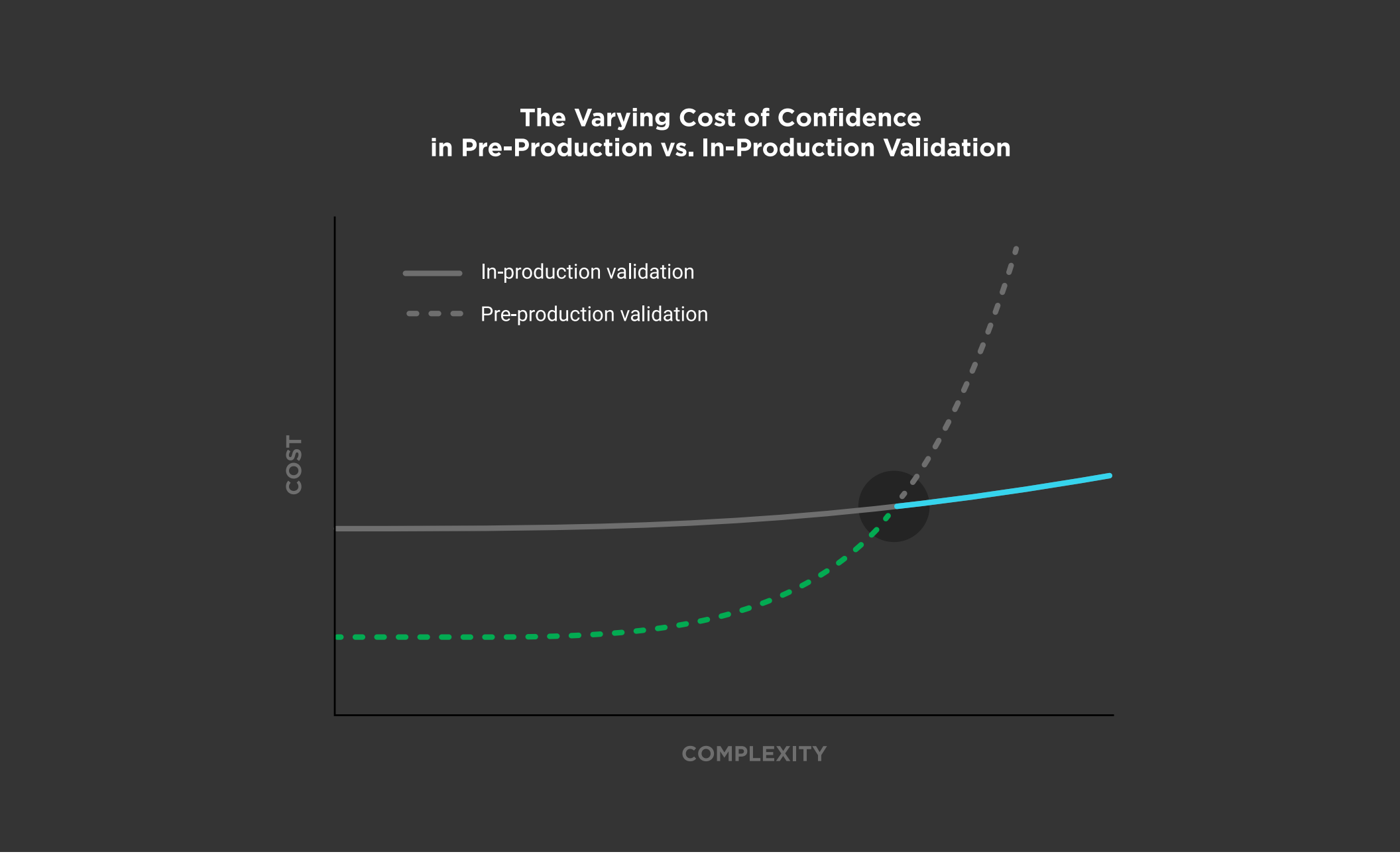
We've now looked at several reasons why catching specific classes of issues in a pre-production environment has become more complex, and therefore more costly. It stands to reason that these costs can and will increase to the point where we

> *"And this inflection point — where the cost of pre-production validation supasses that of in-production validation — isn't static; it's moving."*

become more willing to take on the risk of finding some issues in production instead, and subsequently fixing them as quickly as possible. This is where testing in production outgrows its history as a meme and steps in as a viable methodology.

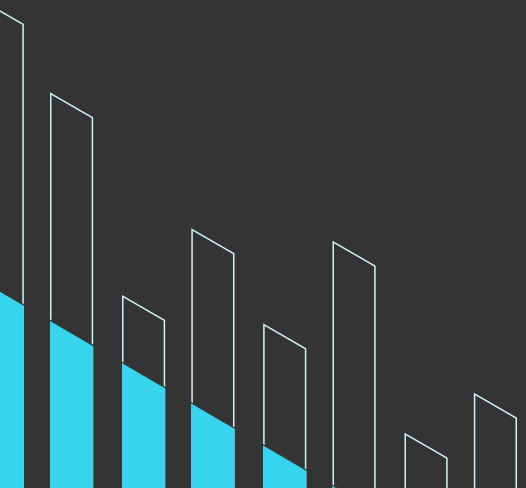
As some classes of bugs or issues are prohibitively expensive (or impossible) to find via pre-production testing, and we find ourselves testing *in* production, whether we ever wanted to or not, it's natural to think about investing some of our energy in limiting blast radius and speeding up remediation of production bugs.

And this inflection point — where the cost of pre-production validation supasses that of in-production validation — isn't static; it's *moving*. Since the cost of testing in production is somewhat fixed, and the cost of thorough pre-production testing is increasing with the growing complexity of our development processes, the threshold is something we need to be dynamically assessing.



In this increasingly complex world of risk management, testing in production is now a key tool in the toolbox. Let's look at what it would mean, and what it would cost, to do it well.

REDUCING THE COST OF TESTING IN PRODUCTION



Measuring the costs of testing in production

The cost of testing in production is primarily associated with impact to users. This cost can be measured across four axes:

- Number of users affected
- Category of users affected
- Severity of the impact
- Duration of the impact

NUMBER OF USERS AFFECTED

While this metric is fairly straightforward, to infer the impact of a failure from the raw user numbers, you need to account for the total number of users on your service, so I advise taking user quantity as a percent. Note: in order to minimize the number of affected users, it needs to be possible to serve specific users with new code vs. old code, using tools like canaries and feature flags, discussed later.

SEVERITY OF THE IMPACT

This refers to the gravity of the impact on a user and should be focused on the most important pathways a user follows through the system. For example, at CircleCI, an issue that keeps a user from building code is typically much more severe than an issue that prevents them from changing a build setting.

CATEGORY OF USERS AFFECTED

This refers to different categories of users you might have on your system, such as free/trial users, small business vs. enterprise, or ideally, users who are opted-in to beta or early release access. The specific groups which are most costly to affect is a matter of your business, but knowing which they are and targeting changes accordingly is a great risk-mitigation strategy.

DURATION OF IMPACT

This is the time during which users see the effect of a bad change.

Tooling that can reduce costs

As testing in production finds its place, many vendors and tool providers are releasing tools that support this approach, helping constrain blast radius and mitigate risk. Let's talk about some of these options and explore the ways in which they might help you reduce the cost of catching issues in production. We'll explore the tools in two categories: monitoring and deployment/release tools.

MONITORING TOOLS

Monitoring is a category of tools, generally referring to the combination of logs, metrics, and tracing. There are overlaps in what these tools can support as well as how you can best apply them in your environment, but let's start with the basics.

One more note before we dive in: regardless of whether you choose to employ testing in production, you know you need debuggability in your production environment. By laying the groundwork for debugging, you've already made a huge investment in monitoring. So in going the last mile to do it really well, you can get double duty out of these tools for your production testing as well, and you'll be offsetting costs elsewhere. This is a smart investment.

LOGS

Logs help reduce the time of impact by assisting in debugging. While logs have been in use for a long time, we now have the tools to aggregate them. However, many logs remain unstructured — they represent the concerns of the developer at the time they were written, rather than with the intent to help someone debug at the time of failure. Anomaly detection helps by giving us some ability to identify what matters in logs more quickly. When writing new log lines, we should invest in structuring, and make sure we write them in a way that is operation-oriented before shipping them.

METRICS

When we talk about metrics related to monitoring, we think of the dashboards using stats, and counters that we emit from our systems, and that allow developers to identify changes in system behavior

with little effort. While these are generally more intentional than logs, they still suffer from being oriented around what the developer thought would go wrong instead of what is actually going wrong. Similar to logs, anomaly detection also helps here, but to get to real confidence, we have to be confident in our anomaly detection as well as in our coverage through metrics. Metrics can often tell you that something is going wrong, but rarely why, unless you can quickly correlate the cause and effect. The pure cost of thorough metrics can be high as you scale, but since these have invaluable applications beyond just testing in production, they are well worth the investment.

TRACING

Tracing refers to the tools that are used to capture and visualize the flow of a request through a system. While tracing tools are helpful in monoliths (for the purposes of logging a request through the codebase), tracing really shines when you're faced with a single customer ask that requires requests to multiple backend services — and possibly multiple roundtrips to the same service.

Tracing is far less common than logging and metrics, but as we build more distributed systems, it's becoming essential for basic operations. Teams should start by investing time on tracing, then, they'll have the tools to handle more 'test-in-prod' scenarios. The cost of implementing tracing is similar to other monitoring tools, and like other monitoring tools, it reduces the duration of impact through enhanced debugging. It's worth calling out that all monitoring tools can also prevent some of the impact caused by things like scaling, because the tools allow you to see when your system is reaching its threshold before usability starts degrading.

Deploy and release

This group of tools refers to the ways in which users gain access to new code.

BLUE/GREEN DEPLOYS

Blue/green deploys (sometimes called “red/black” deploys) are deploys in which a full copy of a system such as a service or monolith is kept running while traffic is cut over to a new full deploy of that system. This allows developers to monitor performance and cut back immediately if there are any issues. Blue/green deploys can be complex to set up and tend to be very system-dependent. While the cost used to be high in the days of servers, the addition of elastic compute and the cloud have lowered the cost significantly. This protocol is only useful if you can determine the impact of turning them on in real-time (see monitoring). If so, they can greatly reduce the duration of impact.

CANARY DEPLOYS

These are deploys of software updates where, rather than replacing all instances of the codebase (whether it’s a monolith, a service, or something else), a small subset of instances are replaced for the purpose of validating functionality. In some cases, specific traffic can be routed to those canaries, and the options for routing are limitless: specific work types, specific customers, specific geographies, times, etc. The canary’s behavior is monitored, and when it has both proven its intended effect and proven not to cause any unexpected negative effects, the remainder of the instances are generally replaced as well. The timeline for this overall deploy is completely open based on the needs of the business. Like blue/green deploys, canaries are only useful if you can determine the impact in real-time of turning them on (see monitoring). You can increase the value of your canaries by making it easier and faster to remove any routing sending traffic to that canary; ideally, it can even be automated when an error condition is detected. Canaries will help control the number of users affected, the category of users affected, and the duration of the impact.

FEATURE FLAGS

Feature flags are code-level wrappers around sections of a system that are concerning or high

risk. Most systems allow configuration to expose new capabilities to subsets of users, including by geography, user type, and organization type. A major benefit of feature flags is that they allow you to quickly remove buggy code from production — you just need to turn the flag off. The cost of implementing feature flags can be low, but tends to grow over time. They become a major source of technical debt with the combinatorics of exploding, poorly understood, poorly tested code paths. Feature flags can be useful in combination with monitoring tools, but often come with significant management requirements. Typically, feature flags can help control the number of users affected, the categories of users affected, and the duration of impact.

REVERSIONS & QUICK FIXES

Reversions and quick fixes are the result of great deployment automation. When you can deploy quickly and confidently, you can push a fix or a revert to a historical known-good state to deal with any issues that arise. Even if you're not doing continuous deployment, if you have proper tooling in place, you should be able to confidently deploy at any time — knowing that you can quickly respond to a failure as soon as it happens.

Reversions and quick fixes help with duration of impact. While we as a community have had access to many of these tools for a while, there are myriad ways to apply them. Smartly combining tools with an approach that treats them as part of your validation cycle makes it possible to think of these as part of your testing suite, not just as tools for crisis management.

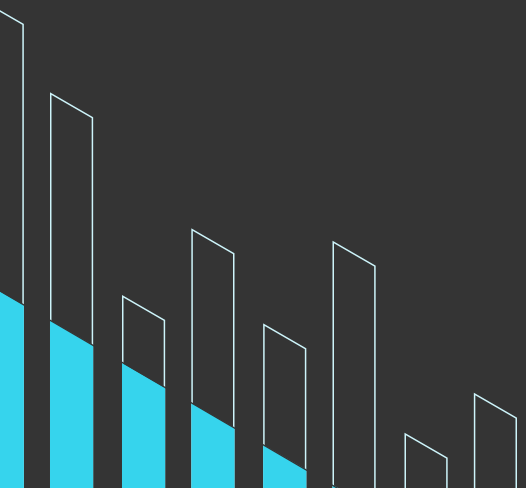
If you're using these tools, even just for crisis management, then you've already made the investment required to get them set up. The real point of leverage is in recognizing the scenarios in which it makes more sense to catch things with these tools that we would have previously thought about catching in a test environment, and planning for these situations intentionally.

A summary of tooling and risk

Below is a chart summarizing the contribution of these various tools to managing risk and user impact in a production environment. One notable takeaway is the limited capability to address the severity of an impact. A small number of users impacted for a short period can still witness a problem in a critical use case in your product. For this reason, it's helpful to have a clear understanding of which of your flows are absolutely critical to the core function of your product and which are merely inconvenient if they are temporarily unavailable. This knowledge will help define the acceptable level of risk in different areas more clearly, and help identify which classes of errors are acceptable to catch in production.


Number of users
Category of users
Severity of impact
Duration of impact
Logs
Metrics
Tracing
Blue / green deploys
Canary deploys
Feature flags
Reversions & quick fixes
MONITORING TOOLS
DEPLOYMENT / RELEASE TOOLS

A REAL-WORLD EXAMPLE



Now that we have an expanded toolkit, let's take a look at a real-world example, evaluating ROI and looking for the cases that are better caught in production instead of in a pre-production environment. For the purposes of illustration, we'll focus on the large dataset case, but the approach should apply wherever you are trying to make this decision.

It is not unusual or uncommon for production OLTP datastores to be operating on data sizes on the order of 10TB–this is certainly the case at CircleCI. The volume of data at CircleCI matters for a couple of reasons. First, there are likely some surprising entries in the dataset after 9 years of operation – entries that may not be considered by a brand new developer modifying functionality of our systems in novel ways. Second, any database engine with a query planner introduces the possibility of

executing queries differently on datasets that are vastly different in size. An index that gets chosen in a small testing environment may no longer be selected in the larger production dataset.

Once we decide to execute at least a portion of our testing on a production-sized dataset, we have three major considerations to try to solve for:

- Getting production data into our testing environment
- Ensuring datastore capacity in testing with either multiple users or largely parallel test systems
- Maintaining privacy in our production data, likely through anonymization

Starting with making production data available for testing, there are two fairly obvious approaches: load data for individual test runs, or maintain datastores that can be accessed by those test runs.

If we are loading the data into a dynamic test environment, the lowest overhead option is to transfer a disk-level image, as this avoids all the time and effort involved with writing through our database engine. At 10TB, if we had perfect Gb ethernet (an unreasonable expectation), the data transfer alone would take about 22 hours. Not great for kicking off a CI run.

A managed instance of the DB is much more viable at this size, especially if you have any goals for performance.There are some initial barriers, such as the desire to split our tests to run in parallel and the volume of CI runs we are doing at any given time. Let's start from the minimum possible cost by assuming we only test merges to master (ie, right before production) and we somehow manage to identify the tests that cover the likely challenging scenario so they can be serialized in a subset of our workload and use a single shared DB instance.

Assuming no requirements for high availability or resilience in this test instance, along with modest performance, we can probably use a basic xlarge instance from EC2 (around $150/month). Adding gp2 EBS storage to hold the data brings that up to $250/month or $3k/year. That actually sounds pretty good. Except now we have a replica of production data that needs maintaining and it's in the deploy path for our core application, so it needs dedicated engineering time. Assuming there is ETL tooling, security, and operational maintenance, let's estimate that takes up 50% of an engineer's time (or smaller amounts from multiple individuals).

Since our dataset includes Personally Identifiable Information (PII) in it so our engineer also has to anonymize the data. We'll pretend we can get that done within that 50% time allocation.

In theory, we now have a viable production-like dataset. Unfortunately, when we anonymized the data, we removed our two biggest classes of error detection. The old data of surprising shape is now a string of random characters like every other record. And we've changed the cardinality of the data in a way that makes the effectiveness of indexes impossible to predict.

Admittedly, the cost of engineering time is highly variable, but let's estimate we're now spending over $50k/year. We have single point of failure in our delivery pipeline, we've manipulated the data in a way that makes it decidedly *not* production, and we've chosen a subset of our tests to run to keep things flowing. The usefulness of this effort is also quite variable, but at CircleCI I'd estimate we'd catch around 5-10% of the edge cases that we'd find in production.

To increase test coverage, we'd either have to ramp up our database instance count so we could run in parallel, or slow down all delivery to share a single instance on more tests. Let's choose the former and spend $30k/year on database instances so we can

get 10x parallel builds talking to these DBs. Maybe now we'll catch 25% of cases.

This entire model assumes that we write all of our tests in a way that will identify a slow response to a query during our test cycle. That is very uncommon from my experience. Even teams that are good at performance testing are making conscious decisions about which code paths to test. Let's assume that building and managing the tooling to support broader validation of response times throughout the application is equivalent to at least another 50% of an engineer's time.

All in, we're getting close to $150k/year. What happens if we don't do this at all?

The cost when we catch these issues in production is mostly measured in terms of the impact to users. So we need some reasonable assumptions on two values. First, the likelihood of creating a bug that is difficult to catch in a developer or CI environment, related to the scale of data, and that has a severe impact on users once it's in production. Based on my own experience, I'd call this a once-per-year event. Second, we need to estimate what the financial impact is of this event. At once per year, we're trading off against a $150k alternative.

With some basic monitoring tools and an all-or-nothing deployment strategy, you'd notice this event pretty quickly. Based on what we see in our customer base, a team with the ability to revert the changes might identify and fix them in about 15 mins. Is a 15-minute degradation in your application going to cost you $150k? Only you know that.

As an example, on August 1, 2012 Knight Capital lost USD $440m in 45 mins–close to $150 *million* every 15 minutes. If their issue was preventable for $150k / year (it was actually preventable for much less), they could have spent that for a thousand years to reach their inflection point. On the other hand, the average blogging engine with no SLAs to their customers

would gain very little from this level of investment.

So far, that impact assessment assumes all of your customers are affected. If we layer in some better operational tooling, like a canary deployment, we can put our updated software into production and only route 2% of traffic to it. Assuming your usage is high enough, you should quickly see the same effects but only affect 2% of your traffic. Since we're talking about code changes over large data sets, we just have to remove the routing to the canary. This is near instant, but let's call it 5 minutes to recover to be generous.

One third of the impact time and one fiftieth of the impacted traffic. Now you'd have to be in a position where a 5-minute degradation for 2% of your traffic (possibly only 2% of users) would be worth $150k. Put differently, the previously described 15-minute degradation for all users would be worth $22 million to your business.

It's important to highlight that none of these numbers are correct for your business. In fact, every detail will likely be different based on your team, your product, and your users. But the two things that you should take away are the simplicity and the precision. This exercise is not one of itemizing every detail. Think of it as a back-of-the-envelope calculation or Fermi Problem. Identify the dominant terms and you'll quickly spot your answer. We all have limited dollars to invest and want to maximize our returns. Expanding your horizons to include production validation and knowing how to make this tradeoff will help you do just that.

THE EVOLVING VALIDATION TOOLKIT

Technology trends and shifting cost models mean that testing in production is here to stay. And that's a good thing. It's another tool to add to the validation toolkit — a collection of tools and strategies that has grown and evolved over the last 20 years.

While history has shown us that we can't prevent things from going wrong, we as a community have also matured in our agility in accepting risk. But when things do go wrong, we still need to fix them. Being able to manipulate your production environment quickly and with confidence is more important than ever in a testing-in-production world. And that depends on a reliable CI/CD pipeline, solid deployment practices, and a pipeline that continuously validates your code at every step. Once you've identified what isn't working by testing in production, the crucial work becomes discerning how it got there in the first place and how you'll get the fix out to replace it. If you can't do that, you've taken on unnecessary risks in your test-in-production model.

Another way of looking at it is seeing production validation as just a logical continuation of your delivery process. Leaning on both fast CI and testing in production strategies can help you maintain a tight feedback loop and keep your team in flow.

> *"Once you've identified what isn't working by testing in production, the crucial work becomes discerning how it got there in the first place and how you'll get the fix out to replace it."*

Using a combination of strategies helps you identify issues faster, and makes a high-throughput CI/CD pipeline even more important — once you've understood the problems, you need to be able to quickly push reliable fixes.

A final thought: the cost threshold for testing in production used to be high, and now it's considerably lower. But embracing testing in production is only the first step in making cost optimization tradeoffs. Let's say your system has just crossed the inflection point at which testing in production becomes the more economical option. Excellent: you've put another tool in your toolkit and it's a great way to manage complex risks. But ask, "Why did I build something so complex that I can *only* test it in production? Can I build it better or design it more thoughtfully?" No matter how cheap testing in production becomes, you will always be able to drive down the cost of testing your service through better design.

CircleCI was founded on creating confidence in code and helping our customers move quickly.

We've seen the world of software development evolve at an unprecedented pace as our customers have moved from Rails to iOS to Docker to today's massive explosion of services.

Our highest priority has always been enabling your confidence through validation of your code. And pre-production as well as production validation are both part of that process. I encourage you to think both more analytically as well as holistically about your CI, your in-production testing, and any and all of your validation strategies as components of the same cycle, supporting the same aim: confidence in code and speed to market. My hope is that by looking at these pieces of the validation puzzle all together, instead of separating them into the concern of developers vs. operators, you'll be able to make better cost and risk decisions, ship better code, and create better products.